EXPERT INSIGHT

SPRINT book

Kubernetes

Полное руководство по развертыванию и управлению Kubernetes в облачных и локальных средах

Второе издание

Предисловие

Эд Прайс

технический писатель, контент-менеджер
по облачным архитектурам

Гиниш Мадаппарамбат
Расс МакКендрик

packt

The Kubernetes Bible

Second Edition

The definitive guide to deploying and managing Kubernetes across cloud and on-prem environments

Gineesh Madapparambath
Russ McKendrick

‹packt›

Kubernetes

Второе издание

Полное руководство по развертыванию и управлению Kubernetes в облачных и локальных средах

Гиниш Мадаппарамбат
Расс МакКендрик

SPRINT book 2026

Гиниш Мадаппарамбат, Расс МакКендрик

Kubernetes. Полное руководство по развертыванию и управлению Kubernetes в облачных и локальных средах

2-е издание

Перевел с английского С. Черников

Научный редактор С. Метелёв

ББК 32.988.02-018
УДК 004.738.2

Мадаппарамбат Гиниш, МакКендрик Расс

М13 Kubernetes. Полное руководство по развертыванию и управлению Kubernetes в облачных и локальных средах. 2-е изд. — Астана: «Спринт Бук», 2026. — 752 с.: ил.

ISBN 978-601-12-8674-9

Kubernetes стал основной платформой оркестрации контейнеризованных приложений. Но его мощь часто скрыта за сложной терминологией и тонкостями настройки. Книга поможет разобраться в этом многообразии, предлагая практические примеры и советы экспертов для решения самых непростых задач.

В новом издании описываются передовые методы безопасности и приемы бесперебойного развертывания и масштабирования, гарантирующие неизменную доступность сервисов. Получите навыки создания приложений, готовых к продакшену, обеспечьте безопасность сред разработки и станьте настоящим гуру в области защиты. К концу работы с этим исчерпывающим руководством вы будете обладать всеми необходимыми навыками и знаниями, чтобы оркестрировать свои контейнерные приложения с ювелирной точностью, добиваясь их максимальной производительности и масштабируемости.

Хватит довольствоваться базовым управлением контейнерами! Выведите оркестрацию на новый уровень.

16+ (В соответствии с Федеральным законом от 29 декабря 2010 г. № 436-ФЗ.)

Все права защищены. Никакая часть данной книги не может быть воспроизведена в какой бы то ни было форме без письменного разрешения владельцев авторских прав.

Информация, содержащаяся в данной книге, получена из источников, рассматриваемых издательством как надежные. Тем не менее, имея в виду возможные человеческие или технические ошибки, издательство не может гарантировать абсолютную точность и полноту приводимых сведений и не несет ответственности за возможные ошибки, связанные с использованием книги. В книге возможны упоминания организаций, деятельность которых запрещена на территории Российской Федерации, таких как Meta Platforms Inc., Facebook, Instagram и др. Издательство не несет ответственности за доступность материалов, ссылки на которые вы можете найти в этой книге. На момент подготовки книги к изданию все ссылки на интернет-ресурсы были действующими.

ISBN 978-601-12-8674-9

© Packt Publishing 2024.
First published in the English language under the title ‘The Kubernetes Bible - Second Edition – (9781835464717)’
© Перевод на русский язык ТОО «Спринт Бук», 2026
© Издание на русском языке, оформление ТОО «Спринт Бук», 2026

Изготовитель: ТОО «Спринт Бук». Место нахождения и фактический адрес:
010000, Казахстан, город Астана, район Алматы, проспект Ракымжан Кошкарбаев, д. 10/1, н. п. 18.

Дата изготовления: 06.2026. Наименование: книжная продукция. Срок годности: не ограничен.

Подписано в печать 24.04.26. Формат 70×100/16. Бумага офсетная. Усл. п. л. 60,630. Тираж 700. Заказ 0000.

Краткое содержание

Об авторах 21
О научных редакторах 22
Предисловие 24
Введение 26
Глава 1. Основы Kubernetes 34
Глава 2. Архитектура Kubernetes: от образов контейнеров к работающим подам 71
Глава 3. Установка первого кластера Kubernetes 115
Глава 4. Запуск контейнеров в Kubernetes 143
Глава 5. Использование многоконтейнерных подов и паттерны проектирования 181
Глава 6. Пространства имен, квоты и лимиты для реализации мультитенантности в Kubernetes 218
Глава 7. Настройка подов с помощью объектов ConfigMap и Secret 248
Глава 8. Публикация подов с помощью сервисов 281
Глава 9. Постоянное хранилище в Kubernetes 326
Глава 10. Запуск рабочих нагрузок производственного уровня в Kubernetes 366
Глава 11. Использование объекта Deployment для управления рабочими нагрузками без сохранения состояния 384
Глава 12. Объект StatefulSet: развертывание приложений с сохранением состояния 420
Глава 13. Объект DaemonSet: запуск одного пода на каждом узле 457
Глава 14. Работа с Helm-чартами и операторами 474
Глава 15. Запуск кластера на платформе Google Cloud Platform (GCP) с помощью Google Kubernetes Engine (GKE) 515
Глава 16. Запуск кластера на платформе Amazon Web Services с помощью Amazon Elastic Kubernetes Service 537
Глава 17. Запуск кластера на платформе Microsoft Azure с помощью Azure Kubernetes Service 561
Глава 18. Безопасность в системе Kubernetes 587
Глава 19. Продвинутые методы планирования подов 641
Глава 20. Автоматическое масштабирование подов и узлов в Kubernetes 668
Глава 21. Продвинутые возможности Kubernetes 705

Оглавление

Об авторах 21
О научных редакторах 22
Предисловие 24
Введение 26
Кому адресована эта книга 27
Содержание книги 28
Как извлечь максимальную пользу из этой книги 30
Загрузка файлов с кодом из примеров 31
Загрузка цветных изображений 32
Условные обозначения 32
От издательства 33
О научном редакторе русскоязычного издания 33

Глава 1. Основы Kubernetes 34
Монолиты и микросервисы 35
Стремительное развитие Интернета, начавшееся в конце 1990-х годов 35
Потребность в более частых релизах программного обеспечения 35
Переход к гибким методологиям разработки 36
Переход от локальной инфраструктуры к облаку 39
Почему облако хорошо подходит для масштабирования 42
Монолитная архитектура 43
Микросервисная архитектура 44
Выбор между монолитной и микросервисной архитектурами 47
Контейнеры 48
Почему контейнеры подходят для развертывания микросервисов 48
Преимущества изоляции с помощью контейнеров 49
Контейнерные движки 53
Основы работы с контейнерами 53
Управление контейнерами с помощью Kubernetes 57
Kubernetes — это платформа для запуска рабочих нагрузок в рабочей среде 57
KubeVirt — мост между контейнерами и виртуальными машинами 58

История развития Kubernetes 59
Истоки Kubernetes 59
Кто управляет Kubernetes сегодня 60
Место Kubernetes в современном ландшафте 60
Дальнейшее развитие Kubernetes 62
Задачи, решаемые системой Kubernetes 62
Обеспечение высокой доступности 63
Управление релизами и развертыванием контейнеров 64
Автоматическое масштабирование контейнеров 66
Сетевая изоляция 67
Управление доступом на основе ролей (RBAC) 68
Поддержка рабочих нагрузок с сохранением состояния 68
Управление ресурсами 68
Когда Kubernetes не является оптимальным решением 69
Резюме 69
Дополнительные источники информации 70

Глава 2. Архитектура Kubernetes: от образов контейнеров к работающим подам 71
Технические требования 71
Происхождение названия Kubernetes 72
Различие между управляющими и вычислительными узлами 72
Мастер-узлы и рабочие узлы 72
Контейнеры Linux и Windows 73
Компоненты Kubernetes 73
Компоненты управляющего слоя 77
Компоненты вычислительных узлов 78
Дополнения 78
Управляющий слой в готовых кластерах Kubernetes 78
Компоненты управляющего слоя 79
Компонент kube-apiserver 79
Хранилище данных etcd 83
Компонент kube-scheduler 86
Компонент kube-controller-manager 88
Компонент cloud-controller-manager 89
Компоненты вычислительных узлов 90
Контейнерный движок и среда выполнения контейнеров 90
Агент kubelet 94
Компонент kube-proxy 95
Обзор утилиты командной строки kubectl и синтаксиса YAML 96
Установка утилиты командной строки kubectl 96
Роль утилиты kubectl 98

Принцип работы утилиты kubectl ... 100
Автодополнение команд kubectl ... 103
Императивный синтаксис ... 104
Декларативный синтаксис ... 105
Обеспечение высокой доступности Kubernetes ... 108
Одноузловой кластер ... 108
Кластер с одним управляющим узлом ... 110
Кластер с несколькими управляющими узлами ... 111
Управление компонентом etcd в кластере Kubernetes с несколькими управляющими узлами ... 112
Резюме ... 114
Дополнительные источники информации ... 114

Глава 3. Установка первого кластера Kubernetes ... 115
Технические требования ... 116
Установка кластера Kubernetes с помощью minikube ... 116
Установка инструмента minikube ... 117
Конфигурации minikube ... 118
Драйверы для minikube ... 119
Создание одноузлового кластера Kubernetes с помощью minikube ... 120
Получение доступа к кластеру Kubernetes, созданному с помощью minikube ... 124
Остановка и удаление локального кластера minikube ... 126
Создание многоузлового кластера Kubernetes с использованием minikube ... 127
Создание кластера Kubernetes с несколькими управляющими узлами с помощью minikube ... 128
Создание нескольких кластеров Kubernetes с помощью minikube ... 128
Создание многоузлового кластера Kubernetes с помощью kind ... 129
Установка kind в локальную систему ... 130
Создание кластера Kubernetes с помощью kind ... 131
Остановка и удаление локального кластера kind ... 134
Альтернативные учебные среды для освоения Kubernetes ... 135
Killercoda ... 135
Кластеры Kubernetes производственного уровня ... 136
Готовые кластеры Kubernetes, предоставляемые облачными сервисами ... 136
Дистрибутивы Kubernetes ... 137
Инструменты для установки Kubernetes ... 137
Гибридные и мультиоблачные решения ... 138
Выбор подходящей среды ... 139
Запуск Kubernetes в локальной инфраструктуре: вызовы и факторы, которые следует учитывать ... 139
Резюме ... 141
Дополнительные источники информации ... 142

Глава 4. Запуск контейнеров в Kubernetes ... 143
Технические требования ... 144
Понятие пода ... 144
Что такое под ... 144
Каждый под получает IP-адрес ... 147
Проектирование подов ... 148
Запуск подов ... 150
Создание пода с помощью императивного синтаксиса ... 150
Создание пода с помощью декларативного синтаксиса ... 152
Чтение информации и метаданных пода ... 153
Вывод списка объектов в формате JSON или YAML ... 155
Резервное копирование ресурса с помощью операции перечисления ... 155
Получение дополнительных сведений с помощью операции перечисления ... 156
Получение доступа к поду из внешнего мира ... 156
Получение доступа к контейнеру внутри пода ... 157
Удаление пода ... 158
Снабжение подов метками и аннотациями ... 159
Метки и их назначение ... 159
Аннотации и их отличие от меток ... 161
Добавление метки ... 161
Вывод списка меток, присвоенных поду ... 162
Добавление или обновление метки запущенного пода ... 162
Удаление метки, присвоенной запущенному поду ... 163
Добавление аннотации ... 164
Запуск объекта Job ... 165
Что такое Job ... 165
Создание объекта Job с параметром restartPolicy ... 166
Параметр backoffLimit ... 168
Многократное выполнение задания с помощью параметра completions ... 169
Многократное выполнение задания в параллельном режиме ... 170
Завершение задания по истечении определенного времени ... 171
Что происходит при успешном завершении задания ... 172
Удаление задания ... 172
Запуск объекта CronJob ... 173
Что такое CronJob ... 173
Подготовка к созданию объекта CronJob ... 174
Ключ schedule ... 174
Роль раздела jobTemplate ... 175
Управление сроком выполнения объекта CronJob ... 176
Управление историей выполнения заданий ... 177
Создание объекта CronJob ... 177
Удаление объекта CronJob ... 178
Резюме ... 178
Дополнительные источники информации ... 180

Глава 5. Использование многоконтейнерных подов и паттерны проектирования 181
Технические требования 182
Многоконтейнерные поды 182
Сценарии использования многоконтейнерных подов 182
Создание пода, состоящего из двух контейнеров 184
Что происходит при неудачном запуске контейнера внутри пода 186
Удаление многоконтейнерного пода 187
Период корректного завершения работы пода 188
Доступ к конкретному контейнеру внутри многоконтейнерного пода 189
Выполнение команд в контейнерах 191
Переопределение команд, выполняемых контейнерами по умолчанию 192
Контейнеры инициализации 194
Получение логов конкретного контейнера 197
Совместное использование томов контейнерами одного пода 199
Тома в Kubernetes 199
Создание и монтирование тома emptyDir 201
Создание и монтирование тома hostPath 204
Паттерн проектирования Ambassador 206
Как устроен паттерн Ambassador 207
Пример пода с контейнером-посредником 208
Паттерн проектирования Sidecar 209
Как устроен паттерн Sidecar 209
Когда следует использовать паттерн Sidecar 210
Пример пода с контейнером-прицепом 211
Паттерн проектирования Adapter 212
Пример пода с контейнером-адаптером 213
Контейнеры-прицепы и их нативная реализация в Kubernetes 215
Резюме 217
Дополнительные источники информации 217

Глава 6. Пространства имен, квоты и лимиты для реализации мультитенантности в Kubernetes 218
Технические требования 218
Пространства имен в Kubernetes 218
Важность пространств имен в Kubernetes 219
Использование пространств имен для разделения ресурсов 220
Пространства имен по умолчанию 221
Влияние пространств имен на ресурсы и сервисы 222
Просмотр пространств имен в кластере 223
Получение данных о конкретном пространстве имен 223
Создание пространства имен с помощью императивного синтаксиса 224
Создание пространства имен с помощью декларативного синтаксиса 224
Удаление пространства имен 225

Создание ресурса внутри пространства имен ... 225
Вывод списка ресурсов внутри конкретного пространства имен ... 227
Задание текущего пространства имен с помощью kubectl config set-context ... 228
Перечисление ресурсов, находящихся в конкретном пространстве имен ... 229
Область видимости и пространство имен ... 229
Не все ресурсы принадлежат пространству имен ... 230
Разрешение DNS-имен сервисов с помощью пространств имен ... 231
Лучшие практики работы с пространствами имен в Kubernetes ... 231
Настройка квот и лимитов на уровне пространства имен ... 233
Зачем задавать квоты ... 233
Как поды потребляют ресурсы ... 234
Как поды могут запрашивать вычислительные ресурсы ... 235
Что означают эти метрики ... 235
Как можно ограничить потребление ресурсов ... 236
Зачем нужны объекты ResourceQuota ... 240
Создание объекта ResourceQuota ... 241
Квоты на использование ресурсов хранилища ... 244
Вывод списка объектов ResourceQuota ... 244
Удаление объекта ResourceQuota ... 244
Знакомство с объектом LimitRange ... 244
Вывод списка объектов LimitRange ... 246
Удаление объекта LimitRange ... 246
Резюме ... 246
Дополнительные источники информации ... 247

Глава 7. Настройка подов с помощью объектов ConfigMap и Secret ... 248
Технические требования ... 248
Объекты ConfigMap и Secret ... 249
Отделение приложения от его конфигурации ... 249
Как поды потребляют объекты ConfigMap и Secret ... 252
Настройка подов с помощью ConfigMap ... 253
Перечисление объектов ConfigMap ... 254
Создание объекта ConfigMap ... 254
Создание объекта ConfigMap на основе литеральных значений ... 255
Хранение целых конфигурационных файлов в объекте ConfigMap ... 257
Создание объекта ConfigMap на основе env-файла ... 259
Чтение значений, содержащихся в объекте ConfigMap ... 260
Подключение данных ConfigMap в виде переменных окружения ... 261
Использование команды kubectl port-forward ... 263
Монтирование объекта ConfigMap в виде тома ... 264
Удаление объекта ConfigMap ... 266
Обновление объекта ConfigMap ... 267
Неизменяемые объекты ConfigMap ... 267

Управление конфиденциальной конфигурацией с помощью объекта Secret 268
Перечисление объектов Secret 270
Создание объекта Secret императивным способом 270
Создание объекта Secret декларативным способом 271
Создание объекта Secret на основе содержимого файла 273
Чтение значений, содержащихся в объекте Secret 273
Использование значения из объекта Secret в виде переменной окружения 275
Монтирование объекта Secret в виде тома 277
Удаление объекта Secret 278
Обновление объекта Secret 279
Резюме 279
Дополнительные источники информации 280

Глава 8. Публикация подов с помощью сервисов 281
Технические требования 281
Причины для публикации подов 282
Сетевое взаимодействие в Kubernetes 282
Сетевые плагины 283
Что такое сервисная сетка 284
Назначение IP-адреса поду 285
Динамическое назначение IP-адресов подам в Kubernetes 286
Почему не стоит прописывать IP-адреса подов в коде приложений 287
Как сервисы направляют трафик к подам 288
Циклический алгоритм балансировки нагрузки в Kubernetes 289
Обращение к сервису в Kubernetes 290
Процесс генерации DNS-имен сервисов 291
Обнаружение и направление трафика к подам Kubernetes 291
Использование пода k8sutils для отладки сервисов 294
Недостатки использования команды kubectl expose в Kubernetes 297
Принцип взаимодействия сервисов и подов 297
Типы сервисов 298
Сервис типа NodePort 299
Назначение сервисов типа NodePort 299
Создание двух подов на основе образа containous/whoami 299
YAML-манифест для создания сервиса NodePort 300
Проверка работоспособности сервиса NodePort 302
Оценка готовности сервиса к развертыванию в рабочей среде 302
Просмотр списка сервисов NodePort 303
Добавление подов в сервис NodePort 303
Просмотр подробных сведений о сервисе NodePort 304
Удаление сервисов 304
Сервис NodePort и команда kubectl port-forward 305

Сервис типа ClusterIP ... 306
Назначение сервисов типа ClusterIP ... 306
Выбор между сервисами NodePort и ClusterIP ... 306
Просмотр списка сервисов ClusterIP ... 308
Императивный способ создания сервисов ClusterIP ... 308
Просмотр подробных сведений о сервисе ClusterIP ... 309
Декларативный способ создания сервисов ClusterIP ... 310
Headless-сервисы ... 311
Сервис типа LoadBalancer ... 312
Облачные провайдеры, поддерживающие сервисы LoadBalancer ... 314
Стоит ли использовать сервис LoadBalancer ... 314
Сервис типа ExternalName ... 314
Обеспечение работоспособности сервисов с помощью проверок ... 316
Что такое проверка готовности и зачем она нужна ... 316
Реализация проверки готовности ... 317
Что такое проверка жизнеспособности и зачем она нужна ... 318
Реализация проверки жизнеспособности ... 320
Проверка успешности запуска ... 323
Совместное использование проверок готовности и жизнеспособности ... 324
Резюме ... 324
Дополнительные источники информации ... 325

Глава 9. Постоянное хранилище в Kubernetes ... 326
Технические требования ... 326
Назначение постоянного хранилища данных ... 326
Знакомство с томами ... 327
Объект PersistentVolume ... 332
Типы объектов PersistentVolume ... 333
Преимущества объектов PersistentVolume ... 335
Режимы доступа, поддерживаемые объектом PersistentVolume ... 336
Создание первого объекта PersistentVolume ... 338
Управление хранением данных с помощью объектов PersistentVolume ... 339
Создание объекта PersistentVolume на основе блочного тома ... 340
Может ли Kubernetes выделять или создавать ресурсы хранилища самостоятельно ... 340
Монтирование объекта PersistentVolume в под ... 341
Объект PersistentVolumeClaim ... 341
Разделение этапов создания и потребления ресурсов хранилища ... 341
Процесс использования объекта PersistentVolume ... 342
Создание пода с объектом PersistentVolumeClaim ... 344
Жизненный цикл объекта PersistentVolume в Kubernetes ... 348
Почему объекты PersistentVolume не связаны с пространствами имен ... 348
Освобождение объекта PersistentVolume ... 349

Обновление политики освобождения ... 350
Статусы объектов PersistentVolume и PersistentVolumeClaim ... 351
Статическое и динамическое выделение объектов PersistentVolume ... 352
Сравнение статического и динамического выделения ... 352
Динамическое выделение ресурсов ... 353
Знакомство с CSI ... 353
Знакомство с объектом StorageClass ... 355
Роль PersistentVolumeClaim в динамическом выделении ресурсов хранилища ... 358
Продвинутые темы, связанные с хранением данных ... 361
Эфемерные тома для временного хранения данных в Kubernetes ... 361
Клонирование томов CSI и снимки состояния томов ... 362
Увеличение размера тома через PersistentVolumeClaim ... 363
Резюме ... 364
Дополнительные источники информации ... 365

Глава 10. Запуск рабочих нагрузок производственного уровня в Kubernetes ... 366
Технические требования ... 366
Обеспечение высокой доступности и отказоустойчивости в Kubernetes ... 367
Высокая доступность ... 367
Отказоустойчивость ... 368
Обеспечение высокой доступности и отказоустойчивости приложений в Kubernetes ... 368
Объект ReplicationController ... 370
Объект ReplicaSet ... 370
Отличия ReplicaSet от ReplicationController ... 371
Создание объекта ReplicaSet ... 372
Тестирование поведения объекта ReplicaSet ... 375
Тестирование доступности и отказоустойчивости с помощью ReplicaSet ... 377
Масштабирование объекта ReplicaSet ... 378
Использование проверок жизнеспособности подов с объектом ReplicaSet ... 379
Удаление объекта ReplicaSet ... 382
Резюме ... 382
Дополнительные источники информации ... 383

Глава 11. Использование объекта Deployment для управления рабочими нагрузками без сохранения состояния ... 384
Технические требования ... 384
Знакомство с объектом Deployment ... 385
Создание объекта Deployment ... 388
Публикация подов Deployment с помощью объектов Service ... 392
Масштабирование объекта Deployment ... 402
Удаление объекта Deployment ... 404

Управление ревизиями и версиями приложений в Kubernetes с помощью объекта Deployment ... 404
Обновление объекта Deployment ... 406
Откат обновлений объекта Deployment ... 409
Канареечное развертывание ... 412
Лучшие практики работы с объектами Deployment ... 414
Управляйте объектами Deployment декларативным способом ... 414
Не используйте стратегию Recreate в рабочей среде ... 414
Не создавайте поды с метками, совпадающими с селектором существующего объекта Deployment ... 415
Внимательно настраивайте проверки контейнеров ... 415
Используйте осмысленные и семантические теги образов ... 416
Переходите на новые версии Kubernetes ... 417
Задействуйте механизмы управления ресурсами в объекте Deployment ... 417
Масштабирование и управление репликами ... 417
Вопросы безопасности ... 418
Резюме ... 418
Дополнительные источники информации ... 419

Глава 12. Объект StatefulSet: развертывание приложений с сохранением состояния ... 420
Технические требования ... 420
Знакомство с объектом StatefulSet ... 421
Управление состоянием в контейнерах ... 421
Управление состоянием в подах Kubernetes ... 422
Объект StatefulSet и его отличие от объекта Deployment ... 424
Ограничения объекта StatefulSet ... 426
Управление данными в объекте StatefulSet ... 426
Управление репликацией ... 427
Управление объектом StatefulSet ... 428
Создание объекта StatefulSet ... 428
Использование headless-сервиса и стабильных сетевых идентификаторов ... 435
Сохранение состояния ... 438
Масштабирование объекта StatefulSet ... 439
Удаление объекта StatefulSet ... 442
Выпуск новой версии приложения, развернутого в виде StatefulSet ... 443
Обновление объекта StatefulSet ... 445
Откат обновлений объекта StatefulSet ... 452
Лучшие практики работы с объектами StatefulSet ... 452
Управляйте объектами StatefulSet декларативным способом ... 452
Не задавайте нулевое значение параметра terminationGracePeriodSeconds для подов StatefulSet ... 453

Перед удалением StatefulSet уменьшайте количество реплик до нуля ... 453
Обеспечьте совместимость состояния при откате изменений ... 454
Не создавайте поды с метками, совпадающими с селектором существующего объекта StatefulSet ... 454
Используйте удаленное хранилище для объектов PV ... 454
Используйте проверки жизнеспособности и готовности ... 455
Следите за состоянием объектов StatefulSet ... 455
Резюме ... 455
Дополнительные источники информации ... 456

Глава 13. Объект DaemonSet: запуск одного пода на каждом узле ... 457
Технические требования ... 457
Знакомство с объектом DaemonSet ... 458
Планирование подов DaemonSet ... 460
Проверка объектов DaemonSet ... 460
Создание объектов DaemonSet и управление ими ... 462
Создание объекта DaemonSet ... 463
Приоритизация критических объектов DaemonSet в Kubernetes ... 466
Изменение объекта DaemonSet ... 467
Откат обновлений объекта DaemonSet ... 469
Удаление объекта DaemonSet ... 469
Типичные сценарии использования объектов DaemonSet ... 470
Лучшие практики реализации объектов DaemonSet ... 471
Альтернативы объекту DaemonSet ... 472
Резюме ... 473
Дополнительные источники информации ... 473

Глава 14. Работа с Helm-чартами и операторами ... 474
Технические требования ... 474
Знакомство с инструментом Helm ... 475
Развертывание приложений в Kubernetes с помощью Helm ... 477
Установка Helm в Linux ... 477
Установка Helm в Windows ... 478
Установка Helm в macOS ... 478
Установка из двоичных файлов ... 479
Развертывание чарта WordPress ... 479
Удаление релиза Helm ... 486
Структура Helm-чарта ... 487
Установка веб-интерфейса Kubernetes с использованием Helm-чартов ... 489
Безопасный доступ к Kubernetes Dashboard ... 490
Получение доступа к веб-интерфейсу Dashboard ... 491
Установка других популярных решений с помощью Helm-чартов ... 493
Вопросы безопасности, связанные с Helm-чартами ... 494

Знакомство с операторами Kubernetes ... 495
От ручного управления к программному ... 495
Helm-чарты и операторы Kubernetes ... 496
Роль операторов в развертывании приложений ... 497
Повторное использование автоматизированных решений ... 498
Поддержание состояния приложения с помощью операторов ... 498
Определения пользовательских ресурсов — основа операторов ... 498
Механизм распространения операторов ... 499
Создание собственного оператора ... 500
Компонент Operator Lifecycle Manager (OLM) ... 500
Мониторинг кластеров Kubernetes с помощью Prometheus и Grafana ... 501
Установка компонента Operator Lifecycle Manager ... 502
Установка операторов Prometheus и Grafana с использованием OLM ... 503
Настройка экземпляров Prometheus и Grafana с помощью операторов ... 504
Резюме ... 513
Дополнительные источники информации ... 514

Глава 15. Запуск кластера на платформе Google Cloud Platform (GCP) с помощью Google Kubernetes Engine (GKE) ... 515
Технические требования ... 515
Знакомство с Google Cloud Platform и Google Kubernetes Engine ... 515
Платформа Google Cloud Platform ... 516
Сервис Google Kubernetes Engine ... 516
Подготовка локальной среды ... 517
Создание проекта ... 518
Установка интерфейса командной строки GCP ... 519
Запуск кластера GKE ... 524
Развертывание рабочей нагрузки и взаимодействие с кластером ... 525
Настройка локального клиента ... 526
Изучение консоли Google Cloud ... 528
Дополнительные сведения об узлах кластера ... 533
Резюме ... 536
Дополнительные источники информации ... 536

Глава 16. Запуск кластера на платформе Amazon Web Services с помощью Amazon Elastic Kubernetes Service ... 537
Технические требования ... 537
Знакомство с Amazon Web Services и Amazon Elastic Kubernetes Service ... 537
Платформа Amazon Web Services ... 538
Сервис Amazon Elastic Kubernetes Service ... 539
Подготовка локальной среды ... 540
Регистрация учетной записи AWS ... 540
Установка интерфейса командной строки AWS ... 541

Настройка интерфейса командной строки AWS ... 542
Установка eksctl — официального CLI для Amazon EKS ... 545
Запуск кластера Amazon EKS ... 546
Развертывание рабочей нагрузки и взаимодействие с кластером ... 549
Развертывание рабочей нагрузки ... 550
Изучение консоли AWS ... 551
Удаление кластера Amazon EKS ... 557
Резюме ... 559
Дополнительные источники информации ... 560

Глава 17. Запуск кластера на платформе Microsoft Azure с помощью Azure Kubernetes Service ... 561
Технические требования ... 561
Знакомство с Microsoft Azure и Azure Kubernetes Service ... 562
Платформа Microsoft Azure ... 562
Сервис Azure Kubernetes Service ... 563
Подготовка локальной среды ... 564
Создание бесплатной учетной записи Microsoft Azure ... 564
Установка интерфейса командной строки Azure ... 565
Запуск кластера AKS ... 569
Развертывание рабочей нагрузки и взаимодействие с кластером ... 572
Развертывание рабочей нагрузки ... 572
Изучение портала Azure ... 573
Удаление кластера AKS ... 582
Сравнение трех публичных облачных платформ ... 583
Резюме ... 585
Дополнительные источники информации ... 586

Глава 18. Безопасность в системе Kubernetes ... 587
Технические требования ... 588
Аутентификация и авторизация: управление пользовательским доступом ... 589
Аутентификация и управление пользователями ... 589
Авторизация и знакомство с механизмом RBAC ... 611
Контроллеры допуска: политики и проверки безопасности ... 619
Двухэтапный процесс работы контроллеров допуска ... 620
Включение и отключение контроллеров допуска ... 621
Распространенные контроллеры допуска ... 622
Преимущества контроллеров допуска ... 623
Защита подов и контейнеров ... 623
Защита подов и контейнеров в Kubernetes с помощью контекста безопасности ... 624
Защита подов с помощью объекта NetworkPolicy ... 628

Защита взаимодействий компонентов Kubernetes с помощью TLS-сертификатов ... 634
Защита контейнеров с помощью gVisor и Kata Containers ... 636
Управление секретами и учетными данными реестров ... 638
Создание секрета для реестра Docker с помощью kubectl ... 639
Резюме ... 639
Дополнительные источники информации ... 640

Глава 19. Продвинутые методы планирования подов ... 641
Технические требования ... 641
Обзор компонента kube-scheduler ... 642
Правила предпочтения узлов ... 645
Использование поля nodeName в спецификации подов ... 645
Использование поля nodeSelector в спецификации подов ... 648
Использование поля nodeAffinity в спецификации подов ... 652
Выбор узлов на основе ограничений и допусков ... 656
Статические поды в Kubernetes ... 663
Расширенные конфигурации планировщика Kubernetes ... 664
Конфигурация планировщика ... 664
Изоляция узлов и ограничение их возможностей ... 665
Повышение производительности планировщика Kubernetes ... 665
Резюме ... 666
Дополнительные источники информации ... 667

Глава 20. Автоматическое масштабирование подов и узлов в Kubernetes ... 668
Технические требования ... 669
Запросы и лимиты ресурсов, доступных подам ... 669
Вертикальное автомасштабирование подов с помощью объекта VerticalPodAutoscaler ... 674
Включение механизма InPlacePodVerticalScaling ... 675
Включение VPA в кластере GKE ... 676
Включение VPA в других Kubernetes-кластерах ... 677
Использование VPA ... 677
Горизонтальное автомасштабирование подов с помощью объекта HorizontalPodAutoscaler ... 683
Развертывание приложения для демонстрации принципа работы HPA ... 685
Реализация HPA ... 688
Автомасштабирование узлов Kubernetes с помощью кластерного автоскейлера ... 691
Ограничения кластерного автоскейлера ... 692
Включение кластерного автоскейлера в GKE ... 693
Включение кластерного автоскейлера в Amazon EKS ... 694
Включение кластерного автоскейлера в AKS ... 694
Использование кластерного автоскейлера ... 695

Альтернативные автоскейлеры для Kubernetes ... 701
Автоскейлер KEDA ... 701
Автоскейлер Karpenter ... 702
Резюме ... 703
Дополнительные источники информации ... 704

Глава 21. Продвинутые возможности Kubernetes ... 705
Технические требования ... 706
Продвинутая маршрутизация трафика с помощью объекта Ingress ... 706
Типы сервисов Kubernetes ... 706
Обзор объекта Ingress ... 707
Использование Nginx в качестве Ingress-контроллера ... 710
Развертывание Ingress-контроллера Nginx в кластере minikube ... 712
Развертывание ресурсов Ingress в Kubernetes ... 712
Использование ресурса IngressClass и нескольких Ingress-контроллеров ... 716
Контроллер Azure AGIC для кластера AKS ... 717
Причины для использования AGIC в AKS ... 717
Gateway API ... 720
Ресурсы Endpoints и EndpointSlice ... 723
Современные технологии в контексте Kubernetes ... 724
Бессерверные вычисления с Knative и OpenFaaS ... 724
Kubeflow — машинное обучение в Kubernetes ... 725
KubeVirt — виртуальные машины в Kubernetes ... 726
Обслуживание кластеров Kubernetes: задачи второго дня эксплуатации ... 727
Резервное копирование и восстановление Kubernetes-кластера ... 727
Обновление Kubernetes-кластера ... 730
Управление несколькими кластерами ... 733
Лучшие практики защиты кластера Kubernetes ... 734
Контроль доступа к API Kubernetes ... 735
Устранение неполадок в Kubernetes ... 739
Получение сведений о ресурсах ... 739
Использование логов и событий для устранения неполадок в Kubernetes ... 740
Встроенный помощник kubectl explain ... 742
Интерактивное устранение неполадок с помощью команды kubectl exec ... 742
Эфемерные контейнеры в Kubernetes ... 744
Распространенные проблемы и методы их решения ... 745
Резюме ... 747
Дополнительные источники информации ... 748

Об авторах

Гиниш Мадаппарамбат — специалист с более чем 15-летним опытом в области управления ИТ-сервисами и консультирования. В круг его профессиональных интересов входят Linux, автоматизация и контейнеризация. Он активно участвовал в проектировании, развертывании и сопровождении решений на базе Ansible и Ansible Automation Platform в различных инфраструктурах — от частных и публичных облаков до физических и сетевых сред. За свою карьеру успел поработать в разных странах и на разных должностях, в том числе системным инженером, специалистом по автоматизации и проектировщиком инфраструктуры. Написал также книгу *Ansible for Real Life Automation*.

> *Я посвящаю эту книгу своей супруге Дипти за неизменную поддержку и вдохновение. Моему сыну Абхаю и дочери Ану — за терпение и понимание, позволившие мне уделить время работе. Моим родителям и друзьям — за помощь и веру в меня.*

Расс МакКендрик — DevOps-инженер и системный администратор, специализирующийся на автоматизации и контейнерных технологиях. Работает в ИТ и смежных областях уже около 30 лет. За свою карьеру прошел путь от специалиста технической поддержки начального уровня до старшего специалиста, работая как с клиентами, так и с внутренними командами в компаниях самого разного масштаба.

В настоящее время Расс руководит практикой SRE и DevOps в компании Node4, работая преимущественно с Linux и системами с открытым исходным кодом, которые он использует как на выделенном оборудовании, так и на виртуальных машинах, развернутых в публичных и частных облаках. А еще он безудержно коллекционирует музыкальные пластинки!

О научных редакторах

Ром Адамс (урожденный Ромуальд Вандепул) — консультант по вопросам открытого программного обеспечения и стратегического управления с 20-летним опытом работы в ИТ-индустрии. Специализируется на облачных решениях и помогает компаниям модернизировать инфраструктуру и проводить цифровую трансформацию с использованием open-source-технологий. Ром консультирует различные компании по вопросам работы с открытым кодом и внутренними репозиториями. Ранее он занимал должность ведущего архитектора в компании Ondat — разработчике облачных хранилищ данных, впоследствии приобретенной компанией Akamai. В этой роли он занимался проектированием продуктов и гибридных облачных решений. До этого работал в Tyco, NetApp и Red Hat, где стал признанным экспертом в области гибридных облачных технологий.

Ром неоднократно выступал модератором и докладчиком на профильных мероприятиях, делясь своими взглядами на корпоративную культуру, организационные процессы и внедрение технологий, а также заражая своей приверженностью идее открытых инноваций.

Я благодарю бабушку — за ее доброту, дедушку — за его мудрость и мою спутницу жизни и лучшую подругу Мерседес Адамс — за любовь, терпение и постоянную поддержку.

Шейн Боулден — архитектор решений в компании Red Hat, который помогает организациям развертывать и защищать платформы и технологии с открытым исходным кодом, а также управлять ими.

Он участвует в разработке нескольких открытых проектов в области информационной безопасности, включая Compliance as Code, разрабатывает SCAP-профили и Ansible-плейбуки в соответствии с австралийскими стандартами защиты данных. Шейн также помог проекту Keycloak пройти сертификацию на соответствие требованиям австралийской инициативы Consumer Data Right (CDR) при участии фонда OpenID и внес ряд новых политик в проект StackRox, ориентированный на обеспечение безопасности Kubernetes-сред.

Шейн активно интересуется искусственным интеллектом и машинным обучением. Он публикует научные работы о генетических алгоритмах и их практическом применении, а также проводит мастер-классы на конференциях, помогая новичкам осваивать открытые фреймворки для разработки генеративного ИИ.

Кроме того, Шейн регулярно выкладывает статьи об эксплуатации платформ и обеспечении безопасности в своем блоге stb.id.au.

Предисловие

Контейнеры позволяют упаковать приложение вместе со всеми зависимостями, библиотеками и фреймворками в единый исполняемый модуль. Это упрощает развертывание, но порождает новую задачу, связанную с управлением этими модулями, то есть с их оркестрацией. Контейнеры необходимо распределять по узлам кластера, автоматически перезапускать их при сбоях и заменять отказавшие узлы. Именно для удовлетворения этой потребности компания Google разработала Kubernetes.

Я впервые столкнулся с системой Kubernetes вскоре после ее появления в конце 2015 года. В то время я занимался реализацией программ по сбору обратной связи от разработчиков, создававших решения на платформе Microsoft Azure. Нам требовалось, чтобы наши сервисы работали с системами оркестрации контейнеров, в том числе с Kubernetes. К моменту запуска *Azure Kubernetes Service (AKS)* в 2018 году я уже публиковал статьи, руководства и техническую документацию по архитектуре облачных решений в Azure Architecture Center. Вместе с ведущими специалистами компании Microsoft мы разрабатывали рекомендации по проектированию архитектур AKS и создавали эталонные архитектуры. В 2022 году я участвовал в создании серии материалов под названием *AKS for Amazon EKS professionals*, а позднее — в публикации контента о *Google Kubernetes Engine (GKE)*, организованной центром Google Cloud Architecture Center.

> *Мощь Kubernetes обусловлена многолетним опытом, накопленным Google при работе с системой Borg. Масштаб Google не имеет себе равных: Borg запускает более двух миллиардов контейнеров в неделю — в среднем 3300 в секунду. Система Kubernetes родилась в огне, прошла боевое крещение и готова к огромным нагрузкам.*
>
> *Свапнил Бхартиа, OpenSource.com*

Свапнил Бхартиа точно определил причину успеха Kubernetes: он появился из реальной потребности и был доведен до исключительной эффективности. Но одни слова не помогут вашему приложению. Давайте узнаем, что персонажи комикса Скотта Адамса говорят о проектировании облачно-ориентированных приложений, контейнеризации и Kubernetes.

Босс: Я хочу понять, почему перенос нашего приложения в облако не решил все наши проблемы автоматически.

Дилберт: Вы не позволили мне перестроить архитектуру под облако.

Босс: Просто поместите все в контейнеры.

Дилберт: Нельзя решить проблему, просто произнося технические термины.

Босс: Kubernetes.

И здесь на свет появляется книга «Kubernetes. Полное руководство». Как сказал Скотт Адамс, недостаточно просто произносить технические термины — нужно разбираться в соответствующих концепциях и уметь применять эти знания на практике. Вы должны понимать, как запускать Kubernetes, как проектировать и развертывать крупные кластеры, как размещать, отлаживать и восстанавливать работу контейнеризованных приложений. Авторы этой книги Гиниш Мадаппарамбат и Расс МакКендрик проведут вас через весь путь — от установки первого кластера Kubernetes до настройки подов и развертывания приложений с сохранением и без сохранения состояния. Вы научитесь запускать кластеры Kubernetes на платформах Google Cloud (GKE), Amazon Web Services (EKS) и Microsoft Azure (AKS). Кроме того, вы освоите работу с Helm-чартами, методы обеспечения безопасности, продвинутые техники использования подов, способы управления трафиком и многое другое.

Изучайте каждую главу этой книги максимально внимательно, впитывая опыт Гиниша и Расса как губка. Чем глубже вы погрузитесь в материал, тем лучше его запомните и тем более эффективно сможете применить на практике. Всегда держите эту книгу под рукой. Она станет вашим незаменимым помощником в деле управления и оркестрации контейнеров.

Эд Прайс,
технический писатель, менеджер
контента в сфере облачных архитектур,
соавтор восьми книг, включая The Azure
Cloud Native Architecture Mapbook

Введение

Контейнеры совершили настоящий прорыв в мире виртуализации, обеспечив беспрецедентную гибкость, особенно актуальную сегодня, когда такие модные слова, как *облако*, *Agile* и *DevOps*, на слуху буквально у всех.

В настоящее время почти никто не ставит под сомнение целесообразность использования контейнеров — они распространились повсюду, особенно после успеха Docker и появления системы Kubernetes, быстро ставшей ведущей платформой для оркестрации контейнеров.

Однако, несмотря на всю гибкость контейнеров, организации не спешили с их развертыванием в рабочей среде. Годами компании использовали контейнеры для создания прототипов, локальной разработки и решения аналогичных задач, но идея их использования для запуска полноценных рабочих нагрузок многим казалась немыслимой.

Переломным моментом стало появление систем оркестрации контейнеров, и прежде всего Kubernetes. Изначально созданный в компании Google, сегодня Kubernetes является ведущим оркестратором, предоставляющим весь необходимый функционал для масштабного развертывания контейнеров в рабочей среде. Эта платформа невероятно популярна, но в то же время весьма сложна. Kubernetes представляет собой универсальный инструмент, освоение которого требует времени и усилий.

Система оркестрации Kubernetes опирается на собственную концептуальную модель, отличную от модели контейнерных движков. Однако в сочетании с контейнерными движками Kubernetes формирует по-настоящему мощную

платформу, позволяющую развертывать облачно-ориентированные приложения в рабочей среде. Мы, как инженеры, ежедневно работающие с Kubernetes, разделяем распространенное мнение о том, что эту технологию непременно стоит освоить. Именно поэтому мы решили поделиться своими знаниями, чтобы сделать Kubernetes более доступным, охватив большую часть функционала этого оркестратора.

Эта книга целиком посвящена Kubernetes и является результатом нашей работы. Она предоставляет широкий обзор платформы и охватывает множество аспектов ее использования — от создания простых подов до развертывания самого оркестратора в публичном облаке. Мы не стремились сделать из нее руководство для начинающих.

Мы надеемся, что в этой книге вы найдете все, что хотите знать о Kubernetes!

Кому адресована эта книга

Издание адресовано людям, планирующим использовать Kubernetes совместно с контейнерными движками. Хотя Kubernetes поддерживает различные контейнерные движки через *интерфейс среды выполнения контейнеров* (Container Runtime Interface, CRI) и не привязан к какому-либо конкретному решению, на практике одной из самых распространенных конфигураций является связка Kubernetes и `containerd`.

Эта книга представляет собой сугубо техническое издание, где основное внимание уделено Kubernetes и контейнерным средам. Она ориентирована на инженеров, будь то разработчики или системные администраторы, и не предназначена для менеджеров проектов. Это настоящая библия Kubernetes для тех, кто намерен работать с данной системой на повседневной основе. Если вас не пугает необходимость вводить команды в терминале, то эта книга для вас.

Полное отсутствие опыта работы с Kubernetes или средний уровень подготовки не станут помехой для освоения материала. Хотя в главах затрагиваются базовые понятия контейнеризации, читателю желательно иметь хотя бы минимальное техническое представление о контейнерах. Издание также может послужить практическим руководством для тех, кто занимается переносом существующего приложения в Kubernetes.

Эта книга содержит практические руководства по развертыванию Kubernetes в публичных облаках, например Amazon EKS или Google GKE. Она будет особенно полезна пользователям облачных платформ, желающим включить Kubernetes в свой технологический стек.

Содержание книги

Глава 1, «Основы Kubernetes», содержит общий обзор системы. В ней вы узнаете, что такое Kubernetes, с какой целью и кем он был создан, кто поддерживает проект сегодня и при каких условиях этот инструмент стоит включать в свой технологический стек.

Глава 2, «Архитектура Kubernetes: от образов контейнеров к работающим подам», посвящена устройству Kubernetes как распределенной программной системы. С технической точки зрения Kubernetes не является единым монолитным приложением, а состоит из набора микросервисов, взаимодействующих друг с другом. В этой главе мы разберем архитектуру Kubernetes и посмотрим, как эта система передает ваши инструкции защищенным контейнерам.

В главе 3, «Установка первого кластера Kubernetes», вы узнаете о том, что распределенная природа системы Kubernetes затрудняет ее установку. Чтобы упростить процесс освоения, можно развернуть кластер, используя один из ее дистрибутивов. В этой главе мы познакомимся с инструментами `kind` и `minikube`, позволяющими запустить рабочий кластер Kubernetes на вашем компьютере.

Глава 4, «Запуск контейнеров в Kubernetes», вводит понятие пода.

Глава 5, «Использование многоконтейнерных подов и паттерны проектирования», знакомит читателей с многоконтейнерными подами и такими паттернами проектирования, как Ambassador (Посредник), Adapter (Адаптер) и Sidecar (Прицеп), которые можно использовать при запуске нескольких контейнеров в составе одного пода.

Глава 6, «Пространства имен, квоты и лимиты для реализации мультитенантности в Kubernetes», знакомит читателя с пространством имен — ключевым аспектом управления кластером, с которым неизбежно придется столкнуться в процессе работы с Kubernetes. Несмотря на простоту, это понятие является одним из базовых, и для успешной работы с системой необходимо овладеть им в совершенстве. В этой главе мы также поговорим о реализации мультитенантности в Kubernetes с помощью пространств имен, квот и лимитов.

В главе 7, «Настройка подов с помощью объектов ConfigMap и Secret», вы узнаете о том, как отделить приложения от их конфигураций. В системе Kubernetes приложения и конфигурации имеют собственный жизненный цикл благодаря объектам ConfigMap и Secret. Данная глава будет посвящена этим двум объектам и способам подключения их данных в виде переменных окружения или томов, смонтированных в под.

Глава 8, «Публикация подов с помощью сервисов», знакомит читателей с понятием сервиса в Kubernetes. Каждому поду в системе динамически назначается собственный IP-адрес. Сервисы позволяют предоставить доступ к поду другим

подам внутри кластера, а также внешним клиентам через единое статическое DNS-имя. В этой главе мы рассмотрим три основных типа сервисов: ClusterIP, NodePort и LoadBalancer, каждый из которых предназначен для определенного сценария публикации подов.

В главе 9, «Постоянное хранилище в Kubernetes», вы узнаете о том, что по умолчанию поды не являются постоянными сущностями: так как они управляют только контейнерами, их уничтожение приводит к потере данных. Решением этой проблемы является использование постоянного хранилища, обеспечиваемого механизмами PersistentVolume и PersistentVolumeClaim. Глава посвящена этим двум ресурсам, а также объекту StorageClass. В ней вы узнаете, что система Kubernetes отличается большой гибкостью в плане работы с хранилищами и что поды можно интегрировать со множеством различных технологий хранения данных.

В главе 10, «Запуск рабочих нагрузок производственного уровня в Kubernetes», подробно рассматриваются вопросы обеспечения высокой доступности и отказоустойчивости в системе Kubernetes с помощью объектов ReplicationController и ReplicaSet.

Глава 11, «Использование объекта Deployment для управления рабочими нагрузками без сохранения состояния», продолжает тему предыдущей главы и показывает, как управлять несколькими версиями ReplicaSet с помощью объекта Deployment. Этот объект является базовым строительным блоком для приложений без сохранения состояния, работающих в Kubernetes.

Глава 12, «Объект StatefulSet: развертывание приложений с сохранением состояния», знакомит со следующим важным объектом Kubernetes — StatefulSet, который служит основой для запуска приложений с сохранением состояния. В этой главе мы обсудим ключевые различия между запуском приложений без сохранения и с сохранением состояния в Kubernetes.

Глава 13, «Объект DaemonSet: запуск одного пода на каждом узле», посвящена объекту DaemonSet, предназначенному для запуска эксплуатационных или вспомогательных рабочих нагрузок в кластерах Kubernetes. Если вам нужно, чтобы на каждом узле работал ровно один под, используйте DaemonSet.

Глава 14, «Работа с Helm-чартами и операторами», посвящена Helm-чартам — специализированному инструменту для упаковки и распространения приложений в Kubernetes. Освоив материал этой главы, вы сможете быстро развернуть среду разработки Kubernetes и даже подготовить собственное приложение к распространению в виде отдельного Helm-чарта. В этой же главе вы познакомитесь с операторами Kubernetes и узнаете, как они помогают развертывать стеки приложений.

В главе 15, «Запуск кластера на платформе Google Cloud Platform (GCP) с помощью Google Kubernetes Engine (GKE)», описывается процесс переноса

рабочих нагрузок Kubernetes на облачную платформу Google с использованием как встроенного клиента командной строки, так и консоли Google Cloud.

Глава 16, «Запуск кластера на платформе Amazon Web Services с помощью Amazon Elastic Kubernetes Service», описывает процесс переноса рабочей нагрузки, запущенной в предыдущей главе, на платформу Kubernetes от компании Amazon.

Глава 17, «Запуск кластера на платформе Microsoft Azure с помощью Azure Kubernetes Service», посвящена запуску кластера Kubernetes на платформе Microsoft Azure.

Глава 18, «Безопасность в системе Kubernetes», охватывает ключевые механизмы авторизации в Kubernetes, в том числе управление доступом на основе ролей (RBAC), а также другие схемы авторизации и управления пользователями. Кроме того, в этой главе рассматриваются контроллеры допуска, взаимодействие по протоколу TLS с использованием сертификатов и применение контекстов безопасности.

В главе 19, «Продвинутые методы планирования подов», мы подробно обсудим правила предпочтения узлов, ограничения и допуски, а также расширенные политики планирования в целом.

Глава 20, «Автоматическое масштабирование подов и узлов в Kubernetes», знакомит читателя с принципами автомасштабирования в системе Kubernetes и способами применения таких компонентов, как вертикальный и горизонтальный автоскейлер и инструмент автомасштабирования кластера.

Глава 21, «Продвинутые возможности Kubernetes», посвящена объектам Ingress и Ingress-контроллерам. В этой главе вы научитесь использовать Nginx в качестве реализации Ingress-контроллера и применять Azure Application Gateway в качестве нативного Ingress-контроллера в средах Azure. Кроме того, в этой главе рассматриваются продвинутые темы Kubernetes, включая управление трафиком, стратегии работы с несколькими кластерами, задачи второго дня эксплуатации кластера, передовые практики и методы устранения неполадок.

Как извлечь максимальную пользу из этой книги

Хотя в главах книги затрагиваются основы работы с контейнерами, по большей части она посвящена Kubernetes. Данная система поддерживает несколько контейнерных движков, однако в книге в основном используется среда выполнения `containerd`. Вам не обязательно быть экспертом, но базовое понимание процесса запуска и управления приложениями в контейнерах поможет при изучении материала.

Хотя Kubernetes может работать с контейнерами Windows, большинство тем, раскрываемых в этой книге, будет связано с Linux. Глубокие знания Linux могут пригодиться, но не являются обязательными — достаточно уметь пользоваться терминалом и владеть основами написания Bash-сценариев.

Нелишним также будет общее представление об архитектуре программного обеспечения, например о REST API.

Программное/аппаратное обеспечение, используемое в этой книге	Требования к операционной системе
Kubernetes ⩾ 1.31	Windows, macOS, Linux
kubectl ⩾ 1.31	Windows, macOS, Linux

Мы настоятельно рекомендуем пока не устанавливать Kubernetes или `kubectl` на свой компьютер. Kubernetes представляет собой не единый исполняемый файл, а распределенную систему, состоящую из нескольких компонентов, что усложняет развертывание полноценного кластера с нуля. Вместо этого мы советуем обратиться к главе 3, посвященной настройке Kubernetes.

Даже если вы используете цифровую версию книги, мы советуем вам набирать код самостоятельно или использовать код из репозитория GitHub (ссылку на него вы найдете в следующем разделе). Это поможет вам избежать возможных ошибок, связанных с копированием и вставкой кода.

Обратите внимание на то, что `kubectl`, `helm` и другие подобные инструменты будут использоваться в этой книге чаще всего, однако вокруг Kubernetes существует обширная экосистема, поэтому в дальнейшем нам может потребоваться установить и другое ПО, не упомянутое в этом разделе. Помимо всего прочего, книга посвящена использованию Kubernetes в облаке, поэтому мы также рассмотрим способы развертывания кластеров Kubernetes на таких публичных облачных платформах, как Amazon Web Services и Google Cloud Platform. Это может потребовать установки дополнительного программного обеспечения, связанного не только с Kubernetes, но и с соответствующими платформами.

Загрузка файлов с кодом из примеров

Код из примеров, приведенных в книге, опубликован на GitHub по адресу https://github.com/PacktPublishing/The-Kubernetes-Bible-Second-Edition. В случае обновления кода этих примеров он будет обновлен и в репозитории GitHub.

Вы также можете найти другие коллекции исходного кода из нашего обширного каталога книг и видеокурсов по адресу https://github.com/PacktPublishing/.

Загрузка цветных изображений

PDF-файл с цветными версиями снимков экрана и схем, приведенных в оригинальной книге, можно загрузить по ссылке https://static.packt-cdn.com/downloads/9781835464717_ColorImages.pdf.

Условные обозначения

В книге используются следующие условные обозначения.

Курсивом выделены новые термины и важные понятия.

Фрагменты кода в тексте, названия таблиц базы данных, каталогов, файлов, а также расширения файлов, пути к ним, URL-адреса из примеров, данные, введенные пользователем, выделяются моноширинным шрифтом. Например: «Теперь нам нужно создать файл `kubeconfig` для локального интерфейса командной строки `kubectl`».

Блок кода оформляется следующим образом:

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx-Pod
```

Если мы хотим обратить ваше внимание на определенную часть блока кода, соответствующие строки или элементы выделяются серым фоном:

```
apiVersion: v1
kind: ReplicationController
metadata:
  name: nginx-replicationcontroller-example
```

Любой ввод или вывод командной строки оформляется следующим образом:

```
$ kubectl get nodes
```

Шрифтом без засечек выделяются названия клавиш, URL и слова, которые вы видите на экране, в частности названия элементов меню или диалоговых окон, упомянутые в тексте. Например: «На этом экране должна отображаться кнопка Enable Billing».

Так оформляется примечание.

Так оформляется совет или предложение.

От издательства

Мы выражаем огромную благодарность компании Orion soft за помощь в работе над русскоязычным изданием книги и их вклад в повышение качества переводной литературы.

Ваши замечания, предложения, вопросы отправляйте по адресу comp@sprintbook.kz (издательство SprintBook, компьютерная редакция).

Мы будем рады узнать ваше мнение!

О научном редакторе русскоязычного издания

Сергей Метелёв — ведущий системный инженер компании Orion soft. Имеет обширный опыт внедрения DevOps-практик в различных командах разработки, от автоматизации CI-процессов и сборки кода в контейнеры до мониторинга межсервисного взаимодействия. Участвовал в запуске и развитии региональной медицинской информационной системы под Kubernetes, а также в модернизации и расширении распределенной инфраструктуры в международной финтех-команде.

1 Основы Kubernetes

Приветствуем вас на страницах книги «Kubernetes. Полное руководство»! Мы рады сопровождать вас в процессе освоения этой системы. Если вы занимаетесь разработкой программного обеспечения, то, вероятно, уже слышали о Kubernetes. Это неудивительно, так как за последние годы популярность этой системы значительно возросла.

Созданный компанией Google, Kubernetes представляет собой самый популярный и широко используемый инструмент для оркестрации контейнеров. Если вам нужна надежная система управления контейнеризованными приложениями в рабочей среде, будь то локальная инфраструктура или публичное облако, Kubernetes станет идеальным решением. Ключевым моментом здесь является масштабирование. Развертывание и управление контейнерами в больших масштабах — это весьма сложная задача, поскольку контейнерные движки (такие как Docker) не предусматривают встроенных механизмов поддержания доступности и масштабируемости контейнеров[1].

Система Kubernetes изначально разрабатывалась компанией Google для развертывания огромного количества контейнеров в ее распределенной инфраструктуре. Включив Kubernetes в свой технологический стек, вы получите открытую платформу, созданную одним из интернет-гигантов, предъявляющим строжайшие требования к стабильности.

Хотя система Kubernetes поддерживает множество сред выполнения контейнеров, в этой книге основное внимание уделяется ее работе с Docker и Podman.

Возможно, вы уже активно используете Docker в своей работе, но мир оркестрации контейнеров вам пока незнаком. Вы даже можете не видеть смысла в применении таких инструментов, если Docker и так позволяет вам справляться

---

[1] На самом деле проверка доступности и масштабируемость реализованы в Docker (режим swarm) и его альтернативе — Podman (который также позволяет запускать поды — несколько контейнеров в общем пространстве имен), однако их гибкость и функционал, несомненно, сильно уступают возможностям Kubernetes. — *Здесь и далее примеч. науч. ред.*

со своими задачами. Именно поэтому в первой главе мы не будем углубляться в детали Kubernetes. Вместо этого разберем, что собой представляет эта система и как она помогает управлять контейнерами в рабочей среде. Вам будет проще освоить эту новую технологию, если вы будете понимать, зачем она была создана.

Примеры кода для этой главы можно загрузить из официального репозитория GitHub по адресу https://github.com/PacktPublishing/The-Kubernetes-Bible-Second-Edition/tree/main/Chapter01.

Монолиты и микросервисы

Давайте на мгновение отвлечемся от Kubernetes и Docker и немного поговорим о том, как развивались Интернет и сфера разработки программного обеспечения на протяжении последних 20 лет. Это поможет лучше понять, какое место занимает система Kubernetes и какие задачи она решает.

Стремительное развитие Интернета, начавшееся в конце 1990-х годов

В конце 1990-х годов популярность Интернета начала быстро расти. Если в 1990-е и даже начале 2000-х годов Интернетом пользовались всего несколько сотен тысяч человек во всем мире, то сегодня почти 2 миллиарда людей регулярно выходят в Интернет, чтобы отправлять электронную почту, просматривать веб-страницы, играть в видеоигры и решать другие задачи. Их потребности удовлетворяются десятками приложений, развернутых на множестве устройств.

При этом количество подключенных устройств также существенно выросло: теперь у одного человека может быть несколько разных гаджетов с доступом в Сеть, например ноутбук, настольный компьютер, смартфон, телевизор, планшет и многое другое.

Сегодня Интернет можно использовать для работы, развлечения, совершения покупок, чтения и не только. Он проник почти во все сферы общественной жизни и за последние 20 лет привел к глубочайшему изменению парадигмы. Все это сделало разработку программного обеспечения как никогда важной.

Потребность в более частых релизах программного обеспечения

Чтобы справиться с постоянно растущим числом пользователей, которые требуют все больше в плане функциональности, индустрия разработки ПО была вынуждена измениться, увеличив скорость и частоту выпуска новых версий.

В 1990-е годы можно было создать приложение, развернуть его в рабочей среде и обновлять всего один-два раза в год. Сегодня компании вынуждены выпускать обновления до нескольких раз в день, например, чтобы добавить новую функцию, обеспечить интеграцию с платформой социальной сети, реализовать поддержку разрешения экрана нового смартфона или выпустить патч для устранения уязвимости, обнаруженной накануне. Поскольку все стало гораздо сложнее, скорость разработки должна быть существенно выше, чем раньше.

Потребность в постоянном обновлении программного обеспечения стала нормой, и теперь выживание компаний во многом зависит от частоты релизов. Но как ускорить жизненный цикл разработки, чтобы чаще доставлять новые версии ПО нашим пользователям?

ИТ-отделы компаний были вынуждены эволюционировать как в организационном, так и в техническом плане. Изменение подхода к управлению проектами и командами позволило переключиться на гибкие методологии разработки, а широкое распространение таких технологий, как облачные платформы, контейнеры и виртуализация, помогло согласовать техническую гибкость с организационной. Все это было направлено на повышение частоты выпусков программного обеспечения. Давайте подробнее рассмотрим, как происходила эта эволюция.

Переход к гибким методологиям разработки

В настоящее время стандартным способом организации работы ИТ-команд считаются такие гибкие (agile) методологии, как Scrum, Kanban и DevOps.

Типичный ИТ-отдел, не использующий гибкие подходы, часто состоит из трех отдельных команд, каждая из которых отвечает за один этап жизненного цикла разработки и выпуска продукта.

Хотя сейчас мы говорим о гибких методологиях и истории Интернета, эта книга на самом деле посвящена системе Kubernetes! Просто перед тем, как перейти к ее обсуждению, нам следует объяснить кое-какие проблемы, с которыми сталкивалась индустрия до ее появления.

До внедрения гибких методологий специалисты, отвечавшие за разработку и эксплуатацию ПО, часто существовали в изоляции друг от друга, что приводило к неэффективности и сбоям в коммуникации. Гибкие методологии помогли устранить эти разрывы и наладить сотрудничество. Ниже перечислены три команды, которые раньше работали независимо друг от друга (рис. 1.1).

- *Бизнес-команда* представляет интересы клиента. Ее задача — объяснить, какие функции нужно включить в приложение, чтобы удовлетворить запросы пользователей. Эта команда переводит бизнес-цели в понятные разработчикам инструкции.
- *Команда разработки* состоит из инженеров, которые воплощают приложение в жизнь. Они превращают запросы бизнес-команды в код, создавая функциональность, с которой предстоит взаимодействовать пользователям. Здесь особенно важна четкая коммуникация: расплывчатые инструкции чреваты недопониманием, задержками и тратой лишних ресурсов.
- *Операционная команда (команда эксплуатации)* состоит из хранителей серверов, чья главная задача заключается в поддержании стабильной работы приложения. Добавление новых функций может нарушить его работу, так как требует обновлений, которые несут с собой определенные риски. В прошлом эти специалисты могли даже не знать о готовящихся изменениях, потому что их не привлекали к планированию.

Рис. 1.1. Изолированные команды в типичном ИТ-отделе

Роли в такой структуре четко определены, сотрудники из разных команд взаимодействуют не слишком активно, а при возникновении проблем теряют время, пытаясь получить нужную информацию от нужного человека.

Такая изоляция:

- значительно увеличивает время разработки;
- повышает риск того, что новая версия при развертывании в рабочей среде может не заработать вовсе.

Гибкие методологии и DevOps были призваны решить именно эти проблемы. Их внедрение побудило людей к совместной работе и формированию междисциплинарных команд.

DevOps — это культура сотрудничества и набор практик, направленных на улучшение взаимодействия между командой разработки (development, Dev) и операционной командой (operations, Ops). DevOps предполагает совместную работу и автоматизацию на всех этапах жизненного цикла программного обеспечения — от разработки и тестирования до развертывания и сопровождения.

В состав agile-команды входит владелец продукта, который описывает конкретные функции в виде пользовательских историй, понятных разработчикам, сотрудничающих с ним. Разработчики должны иметь представление о рабочей среде и возможность развертывать в ней приложения, предпочтительно с использованием подхода *непрерывной интеграции и непрерывного развертывания* (continuous integration and continuous deployment, CI/CD). Тестировщики также должны входить в состав таких команд, чтобы иметь возможность писать тесты.

Благодаря совместной работе над проектом, проиллюстрированной на рис. 1.2, команды получают более полное и точное представление о нем.

Рис. 1.2. Совместная работа над проектом

Таким образом, внедрение гибких методологий и DevOps привело к появлению междисциплинарных команд, способных формализовать потребность пользователей, реализовать нужную функцию, протестировать ее, выпустить и сопровождать в рабочей среде. В табл. 1.1 сравнивается традиционный подход к разработке ПО с гибкими методологиями и DevOps.

Таблица 1.1. Сравнение традиционного подхода к разработке ПО с гибкими методологиями DevOps

Характеристика	Традиционный подход к разработке ПО	Гибкие методологии и DevOps
Структура команды	Изолированные отделы (разработка, эксплуатация)	Кросс-функциональные, междисциплинарные команды
Стиль работы	Независимые рабочие процессы, ограниченное взаимодействие	Совместная работа, итеративный подход к разработке
Ответственность	Разработчики передают готовое решение операционной команде для развертывания и сопровождения	Принцип «Кто создает, тот и запускает». Вся команда отвечает за весь жизненный цикл продукта
Фокус	Функциональные возможности	Бизнес-ценность, постоянное улучшение
Релизный цикл	Длительные релизные циклы, редкие развертывания	Короткие спринты, частые релизы со сбором обратной связи
Тестирование	Отдельная фаза тестирования после разработки	Интегрированное тестирование на протяжении всего цикла разработки
Инфраструктура	Статическая инфраструктура, управляемая вручную	Автоматизированное предоставление и управление инфраструктурой (DevOps)

Итак, мы рассмотрели организационные изменения, вызванные внедрением гибких методологий. Теперь обсудим техническую эволюцию, которая произошла за последние несколько лет.

Переход от локальной инфраструктуры к облаку

Наличие agile-команд — это замечательно, но гибкость должна распространяться не только на организацию работы, но и на способы создания и размещения программного обеспечения.

Чтобы увеличить частоту релизов, agile-команды были вынуждены пересмотреть такие важные аспекты процесса разработки и выпуска ПО, как:

- размещение приложений;
- архитектура программного обеспечения.

Современные приложения рассчитаны не на сотни, а на миллионы одновременных подключений. Рост количества пользователей потребовал соответствующего увеличения вычислительных мощностей. В результате организация хостинга приложений превратилась в серьезную технологическую задачу.

На заре развития веб-хостинга компании использовали два основных подхода к размещению своих приложений, одним из которых был локальный хостинг (on-premises hosting). Этот способ подразумевал физическое владение и управление серверами, на которых работали приложения. Для организации локального хостинга могли использоваться следующие средства.

1. *Выделенные серверы*. Этот вариант предполагал аренду физических серверов у специализированных провайдеров. При этом провайдер обеспечивал работу физической инфраструктуры (электропитание, охлаждение, безопасность), а ответственность за настройку серверов, установку ПО и текущее обслуживание брала на себя сама компания. Такой подход давал больше контроля и возможностей для настройки по сравнению с виртуальным хостингом, но по-прежнему требовал наличия в штате компании квалифицированных ИТ-специалистов.
2. *Собственный дата-центр*. Такой подход предполагал строительство и обслуживание частного центра обработки данных. Этот вариант требовал колоссальных инвестиций в создание и эксплуатацию собственной физической инфраструктуры, включающей серверное и сетевое оборудование, хранилища данных, а также надежные системы электропитания, охлаждения и обеспечения безопасности. Такой подход предоставлял максимальный уровень контроля и безопасности, но являлся крайне затратным и ресурсоемким, поэтому обычно применялся только крупными корпорациями, располагающими серьезными ИТ-ресурсами.

Стоит отметить, что локальный хостинг также предполагает управление ОС, обновление системы безопасности, создание резервных копий и разработку планов восстановления после сбоев в работе серверов. В связи с этим компаниям, выбравшим данный подход, часто приходилось организовывать отдельную ИТ-команду для обслуживания локальной инфраструктуры и управления ею, что дополнительно увеличивало затраты.

По мере роста пользовательской базы компании были вынуждены наращивать вычислительные мощности, арендуя более производительные серверы или устанавливая новое оборудование в собственном дата-центре. Такой подход не отличается гибкостью. И сегодня многие компании все еще используют такие малогибкие локальные решения.

Переломным моментом стало появление публичного облака, служащего альтернативой локальной инфраструктуре. Такие крупные компании, как Amazon,

Google и Microsoft, владеющие множеством дата-центров, создали слой виртуализации поверх своей масштабной инфраструктуры, предоставив доступ к виртуальным машинам через API. Благодаря этому виртуальную машину можно получить с помощью всего нескольких щелчков кнопкой мыши или путем ввода пары команд в консоли.

В табл. 1.2 перечислены основные выгоды облачных вычислений для организаций.

Таблица 1.2. Важность облачных вычислений для организаций

Характеристика	Локальная инфраструктура	Облако
Масштабируемость	Ограниченная — для увеличения ресурсов требуется закупка нового оборудования	Высокая — ресурсы легко добавляются или удаляются по запросу
Гибкость	Низкая — для внесения изменений требуется модернизация физического оборудования	Высокая — ресурсы можно быстро выделить и освободить
Стоимость	Высокие первоначальные затраты на покупку оборудования, лицензионного ПО и формирование ИТ-отдела	Низкие первоначальные затраты — ресурсы оплачиваются по мере использования
Обслуживание	Для обслуживания и обновления системы требуется отдельная команда ИТ-специалистов	Минимальные затраты на обслуживание, так как инфраструктурой управляет облачный провайдер
Безопасность	Практически полный контроль над безопасностью, но для его осуществления требуются квалифицированные специалисты	Надежные меры безопасности, реализуемые облачным провайдером
Время простоя	Восстановление после аппаратных сбоев может занимать много времени	Облачные провайдеры обеспечивают высокую доступность и возможности для аварийного восстановления
Расположение	Привязка к физическому местоположению дата-центра	Доступ из любой точки мира при наличии интернет-соединения

В следующем подразделе вы узнаете о том, как облачные технологии помогли организациям масштабировать свою ИТ-инфраструктуру.

Почему облако хорошо подходит для масштабирования

Сегодня практически любой человек может нажать буквально несколько кнопок, чтобы получить сотни или даже тысячи серверов в виде виртуальных машин или экземпляров, созданных на базе физической инфраструктуры, обслуживаемой такими облачными провайдерами, как *Amazon Web Services*, *Google Cloud Platform* и *Microsoft Azure*. За последние годы многие компании перенесли свои рабочие нагрузки из локальной инфраструктуры в облако, и масштаб этого перехода был колоссальным.

Благодаря этому в настоящее время вычислительные мощности являются одним из самых легкодоступных ресурсов.

Облачный хостинг превратился в стандартное решение для размещения приложений, доступное agile-командам, поскольку облако идеально соответствует современным подходам к разработке ПО.

Конфигурации виртуальных машин, процессоры, операционные системы, правила обработки сетевого трафика и многое другое доступно для просмотра и полностью настраивается, поэтому ваша команда точно знает, как устроена рабочая среда. Программируемая природа облачных платформ позволяет легко воспроизвести рабочую среду в среде разработки или тестирования. Это повышает гибкость и помогает командам справляться с проблемами, возникающими в процессе создания программного обеспечения. Данное преимущество особенно важно для agile-команд, работающих в соответствии с принципами DevOps и отвечающих за разработку, выпуск и сопровождение приложений в рабочей среде.

Использование облачных сервисов дает такие преимущества, как:

- эластичность и масштабируемость;
- помощь в преодолении изоляции команд и внедрении гибких методологий;
- полная совместимость с гибкими методологиями и DevOps;
- низкая стоимость и гибкие модели оплаты;
- отсутствие необходимости управлять физическими серверами;
- возможность беспрепятственно удалять и заново создавать виртуальные машины;
- большая гибкость по сравнению с арендой физических серверов.

Благодаря этим преимуществам облако стало ценнейшим инструментом в арсенале agile-команд разработчиков. По сути, вы можете создавать и многократно воспроизводить рабочую среду, не занимаясь при этом обслуживанием

физического оборудования. Облако позволяет масштабировать приложение в зависимости от количества пользователей или объема потребляемых ими вычислительных ресурсов. Это делает приложение высокодоступным и отказоустойчивым, что в конечном счете улучшает пользовательский опыт.

Kubernetes можно запускать как в облаке, так и в локальной инфраструктуре. Эта система чрезвычайно универсальна, ее можно развернуть даже на одноплатном компьютере Raspberry Pi. Хотя Kubernetes и публичное облако прекрасно сочетаются друг с другом, вы не обязаны использовать этот инструмент именно в облаке.

Теперь, когда мы обсудили изменения, связанные с появлением облачных технологий, обратимся к архитектуре программного обеспечения, так как и в этой области за последние годы произошло немало перемен.

Монолитная архитектура

В прошлом большинство приложений создавалось в виде монолитов. Типичное монолитное приложение представляет собой единый процесс, один исполняемый файл или один пакет, как показано на рис. 1.3.

Рис. 1.3. Монолитное приложение представляет собой один большой компонент, содержащий все программное обеспечение

Этот единый компонент отвечает за реализацию всей бизнес-логики, предусмотренной в программном обеспечении. Монолиты могут быть хорошим выбором, если вы разрабатываете простое приложение, которое не требуется часто обновлять в рабочей среде. Почему? Потому что у монолитов есть один серьезный недостаток: если он становится нестабильным или по какой-либо причине выходит из строя, все приложение оказывается недоступным.

Монолитная архитектура позволяет заметно сократить время разработки ПО, и, пожалуй, это ее единственное преимущество. Однако у нее есть и множество недостатков. Вот лишь некоторые из них:

- неудачное развертывание в рабочей среде может вывести из строя все приложение;
- масштабирование становится сложной задачей, а при его сбое все экземпляры приложения могут оказаться недоступными;
- любой сбой в монолите способен привести к полной остановке работы приложения.

В 2010-х годах эти недостатки начали вызывать серьезные проблемы. С ростом частоты развертываний возникла необходимость в архитектуре, способной поддерживать частые релизы и короткие циклы обновлений, а также снижающей риск возникновения полной недоступности приложения. Именно поэтому была разработана микросервисная архитектура.

Микросервисная архитектура

Микросервисная архитектура предполагает создание программного обеспечения в виде набора независимых микроприложений. Каждое из них называется *микросервисом* и имеет собственную систему версионирования, жизненный цикл, окружение и зависимости, а при необходимости и отдельный цикл развертывания. Каждый микросервис отвечает за ограниченное число бизнес-правил, а все они в совокупности составляют полноценное приложение. По сути, микросервис можно рассматривать как самостоятельное полнофункциональное ПО со своим жизненным циклом и процессом версионирования.

Поскольку микросервисы реализуют лишь часть функциональных возможностей всего приложения, для использования их функций к ним необходимо предоставить доступ. Как минимум нужно обеспечить возможность получать данные от микросервиса, а иногда и передавать их ему. Для предоставления доступа к микросервисам можно использовать такие широко поддерживаемые протоколы, как HTTP или AMQP. Кроме того, важно обеспечить взаимодействие микросервисов между собой.

В связи с этим микросервисы обычно создаются как веб-сервисы, предоставляющие свою функциональность через четко определенные API. Хотя REST API, реализованные поверх HTTP (или HTTPS), остаются популярным выбором благодаря своей простоте и повсеместному использованию, все большее распространение получают такие альтернативные протоколы, как GraphQL, AMQP и gRPC.

Главное требование к микросервису заключается в том, чтобы он предоставлял хорошо документированную и легко обнаруживаемую конечную точку API независимо от выбранного протокола. Это обеспечивает беспрепятственное взаимодействие микросервисов и обмен данными между ними.

Именно в этом состоит принципиальное отличие микросервисной архитектуры от монолитной (рис. 1.4).

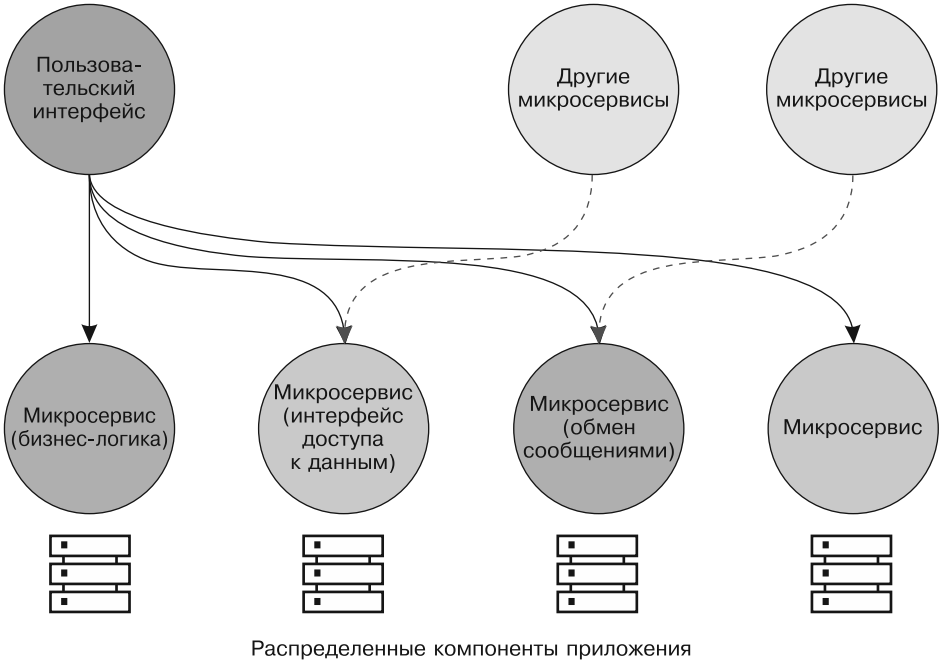

Рис. 1.4. Микросервисная архитектура, в которой различные микросервисы взаимодействуют по протоколу HTTP

Еще один ключевой принцип построения микросервисной архитектуры состоит в том, что микросервисы должны быть слабо связаны между собой. Если какой-то микросервис становится недоступным или нестабильным, это не должно

влиять на работу остальных микросервисов или на стабильность приложения в целом. Вы должны иметь возможность выделять ресурсы, масштабировать, запускать, обновлять или останавливать каждый микросервис, не затрагивая остальные. Если микросервису требуется взаимодействовать с СУБД, то и база данных должна быть изолирована, то есть каждый микросервис должен иметь собственную базу данных. В этом случае сбой в базе данных *микросервиса А* не повлияет на работу *микросервиса Б* (рис. 1.5).

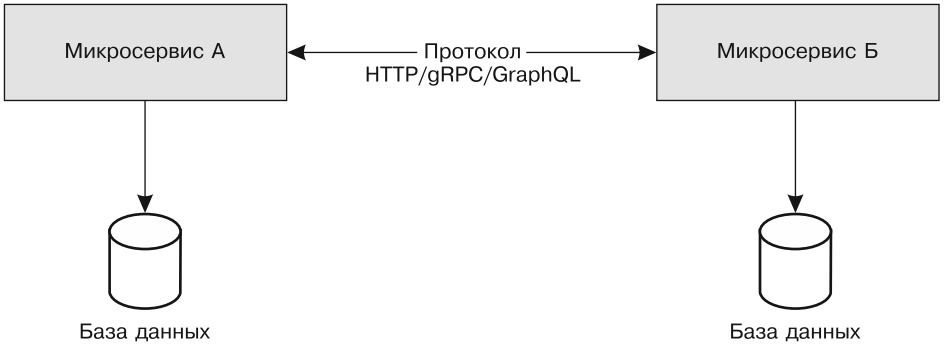

Рис. 1.5. Микросервисная архитектура, в которой отдельные микросервисы взаимодействуют между собой и с выделенным сервером баз данных, что обеспечивает их изоляцию и отсутствие общих зависимостей

Главное правило — максимально изолировать микросервисы, чтобы они были полностью независимы друг от друга. Такая независимость означает, что микросервисы могут использовать совершенно разные технические среды и реализовываться на разных языках программирования. Например, один микросервис может быть написан на Go, другой — на Java, а третий — на PHP, и вместе они будут составлять единое приложение. В рамках микросервисной архитектуры это не является проблемой. Поскольку протокол HTTP — это стандарт, микросервисы смогут взаимодействовать между собой даже при различиях в реализации.

Микросервисы должны быть изолированы не только друг от друга, но и от операционной системы, в которой они работают. Они должны функционировать не на уровне хостовой системы, а на более высоком уровне, чтобы их можно было разворачивать на разных машинах без жесткой привязки к ОС. Именно поэтому микросервисная архитектура отлично сочетается с контейнерами.

Если вам требуется внедрить новую функцию в рабочую среду, достаточно развернуть только те микросервисы, которые затрагивает новая версия ПО. Остальные могут продолжать работу без изменений.

Как вы, наверное, уже догадались, микросервисная архитектура обладает значительными преимуществами в контексте современной разработки приложений, а именно:

- упрощает организацию регулярных релизов в рабочей среде при минимальном влиянии на стабильность всего приложения;
- позволяет обновлять только конкретный микросервис, а не приложение целиком;
- облегчает масштабирование за счет увеличения ресурсов отдельных сервисов.

Однако у микросервисной архитектуры есть и недостатки:

- она требует более тщательного планирования и оказывается более сложной в разработке;
- управление зависимостями каждого микросервиса вызывает определенные трудности.

Разработка микросервисных приложений считается довольно сложной задачей. Такой подход может показаться трудным для понимания, особенно начинающим разработчикам. Кроме того, управление зависимостями может осложняться наличием собственного набора зависимостей у каждого микросервиса.

Выбор между монолитной и микросервисной архитектурами

Создание успешного приложения требует тщательного планирования, и одним из ключевых решений, которое вам предстоит принять, является выбор архитектуры. Сегодня в отрасли доминируют два основных подхода: монолиты и микросервисы.

- *Монолиты*. Представьте себе компактную систему, построенную по принципу «все в одном», — это и есть монолит. Весь его код содержится в одной кодовой базе, что упрощает разработку и начальное развертывание. Данный вариант хорошо подходит для небольших проектов или команд с ограниченными ресурсами. Кроме того, обновление монолитов обычно выполняется относительно быстро, поскольку при этом приходится управлять только одной системой.
- *Микросервисы*. Этот вариант предполагает разделение приложения на независимые модульные компоненты, каждый из которых можно разрабатывать, масштабировать и развертывать отдельно. Такой подход особенно эффективен для реализации крупных, функционально насыщенных проектов, над которыми работают команды, обладающие разнообразной экспертностью. Микросервисы обеспечивают гибкость и ускоряют разработку, но при этом усложняют отладку и управление безопасностью.

В конечном счете выбор между монолитом и микросервисами зависит от ваших конкретных потребностей. Учитывайте масштаб проекта, структуру команды и желаемый уровень гибкости. Не поддавайтесь модным тенденциям — выбирайте архитектуру, которая позволит вашей команде разрабатывать и сопровождать приложение максимально эффективно.

Kubernetes одинаково хорошо работает как с монолитами, так и с микросервисами, позволяя выбрать архитектуру, наилучшим образом соответствующую потребностям проекта.

В следующем разделе мы поговорим о контейнерах и узнаем, как они помогают реализовывать микросервисную архитектуру.

Контейнеры

Сравнение монолитной и микросервисной архитектур показало, что микросервисная архитектура лучше всего сочетает в себе принципы гибкой разработки и DevOps. Поскольку Kubernetes особенно эффективно управляет приложениями, построенными на основе микросервисов, именно этому виду архитектуры будет посвящена эта книга.

Теперь перейдем к обсуждению Docker, контейнерного движка для Linux, который является отличным инструментом для управления микросервисами. Если вы уже хорошо знакомы с Docker, то можете пропустить этот раздел, в противном случае рекомендуем вам изучить его внимательно.

Почему контейнеры подходят для развертывания микросервисов

Вспомним два ключевых принципа построения микросервисной архитектуры.

1. Каждый микросервис может иметь собственное техническое окружение и набор зависимостей.
2. При этом он должен быть изолирован от операционной системы, в которой работает.

Пока отвлечемся от второго пункта и сосредоточимся на первом. Два микросервиса одного приложения могут быть написаны на разных языках или на одном языке, но в виде двух разных версий. Попытка развернуть эти два микросервиса на одной машине с Linux может превратиться в настоящий кошмар.

Причина заключается в том, что вам придется установить все версии необходимого окружения выполнения и зависимостей, что может привести к воз-

никновению конфликтов. При этом все компоненты будут работать в одной хостовой операционной системе. Теперь представьте, что вы решили удалить один из этих микросервисов с сервера, чтобы развернуть его на другой машине, и при этом полностью очистить первый сервер от всех зависимостей, которые использовал данный микросервис. Разумеется, опытный Linux-инженер справится с этой задачей, однако для большинства пользователей риск возникновения конфликтов между зависимостями чрезвычайно высок. В итоге приложение, работающее в такой сложной инфраструктуре, может оказаться полностью недоступным.

Чтобы решить эту проблему, можно создать образ машины для каждого микросервиса и запустить каждый из них на отдельной виртуальной машине, то есть отказаться от развертывания нескольких микросервисов на одном сервере. Однако в этом случае потребуется по одной машине на каждый микросервис. Разумеется, с помощью AWS или GCP можно без труда развернуть десятки серверов, каждый из которых будет обслуживать только один микросервис, но это приведет к лишним расходам, поскольку при таком подходе вычислительные ресурсы хоста не используются совместно.

В мире контейнеров существуют аналогичные решения, однако стандартные среды выполнения контейнеров не обеспечивают полной изоляции микросервисов. Именно здесь в игру вступают среда выполнения *Kata* и проект *Confidential Containers*. Эти технологии обеспечивают более высокий уровень безопасности и улучшенную изоляцию контейнеризованных приложений. Более подробно мы поговорим о них далее в книге.

В следующем подразделе вы узнаете о том, как контейнеры обеспечивают изоляцию приложений.

Преимущества изоляции с помощью контейнеров

Такие контейнерные движки, как Docker и Podman, играют ключевую роль в управлении микросервисами. В отличие от *виртуальных машин* (ВМ), которым требуется полноценная гостевая операционная система, контейнеры представляют собой легковесные модули, использующие ядро Linux хостовой системы. Благодаря этому они запускаются и останавливаются значительно быстрее, чем виртуальные машины.

Контейнерные движки предоставляют удобный API для сборки, развертывания и управления контейнерами. Они не создают дополнительный слой виртуализации, а используют встроенные возможности ядра Linux для изоляции процессов, обеспечения безопасности и распределения ресурсов. Столь эффективный подход делает контейнеризацию оптимальным решением для развертывания микросервисов.

На рис. 1.6 показано, чем контейнеры отличаются от виртуальных машин.

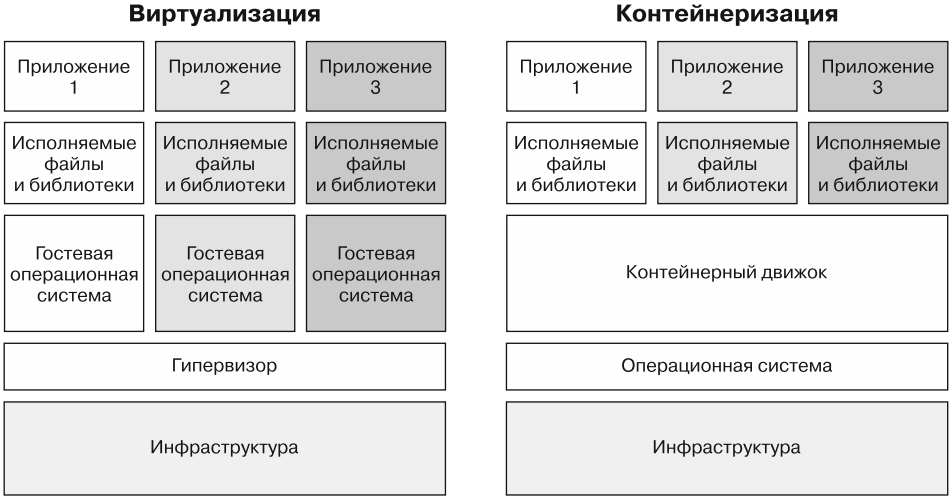

Рис. 1.6. Различие между виртуальными машинами и контейнерами

Ваши микросервисы будут запускаться поверх этого слоя, а не напрямую в хостовой системе, функция которой ограничивается выполнением контейнеров.

Изоляция контейнеров позволяет создавать их в любом количестве и использовать для запуска приложений, написанных на разных языках, не опасаясь возникновения конфликтов. При этом перенос микросервиса сводится к остановке работающего контейнера и запуску нового экземпляра на другой машине из того же образа.

Использование контейнеров с микросервисами дает три основных преимущества:

- снижение влияния на изменения хостовой системы;
- совместное использование ресурсов хоста без возникновения конфликтов между микросервисами;
- исключение зависимости микросервиса от хостовой системы.

В результате контейнеризации микросервис оказывается почти полностью отделенным от хостовой операционной системы и зависит исключительно от контейнера, в котором работает. Поскольку контейнер является более легковесным по сравнению с полноценной Linux, его распространение и развертывание на множестве машин не составляет труда. Таким образом, контейнер и микросервис могут работать на любой машине с установленным контейнерным движком.

На рис. 1.7 показана микросервисная архитектура, в которой каждый микросервис упакован в контейнер.

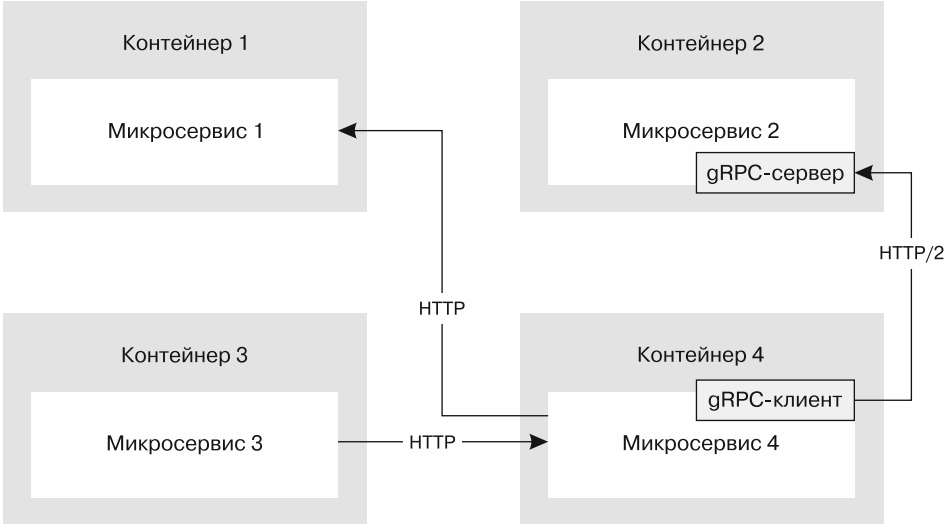

Рис. 1.7. Приложение на основе микросервисов, каждый из которых упакован в контейнер. Благодаря тому что жизненный цикл приложения связан с контейнером, его можно легко развернуть на любой машине с установленным контейнерным движком

Контейнеры также органично вписываются в методологию DevOps. Разрабатывая приложение локально в контейнере, который впоследствии будет собран и развернут в рабочей среде, вы гарантируете полное соответствие между средами разработки и эксплуатации ПО.

Контейнерные движки способны управлять не только жизненным циклом контейнера, но и целой экосистемой вокруг него. Они позволяют настраивать сети, налаживать взаимодействие между контейнерами и соответствуют многим особенностям микросервисной архитектуры, о которых мы говорили ранее.

Используя контейнеры в сочетании с облачными технологиями, вы можете построить надежную инфраструктуру для размещения своих микросервисов. Облачный сервис предоставляет столько машин, сколько вам необходимо. Установив на них контейнерный движок, вы сможете развернуть на каждой по несколько контейнеризованных микросервисов.

Контейнерные движки, такие как Docker или Podman, сами по себе являются весьма удобными инструментами. Однако их сложно использовать в рабочей среде в чистом виде.

Контейнерные движки особенно хорошо проявляют себя в средах разработки благодаря:

- *простоте* — их легко установить и использовать, что позволяет быстро собирать, тестировать и запускать контейнеризованные приложения;
- *гибкости* — разработчики могут экспериментировать с различными конфигурациями контейнеров и исследовать возможности контейнеризации;
- *изоляции* — контейнерные движки обеспечивают изоляцию приложений, предотвращая конфликты и упрощая отладку.

Однако рабочие среды предъявляют гораздо более строгие требования, которые контейнерные движки не способны полностью удовлетворить.

- *Масштабирование*. Контейнерные движки (такие как Docker или Podman) не имеют встроенных средств автоматического масштабирования, позволяющих динамически адаптировать процесс развертывания контейнеров в зависимости от уровня потребления ресурсов.
- *Восстановление после сбоев*. Контейнерные движки не предусматривают полноценного механизма аварийного восстановления, обеспечивающего доступность сервисов при возникновении сбоев.
- *Безопасность*. Хотя контейнерные движки обеспечивают базовую изоляцию, управление политиками безопасности при масштабном развертывании контейнеров на множестве машин является весьма сложной задачей.
- *Стандартизация*. Для настройки взаимодействия контейнерных движков с такими внешними системами, как CI/CD-конвейеры или инструменты мониторинга, может потребоваться написание специальных скриптов или применение дополнительных средств интеграции.

Итак, контейнерные движки отлично проявляют себя в средах разработки, однако для развертывания приложений в рабочей среде требуется более надежное решение, например Kubernetes — мощная платформа для оркестрации контейнеров, предлагающая полный набор функциональных возможностей. Она управляет всем жизненным циклом контейнеров: от планирования запуска на доступных ресурсах до масштабирования развертываний в зависимости от уровня нагрузки и распределения трафика для достижения оптимальной производительности. В отличие от пользовательских скриптов для контейнерных движков Kubernetes предоставляет четко определенный API для взаимодействия с контейнеризованными приложениями, что упрощает интеграцию с другими инструментами, используемыми в рабочей среде. Помимо базовой изоляции приложений, Kubernetes предлагает расширенные средства безопасности, в частности механизм управления доступом на основе ролей и сетевые политики. Это позволяет эффективно управлять контейнеризованными рабочими нагрузками, имеющими отношение к разным командам или проектам, в рамках одной инфраструктуры, оптимизировать использование ресурсов и упрощать процесс развертывания.

Прежде чем приступить к изучению Kubernetes, давайте кратко обсудим тему контейнеров и контейнерных движков.

Контейнерные движки

Контейнерный движок служит интерфейсом для конечных пользователей и REST-клиентов. Он обрабатывает пользовательский ввод, загружает образы контейнеров из реестров, распаковывает их на диск, преобразует данные, полученные от пользователя или REST-клиента, для дальнейшего использования, подготавливает точки монтирования контейнеров и обеспечивает взаимодействие со средами выполнения. По сути, движок — это внешний слой, упрощающий управление образами и контейнерами, в то время как лежащие в его основе среды выполнения осуществляют все сложные низкоуровневые операции.

Docker является одним из самых распространенных контейнерных движков, однако у него есть немало альтернатив, среди которых можно выделить *LXD*, *Rkt*, *CRI-O* и *Podman*.

В основе работы Docker лежит среда выполнения контейнеров `containerd`, которая отвечает за такие ключевые аспекты управления контейнерами, как их жизненный цикл, передача и хранение образов, запуск и контроль их работы, а также подключение хранилищ и сетевых ресурсов. В свою очередь, среда `containerd` использует такие компоненты, как `runc` и `hcsshim`. Первый представляет собой инструмент командной строки, предназначенный для создания и запуска контейнеров в Linux, а второй играет ключевую роль в создании и управлении контейнерами в Windows.

Следует отметить, что среда `containerd` не предназначена для прямого взаимодействия с конечным пользователем. Контейнерные движки вроде Docker обычно работают с такими средами выполнения, упрощая процесс создания и управления контейнерами. Компонент `runc` играет ключевую роль — он используется не только средой `containerd`, но и движками Podman, CRI-O и косвенно самой платформой Docker.

Основы работы с контейнерами

Как было сказано в предыдущем разделе, Docker является одним из самых известных и широко используемых контейнерных движков. Теперь разберемся с базовой терминологией, связанной с контейнерами.

Образ контейнера

Образ контейнера — это своего рода шаблон, который контейнерные движки используют для запуска контейнеров. Он представляет собой автономный исполняемый пакет, включающий приложение и все необходимые зависимости. В него входит все, что нужно для работы программного обеспечения: код, среда

выполнения, библиотеки и системные утилиты. Образы контейнеров создаются на основе файлов Dockerfile или Containerfile, в которых описаны шаги сборки. Эти образы хранятся в репозиториях и распространяются через такие реестры контейнеров, как Docker Hub, что делает их ключевым элементом технологии контейнеризации.

Контейнер

Контейнер можно рассматривать как запущенный экземпляр образа. Подобно морским грузовым контейнерам, они служат упаковкой для кода приложения, его зависимостей и среды выполнения, объединяя их в один легковесный пакет. Благодаря тому что контейнер включает все необходимое, он одинаково работает в разных средах. Каждый контейнер выполняется изолированно, что предотвращает конфликты с другими приложениями, работающими в той же системе. Контейнеры совместно используют ядро хостовой операционной системы, благодаря чему они запускаются и останавливаются значительно быстрее, чем виртуальные машины.

Реестр контейнеров

Реестр контейнеров представляет собой централизованный репозиторий для хранения и обмена образами контейнеров. Он играет роль механизма распространения, позволяя пользователям отправлять образы в реестр и загружать их оттуда. Среди популярных публичных реестров можно выделить Docker Hub, Red Hat Quay, *Elastic Container Registry* (ECR) от Amazon, Azure Container Registry, Google Container Registry и GitHub Container Registry. Многие организации используют частные реестры для безопасного хранения и обмена пользовательскими образами. Реестры играют ключевую роль в экосистеме Docker, способствуя совместной работе и эффективному управлению контейнеризованными приложениями.

Файлы Dockerfile и Containerfile

Dockerfile или Containerfile — это текстовый файл, содержащий набор инструкций для сборки образа контейнера. Он определяет базовый образ, настраивает окружение, копирует код приложения, устанавливает зависимости и задает параметры среды выполнения. Эти файлы автоматизируют процесс создания полноценных образов, позволяя разработчикам версионировать и распространять конфигурации их приложений.

Пример файла Dockerfile приведен в следующем фрагменте кода:

```
# syntax=docker/dockerfile:1

FROM node:18-alpine
WORKDIR /app
```

```
COPY . .
RUN yarn install --production
CMD ["node", "src/index.js"]
EXPOSE 3000
```

Разберем его построчно.

1. Строка `#syntax=docker/dockerfile:1` определяет версию синтаксиса Dockerfile, используемую для сборки образа. В данном случае указана версия 1 стандартного синтаксиса Dockerfile.
2. Строка `FROM node:18-alpine` определяет базовый образ контейнера. Здесь используется официальный образ Node.js 18 на базе Alpine Linux, который служит легковесной и достаточной основой для приложения.
3. Строка `WORKDIR /app` задает рабочий каталог внутри контейнера. В данном случае это `/app`. Все последующие команды Dockerfile будут выполняться относительно этого каталога.
4. Строка `COPY . .` копирует все файлы и каталоги из текущего контекста (каталога, в котором находится Dockerfile) в рабочий каталог (`/app`), указанный на предыдущем шаге. Таким образом, в контейнер помещается весь код приложения[1].
5. Строка `RUN yarn install --production` приказывает контейнерному движку выполнить команду внутри контейнера. В данном случае выполняется команда `yarn install --production`, которая устанавливает все рабочие зависимости, перечисленные в файле `package.json`, с помощью пакетного менеджера `yarn`. Флаг `--production` гарантирует установку только тех зависимостей, которые необходимы в рабочей среде, при этом зависимости, используемые при разработке, игнорируются.
6. Строка `CMD ["node", "src/index.js"]` задает команду по умолчанию, выполняемую при запуске контейнера. Здесь указаны два элемента: `node` и `src/index.js`. Это означает, что при запуске контейнера будет запущен интерпретатор Node.js (`node`), который выполнит основной сценарий приложения (`src/index.js`).
7. Строка `EXPOSE 3000` открывает порт внутри контейнера. В данном случае будет открыт порт `3000`. По умолчанию эта инструкция не сопоставляет порт контейнера с портом хостовой машины, однако вы можете сделать это позже при запуске контейнера с флагом `-p` (например, `docker run -p 3000:3000 my-image`). Открытие порта `3000` предполагает, что ваше приложение ожидает входящие подключения именно на этом порте.

---

[1] Главное не забывать держать рядом подходящую конфигурацию .dockerignore, которая исключит из копирования такие очевидно ненужные в образе файлы, как сам Dockerfile и, к примеру, node_modules для данного примера, так как образ не должен содержать лишнего.

Для сборки образа контейнера можно использовать поддерживаемый контейнерный движок (например, Docker или Podman) либо инструмент для сборки контейнеров, такой как Buildah или kaniko.

Инструменты Docker Compose и Podman Compose

Docker Compose — это инструмент для определения и запуска многоконтейнерных приложений. Он использует YAML-файл для настройки необходимых приложению сервисов, сетей и томов, позволяя разработчикам определить всю программную инфраструктуру в одном месте. Инструменты Docker Compose и Podman Compose упрощают оркестрацию сложных приложений, помогая управлять множеством контейнеров как единым стеком.

Следующий файл, `compose.yaml`, создает два контейнера для стека приложений WordPress с помощью одной команды `docker compose` или `podman compose`:

```
# compose.yaml
services:
  db:
    image: docker.io/library/mariadb
    command: '--default-authentication-plugin=mysql_native_password'
    volumes:
      - db_data:/var/lib/mysql
    restart: always
    environment:
      - MYSQL_ROOT_PASSWORD=somewordpress
      - MYSQL_DATABASE=wordpress
      - MYSQL_USER=wordpress
      - MYSQL_PASSWORD=wordpress
    expose:
      - "3306"
      - "33060"
    networks:
      - wordpress
  wordpress:
    image: wordpress:latest
    ports:
      - "8081:80"
    restart: always
    environment:
      - WORDPRESS_DB_HOST=db
      - WORDPRESS_DB_USER=wordpress
      - WORDPRESS_DB_PASSWORD=wordpress
      - WORDPRESS_DB_NAME=wordpress
    networks:
      - wordpress
volumes:
  db_data:
networks:
  wordpress: {}
```

В следующем разделе будет показано, насколько эффективно система Kubernetes способна оркестровать все эти операции с контейнерами.

Управление контейнерами с помощью Kubernetes

В этом разделе мы сосредоточимся на системе Kubernetes, которой посвящена эта книга.

Kubernetes — это платформа для запуска рабочих нагрузок в рабочей среде

Если открыть официальный сайт Kubernetes (https://kubernetes.io), то в заголовке можно увидеть надпись Production-Grade Container Orchestration (рис. 1.8).

Рис. 1.8. Главная страница сайта Kubernetes с заголовком, представляющим эту систему как платформу для оркестрации контейнеров в рабочей среде

Эта фраза прекрасно передает суть Kubernetes: это платформа для оркестрации контейнеров в рабочей среде. Система Kubernetes не пытается заменить Docker (или другие контейнерные движки). Она предназначена для управления кластерами машин, на которых работают среды выполнения контейнеров. При работе с Kubernetes, помимо самой этой системы, вы используете полнофункциональные стандартные установки контейнерных движков.

В заголовке на сайте упоминается *рабочая среда* (production, продакшен), и это понятие действительно является ключевым для Kubernetes. Данная система была разработана для решения задач, связанных с эксплуатацией приложений в современной рабочей среде. Управление рабочими нагрузками сегодня существенно отличается от того, каким оно было в 2000-х годах. В те времена рабочая среда обычно представляла собой всего несколько физических серверов (или даже один локальный сервер), на которых запускались в основном монолитные приложения, установленные непосредственно в хостовой ОС Linux. Сегодня благодаря публичным облачным платформам, таким как *Amazon Web Services (AWS)* и *Google Cloud Platform (GCP)*, любой пользователь может с легкостью получить сотни или даже тысячи серверов в виде экземпляров или виртуальных машин. Более того, приложения теперь разворачиваются не в хостовой системе, а поверх движка Docker в виде контейнеризованных микросервисов, что позволяет уменьшить влияние на конфигурацию хоста.

Теперь проблема связана с необходимостью управлять установками Docker на каждой из этих виртуальных машин в облаке. Представьте, что у вас есть 10 (100 или 1000) машин, запущенных в облаке, и вам нужно выполнить простую задачу — развернуть контейнеризованное Docker-приложение на каждой из них.

Это можно сделать, выполнив команду `docker run` на каждой машине. Однако существует и более эффективный способ — использовать *оркестратор контейнеров*, например *Kubernetes*. В двух словах Kubernetes можно описать как *REST API*, который ведет учет всех машин с запущенным демоном Docker.

Однако это определение является слишком упрощенным. На самом деле Kubernetes представляет собой не один централизованный REST API, а набор микросервисов.

Стоит также отметить, что, хотя система Kubernetes отлично управляет контейнеризованными рабочими нагрузками, она не может полностью заменить виртуальные машины (ВМ). ВМ по-прежнему полезны для решения определенных задач, например для запуска устаревших приложений или ПО со сложными зависимостями, которые трудно контейнеризовать. Тем не менее Kubernetes продолжает развиваться в сторону устранения разрыва между контейнерами и виртуальными машинами.

KubeVirt — мост между контейнерами и виртуальными машинами

KubeVirt — это проект, расширяющий возможности Kubernetes за счет управления виртуальными машинами через привычный API Kubernetes. Такой подход позволяет пользователям задействовать всю мощь и гибкость Kubernetes для развертывания ВМ наряду с контейнеризованными приложениями. KubeVirt следует принципам *IaC* (Infrastructure as Code — инфраструктура как код), что дает пользователям возможность определять и управлять ВМ декларативно, прямо в манифестах Kubernetes. Это упрощает администрирование ВМ и органично интегрирует их в существующие рабочие процессы Kubernetes.

Включая виртуальные машины в зону ответственности Kubernetes, KubeVirt обеспечивает эффективный способ организации гибридной среды, в которой могут сосуществовать контейнеры и ВМ. Этот проект демонстрирует продолжающуюся эволюцию Kubernetes как платформы для управления разнородными рабочими нагрузками, открывая путь к разработке более унифицированного подхода к развертыванию и сопровождению приложений.

Итак, мы обсудили работу с контейнерами и сложности их оркестрации в масштабах крупной инфраструктуры. В следующем разделе вы познакомитесь с историей развития системы Kubernetes.

История развития Kubernetes

Теперь поговорим о том, как развивалась система Kubernetes, а также о людях, поддерживающих этот проект сегодня.

Истоки Kubernetes

С момента своего основания в 1998 году компания Google накопила большой опыт управления масштабными и ресурсоемкими рабочими нагрузками, в частности контейнерными. Начиная с середины 2000-х годов Google разрабатывала свои приложения в виде контейнеров Linux. Она осознала преимущества контейнеризации задолго до появления системы Docker, упростившей использование контейнеров для широкой аудитории, что привело к запуску внутреннего проекта компании под названием Borg. Чтобы сделать архитектуру Borg более расширяемой и надежной, Google инициировала разработку еще одного оркестратора контейнеров под названием Omega. Впоследствии ряд улучшений, внедренных в Omega, были интегрированы в систему Borg.

Kubernetes зародился как внутренний проект Google. Первый коммит в репозиторий этой системы был сделан в 2014 году несколькими разработчиками, среди которых были Брендан Бернс, Джо Беда и Крейг Маклендон. Однако поначалу Google не собиралась открывать исходный код Kubernetes. Важную роль в продвижении этой идеи сыграл Клейтон Коулман из Red Hat, который активно отстаивал необходимость сделать проект открытым и передать его развитие в руки сообщества. Келси Хайтауэр, один из первых сторонников Kubernetes в компании CoreOS, стал заметным популяризатором данной технологии. Как спикер, автор и сооснователь конференции KubeCon, он оказал значительное влияние на распространение Kubernetes и рост сообщества пользователей этой системы.

Сегодня в развитии Kubernetes, помимо Google, активно участвуют Red Hat, Amazon, Microsoft и многие другие компании.

Проект Borg нельзя назвать предшественником Kubernetes, потому что он до сих пор используется внутри Google. Корректнее будет сказать, что при создании Kubernetes было заимствовано множество идей из системы Borg. Важно помнить о том, что Kubernetes — это не Borg и не Omega. Borg был написан на языке C++, а Kubernetes — на Go. Фактически это два совершенно разных проекта, хотя один и был в значительной степени вдохновлен другим. Borg и Omega — это внутренние проекты Google, которые изначально не предназначались для широкой аудитории.

Система Kubernetes была разработана с учетом опыта, накопленного компанией Google при управлении контейнерами в рабочей среде. При этом она

унаследовала идеи, концепции и архитектурные решения, использованные в проектах Borg и Omega. Вот краткий список этих заимствованных идей и концепций.

- Для создания, обновления и удаления контейнеров в Kubernetes используется логический объект под названием «под» (pod).
- Каждый под получает собственный IP-адрес внутри кластера.
- Распределенные компоненты отслеживают центральный API Kubernetes, чтобы получать актуальное состояние кластера.
- В системе предусмотрен встроенный механизм балансировки нагрузки между подами и сервисами.
- Для управления ресурсами в Kubernetes используются метаданные в виде меток и селекторов.

Все это делает Kubernetes исключительно эффективным инструментом для управления большим количеством контейнеров в рабочей среде. По сути, концепции, используемые в этой системе, появились задолго до нее самой. Несмотря на то что Kubernetes — это относительно молодой проект, он был основан на прочном фундаменте.

Кто управляет Kubernetes сегодня

Система Kubernetes больше не находится под управлением Google, поскольку 29 августа 2018 года компания передала операционный контроль над проектом некоммерческой организации *Cloud Native Computing Foundation (CNCF)*, цель которой состоит в поддержке и развитии открытой экосистемы облачных технологий.

Google является одним из учредителей CNCF наряду с такими компаниями, как Cisco, Red Hat и Intel. Исходный код Kubernetes размещен на GitHub и является одним из наиболее активно развивающихся проектов на этой платформе. Этот код распространяется на условиях лицензии Apache License версии 2.0, то есть с минимальными ограничениями. Использование Kubernetes не требует оплаты, а если вы владеете языком Go, то можете даже внести свой вклад в развитие этой системы.

Место Kubernetes в современном ландшафте

В сфере оркестрации контейнеров у Kubernetes существуют различные альтернативы — как в виде решений с открытым исходным кодом, так и в виде платформозависимых систем. Среди самых заметных ее конкурентов можно выделить:

- Apache Mesos;
- HashiCorp Nomad;
- Docker Swarm;
- Amazon ECS.

У каждого из этих оркестраторов есть свои сильные и слабые стороны, однако Kubernetes остается наиболее популярным и широко используемым решением в данной сфере.

Kubernetes одержал победу в гонке за популярность и стал стандартом для развертывания контейнерных приложений в рабочей среде. Стремительный рост востребованности данной технологии в ИТ-отрасли побудил облачных провайдеров включить собственные Kubernetes-решения в перечень своих услуг. Благодаря этому сегодня Kubernetes поддерживается практически повсеместно.

Ниже приведены примеры сервисов, основанных на Kubernetes, которые позволяют развернуть кластер, сделав всего несколько щелчков кнопкой мыши:

- Google Kubernetes Engine (GKE) на платформе Google Cloud Platform;
- Elastic Kubernetes Service (Amazon EKS);
- Azure Kubernetes Service на платформе Microsoft Azure;
- Alibaba Cloud Container Service for Kubernetes (ACK).

Дело не ограничивается только облачными сервисами — важную роль играет и рынок PaaS (Platform-as-a-Service — платформа как услуга). Компания *Red Hat* начала интегрировать Kubernetes в свою контейнерную платформу OpenShift в 2015 году после выхода версии OpenShift 3. Это привело к значительному изменению архитектуры OpenShift: система перешла от своей исходной модели к оркестрации контейнеров на базе Kubernetes, предоставив пользователям расширенные возможности управления контейнерами и полный набор корпоративных инструментов для создания, развертывания и администрирования контейнеров непосредственно поверх Kubernetes. Кроме того, появились и другие проекты, например Rancher, разработанный в качестве *дистрибутива Kubernetes* и предлагающий полный набор инструментов, совместимых с этим оркестратором, а также *Knative*, предназначенный для управления бессерверными рабочими нагрузками с использованием Kubernetes.

Платформа AWS предлагает два сервиса для оркестрации контейнеров: Amazon ECS и Amazon EKS. Второй появился позднее и представляет собой полноценную реализацию Kubernetes в инфраструктуре AWS. Эти сервисы имеют схожие названия, но являются совершенно разными решениями.

Дальнейшее развитие Kubernetes

Система Kubernetes не ограничивается контейнерами — она развивается, поддерживая все более широкий спектр рабочих нагрузок. Проект KubeVirt позволяет ей работать с виртуальными машинами, а интеграция с такими ИИ/МО-фреймворками, как TensorFlow, создает возможности для оркестрации задач машинного обучения. Благодаря своей беспрецедентной гибкости в будущем Kubernetes может стать универсальной платформой для управления самыми разными рабочими нагрузками — от контейнеров и виртуальных машин до рабочих процессов ИИ/МО.

Сегодня освоение Kubernetes является одним из самых разумных решений для всех, кто занимается управлением облачными приложениями в рабочей среде. Kubernetes быстро развивается, и нет никаких оснований полагать, что в ближайшем будущем эта тенденция прекратится.

Овладев этим инструментом, вы приобретете одну из самых востребованных компетенций на современном ИТ-рынке. Надеемся, что смогли вас в этом убедить!

В следующем разделе вы узнаете о том, как Kubernetes упрощает процесс эксплуатации систем.

Задачи, решаемые системой Kubernetes

Kubernetes отлично подходит для команд DevOps, потому что является мощной платформой для оркестрации контейнеров, позволяющей управлять развертыванием, масштабированием и сетевыми взаимодействиями контейнеризованных приложений. Контейнеры представляют собой легковесные пакеты, содержащие приложение вместе со всеми его зависимостями, что ускоряет и повышает надежность развертывания в различных средах. Для использования Kubernetes есть несколько важных причин.

- *Автоматизация*. Kubernetes автоматизирует множество рутинных задач, связанных с развертыванием и управлением контейнеризованными приложениями, позволяя разработчикам сосредоточиться на инновациях.
- *Масштабируемость*. Kubernetes позволяет легко увеличивать или уменьшать количество экземпляров приложения в зависимости от уровня нагрузки, обеспечивая оптимальное использование ресурсов.
- *Согласованность*. Kubernetes гарантирует единообразное развертывание приложения в различных средах, снижая риск возникновения ошибок конфигурации и упрощая процесс доставки ПО.

- *Гибкость*. Система Kubernetes совместима со множеством инструментов и технологий, которыми обычно пользуются команды DevOps, что облегчает ее интеграцию в существующие рабочие процессы.

Очевидно, что запуск контейнеров на локальной машине или в среде разработки не требует такого же уровня планирования, как их развертывание на удаленных серверах, обслуживающих миллионы пользователей. При запуске контейнеров в рабочей среде возникают следующие специфические задачи, и Kubernetes является отличным инструментом для их решения:

- обеспечение высокой доступности;
- управление релизами и развертыванием контейнеров;
- автоматическое масштабирование контейнеров;
- сетевая изоляция;
- управление доступом на основе ролей (RBAC);
- поддержка рабочих нагрузок с сохранением состояния;
- управление ресурсами.

Обеспечение высокой доступности

Обеспечение высокой доступности — это основной принцип эксплуатации ПО в рабочей среде. Он означает, что перебои в работоспособности и доступности приложения должны быть полностью исключены. Разумеется, это утопия: перебои в работе происходят даже у крупнейших компаний. Однако к этому все равно следует стремиться. Kubernetes предлагает целый арсенал средств, которые обеспечивают высокую доступность контейнеров, реплицируя их на нескольких хостах и регулярно проверяя их работоспособность.

При развертывании контейнеров доступность приложения напрямую зависит от их состояния. Представьте, что по какой-то причине контейнер с одним из ваших микросервисов стал недоступен. Используя только систему Docker, вы не сможете гарантировать, что этот контейнер будет остановлен и воссоздан для восстановления работы сервиса[1]. С Kubernetes это возможно: данная система позволяет проектировать приложения, способные к самовосстановлению за счет автоматического выполнения таких операций, как проверка работоспособности и замена контейнеров.

Если одна из машин в вашем кластере выйдет из строя, все контейнеры, запущенные на ней, перестанут работать. Система Kubernetes обнаружит это и немедленно перезапустит данные контейнеры на другой машине. Это обеспечит не только высокую доступность ваших приложений, но и их устойчивость к сбоям.

---

[1] Встроенный режим Swarm все же позволяет это реализовать.

Управление релизами и развертыванием контейнеров

Управление процессом развертывания — это еще одна из специфических задач, эффективно решаемых системой Kubernetes. Развертывание предполагает обновление приложения в рабочей среде, при котором старая версия конкретного микросервиса заменяется новой.

Развертывание в рабочей среде осложняется необходимостью обновлять контейнеры, обрабатывающие запросы от конечных пользователей. Некорректное выполнение этого процесса может привести к таким серьезным последствиям, как нестабильная работа или недоступность приложения. Именно поэтому вам необходимо иметь возможность отменить изменения, выполнив откат к предыдущей версии приложения. Самое главное — осуществить развертывание максимально незаметно для пользователя и с минимальными перебоями в работе.

Процесс выпуска новой версии приложения состоит из нескольких этапов.

1. Обновление информации о приложении в файле Dockerfile или Containerfile (при необходимости).
2. Сборка нового Docker-образа контейнера с последней версией приложения.
3. Отправка нового образа в реестр контейнеров.
4. Загрузка нового образа контейнера из реестра в промежуточную, тестовую или рабочую среду (на Docker-хосты).
5. Остановка и удаление контейнера с предыдущей версией приложения из системы.
6. Запуск нового контейнера с последней версией приложения в промежуточной, тестовой или рабочей среде.

На рис. 1.9 представлена схема, иллюстрирующая процесс управления контейнерами (учтите, что на ней изображен идеализированный сценарий, так как в реальных условиях для разных сред могут использоваться разные изолированные реестры контейнеров).

Процесс сборки контейнера не имеет никакого отношения к системе Kubernetes — он относится исключительно к управлению образами контейнеров. Kubernetes вступает в игру только на этапе развертывания новых контейнеров на основе уже собранного образа.

Без использования Kubernetes вам пришлось бы вручную выполнить все операции, включая `docker pull`, `docker stop`, `docker delete` и `docker run`, для развертывания новой версии контейнера на конкретной машине, а затем повторить эти действия на каждом сервере с запущенной копией контейнера. Такой подход вполне работоспособен, но крайне утомителен из-за отсутствия автоматизации.

Однако хорошая новость заключается в том, что система Kubernetes способна сделать все это за вас.

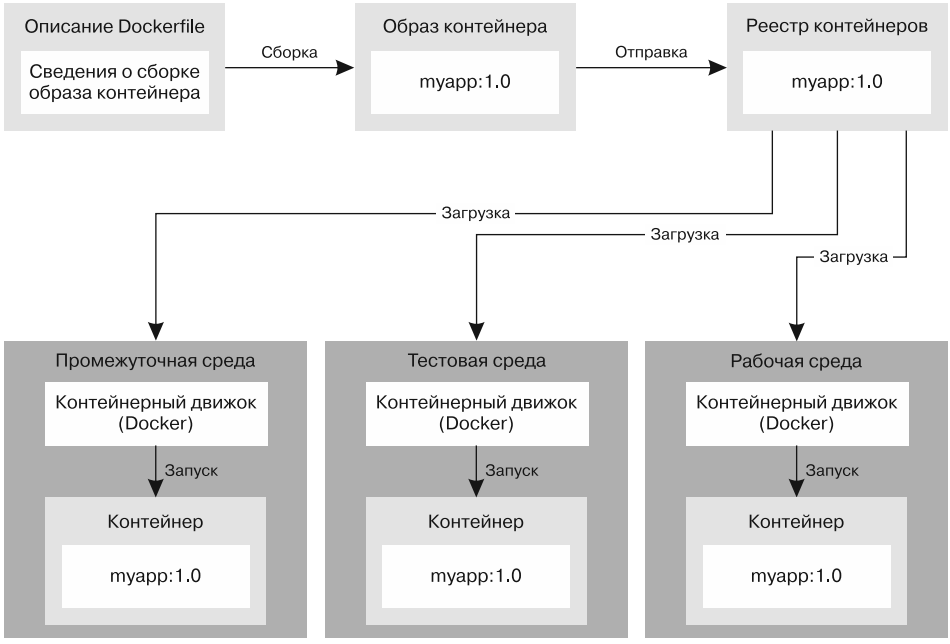

Рис. 1.9. Высокоуровневая схема процесса управления контейнерами

Она предусматривает средства управления развертываниями и откатами контейнеров, способные значительно упростить вам жизнь. Например, чтобы обновить контейнеры на всех машинах, достаточно одной команды:

```
$ kubectl set image deploy/myapp myapp_container=myapp:1.0.0
```

В реальном кластере Kubernetes эта команда обновит контейнер `myapp_container`, запущенный в рамках объекта развертывания `myapp`, до версии с тегом `1.0.0` на каждой машине, где запущен данный контейнер.

При этом неважно, идет ли речь об одном контейнере на одной машине или о миллионах контейнеров, запущенных в разных дата-центрах, — команда будет работать одинаково, обеспечивая высокую доступность приложения.

Помните о том, что ваша главная цель всегда состоит в поддержании высокой доступности: процесс развертывания не должен приводить к программным сбоям или прерыванию работы сервиса. Система Kubernetes по умолчанию поддерживает несколько вариантов стратегий развертывания (к примеру, последовательное обновление (rolling updates)), направленных на предотвращение перебоев в работе сервиса.

Кроме того, Kubernetes хранит в памяти все ревизии конкретного развертывания (deployment) и позволяет выполнить откат к предыдущей версии с помощью всего одной команды. Это исключительно мощный инструмент, который дает возможность обновить весь кластер контейнеров Docker путем выполнения единственной команды.

Автоматическое масштабирование контейнеров

Масштабирование — это еще одна из специфических задач, возникающих в рабочей среде, решение которой существенно упростило появление таких публичных облачных платформ, как *Amazon Web Services (AWS)* и *Google Cloud Platform (GCP)*. Масштабирование позволяет адаптировать уровень используемых вычислительных ресурсов под текущие потребности с целью обеспечения высокой доступности приложения и балансировки нагрузки. Помните, что основная задача всегда заключается в предотвращении сбоев.

Когда рабочие серверы сталкиваются с резким всплеском трафика и один из контейнеров перестает справляться с нагрузкой, возникает потребность в эффективном способе масштабирования. Для этого существует два основных метода.

- *Вертикальное масштабирование* позволяет контейнеру задействовать дополнительные вычислительные ресурсы, предоставляемые хост-машиной.
- *Горизонтальное масштабирование* предполагает создание копий контейнера на той же или на другой машине с последующим распределением трафика между ними.

Система Docker не способна в полной мере решить эту задачу самостоятельно. Однако при управлении контейнерами Docker с помощью Kubernetes это становится возможным (рис. 1.10).

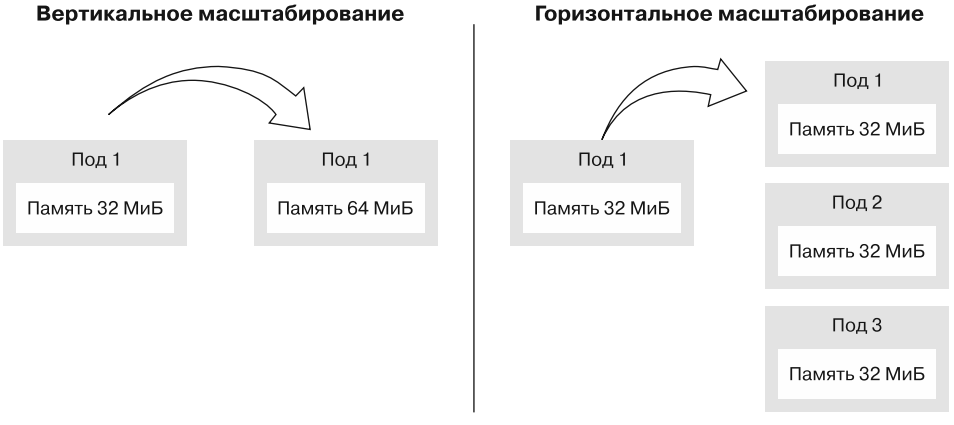

Рис. 1.10. Вертикальное и горизонтальное масштабирование подов

Система Kubernetes способна автоматически выполнять как вертикальное, так и горизонтальное масштабирование. Это достигается либо за счет предоставления контейнерам доступа к большему объему вычислительных ресурсов хост-машины, либо путем создания дополнительных контейнеров, которые можно развернуть на том же самом или на другом узле кластера. А если ваш кластер Kubernetes не сможет разместить новые контейнеры из-за загруженности всех узлов, то данная система сможет автоматически запустить дополнительные виртуальные машины, взаимодействуя с вашим облачным провайдером с помощью *инструмента автомасштабирования кластера* (Cluster Autoscaler).

Инструмент автомасштабирования кластера работает только в том случае, если кластер Kubernetes развернут на платформе поддерживаемого облачного провайдера (в частном или публичном облаке).

Достичь этих целей без использования оркестратора контейнеров невозможно, поскольку вы не можете выполнять все необходимые операции вручную. Чтобы сделать свои приложения способными к самовосстановлению, устойчивыми к сбоям и высокодоступными, вам следует мыслить в духе культуры DevOps, стремиться к гибкости и автоматизировать рабочие процессы.

Помимо масштабирования кластера и увеличения числа контейнеров, вам также необходимо уметь сокращать их количество при снижении нагрузки, подстраивая объем потребляемых ресурсов под текущий уровень. В этом Kubernetes тоже может помочь.

Сетевая изоляция

Когда количество пользователей исчисляется миллионами, обеспечение безопасного взаимодействия между контейнерами приобретает первостепенное значение. Традиционные методы часто требуют сложной ручной настройки, и именно здесь Kubernetes демонстрирует свои сильные стороны.

- *Сетевое взаимодействие подов*. Kubernetes создает виртуальную оверлейную сеть для обеспечения связи между подами. По умолчанию контейнеры внутри одного пода могут обмениваться данными напрямую, тогда как контейнеры в разных подах изолированы друг от друга. Это предотвращает нежелательные взаимодействия между контейнерами и повышает уровень безопасности.
- *Сетевые политики*. Kubernetes позволяет задавать детализированные сетевые политики, которые дополнительно ограничивают взаимодействие между подами. Мы можем определить правила распределения входящего (ingress) и исходящего (egress) трафика для подов, чтобы они обращались только к тем ресурсам, которые им необходимы. Такой подход упрощает конфигурацию сети и повышает безопасность в рабочей среде.

Управление доступом на основе ролей (RBAC)

При наличии большого количества пользователей крайне важно грамотно управлять доступом к ресурсам контейнеров в рабочей среде. Для этого в Kubernetes предусмотрены следующие механизмы.

- *Пользовательские роли*. Система Kubernetes позволяет определить пользовательские роли, задающие разрешения на доступ к ресурсам контейнеров и управление ими. Эти роли могут назначаться как отдельным пользователям, так и их группам, что обеспечивает точный контроль над тем, кто может выполнять те или иные действия (например, просмотр логов пода или развертывание новых контейнеров).
- *Учетные записи сервисов*. Kubernetes использует учетные записи сервисов для назначения идентификаторов подам, работающим в кластере. Этим учетным записям могут назначаться роли, предоставляющие подам доступ только к тем ресурсам, которые им необходимы.

Такой многоуровневый подход, сочетающий пользовательские роли с учетными записями сервисов, усиливает безопасность и улучшает управляемость ресурсами в рабочей среде.

Поддержка рабочих нагрузок с сохранением состояния

Хотя контейнеры обычно не сохраняют состояние (stateless), то есть их данные удаляются после остановки, некоторые приложения нуждаются в постоянном хранилище. Для управления рабочими нагрузками с сохранением состояния (stateful) в Kubernetes предусмотрены механизмы *Persistent Volume* (PV, постоянный том) и *Persistent Volume Claim* (PVC, заявка на выделение постоянного тома). PV — это ресурс постоянного хранилища, создаваемый администратором (например, локальная директория хоста или облачное хранилище). Приложения получают доступ к такому хранилищу с помощью заявки PVC. Эта абстракция отделяет управление хранилищем от приложения, позволяя контейнерам использовать постоянное хранилище, не заботясь о его реализации.

Управление ресурсами

В рабочей среде важно эффективно распределять ресурсы контейнеров, чтобы оптимизировать производительность и избегать узких мест. Для этого система Kubernetes предоставляет следующие механизмы.

- *Квоты ресурсов*. Позволяют задавать ограничения на использование процессорного времени, памяти и других ресурсов как для всего пространства имен, так и для отдельных подов. Это гарантирует справедливое распреде-

ление ресурсов и предотвращает ситуацию, в которой один под потребляет слишком много ресурсов, нарушая работу других приложений.

- *Лимиты и запросы ресурсов*. При определении объекта развертывания можно указать минимальное количество ресурсов, гарантированно выделяемое контейнеру (запрос, request), и максимальное количество ресурсов, которое контейнер может потребить (лимит, limit). Это обеспечивает приложению достаточный объем ресурсов для корректной работы и предотвращает их неконтролируемое потребление.

Обо всех этих возможностях мы подробно поговорим в следующих главах.

Когда Kubernetes не является оптимальным решением

У Kubernetes есть несомненные преимущества, однако применение этой системы уместно не во всех случаях. Ниже перечислены ситуации, в которых целесообразнее использовать другие решения.

- *Архитектура без контейнеров*. Если вы не используете контейнеры, Kubernetes вам не нужен.
- *Очень малое число микросервисов или приложений*. Система Kubernetes особенно эффективна при управлении большим количеством контейнеров. Если ваше приложение состоит всего из двух-трех микросервисов, возможно, вам будет достаточно более простого оркестратора.

Резюме

В этой вступительной главе мы обсудили монолиты, микросервисы, контейнеры Docker, облачные вычисления, систему Kubernetes и историю ее появления. К этому моменту у вас должно было сложиться целостное представление о том, как Kubernetes можно использовать для управления контейнерами в рабочей среде. Теперь вы знаете, зачем была создана эта система и как она стала востребованным инструментом для оркестрации контейнеров.

В следующей главе рассмотрим процесс, который Kubernetes использует для запуска контейнера Docker. Вы увидите, что этой системе можно передавать команды, которые она будет интерпретировать как инструкции для запуска контейнеров. Мы также подробно разберем все компоненты системы Kubernetes и обсудим их роль в работе кластера. В частности, мы рассмотрим архитектуру Kubernetes, уделив внимание различиям между мастер-узлами, рабочими узлами и компонентами управляющего слоя.

Дополнительные источники информации

- Документация по Kubernetes: https://kubernetes.io/docs/home/.
- Документация по Podman: https://docs.podman.io/en/latest/.
- Документация по Docker: https://docs.docker.com/.
- Среда Kata Containers: https://katacontainers.io/.
- Инструмент kaniko: https://github.com/GoogleContainerTools/kaniko.
- Инструмент Buildah: https://buildah.io.
- Проект KubeVirt: https://kubevirt.io.
- Проект Knative: https://knative.dev/docs/.
- Kubernetes: The Documentary (часть 1): https://rutube.ru/video/e1ea340b5e6c1ee1f2861c79aae7c6ba/.
- Kubernetes: The Documentary (часть 2): https://rutube.ru/video/c81cc0e942301b242bf4289a811e55e8/.
- Клейтон Коулман об истории Kubernetes: https://rutube.ru/video/d638426c38701d4a870dca54f801697d/.

2 Архитектура Kubernetes: от образов контейнеров до работающих подов

В предыдущей главе мы заложили основу для понимания функциональных возможностей Kubernetes. Теперь, когда вы знаете, как Kubernetes помогает управлять кластерами машин, на которых запускаются контейнеризованные микросервисы, мы можем перейти к рассмотрению более глубоких технических тем. В этой главе мы поговорим о том, как Kubernetes управляет контейнерами, распределенными между разными машинами. После ее изучения вы будете иметь более полное понимание устройства кластера Kubernetes. В частности, вы узнаете, из каких компонентов состоит Kubernetes и какова роль каждого из них в управлении контейнерами.

Система Kubernetes состоит из нескольких распределенных компонентов, каждый из которых играет важную роль в обеспечении работы контейнеров. Чтобы понять назначение каждого компонента, мы проследим весь жизненный цикл контейнера — от момента его создания до запуска на одной из машин, входящих в кластер Kubernetes.

Технические требования

Для работы с этой главой необходимы:

- базовые знания операционной системы Linux и умение выполнять в ней простейшие операции;
- одна или несколько машин с установленной Linux.

Примеры кода, приведенные в этой главе, были протестированы в дистрибутиве Fedora Workstation и доступны в репозитории GitHub по адресу https://github.com/PacktPublishing/The-Kubernetes-Bible-Second-Edition/tree/main/Chapter02.

Происхождение названия Kubernetes

Название Kubernetes происходит от греческого слова *kubernētēs*, которое переводится как «кормчий» или «рулевой». Этот морской термин обозначает человека, умело управляющего судном и прокладывающего его курс. Выбор названия отражает основную роль платформы — направлять и координировать процесс развертывания контейнеризованных приложений, подобно тому как кормчий ведет корабль через сложный цифровой океан.

Помимо официального названия, в сообществе Kubernetes часто используется сокращение *K8s*. Оно образовано с помощью особого приема составления аббревиатур, когда вместо самих букв в середине слова указывается их количество: между K и s в слове Kubernetes насчитывается как раз восемь букв. Такая сокращенная форма названия не только упрощает устную и письменную коммуникацию, но и придает общению внутри экосистемы Kubernetes элемент неформальности.

Различие между управляющими и вычислительными узлами

Для работы Kubernetes необходимы машины под управлением Linux, которые в этой системе называются узлами (node). Узел может быть как физическим сервером, так и виртуальной машиной, предоставленной облачным провайдером, например экземпляром EC2. В Kubernetes выделяют два типа узлов:

- управляющие (или мастер-узлы);
- вычислительные (или рабочие узлы).

Мастер-узлы и рабочие узлы

В некоторых источниках встречаются термины «мастер-узел» (master node) и «рабочий узел» (worker node), которые ранее использовались для описания традиционной иерархической схемы распределения ролей в распределенной системе. В такой схеме мастер-узел управлял процессом и назначал задачи рабочим узлам. Поскольку эти термины могут иметь неуместные исторические и культурные коннотации, сообщество Kubernetes заменило их терминами «управляющий узел» (control plane node или controller node) и «вычислительный узел» (compute node). Под управляющим узлом понимается набор компонентов, отвечающих за управление общим состоянием кластера, а под вычислительным — отдельная машина в кластере, которая выполняет назначенные задачи или рабочие нагрузки приложений. Управляющий слой (control plane) отвечает за поддержание состояния кластера Kubernetes, а вычислительные узлы — за запуск контейнеров с вашими приложениями.

Контейнеры Linux и Windows

Система Kubernetes позволяет использовать узлы на базе Windows для запуска контейнеров, предназначенных для этой операционной системы. Кластер может содержать как Linux-, так и Windows-машины, однако запуск контейнера Windows на вычислительном узле под управлением Linux (и наоборот) невозможен. Правильное соотношение Linux- и Windows-машин в кластере позволяет достичь оптимальной производительности.

В следующих разделах этой главы мы рассмотрим различные компоненты Kubernetes и их назначение.

Компоненты Kubernetes

Kubernetes функционирует как распределенное приложение. Это не единый монолитный программный продукт, выпускаемый в виде одной сборки для установки на выделенную машину, а совокупность небольших проектов, написанных на языке Go, которые образуют целостную систему.

Чтобы развернуть полностью работоспособный кластер Kubernetes, необходимо установить и настроить каждый из этих компонентов по отдельности, обеспечив их беспрепятственное взаимодействие. После выполнения этих шагов можно приступать к запуску контейнеров с помощью оркестратора Kubernetes.

Для целей разработки и локального тестирования ПО все компоненты Kubernetes можно установить на одну машину. Однако в рабочей среде для обеспечения высокой доступности, балансировки нагрузки, организации распределенных вычислений и масштабирования компоненты следует распределять между разными хостами. Такой подход дает два преимущества.

- Кластер становится высокодоступным и устойчивым к сбоям.
- Кластер легче масштабировать, поскольку каждый компонент имеет собственный жизненный цикл и может масштабироваться независимо от остальных.

В итоге отказ одного из серверов приведет к недоступности лишь небольшой части кластера, а добавление новых серверов не будет вызывать никаких затруднений.

Каждый компонент Kubernetes выполняет строго определенную функцию. Чтобы понять принцип работы системы в целом, важно знать назначение каждого компонента и то, как он взаимодействует с другими.

В зависимости от своей роли компонент развертывается либо на управляющем, либо на вычислительном узле. Одни компоненты отвечают за поддержание состояния всего кластера и управление им, а другие — за запуск контейнеров приложений и взаимодействуют напрямую со средой выполнения контейнеров (например, `containerd` или Docker). Таким образом, все компоненты Kubernetes

можно разделить на две группы: компоненты управляющего слоя и компоненты вычислительных узлов.

Вы не запускаете контейнеры самостоятельно, поэтому не взаимодействуете непосредственно с вычислительными узлами. Вместо этого ваши инструкции отправляются в управляющий слой, который делегирует задачу создания и обслуживания контейнеров вычислительным узлам от вашего имени (рис. 2.1).

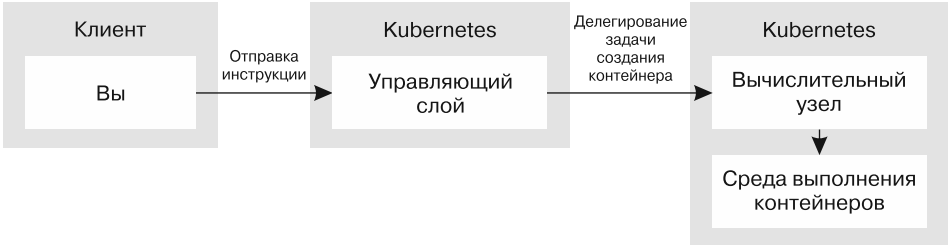

Рис. 2.1. Типовой рабочий процесс Kubernetes

Благодаря распределенной архитектуре Kubernetes компоненты управляющего слоя могут располагаться на нескольких машинах. Существует два подхода к их развертыванию.

- Все компоненты управляющего слоя можно запустить на одной машине либо распределить по разным. Для обеспечения максимальной отказоустойчивости рекомендуется размещать их на отдельных машинах. При этом компоненты Kubernetes должны иметь возможность взаимодействовать между собой, и их установка на разных хостах вполне допускает такое взаимодействие.
- С вычислительными (или рабочими) узлами все намного проще. Их компоненты устанавливаются на стандартную машину с поддерживаемой средой выполнения контейнеров. Эти компоненты взаимодействуют с локальным контейнерным движком, установленным на данной машине, и запускают контейнеры на основе инструкций, поступающих от компонентов управляющего слоя. Для увеличения вычислительных ресурсов кластера достаточно добавить новые вычислительные узлы и присоединить их к кластеру, создав место для дополнительных контейнеров.

Распределение компонентов управляющего слоя и вычислительных узлов по разным машинам делает кластер высокодоступным и масштабируемым. Система Kubernetes изначально проектировалась с учетом принципов облачных технологий: ее компоненты не используют локальные хранилища данных, легко масштабируются и могут быть распределены по разным хостам. Основная идея заключается в том, чтобы избежать появления единой точки отказа, которая возникает при размещении всех компонентов на одном хосте.

Ниже показана упрощенная схема полнофункционального кластера Kubernetes (рис. 2.2). В этой главе мы разберем каждый из перечисленных на ней компонентов и их функции. В данном примере все компоненты управляющего слоя установлены на одной машине.

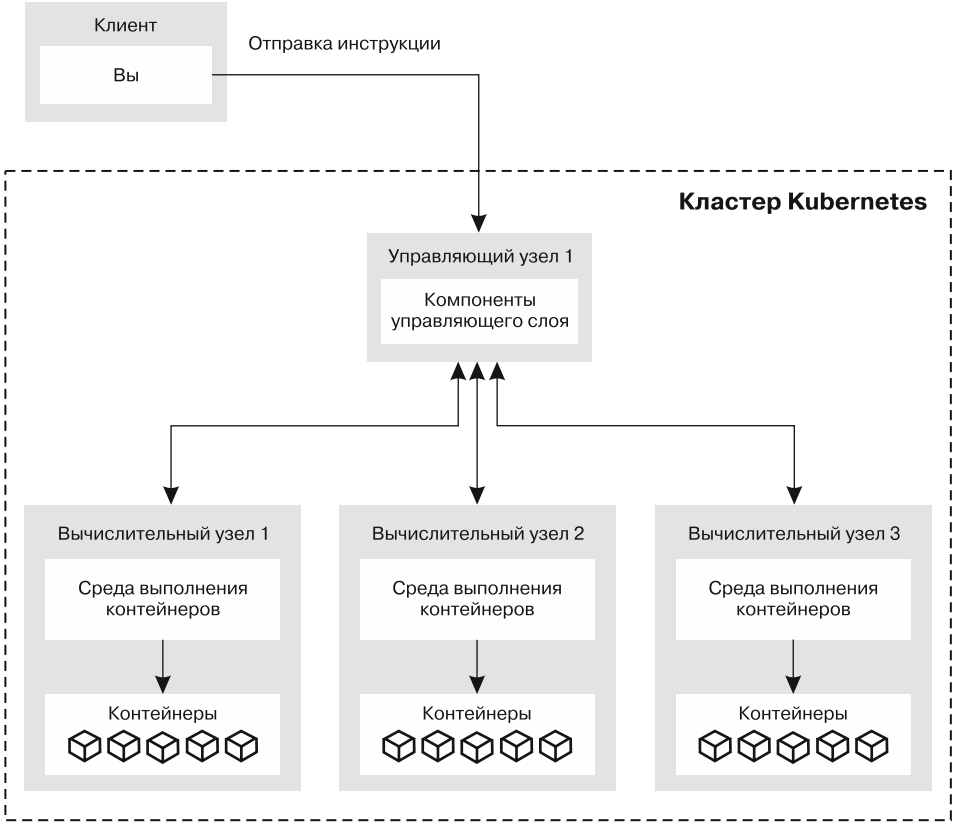

Рис. 2.2. Полнофункциональный кластер Kubernetes с одним управляющим узлом и тремя вычислительными узлами

На этой схеме изображен кластер Kubernetes, состоящий из четырех узлов и содержащий все необходимые компоненты.

Помните, что система Kubernetes может быть адаптирована под конкретную среду. Например, при использовании Kubernetes как части дистрибутивов Amazon EKS или Red Hat OpenShift в ней могут появиться дополнительные компоненты или измениться поведение стандартных. В книге обсуждается «чистый» Kubernetes. Компоненты, описанные в этой главе, являются базовыми, то есть присутствуют в любой установке данной системы.

На следующей схеме (рис. 2.3) перечислены основные компоненты кластера Kubernetes.

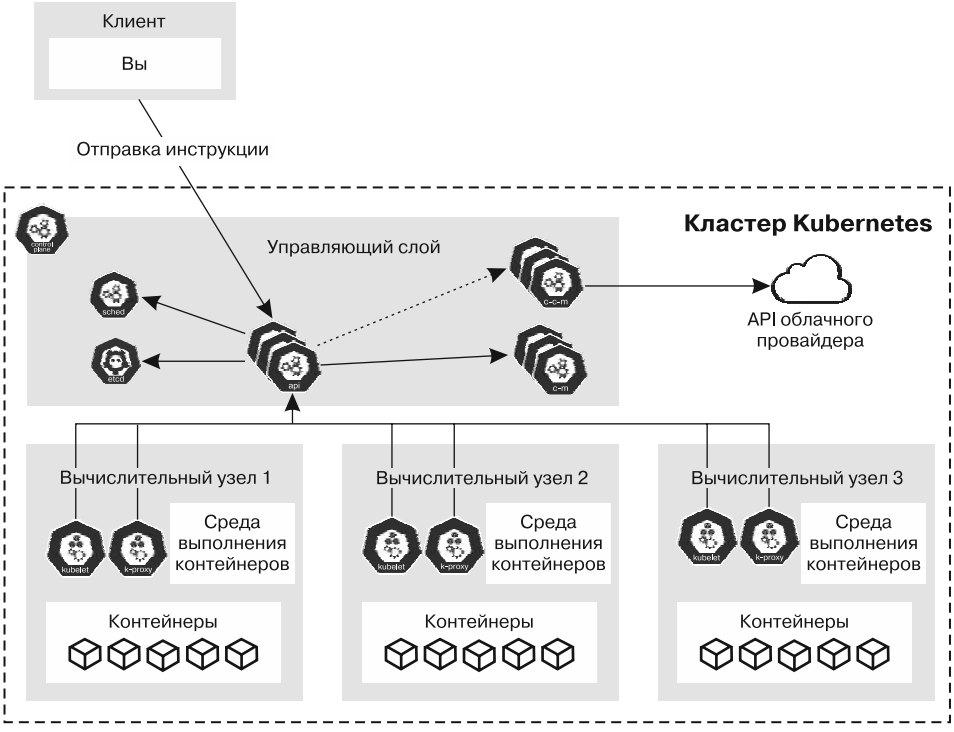

Рис. 2.3. Компоненты кластера Kubernetes (источник изображения: https://kubernetes.io/docs/concepts/overview/components)

Возможно, вы заметили, что названия большинства этих компонентов начинаются с префикса `kube`: это компоненты, входящие в проект Kubernetes. Компоненты `etcd` и `Container Engine` представляют собой внешние зависимости, которые не являются частью проекта Kubernetes, но необходимы для его работы:

- `etcd` — это стороннее хранилище данных, используемое в проекте Kubernetes. Не беспокойтесь, для работы с данной системой вам не придется глубоко осваивать этот инструмент;
- `Container Engine` — это сторонний контейнерный движок.

Не переживайте, вам не придется устанавливать и настраивать эти компоненты вручную. Сегодня практически никто не управляет ими самостоятельно, и, более того, получить рабочий кластер Kubernetes без ручной установки компонентов не составляет никакого труда.

В целях разработки можно использовать `minikube` — инструмент, позволяющий запустить на локальной машине кластер Kubernetes, состоящий из одного узла. Это легковесное и простое в использовании решение для тестирования и разработки Kubernetes-приложений, не требующее развертывания полноразмерного кластера. Однако `minikube` категорически не рекомендуется применять в рабочей среде.

Для развертывания приложений в рабочей среде можно использовать интегрированный и масштабируемый кластер Kubernetes, предоставляемый такими облачными сервисами, как Amazon EKS или Google GKE. В средах, не имеющих доступа к облаку, можно воспользоваться утилитой для установки Kubernetes `kubeadm`.

В качестве хорошего обучающего материала мы можем порекомендовать курс Келси Хайтауэра *Kubernetes the Hard Way*. В нем подробно описан процесс ручной установки Kubernetes, управления инфраструктурой открытых ключей (public key infrastructure, PKI), налаживания сетевых взаимодействий и предоставления вычислительных ресурсов на «чистых» Linux-машинах в Google Cloud. Хотя новичкам этот курс может показаться сложным, мы советуем ознакомиться с ним для более полного понимания внутреннего устройства Kubernetes. При этом следует помнить, что развертывание и сопровождение кластера производственного уровня по методике *Kubernetes the Hard Way* — это трудоемкий и сложный процесс и полученный результат крайне не рекомендуется использовать в рабочей среде. Этот учебный курс получил широкую известность, поэтому во Всемирной паутине можно встретить множество ссылок на него.

В следующих подразделах мы рассмотрим компоненты управляющего слоя и вычислительных узлов Kubernetes.

Компоненты управляющего слоя

Эти компоненты отвечают за поддержание состояния кластера и должны быть установлены на управляющем узле. Именно они хранят сведения о контейнерах, запущенных в кластере Kubernetes, и о количестве машин, входящих в его состав.

Как администратор вы взаимодействуете с Kubernetes через компоненты управляющего слоя. Вот основные из них:

- `kube-apiserver`;
- `etcd`;
- `kube-scheduler`;
- `kube-controller-manager`;
- `cloud-controller-manager`.

Компоненты вычислительных узлов

Эти компоненты взаимодействуют со средой выполнения контейнеров, запуская их в соответствии с инструкциями, полученными от компонентов управляющего слоя. Компоненты вычислительных узлов должны устанавливаться на Linux-машину, на которой работает поддерживаемая среда выполнения контейнеров. Взаимодействовать с ними напрямую вам не придется. В кластере Kubernetes может насчитываться от сотен до тысяч вычислительных узлов. Основными их компонентами являются:

- `kubelet`;
- `kube-proxy`;
- среда выполнения контейнеров.

Дополнения

Дополнения (add-ons) используют такие ресурсы Kubernetes, как DaemonSet, Deployment и т. д., для реализации функций кластера. Поскольку эти функции работают на уровне всего кластера, ресурсы дополнений, относящиеся к определенному пространству имен, размещаются в пространстве имен `kube-system`. Ниже перечислены дополнительные компоненты, которые чаще всего встречаются в кластерах Kubernetes:

- DNS;
- пользовательский веб-интерфейс (dashboard);
- мониторинг использования ресурсов контейнерами;
- логирование на уровне кластера;
- сетевые плагины.

Управляющий слой в готовых кластерах Kubernetes

В отличие от самостоятельного администрирования кластеров Kubernetes, при использовании таких облачных сервисов, как Amazon EKS и Google GKE, установку и настройку большинства компонентов управляющего слоя выполняет сам провайдер. Пользователю предоставляется доступ к конечной точке Kubernetes (или при необходимости к конечной точке `kube-apiserver`) без раскрытия подробных сведений о физических или виртуальных машинах и настроенных балансировщиках нагрузки. Тот же принцип применяется к таким компонентам, как `kube-scheduler`, `kube-controller-manager`, `etcd` и др.

На рис. 2.4 приведен снимок экрана с информацией о кластере Kubernetes, созданном в сервисе Amazon EKS.

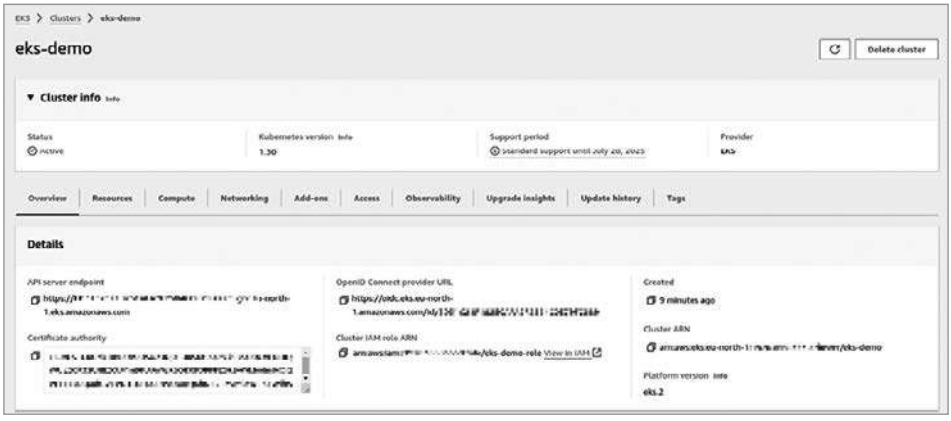

Рис. 2.4. Интерфейс с информацией о кластере Kubernetes, развернутом в Amazon EKS

Подробно о сервисах EKS, GKE и AKS мы поговорим в отдельных главах этой книги.

В следующем разделе мы обсудим компоненты управляющего слоя, отвечающие за поддержание состояния кластера.

Компоненты управляющего слоя

Далее мы рассмотрим различные компоненты управляющего слоя и их функции.

Компонент kube-apiserver

Наиболее важным компонентом Kubernetes является REST API `kube-apiserver`, который предоставляет доступ ко всем возможностям системы. Взаимодействие с Kubernetes происходит путем обращения к этому REST API — через утилиту командной строки `kubectl`, прямые API-вызовы или веб-интерфейс Kubernetes (Dashboard).

Роль компонента kube-apiserver

Компонент `kube-apiserver` входит в состав управляющего слоя Kubernetes. Он написан на языке Go, и его исходный код открыт и доступен на GitHub под лицензией Apache 2.0. Схема взаимодействия с Kubernetes довольно проста: чтобы дать системе инструкцию, вы направляете HTTP-запрос к `kube-apiserver`. Все операции, будь то создание, удаление или обновление контейнера, выполняются через соответствующую конечную точку `kube-apiserver` с использованием

нужного HTTP-метода. Таким образом, `kube-apiserver` является единственной точкой входа для всех инструкций, направляемых оркестратору Kubernetes. Прямого взаимодействия со средой выполнения контейнеров рекомендуется избегать (за исключением тех случаев, когда речь идет об устранении неполадок).

Компонент `kube-apiserver` реализован в соответствии со стандартом REST. Данный архитектурный стиль позволяет эффективно предоставлять функциональные возможности через конечные точки HTTP, доступ к которым осуществляется с использованием таких методов, как `GET`, `POST`, `PUT`, `PATCH` и `DELETE`. Комбинируя HTTP-методы и пути, можно выполнять различные операции (определяемые методом) над ресурсами (доступными по указанному пути).

Стандарт REST обеспечивает значительную гибкость, позволяющую расширять любой REST API за счет добавления новых ресурсов через создание новых путей. Обычно для управления состоянием объектов или ресурсов REST API использует хранилище данных.

Хранение данных в таком API может быть реализовано несколькими способами, включая следующие.

Хранение данных REST API в памяти:

- данные хранятся в собственной памяти API;
- однако это делает API сохраняющим состояние (stateful), что исключает возможность масштабирования.

Для хранения состояния в системе Kubernetes используется компонент etcd — открытое распределенное хранилище данных в формате «ключ — значение», предназначенное для хранения критически важной информации, необходимой для работы распределенных систем.

Использование движка базы данных:

- могут применяться полнофункциональные СУБД, такие как MariaDB или PostgreSQL;
- делегирование функции хранения данных внешнему движку делает API не сохраняющим состояние (stateless) и пригодным для горизонтального масштабирования.

Любой REST API можно легко обновить или расширить, добавив функции, выходящие за рамки его исходного назначения. Вот основные свойства REST API:

- работает на основе протокола HTTP;
- определяет набор ресурсов, идентифицируемых с помощью URL-путей;
- задает набор действий, определяемых HTTP-методами;

- выполняет действия над ресурсами на основе корректно сформированного HTTP-запроса;
- хранит состояние ресурсов в хранилище данных.

Таким образом, `kube-apiserver` — это не что иное, как REST API, лежащий в основе любого кластера Kubernetes, независимо от того, развернут он на локальной машине, в облаке или в собственной инфраструктуре организации. Этот компонент не сохраняет состояние: он хранит данные о состоянии ресурсов в хранилище `etcd`. Благодаря этому `kube-apiserver` можно горизонтально масштабировать, разворачивая его на нескольких машинах и распределяя входящие запросы с помощью балансировщика нагрузки уровня 7 без потери данных.

Поскольку протокол HTTP поддерживается практически повсеместно, взаимодействовать с кластером Kubernetes и передавать ему инструкции очень просто. Тем не менее в большинстве случаев работа с Kubernetes ведется через утилиту командной строки `kubectl` — официальный HTTP-клиент, входящий в состав проекта Kubernetes. При загрузке компонента `kube-apiserver` вы получаете скомпилированный на Go исполняемый файл, готовый к запуску на любой машине с Linux. Разработчики Kubernetes определили набор ресурсов, включаемых непосредственно в этот исполняемый файл. Поэтому в `kube-apiserver` вы обнаружите ресурсы, связанные с управлением контейнерами, сетью и вычислительными процессами в целом.

Вот некоторые из этих ресурсов:

- Pod;
- ReplicaSet;
- PersistentVolume;
- NetworkPolicy;
- Deployment.

Разумеется, этот список не является исчерпывающим. Полный перечень ресурсов Kubernetes можно найти в официальной документации, доступной по адресу https://kubernetes.io/docs/reference/kubernetes-api/.

Возможно, вас интересует, почему здесь нет *контейнерных* ресурсов. Как уже упоминалось в главе 1, для управления контейнерами система использует специальный ресурс Pod. На данном этапе можно рассматривать поды как аналоги контейнеров.

Хотя под может содержать несколько контейнеров, на практике часто встречаются поды с единственным контейнером. Использование нескольких контейнеров в одном поде будет рассмотрено в главе 5 в рамках обсуждения sidecar-контейнеров и контейнеров инициализации.

Вы подробно познакомитесь с этими ресурсами в следующих главах. Каждый из них связан с определенным URL-путем, и изменение HTTP-метода при обращении к этому пути будет приводить к различным результатам. Все соответствующие правила поведения определяются в компоненте `kube-apiserver`. Учтите, что вам не нужно разрабатывать эту логику самостоятельно — она реализована в `kube-apiserver` изначально.

После сохранения объектов Kubernetes в базе данных `etcd` другие компоненты системы смогут *преобразовывать* эти объекты в инструкции для запуска контейнеров.

Помните о том, что компонент `kube-apiserver` — это центральный узел и конечный источник истины для всего кластера Kubernetes. Все действия в этой системе так или иначе выполняются через него. Остальные компоненты, а также администраторы взаимодействуют с `kube-apiserver` по протоколу HTTP, в большинстве случаев избегая прямого обращения к компонентам кластера.

Это связано с тем, что `kube-apiserver` не только управляет состоянием кластера, но и включает в себя многочисленные механизмы аутентификации, авторизации и форматирования HTTP-ответов. Из-за сложности этих процессов прямое вмешательство в его работу крайне не рекомендуется.

Как запускать kube-apiserver

В главе 3 мы рассмотрим процесс установки и настройки кластера Kubernetes в локальной среде.

По сути, существует два способа запуска `kube-apiserver` (и других компонентов):

- запуск `kube-apiserver` в виде образа контейнера;
- загрузка, установка и запуск `kube-apiserver` с помощью unit-файла `systemd`.

Поскольку рекомендуемым методом является запуск `kube-apiserver` в виде образа контейнера, способ с `systemd` мы рассматривать не будем. В зависимости от механизма развертывания кластера Kubernetes `kube-apiserver` и другие компоненты настраиваются как контейнеры путем загрузки соответствующих образов из реестра контейнеров (например, `registry.k8s.io`).

Где запускать kube-apiserver

Будучи компонентом управляющего слоя, `kube-apiserver` должен запускаться на управляющем узле (или узлах). Важно, чтобы этот компонент был установлен на надежной машине, выделенной под задачи управляющего слоя. Если

этот критически важный элемент окажется недоступным, ваши контейнеры продолжат работать, но потеряют связь с системой Kubernetes. Фактически они превратятся в «осиротевшие» контейнеры, работающие на изолированных машинах, не находящиеся под управлением оркестратора.

Кроме того, другие компоненты Kubernetes, запущенные на остальных узлах кластера, постоянно отправляют HTTP-запросы к `kube-apiserver`, чтобы узнать текущее состояние кластера или обновить его. И чем больше вычислительных узлов в кластере, тем больше запросов поступает в `kube-apiserver`. Поэтому данный компонент должен масштабироваться пропорционально размеру кластера.

Как уже было сказано, `kube-apiserver` представляет собой компонент, не сохраняющий состояние: он не управляет состоянием кластера напрямую, а использует для этого внешнюю базу данных. Его можно масштабировать горизонтально, разместив несколько экземпляров на группе машин за балансировщиком нагрузки.

В следующем разделе мы поговорим о том, как Kubernetes хранит сведения о кластере и его ресурсах с помощью `etcd`.

Хранилище данных etcd

Мы уже говорили, что `kube-apiserver` можно масштабировать горизонтально и что для хранения состояния кластера и его параметров данный компонент использует `etcd` — открытое распределенное хранилище данных в формате «ключ — значение». Строго говоря, `etcd` не входит в состав системы Kubernetes, а является отдельным проектом, поддерживаемым сообществом `etcd-io`.

Хотя etcd является наиболее распространенным хранилищем данных для кластеров Kubernetes, некоторые дистрибутивы, например k3s, по умолчанию используют такие альтернативы, как SQLite, или даже внешние базы данных MySQL и PostgreSQL (https://docs.k3s.io/datastore).

etcd — это тоже проект с открытым исходным кодом (как и Kubernetes написанный на языке Go), доступный на GitHub (https://github.com/etcd-io/etcd) под лицензией Apache 2.0. Этот проект развивался под эгидой Cloud Native Computing Foundation (CNCF) — организации, курирующей Kubernetes. В 2018 году он был включен в программу инкубации CNCF, а в 2020-м получил статус «выпущенного» (graduated).

Всякий раз, когда вы обращаетесь к `kube-apiserver` с целью выполнения операции чтения или записи, данные считываются из хранилища `etcd` или записываются в него.

Теперь рассмотрим содержимое управляющего узла (рис. 2.5).

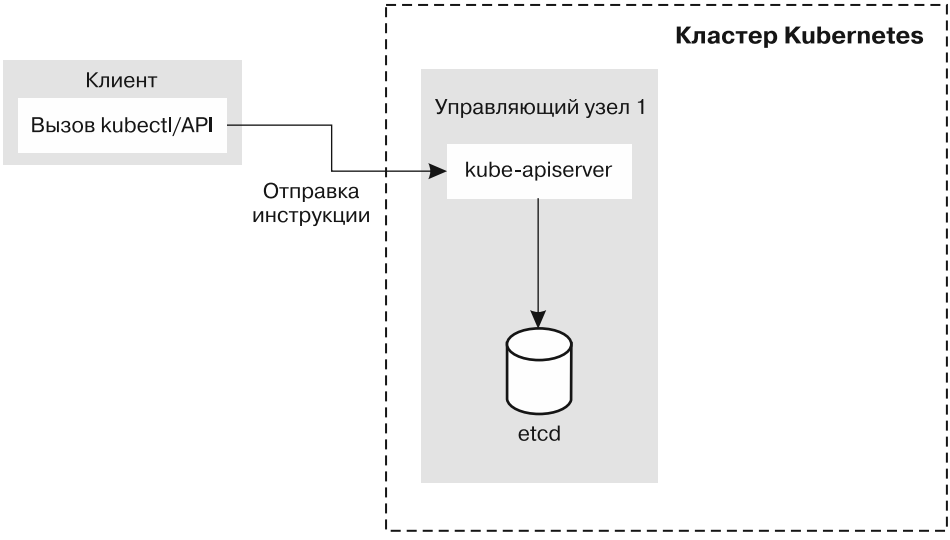

Рис. 2.5. Компонент kube-apiserver располагается перед хранилищем etcd и выполняет роль прокси; только kube-apiserver может читать данные из etcd и записывать их в него

Компонент `etcd` можно назвать «сердцем» кластера. Если данные в `etcd` будут потеряны, кластер Kubernetes перестанет работать. Этот компонент даже важнее `kube-apiserver`: если `kube-apiserver` выйдет из строя, его можно перезапустить, но если данные `etcd` будут повреждены или безвозвратно удалены, кластер будет полностью утрачен.

К счастью, для использования Kubernetes вам не требуется глубокое знание `etcd`. Более того, настоятельно не рекомендуется изменять его содержимое без четкого понимания последствий: некорректная операция может повредить данные в `etcd` и, как следствие, нарушить состояние кластера.

Помните, что согласно принципам архитектуры Kubernetes для чтения или записи данных в `etcd` каждый компонент должен обращаться к `kube-apiserver`. Это связано с тем, что с технической точки зрения утилита `kubectl` проходит аутентификацию в `kube-apiserver` с использованием клиентского TLS-сертификата, который есть только у `kube-apiserver`. Таким образом, это единственный компонент Kubernetes, имеющий право читать и записывать данные в `etcd`. Это принципиально важный момент в архитектуре Kubernetes: остальные компоненты не могут напрямую читать или записывать данные в `etcd`, не обращаясь к конечным точкам `kube-apiserver` по протоколу HTTP.

Следует отметить, что etcd тоже реализован как REST API. По умолчанию он прослушивает порт 2379.

Рассмотрим простой пример команды `kubectl`:

```
$ kubectl run nginx --restart Never --image nginx
```

При ее выполнении утилита `kubectl` сформирует HTTP-запрос `POST`, адресованный компоненту `kube-apiserver`, указанному в файле `kubeconfig`. `kube-apiserver` внесет новую запись в `etcd`, которая будет постоянно храниться на диске.

В этот момент состояние Kubernetes изменяется: теперь уже другие компоненты системы должны будут привести фактическое состояние кластера в соответствие с желаемым (тем, которое хранится в `etcd`).

В отличие от хранилищ Redis или Memcached `etcd` не работает исключительно в оперативной памяти. Его данные хранятся на диске, поэтому не теряются при перезагрузке машины.

Где запускать etcd

В самостоятельно администрируемом кластере Kubernetes компонент `etcd` можно запускать либо в контейнере, либо в виде сервиса, управляемого с помощью unit-файла `systemd`. Этот компонент изначально поддерживает горизонтальное масштабирование путем распределения набора данных между несколькими серверами, что делает его самостоятельным решением для кластеризации.

Существует два варианта размещения `etcd` в системе Kubernetes.

- Совместно с `kube-apiserver` (и другими компонентами управляющего слоя) на управляющих узлах. Этот вариант используется по умолчанию и является наиболее простым. В большинстве кластеров Kubernetes такие компоненты, как `etcd` и `kube-apiserver`, изначально развертываются с использованием статических манифестов. Мы подробно рассмотрим этот и альтернативные подходы далее в книге.
- В виде выделенного кластера `etcd`. Это более сложный вариант, но он обеспечивает повышенную надежность в средах, для которых это критично.

Эксплуатация кластеров etcd в Kubernetes

Подробную информацию о выделенных кластерах `etcd` (состоящих из одного или нескольких узлов) можно найти в официальной документации по Kubernetes: https://kubernetes.io/docs/tasks/administer-cluster/configure-upgrade-etcd/.

Дополнительные сведения о компоненте etcd

Если вы хотите разобраться в принципах работы компонента `etcd` и поэкспериментировать с его набором данных, то можете воспользоваться бесплатной онлайн-площадкой: http://play.etcd.io/play. Там можно потренироваться в управлении кластерами `etcd` и хранимыми в них данными.

В следующем подразделе мы рассмотрим компонент `kube-scheduler`.

Компонент kube-scheduler

Компонент `kube-scheduler` отвечает за выбор вычислительного узла для запуска нового пода.

После создания под находится в состоянии unscheduled («незапланированный»). Это означает, что для его выполнения еще не назначен вычислительный узел. Такой незапланированный под фиксируется в хранилище `etcd` без указания узла, на котором он должен выполняться. В результате ни один активный агент `kubelet` не получает информацию о необходимости запуска этого пода, поэтому ни один контейнер, определенный в спецификации пода, не будет выполнен.

Объект `Pod`, хранящийся в `etcd`, имеет свойство `nodeName`. Как следует из названия, в этом свойстве должно быть указано имя вычислительного узла, на котором будет размещен под. Если это свойство заполнено, под считается *запланированным* (scheduled), а если нет, значит, под находится в состоянии *ожидания* (pending).

Компонент `kube-scheduler` с определенной периодичностью обращается к `kube-apiserver`, чтобы получить список незапланированных подов или подов с пустым свойством `nodeName`. Обнаружив такие поды, он запускает алгоритм выбора вычислительного узла, а затем обновляет свойство пода `nodeName` путем отправки HTTP-запроса компоненту `kube-apiserver`. При выборе вычислительного узла `kube-scheduler` может учитывать дополнительные параметры конфигурации, заданные пользователем (рис. 2.6).

Компонент `kube-scheduler` может учитывать дополнительные параметры конфигурации, что позволяет вам точно управлять процессом выбора вычислительного узла. Ниже перечислены механизмы, которые вы можете задействовать при назначении подов узлам:

- селектор узлов (node selector);
- правила предпочтения (node affinity) и антипредпочтения узлов (node anti-affinity);
- ограничения (taint) и допуски (toleration).

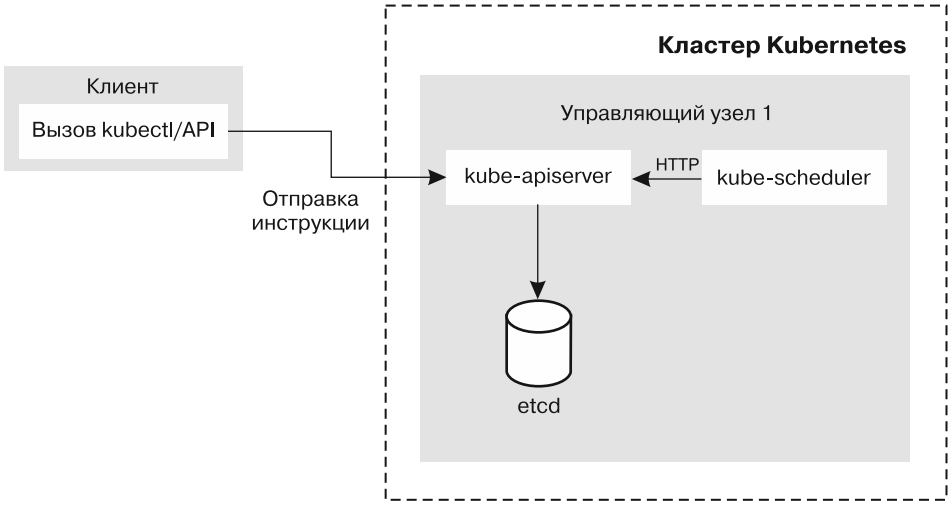

Рис. 2.6. Компонент kube-scheduler опрашивает компонент kube-apiserver, чтобы найти незапланированные поды

Существуют и более продвинутые методы планирования подов, которые не задействуют компонент `kube-scheduler`. Мы рассмотрим их позже.

Компонент kube-scheduler можно заменить собственной реализацией. Вы можете создать свой вариант kube-scheduler с пользовательской логикой выбора узла и задействовать его в своем кластере. Это одна из сильных сторон распределенной архитектуры компонентов Kubernetes.

Где устанавливать kube-scheduler

Компонент `kube-scheduler` можно установить либо на выделенную машину, либо на ту же машину, где установлен `kube-apiserver`. Процесс установки не отличается особой сложностью и не требует значительных ресурсов, однако есть важные моменты, на которые стоит обратить внимание.

Компонент `kube-scheduler` должен быть высокодоступным, поэтому его следует установить на нескольких машинах. При отсутствии рабочего экземпляра `kube-scheduler` в кластере новые поды не будут назначаться узлам, и в результате множество подов будут подолгу находиться в состоянии ожидания. При этом отсутствие `kube-scheduler` никак не повлияет на уже запланированные поды.

В следующем подразделе мы рассмотрим еще один важный компонент управляющего слоя — `kube-controller-manager`.

Компонент kube-controller-manager

Компонент `kube-controller-manager` — это единый исполняемый файл, включающий в себя различные функции, которые в Kubernetes называются контроллерами. Именно этот компонент запускает так называемый цикл согласования (reconciliation loop). `kube-controller-manager` стремится поддерживать фактическое состояние кластера в соответствии с описанным в `etcd`, предотвращая возникновение расхождений между ними.

В некоторых случаях фактическое состояние кластера может отличаться от желаемого состояния, сохраненного в `etcd`. Причиной этого могут быть, например, сбои в работе подов или другие факторы. В таких ситуациях `kube-controller-manager` играет ключевую роль, приводя фактическое состояние в соответствие с желаемым. В качестве примера можно привести контроллер репликации (replication controller), работающий внутри компонента `kube-controller-manager`. Система Kubernetes позволяет задать и поддерживать определенное количество подов на различных вычислительных узлах. Если по какой-либо причине фактическое количество подов отклоняется от заданного, контроллер репликации отправляет запросы в `kube-apiserver`, чтобы создать в `etcd` новый под, заменив тем самым сбойный под на вычислительном узле.

Ниже перечислены некоторые из контроллеров, входящих в состав компонента `kube-controller-manager`.

- *Контроллер узла* (Node Controller) — управляет жизненным циклом узлов, контролируя их добавление, удаление и обновление в кластере.
- *Контроллер репликации* (Replication Controller) — поддерживает заданное количество реплик пода в соответствии с его спецификацией.
- *Контроллер конечных точек* (Endpoints Controller) — заполняет объекты конечных точек, отражая актуальные поды, доступные для каждого сервиса.
- *Контроллер учетных записей сервисов* (Service Account Controller) — управляет учетными записями сервисов (ServiceAccount) в пространстве имен, гарантируя наличие учетной записи `default` в каждом активном пространстве имен.
- *Контроллер пространств имен* (Namespace Controller) — управляет жизненным циклом пространств имен, в том числе их созданием, удалением и изоляцией.
- *Контроллер развертывания* (Deployment Controller) — управляет жизненным циклом развертываний (объектов Deployment), поддерживая требуемое число подов для каждого из них.
- *Контроллер объекта StatefulSet* (StatefulSet Controller) — управляет жизненным циклом объектов StatefulSet, поддерживая требуемое количество реплик, порядок и идентичность подов.

- *Контроллер объекта DaemonSet* (DaemonSet Controller) — управляет жизненным циклом объектов DaemonSet, гарантируя активность копии пода на каждом узле кластера.
- *Контроллер задания* (Job Controller) — управляет жизненным циклом объектов Job, поддерживая заданное количество подов для каждого задания вплоть до его завершения.
- *Контроллер горизонтального автомасштабирования подов* (HPA (Horizontal Pod Autoscaler) Controller) — динамически масштабирует количество реплик для объектов Deployment или StatefulSet на основе уровня потребления ресурсов или других метрик.
- *Сборщик мусора для подов* (Pod Garbage Collector) — удаляет поды, которые больше не контролируются владельцем (например, контроллером репликации или развертыванием).

Как видите, компонент `kube-controller-manager` является довольно объемным. Тем не менее, по сути, он представляет собой единый исполняемый файл, отвечающий за согласование фактического состояния кластера с желаемым состоянием, сохраненным в `etcd`.

Где запускать kube-controller-manager

Как и `kube-apiserver`, компонент `kube-controller-manager` может запускаться на управляющих узлах в виде контейнера или в виде сервиса, управляемого с помощью `systemd`. Кроме того, `kube-controller-manager` можно установить на выделенную машину. Теперь рассмотрим компонент `cloud-controller-manager`.

Компонент cloud-controller-manager

Компонент управляющего слоя `cloud-controller-manager` отвечает за взаимодействие между системой Kubernetes и базовой облачной инфраструктурой. Он предоставляет и управляет облачными ресурсами, включая узлы и тома, обеспечивая выполнение рабочих нагрузок Kubernetes. Этот компонент запускает только контроллеры, адаптированные под конкретного облачного провайдера. Если кластер Kubernetes развернут локально, например на персональном компьютере или в собственной инфраструктуре организации, компонент `cloud-controller-manager` будет в нем отсутствовать.

Подобно `kube-controller-manager`, компонент `cloud-controller-manager` объединяет несколько логически независимых циклов управления (control loops) в двоичный файл, выполняющийся как единый процесс. Для повышения производительности или отказоустойчивости его можно масштабировать горизонтально путем запуска нескольких экземпляров.

Вот примеры контроллеров, зависящих от облачного провайдера.

- *Контроллер узла* (Node Controller) — проверяет, удален ли узел в облаке после того, как он перестал отвечать.
- *Контроллер маршрутов* (Route Controller) — настраивает маршруты в базовой облачной инфраструктуре.
- *Контроллер сервисов* (Service Controller) — управляет созданием, обновлением и удалением балансировщиков нагрузки, предоставляемых облачным провайдером.

Где запускать cloud-controller-manager

Как и `kube-apiserver`, компонент `cloud-controller-manager` может запускаться на управляющих узлах в виде контейнера или в виде сервиса, управляемого с помощью `systemd`.

В следующем разделе мы рассмотрим компоненты вычислительных узлов кластера Kubernetes.

Компоненты вычислительных узлов

В этом разделе мы разберем устройство вычислительного узла, рассмотрев три компонента, которые на нем работают:

- контейнерный движок и среду выполнения контейнеров;
- компонент `kubelet`;
- компонент `kube-proxy`.

Компоненты kubelet, kube-proxy и среда выполнения контейнеров являются обязательными как для узлов управляющего слоя, так и для вычислительных узлов. Далее мы обсудим их роль в обоих контекстах.

Контейнерный движок и среда выполнения контейнеров

Контейнерный движок (container engine) — это программная платформа, отвечающая за создание, запуск и управление жизненным циклом контейнеров. Он обеспечивает более высокий уровень абстракции по сравнению со средой выполнения контейнеров, упрощает управление контейнерами и делает этот процесс более доступным для разработчиков. Наиболее известными контейнерными движками являются Podman, Docker Engine и CRI-O. *Среда выполнения контейнеров* (container runtime) — это базовый программный компонент, осу-

ществляющий создание, запуск и администрирование контейнеров в фоновом режиме по команде контейнерного движка или оркестратора. Она предоставляет ключевые функции для обеспечения работы контейнеров, включая выделение ресурсов, загрузку образов, создание контейнеров и управление их жизненным циклом. Среди известных сред выполнения контейнеров можно выделить `containerd`, `runc`, `dockerd` и Mirantis Container Runtime.

Термины «контейнерный движок» и «среда выполнения контейнеров» иногда используются как синонимы, что может вызывать путаницу. Среда выполнения контейнеров (низкоуровневая) — это ядро, непосредственно выполняющее образы контейнеров, управляющее их жизненным циклом (запуск, остановка, приостановка) и взаимодействующее с базовой операционной системой. Примерами являются runc и CRI-O (при использовании в режиме среды выполнения). Контейнерный движок (высокоуровневый) работает поверх среды выполнения контейнеров, предоставляя такие дополнительные возможности, как сборка образов, работа с реестрами и инструменты управления. Примерами являются Docker, Podman или CRI-O (при использовании с Kubernetes). Таким образом, низкоуровневые среды отвечают за непосредственное выполнение контейнеров, а высокоуровневые движки добавляют слой управления и удобства для пользователя.

Ранее для запуска контейнеров в Kubernetes по умолчанию использовался Docker. Однако сейчас эта система может работать и с другими средами выполнения контейнеров, такими как `containerd`, CRI-O (в связке с `runc`), Mirantis Container Runtime и не только. В книге мы будем использовать Kubernetes с `containerd` или CRI-O по следующим причинам.

- *Узкая специализация и гибкость.* Инструменты `containerd` и CRI-O предоставляют исключительно функции среды выполнения контейнеров, что делает их более легковесными и потенциально более безопасными по сравнению с системой Docker, отличающейся гораздо более обширным функционалом. Эта специализация также обеспечивает более тесную интеграцию с такими платформами оркестрации контейнеров, как Kubernetes. В отличие от Docker они не нуждаются в дополнительных компонентах вроде `cri-dockerd` для работы с Kubernetes.
- *Согласованность с направлением развития Kubernetes.* Kubernetes постепенно отказывается от использования Docker в качестве среды выполнения по умолчанию. Ранее (до выхода версии 1.24) для интеграции Docker с Kubernetes использовался компонент под названием `dockershim`.

  Однако этот подход был признан устаревшим, и теперь Kubernetes рекомендует использовать среды выполнения, совместимые со стандартом *CRI* (Container Runtime Interface — интерфейс среды выполнения контейнеров),

разработанным специально для этой платформы. Выбор среды `containerd` или CRI-O обеспечивает более эффективную интеграцию с системой Kubernetes.

- *Ориентация на Kubernetes*. CRI-O представляет собой легковесную среду выполнения контейнеров, специально разработанную для Kubernetes. Нумерация ее минорных версий синхронизирована с нумерацией минорных версий Kubernetes, что упрощает управление совместимостью. Когда выпуск Kubernetes достигает конца срока поддержки, соответствующую версию CRI-O также можно считать устаревшей, что упрощает процесс поддержания окружения Kubernetes в актуальном и безопасном состоянии.

Интерфейс среды выполнения контейнеров

Система Kubernetes использует среду выполнения контейнеров для запуска контейнеров внутри подов. Для взаимодействия с выбранной средой выполнения Kubernetes по умолчанию использует интерфейс CRI (Container Runtime Interface, интерфейс среды выполнения контейнеров). Этот интерфейс был впервые представлен в Kubernetes версии 1.5, выпущенной в декабре 2016 года.

CRI представляет собой интерфейс плагинов, позволяющий компоненту `kubelet` беспрепятственно интегрироваться с различными средами выполнения контейнеров. Такая гибкость дает возможность выбирать оптимальную среду в зависимости от конкретных требований.

В рамках стандарта CRI определен набор API, позволяющих компоненту `kubelet` эффективно взаимодействовать со средой выполнения контейнеров. Эти API охватывают такие ключевые операции, как запуск, остановка и удаление контейнеров, а также управление песочницами для подов и сетевыми взаимодействиями.

В табл. 2.1 перечислены заданные по умолчанию конечные точки для сред выполнения контейнеров на машинах с Linux.

Таблица 2.1. Конечные точки по умолчанию для сред выполнения контейнеров на машинах под управлением Linux

Среда выполнения	Путь к сокету домена Unix
`containerd`	unix:///var/run/containerd/containerd.sock
CRI-O	unix:///var/run/crio/crio.sock
Docker Engine (при использовании `cri-dockerd`)	unix:///var/run/cri-dockerd.sock

Более подробные сведения можно найти в документации по адресу https://kubernetes.io/docs/setup/production-environment/tools/kubeadm/install-kubeadm.

Kubernetes и Docker. Выпуски Kubernetes до версии 1.24 предусматривали прямую интеграцию с Docker Engine, реализуемую с помощью компонента `dockershim`. Однако эта интеграция была признана устаревшей, и в релизе версии 1.20 было объявлено о ее удалении. В настоящее время осуществляется постепенный отказ от использования Docker в качестве базовой среды выполнения контейнеров, и теперь пользователям Kubernetes рекомендуется выбирать среды выполнения, совместимые со стандартом CRI, разработанным специально для этой платформы.

Несмотря на эти изменения, образы, созданные с помощью Docker, по-прежнему будут работать в вашем кластере с любой средой выполнения контейнеров. Подробнее об этом можно узнать здесь: https://kubernetes.io/blog/2020/12/02/dont-panic-kubernetes-and-docker/.

Таким образом, любая машина под управлением Linux, на которой работает среда `containerd`, может использоваться в качестве базы для построения вычислительного узла Kubernetes. (О вычислительных узлах на машинах под управлением Windows мы поговорим в следующих главах.)

Проект Open Container Initiative. *Open Container Initiative (OCI)* — это проект с открытым исходным кодом, определяющий стандарты для контейнеров, их образов, сред выполнения и контейнерных реестров. Его цель — обеспечить совместимость и взаимную интеграцию контейнерных систем и гарантировать единообразное выполнение контейнеров в различных средах. Система CRI взаимодействует с OCI, предоставляя стандартизированный интерфейс для обмена данными между компонентом `kubelet` и средой выполнения контейнеров. OCI определяет стандарты для образов и сред выполнения контейнеров, поддерживаемых интерфейсом CRI, что способствует эффективному управлению контейнерами и их развертыванию в системе Kubernetes.

Класс среды выполнения контейнеров. Ресурс *RuntimeClass* (класс среды выполнения) системы Kubernetes позволяет определять и назначать подам различные конфигурации среды выполнения контейнеров. Это дает возможность обеспечивать нужный баланс между производительностью и безопасностью приложений. Например, рабочие нагрузки, предъявляющие повышенные требования к безопасности, можно запускать в среде выполнения с аппаратной виртуализацией для обеспечения более строгой изоляции, даже если это приведет к некоторому снижению производительности. RuntimeClass также позволяет использовать одну и ту же среду выполнения с разными настройками для отдельных подов. Для этого необходимо настроить интерфейс CRI на узлах (способ установки зависит от среды) и создать в Kubernetes соответствующие ресурсы типа RuntimeClass.

В следующем подразделе вы познакомитесь с агентом `kubelet` — еще одним важным компонентом узла кластера Kubernetes.

Агент kubelet

Агент `kubelet` — это ключевой компонент вычислительного узла, поскольку именно он взаимодействует с локальной средой выполнения контейнеров, установленной на этом узле.

`kubelet` работает исключительно как системный демон и не может быть запущен внутри контейнера. Он должен выполняться непосредственно в хостовой системе, как правило, в виде сервиса под управлением `systemd`. Это отличает `kubelet` от других компонентов Kubernetes и подчеркивает его строгую привязку к хостовой машине.

При запуске `kubelet` по умолчанию считывает конфигурационный файл, расположенный по адресу `/etc/kubernetes/kubelet.conf`.

В этом файле определены два критически важных параметра:

- адрес конечной точки компонента `kube-apiserver`;
- путь к Unix-сокету локальной среды выполнения контейнеров.

После присоединения вычислительного узла к кластеру агент `kubelet` действует как мост между компонентом `kube-apiserver` и локальной средой выполнения контейнеров. Он регулярно отправляет HTTP-запросы к `kube-apiserver` для получения сведений о подах, подлежащих запуску.

По умолчанию *каждые 20 секунд* агент `kubelet` отправляет `GET`-запрос компоненту `kube-apiserver`, чтобы получить список подов, созданных в `etcd` и назначенных данному узлу.

Получив спецификацию пода в теле HTTP-ответа от `kube-apiserver`, `kubelet` преобразует ее в спецификацию контейнера и передает на выполнение через указанный Unix-сокет. В результате на вычислительном узле создаются контейнеры с использованием локальной среды выполнения (например, `containerd`).

Важно помнить, что, как и другие компоненты Kubernetes, kubelet не обращается к etcd напрямую; он взаимодействует только с kube-apiserver, который предоставляет данные из etcd. Более того, kubelet даже не знает о существовании сервера etcd, работающего за kube-apiserver, к которому он обращается.

Механизмы опроса, называемые в Kubernetes *механизмами наблюдения* (watch mechanisms), определяют, каким образом система выполняет запуск и удаление контейнеров на вычислительных узлах в больших масштабах. Здесь важно учитывать два момента:

- компоненты `kubelet` и `kube-apiserver` должны иметь возможность обмениваться данными по протоколу HTTP, поэтому между вычислительными узлами и управляющим слоем необходимо открыть HTTPS-порт `6443`;

- поскольку kubelet, интерфейс CRI и среда выполнения контейнеров работают на одной машине, их взаимодействие осуществляется через Unix-сокеты.

Поскольку каждый вычислительный узел в кластере Kubernetes должен иметь собственный экземпляр kubelet, увеличение числа узлов приводит к росту количества HTTP-запросов, адресованных компоненту kube-apiserver. В крупных кластерах, содержащих сотни машин, эта повышенная нагрузка может негативно сказаться на производительности kube-apiserver, а в отдельных случаях даже привести к проблемам с доступностью API. Поэтому для обеспечения высокой доступности kube-apiserver и других компонентов управляющего слоя необходимо организовать эффективное масштабирование.

Следует также отметить, что мы можем полностью обойти Kubernetes и создавать контейнеры на рабочих узлах без использования kubelet. Единственная задача kubelet — обеспечить соответствие между локальной средой выполнения контейнеров и конфигурацией, хранящейся в etcd. Поэтому, если вы создаете контейнеры на рабочем узле вручную, kubelet не сможет ими управлять. Однако предоставление доступа к сокету среды выполнения контейнеров контейнеризованным рабочим нагрузкам создает угрозу безопасности, так как позволяет обходить защитные механизмы Kubernetes и служит популярным вектором атак. Одним из ключевых принципов обеспечения безопасности кластера Kubernetes является запрет на монтирование этого сокета в контейнеры.

Учтите, что контейнерный движок, работающий на вычислительном узле, не знает о том, что он управляется системой Kubernetes через локальный агент kubelet. По сути, вычислительный узел представляет собой обычную машину под управлением Linux, на которой установлены среда выполнения контейнеров и агент kubelet, исполняющий инструкции по управлению контейнерами.

В следующем подразделе мы обсудим компонент kube-proxy.

Компонент kube-proxy

Важной частью Kubernetes является сетевая подсистема. Мы подробно рассмотрим принцип ее работы позднее, а сейчас просто имейте в виду то, что в Kubernetes предусмотрено множество механизмов для организации доступа к подам — как извне, так и изнутри кластера.

Эти механизмы реализуются на уровне компонента kube-proxy, поэтому для обеспечения доступности подов, работающих на вычислительных узлах, каждый из этих узлов должен иметь запущенный экземпляр kube-proxy. Чуть позже мы познакомимся с возможностями объекта Kubernetes *Service*, реализация которого осуществляется на уровне компонента kube-proxy. Как и kubelet, компонент kube-proxy взаимодействует с компонентом kube-apiserver.

Существуют и другие подсистемы или расширения, работающие на уровне вычислительного узла, например *cAdvisor* или *CNI* (Container Network Interface — сетевой интерфейс контейнера). Однако обсуждение этих довольно сложных тем мы пока отложим.

Теперь, когда вы познакомились с различными компонентами и концепциями Kubernetes, обсудим утилиту `kubectl` и процесс ее взаимодействия с API Kubernetes.

Обзор утилиты командной строки kubectl и синтаксиса YAML

`kubectl` — это официальная утилита командной строки, предназначенная для управления платформой Kubernetes. Она представляет собой HTTP-клиент, оптимизированный для работы с этой системой и позволяющий отдавать команды кластеру Kubernetes.

Kubernetes и среда обучения на базе Linux

Для эффективного изучения контейнеров Linux и связанных с ними тем лучше всего использовать рабочие станции или лабораторные машины под управлением Linux. Базовые знания этой ОС крайне полезны для работы с контейнерами и системой Kubernetes. Использование Linux в качестве основной ОС на рабочей станции автоматически погружает вас в нужную среду, делая обучение более продуктивным. Вы можете выбрать любой дистрибутив Linux, например Fedora или Ubuntu. Однако мы стремимся к инклюзивности, поэтому предлагаем альтернативные инструкции для пользователей Windows и macOS, чтобы сделать процесс обучения доступным для всех. Таким образом, наличие рабочей станции с Linux не является обязательным условием для изучения Kubernetes. Если вы работаете в Windows, то можете воспользоваться таким альтернативным решением, как WSL (Windows Subsystem for Linux — подсистема Windows для Linux) (https://learn.microsoft.com/en-us/windows/wsl/).

Установка утилиты командной строки kubectl

Утилиту командной строки `kubectl` можно установить на рабочие станции под управлением Linux, Windows или macOS. Для обеспечения оптимальной совместимости необходимо следить за тем, чтобы минорная версия клиента `kubectl` отличалась от минорной версии кластера Kubernetes не более чем на единицу. Это означает, что `kubectl` версии v1.30 может управлять кластерами версий v1.29, v1.30 и v1.31. Использование последней совместимой версии помогает избежать потенциальных проблем.

Поскольку утилита `kubectl` понадобится вам уже в следующей главе, вы можете установить ее прямо сейчас, воспользовавшись инструкциями, приведенными в следующих разделах.

Устаревшие репозитории пакетов Kubernetes

Устаревшие репозитории пакетов Linux `apt.kubernetes.io` и `yum.kubernetes.io` (они же `packages.cloud.google.com`) являются недоступными с 13 сентября 2023 года. Пользователям рекомендуется перейти на новые, управляемые сообществом репозитории Debian- и RPM-пакетов `pkgs.k8s.io`, представленные 15 августа 2023 года. Они заменяют устаревшие репозитории, размещавшиеся на серверах Google (`apt.kubernetes.io` и `yum.kubernetes.io`). Это изменение затрагивает пользователей, устанавливавших официальные версии Kubernetes напрямую, а также тех, кто установил утилиту `kubectl` из устаревших репозиториев пакетов. Дополнительную информацию можно найти в официальном объявлении по адресу https://kubernetes.io/blog/2023/08/31/legacy-package-repository-deprecation/.

Установка kubectl в Linux

Чтобы установить утилиту `kubectl` в Linux, необходимо скачать и скопировать ее в каталог, входящий в путь поиска исполняемых файлов, следующим образом:

```
$ curl -LO "https://dl.k8s.io/release/$(curl -L -s \
https://dl.k8s.io/release/stable.txt)/bin/linux/amd64/kubectl"
$ chmod +x ./kubectl
$ sudo mv ./kubectl /usr/local/bin/kubectl
$ kubectl version
Client Version: v1.30.0
Kustomize Version: v5.0.4-0.20230601165947-6ce0bf390ce3
The connection to the server localhost:8080 was refused - did you specify the
right host or port?
```

Проигнорируйте сообщение об ошибке подключения — она возникает из-за того, что кластер Kubernetes, к которому можно обращаться с помощью `kubectl`, еще не настроен.

В вашем случае путь может отличаться от /usr/local/bin/kubectl[1]. Убедитесь в том, что переменные окружения PATH настроены корректно, чтобы система могла обнаружить утилиту kubectl. Путь к ней также можно прописать в файле /etc/profile.

Чтобы загрузить определенную версию, замените фрагмент `$(curl -L -s https://dl.k8s.io/release/stable.txt)` в команде нужным номером версии.

Например, если вы хотите загрузить версию 1.30.0 для Linux x86-64, выполните команду:

```
$ curl -LO https://dl.k8s.io/release/v1.30.0/bin/linux/amd64/kubectl
```

[1] Определить подходящий путь можно, введя команду echo $PATH. В выводе вы получите список путей поиска, разделенный двоеточиями.

Эта команда загрузит указанную версию утилиты `kubectl`, после чего вы сможете скопировать ее в нужный каталог.

Теперь рассмотрим процесс установки `kubectl` в macOS.

Установка kubectl в macOS

Процесс установки в macOS практически ничем не отличается от описанного выше, за исключением того, что для платформ на базе Intel и Apple Silicon существуют отдельные пакеты `kubectl`.

```
# Intel
$ curl -LO "https://dl.k8s.io/release/$(curl -L -s \
 https://dl.k8s.io/release/stable.txt)/bin/darwin/amd64/kubectl"

# Apple Silicon
$ curl -LO "https://dl.k8s.io/release/$(curl -L -s \
https://dl.k8s.io/release/stable.txt)/bin/darwin/arm64/kubectl"

$ chmod +x ./kubectl
$ sudo mv ./kubectl /usr/local/bin/kubectl
$ sudo chown root: /usr/local/bin/kubectl
```

Установка kubectl в Windows

Скачайте файл `kubectl.exe` (https://dl.k8s.io/release/v1.30.0/bin/windows/amd64/kubectl.exe) через браузер или с помощью утилиты `curl` (или аналогичной утилиты командной строки, если она установлена в Windows):

```
$ curl.exe -LO https://dl.k8s.io/release/v1.30.0/bin/windows/amd64/kubectl.exe
```

Затем добавьте путь к утилите `kubectl` в начало или конец переменной окружения `PATH` и убедитесь, что используемая версия утилиты соответствует загруженной.

Вы также можете установить kubectl с помощью стандартного менеджера пакетов, например apt-get, yum, Zypper, brew (для macOS) или Chocolatey (для Windows). Дополнительную информацию можно найти в документации: https://kubernetes.io/docs/tasks/tools/install-kubectl-linux.

В следующем подразделе мы поговорим об использовании команды `kubectl`.

Роль утилиты kubectl

Поскольку компонент `kube-apiserver` представляет собой HTTP API, для взаимодействия с кластером Kubernetes можно использовать любой HTTP-клиент. Например, вы можете управлять кластером даже с помощью утилиты `curl`, но, разумеется, для этого существует более удобный способ.

Почему же стоит использовать именно специализированный клиент, а не `curl`? Причина заключается в удобстве. Компонент `kube-apiserver` управляет множеством различных ресурсов, каждый из которых имеет свой URL-путь.

Хотя постоянно обращаться к `kube-apiserver` с помощью `curl` технически возможно, это крайне затратно по времени. Необходимость в запоминании путей ко всем ресурсам и способов их вызова создает неудобство для пользователя. Кроме того, нецелесообразность применения `curl` объясняется тем, что утилита `kubectl` управляет и другими аспектами работы, в частности связанными с аутентификацией в системе Kubernetes, управлением контекстами кластера и не только.

При работе через `curl` вам бы пришлось постоянно обращаться к документации, чтобы припомнить URL-путь, HTTP-заголовок или строку запроса. Утилита `kubectl` избавляет вас от этой необходимости, позволяя отправлять команды компоненту `kube-apiserver`, которые являются легко запоминаемыми, безопасными и полностью ориентированными на работу с Kubernetes.

Когда вы вызываете утилиту `kubectl`, она анализирует переданные параметры и на их основе формирует и отправляет HTTP-запросы компоненту `kube-apiserver` вашего кластера Kubernetes (рис. 2.7).

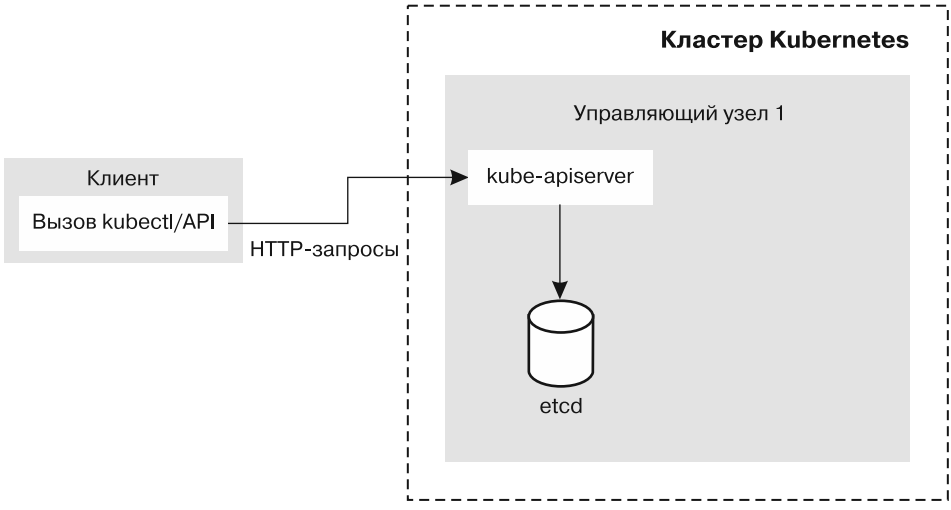

Рис. 2.7. Утилита kubectl отправляет запросы компоненту kube-apiserver по протоколу HTTP; для взаимодействия с кластером Kubernetes вы будете постоянно использовать именно ее

Получив от вас корректно сформированный HTTP-запрос, компонент `kube-apiserver` считывает или изменяет состояние кластера в `etcd` (в зависимости от содержания запроса). Например, если это запрос на обновление образа

запущенного контейнера, `kube-apiserver` вносит изменения в состояние кластера в `etcd`. Затем компоненты, работающие на вычислительном узле, где размещен данный контейнер, выполняют соответствующие команды, чтобы запустить новый контейнер на основе обновленного образа. Это обеспечивает соответствие фактического состояния контейнера данным, хранящимся в `etcd`.

Поскольку вам не требуется напрямую взаимодействовать с контейнерным движком или хранилищем `etcd`, можно сказать, что уровень владения Kubernetes в значительной степени зависит от знания команд `kubectl`. Для эффективной работы с Kubernetes необходимо как можно глубже освоить API данной системы и нюансы его работы. При этом вам предстоит взаимодействовать только с компонентом `kube-apiserver` и утилитой командной строки `kubectl`, которая позволяет обращаться к нему.

Поскольку компонент kube-apiserver доступен по протоколу HTTP(S), подключаться к любому кластеру Kubernetes можно с помощью HTTP-библиотек или программно (используя любой язык программирования). Для kubectl существует множество альтернатив, однако именно эта утилита признана официальным инструментом проекта Kubernetes, часто фигурирует в документации и используется в большинстве примеров, с которыми вам предстоит столкнуться.

Принцип работы утилиты kubectl

При вызове утилиты `kubectl` она пытается прочитать конфигурационный файл `kubeconfig`, находящийся по адресу `$HOME/.kube/config`. Данный файл должен содержать следующую информацию, необходимую для работы `kubectl` и прохождения аутентификации в компоненте `kube-apiserver`:

- URL и порт конечной точки `kube-apiserver`;
- учетную запись пользователя;
- клиентские сертификаты (если используются) для аутентификации в `kube-apiserver`;
- контексты, связывающие пользователя с кластером.

Мы также можем передавать различные параметры (например, сведения о кластере, данные для аутентификации пользователя и т. д.) команде kubectl в виде аргументов, однако этот способ не особенно удобен при необходимости управлять несколькими кластерами в одной среде.

На следующей схеме показан типичный файл `kubeconfig` (рис. 2.8).

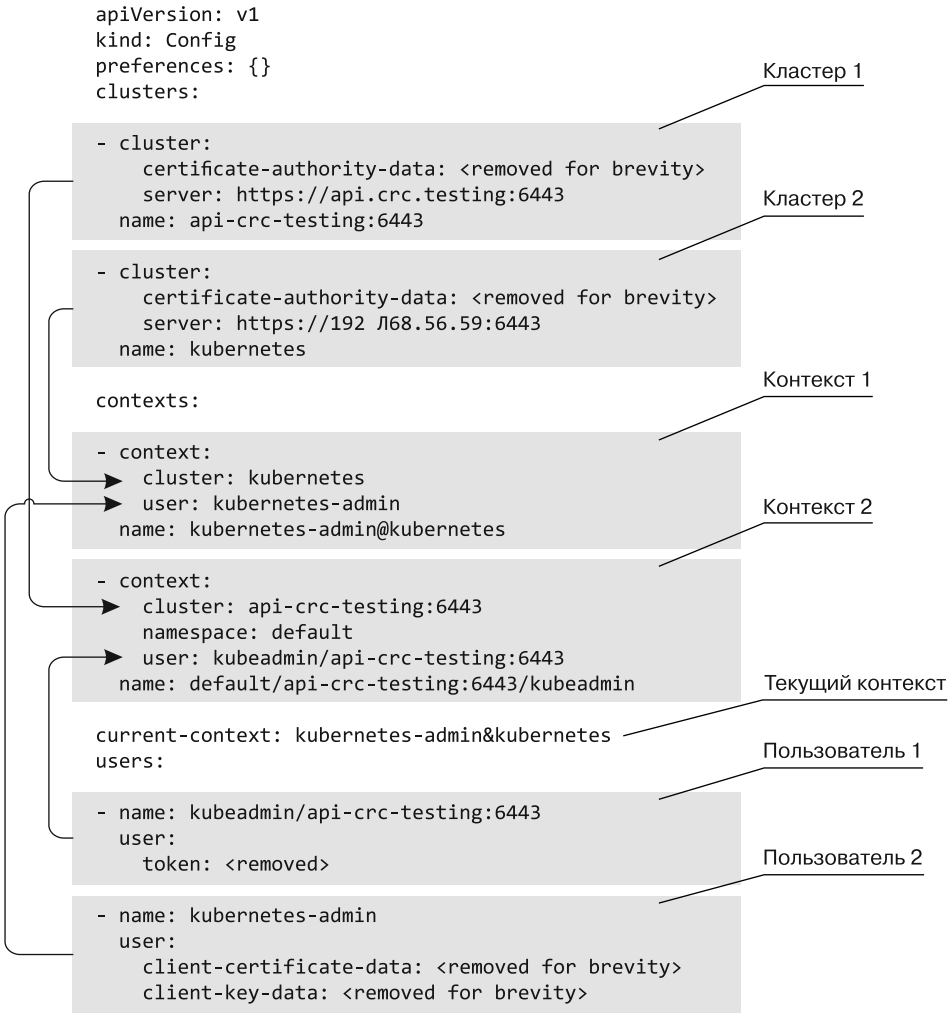

Рис. 2.8. Контекст и структура файла kubeconfig

На представленной выше схеме изображен файл конфигурации с несколькими настроенными кластерами, пользователями и контекстами.

- *Кластеры* (`clusters`) — этот раздел определяет кластеры Kubernetes, с которыми вы можете взаимодействовать. Он содержит такие сведения, как адрес

сервера, версия API и данные центра сертификации для каждого кластера, что позволяет `kubectl` подключаться к ним и отправлять им команды.

- *Пользователи* (`users`) — этот раздел хранит учетные данные для получения доступа к кластерам Kubernetes. Обычно он включает имя пользователя и секрет (например, токен или клиентский сертификат), используемые для аутентификации на API-сервере. Файлы `kubeconfig` также могут ссылаться на сертификаты, указанные в разделе `user`, для безопасной аутентификации пользователей на API-сервере Kubernetes. Такая двусторонняя проверка гарантирует получение доступа к кластеру только авторизованными пользователями с действительными сертификатами и устраняет потенциальные угрозы безопасности.
- *Контексты* (`contexts`) — этот раздел связывает кластеры с пользователями. Каждый контекст ссылается на конкретный кластер и конкретного пользователя в этом кластере. Выбирая контекст, вы определяете, какой кластер и какие учетные данные утилита `kubectl` будет использовать при выполнении последующих команд.

Наличие в файле `kubeconfig` нескольких настроенных кластеров, пользователей и контекстов позволяет легко переключаться между разными кластерами Kubernetes, используя различные учетные данные.

Путь к файлу `kubeconfig` можно переопределить в системе, установив переменную окружения `KUBECONFIG` или передав параметр `--kubeconfig` при вызове `kubectl`:

```
$ export KUBECONFIG="/custom/path/.kube/config"
$ kubectl --kubeconfig="/custom/path/.kube/config"
```

Каждый раз при запуске команды `kubectl` одноименная утилита командной строки ищет файл `kubeconfig`, из которого нужно загрузить конфигурацию, выполняя следующие шаги.

1. Сначала утилита проверяет, был ли передан параметр `--kubeconfig`, и при его наличии загружает указанный файл конфигурации.
2. Если файл конфигурации не найден, `kubectl` проверяет наличие переменной окружения `KUBECONFIG`.
3. В итоге она использует файл по умолчанию, расположенный по адресу `$HOME/.kube/config`.

Чтобы просмотреть файл конфигурации, который в данный момент используется локальной установкой `kubectl`, выполните команду:

```
$ kubectl config view
```

После этого HTTP-запрос отправляется компоненту `kube-apiserver`, который формирует HTTP-ответ. Утилита `kubectl` преобразует этот ответ в удобочитаемый формат и выводит его в терминал.

Ниже приведена команда, которая очень часто используется при работе с Kubernetes:

```
$ kubectl get pods
```

Эта команда выводит список подов. По сути, она отправляет `GET`-запрос компоненту `kube-apiserver` для получения списка контейнеров (подов) в вашем кластере. Внутренне `kubectl` сопоставляет переданный в команду параметр `Pods` с URL-путем `/api/v1/pods`, через который `kube-apiserver` предоставляет доступ к ресурсу Pod.

Рассмотрим еще одну команду:

```
$ kubectl run nginx --restart Never --image nginx
```

Эта команда чуть сложнее, так как `run` не является HTTP-методом. Она отправляет `POST`-запрос компоненту `kube-apiserver`, в результате чего создается контейнер `nginx` на основе одноименного образа, размещенного в контейнерном реестре (например, Docker Hub или quay.io).

На самом деле эта команда создает не контейнер, а под. Данный ресурс будет подробно описан в главе 4. А сейчас давайте перейдем от обсуждения контейнеров к изучению терминологии и концепций Kubernetes. С этого момента, если вы встретите слово «контейнер», оно будет означать реальный контейнер в контексте технологий контейнеризации. Соответственно, под словом «под» мы будем подразумевать ресурс Kubernetes.

В следующем разделе вы узнаете, как активизировать функцию автодополнения команд `kubectl`.

Автодополнение команд kubectl

Утилита `kubectl` имеет встроенную функцию автодополнения команд, которая экономит время и снижает количество ошибок. Данная функция поддерживает следующие популярные оболочки:

- Bash;
- Zsh;
- Fish;
- PowerShell.

Например, чтобы активизировать функцию автодополнения в Linux Bash, выполните команду:

```
# Установка пакета Bash Completion (если нужно)
$ sudo yum install bash-completion   # Для систем на основе RPM
$ sudo apt install bash-completion   # Для систем Debian/Ubuntu

# Добавьте строку source в конфигурацию вашей оболочки
$ echo 'source <(kubectl completion bash)' >> ~/.bashrc
```

Теперь при вводе команды утилита `kubectl` будет предлагать варианты автодополнения на основе доступных ресурсов и параметров. Чтобы принять предложение, нажмите клавишу Tab или продолжайте ввод, чтобы сократить список доступных вариантов.

Процесс активизации функции автодополнения в других оболочках, таких как Zsh или Fish, может быть немного иным. Подробные инструкции приведены в официальной документации по kubectl, доступной по адресу https://kubernetes.io/docs/reference/kubectl/generated/kubectl_completion/.

Такая настройка гарантирует активизацию функции автодополнения при каждом запуске нового сеанса терминала.

В следующем разделе мы поговорим об использовании императивного и декларативного синтаксиса при выполнении команд `kubectl`.

Императивный синтаксис

Практически любую инструкцию, передаваемую компоненту `kube-apiserver` через утилиту `kubectl`, можно записать в *императивном* или *декларативном* стиле. Императивный синтаксис позволяет непосредственно изменять состояние кластера в соответствии с аргументами и параметрами, переданными в команду `kubectl`.

Рассмотрим несколько команд, написанных в императивном стиле:

```
# Создает под с именем my-pod на основе образа контейнера busybox:latest:
$ kubectl run my-pod --restart Never --image busybox:latest
# Показывает все ресурсы ReplicaSet в пространстве имен my-namespace,
# созданные в кластере Kubernetes:
$ kubectl get rs -n my-namespace
# Удаляет под с именем my-pod в пространстве имен default:
$ kubectl delete pods my-pod
```

Императивный синтаксис имеет ряд преимуществ. Если вы уже знаете, какие именно инструкции нужно передать системе Kubernetes и какие команды для этого требуются, то сможете работать очень быстро. Такой синтаксис обеспе-

чивает простоту набора инструкций и позволяет решать многие задачи с помощью всего нескольких команд. Некоторые операции, в частности вывод списка существующих в кластере ресурсов, можно выполнить только в императивной форме.

Однако у императивного синтаксиса есть и серьезный недостаток, заключающийся в сложности отслеживания действий, выполняемых в кластере. Если по какой-либо причине текущее состояние кластера будет утрачено и вам потребуется восстановить его с нуля, то вспомнить все введенные ранее команды будет крайне сложно. Разумеется, можно обратиться к файлу `.bash_history`, но существует более удобный способ, о котором мы поговорим в следующем разделе, посвященном декларативному синтаксису.

Декларативный синтаксис

Термин «декларативный» означает, что мы «декларируем» желаемое состояние кластера, а Kubernetes создает необходимые ресурсы для достижения этого состояния. Поддерживаемыми форматами являются JSON и YAML, однако пользователи Kubernetes обычно предпочитают YAML из-за его простоты.

YAML (YAML ain't markup language — «YAML — не язык разметки») — это человекочитаемый формат сериализации данных, широко применяемый в Kubernetes для четкого и компактного описания конфигурации таких ресурсов системы, как объекты Deployment, Service и Pod. Данный формат упрощает управление и версионирование конфигураций Kubernetes, что способствует совместной работе и воспроизводимости результатов. Важно отметить, что YAML не является языком программирования и не содержит никакой логики — это просто формат описания конфигураций в виде пар «ключ — значение», который используется во многих современных проектах, включая Kubernetes.

Каждая пара «ключ — значение» в YAML-файле описывает конфигурацию создаваемого ресурса Kubernetes.

Например, ранее мы создали под `my-pod` на основе образа контейнера `busybox:latest` с помощью императивной команды:

```
$ kubectl run my-pod --restart=Never --image busybox:latest
```

Теперь сделаем то же самое, но в декларативном стиле:

```
apiVersion: v1
kind: Pod
metadata:
  name: my-pod
spec:
  containers:
    -  name: busybox-container
       image: busybox:latest
```

Предположим, что мы сохранили файл с конфигурацией пода под именем `pod.yaml`. Для создания самого пода необходимо выполнить следующую команду:

```
$ kubectl create -f pod.yaml
```

Результат будет эквивалентен результату выполнения императивной команды.

Каждый YAML-файл, созданный для системы Kubernetes, должен содержать четыре обязательных ключа.

- `apiVersion` — указывает версию API, в которой описан ресурс. Для каждого типа ресурса задается свой ключ `apiVersion`. Например, ресурс типа Pod относится к версии API `v1`.
- `kind` — определяет тип ресурса, создаваемого с помощью YAML-файла. В нашем случае это `Pod`.
- `metadata` — содержит сведения о самом ресурсе, включая его имя. В данном случае под называется `my-pod`. Эти метаданные предназначены для системы Kubernetes, а не для контейнерных движков вроде Docker Engine или Podman.
- `spec` — описывает состав объекта. В приведенном примере под состоит из одного контейнера `busybox-container`, созданного на основе образа `busybox:latest`. Именно эти контейнеры будут запущены в среде выполнения контейнеров на вычислительном узле.

Еще одна важная особенность декларативного синтаксиса состоит в том, что в одном YAML-файле можно описать несколько ресурсов, разделяя их тремя дефисами (`---`). Вот обновленная версия файла, определяющая сразу два пода:

```
apiVersion: v1
kind: Pod
metadata:
  name: my-pod
spec:
  containers:
    -  name: busybox-container
       image: busybox:latest
---
apiVersion: v1
kind: Pod
metadata:
  name: my-second-pod
spec:
  containers:
    - name: nginx-container
      image: nginx:latest
```

Вы сможете прочитать этот файл и без труда понять его содержимое. Как видите, он просто создает два пода. Первый использует образ `busybox`, а второй — `nginx`.

Разумеется, запоминать весь синтаксис и значения для каждого ключа нет необходимости. Вы всегда можете найти примеры декларативных YAML-файлов в официальной документации Kubernetes. Если ее окажется недостаточно, вы можете воспользоваться командой `kubectl explain`, чтобы получить подробное описание ресурса, например:

```
$ kubectl explain pod.spec.containers.image
KIND:       Pod
VERSION:    v1

FIELD: image <string>
DESCRIPTION:
    Container image name. More info:
    https://kubernetes.io/docs/concepts/containers/images
```

Вывод команды kubectl explain содержит подробное описание всех полей конфигурационного файла.

Несмотря на то что написание YAML-файлов занимает больше времени, чем выполнение императивной команды, декларативный синтаксис имеет два существенных преимущества.

- *Управление инфраструктурой как кодом (IaC)* — вы можете хранить конфигурацию в отдельном репозитории Git и версионировать ресурсы Kubernetes. В случае утраты состояния кластера наличие версионированных YAML-файлов в репозитории Git позволит быстро и корректно его восстановить.
- *Создание нескольких ресурсов одновременно* — поскольку в одном YAML-файле можно объявить несколько ресурсов, вы можете хранить в нем полное описание приложения и всех его зависимостей. Это позволяет создавать и воссоздавать сложные приложения одной командой. Позднее вы познакомитесь с инструментом Helm, который реализует механизм шаблонизации YAML-файлов Kubernetes.

Ни один из описанных выше способов использования утилиты `kubectl` не является *лучшим*. Это два разных подхода к взаимодействию с ним, которые следует освоить. Дело в том, что одни функции недоступны при работе в императивном стиле, а другие — в декларативном. В конечном счете оба подхода предполагают взаимодействие с компонентом `kube-apiserver` через протокол HTTP.

Утилита `kubectl` должна быть установлена на любой машине, с помощью которой планируется взаимодействие с кластером.

С технической точки зрения утилиту командной строки `kubectl` необходимо установить и настроить везде, где вы собираетесь работать с кластером Kubernetes.

Это может быть как ваш локальный компьютер, так и сервер, через который вы получаете доступ к кластеру Kubernetes. Однако в рамках масштабных проектов `kubectl` также имеет смысл устанавливать на агентах/раннерах платформы непрерывной интеграции.

На практике нередко возникает необходимость автоматизировать процессы обслуживания или развертывания, выполняемые в кластере Kubernetes, и для этого часто используются такие платформы *непрерывной интеграции* (continuous integration, CI), как GitLab CI, Tekton или Jenkins.

Если вы собираетесь выполнять команды Kubernetes в рамках CI-конвейера, вам необходимо установить утилиту `kubectl` на CI-агенты и разместить в их файловой системе корректно настроенный файл `kubeconfig`. Это позволит вашим CI/CD-конвейерам отдавать команды кластеру Kubernetes и изменять его состояние.

Таким образом, `kubectl` не следует рассматривать в качестве клиента Kubernetes, предназначенного исключительно для использования *человеком*. Это универсальный инструмент для взаимодействия с системой Kubernetes, который следует устанавливать в любом месте, где требуется обеспечить связь с кластером.

Обеспечение высокой доступности Kubernetes

Как уже было показано, система Kubernetes представляет собой решение для организации кластеров. Благодаря своей распределенной архитектуре она может работать на множестве машин. Распределив компоненты между несколькими машинами, можно добиться высокой доступности кластера Kubernetes. Далее мы кратко обсудим различные варианты развертывания этой системы.

Одноузловой кластер

Размещение всех компонентов Kubernetes на одной машине — наименее предпочтительный вариант развертывания системы в рабочей среде. Однако для целей тестирования и разработки он вполне приемлем. Одноузловая схема предполагает объединение всех компонентов Kubernetes на одном хосте или виртуальной машине (рис. 2.9).

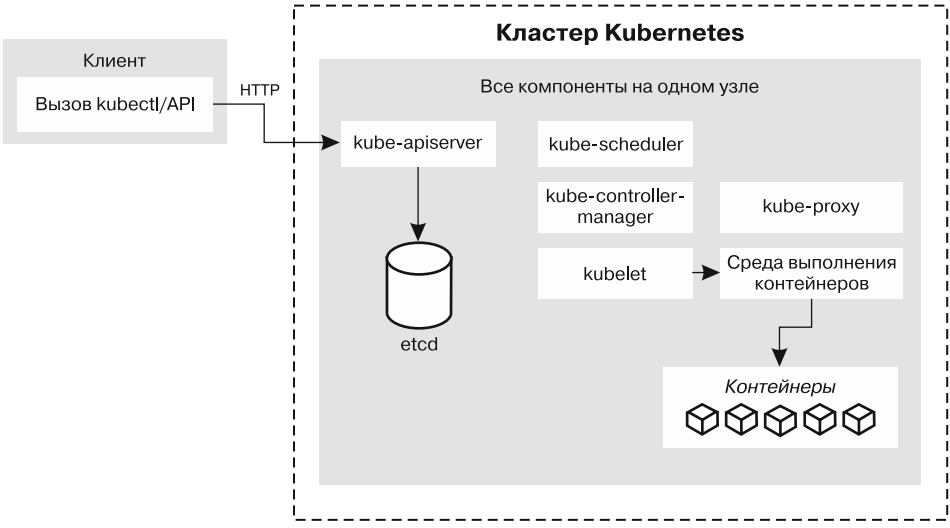

Рис. 2.9. Все компоненты работают на одной машине

Обычно такой подход рассматривается в качестве отправной точки для знакомства с Kubernetes через локальное тестирование. Существует инструмент `minikube`, который упрощает развертывание одноузлового кластера Kubernetes на вашем компьютере. Он запускает виртуальную машину со всеми необходимыми компонентами, уже настроенными для работы. Хотя `minikube` полезен для локального тестирования и теоретически может быть запущен в виде многоузлового кластера, его категорически не рекомендуется использовать в рабочей среде. В табл. 2.2 перечислены основные преимущества и недостатки одноузловых кластеров Kubernetes.

Таблица 2.2. Преимущества и недостатки одноузловых кластеров Kubernetes

Преимущества	Недостатки
Хорошо подходит для тестирования	Невозможно масштабировать
Простая локальная установка	Не отличается высокой доступностью
Поддерживается инструментом `minikube`	Не рекомендуется использовать в рабочей среде

Одноузловой кластер Kubernetes хорошо подходит для периферийных сред с ограниченными ресурсами. Он имеет небольшие системные требования, но при этом обеспечивает надежное развертывание и реализацию стратегий аварийного восстановления.

Кластер с одним управляющим узлом

В данной конфигурации все компоненты управляющего слоя работают на одном узле, при этом вычислительных узлов может быть сколько угодно (рис. 2.10).

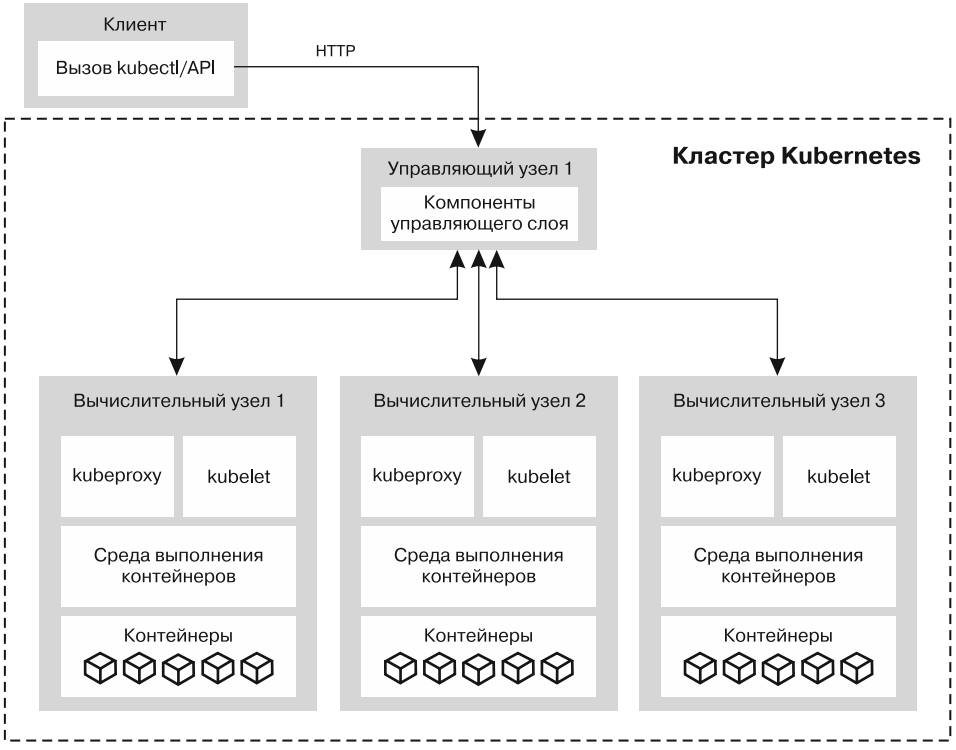

Рис. 2.10. Один управляющий узел управляет всеми вычислительными узлами (в данном примере — тремя)

Эта конфигурация существенно превосходит одноузловые кластеры, а наличие нескольких вычислительных узлов обеспечивает высокую доступность контейнеризованных приложений. Тем не менее у нее есть и недостатки (табл. 2.3).

- Использование только одного управляющего узла создает единую точку отказа. В случае его сбоя вы не сможете управлять запущенными контейнерами средствами Kubernetes. Единственным способом остановки или обновления этих «осиротевших» контейнеров будет подключение к вычислительному узлу по протоколу SSH и выполнение обычных команд управления контейнерами (например, `ctr`, `crictl` или команд Docker — в зависимости от используемой среды выполнения контейнеров).

- При использовании одного экземпляра `etcd` существует высокий риск потери данных в результате повреждения управляющего узла. В таком случае восстановить кластер будет невозможно.
- Наконец, при масштабировании вычислительных узлов в кластере может возникнуть еще одна проблема. У каждого вычислительного узла есть свой агент `kubelet`, который обращается к компоненту `kube-apiserver` с интервалом 20 секунд. Добавление десятков серверов может негативно повлиять на доступность `kube-apiserver` и привести к сбою работы управляющего слоя. Помните о том, что ваш управляющий слой должен иметь возможность масштабироваться и обрабатывать такой трафик.

Таблица 2.3. Преимущества и недостатки кластера Kubernetes с одним управляющим узлом и несколькими вычислительными узлами

Преимущества	Недостатки
Вычислительные узлы обладают высокой доступностью	Управляющий узел является единой точкой отказа
Поддерживает функции многоузловой конфигурации	Использует единственный экземпляр `etcd`
Допускает локальное развертывание с помощью таких инструментов, как `kind` или `minikube`, но этот вариант неидеален	Не обеспечивает эффективного масштабирования

В целом такая конфигурация всегда будет превосходить одноузловой кластер Kubernetes, но она по-прежнему не обеспечивает высокой доступности.

Кластер с несколькими управляющими узлами

Это наилучший способ обеспечить высокую доступность кластера Kubernetes. Данная конфигурация предполагает репликацию запущенных контейнеров и компонентов управляющего слоя, что устраняет единую точку отказа (рис. 2.11).

Используя такой кластер, вы устраняете многие риски, свойственные описанным ранее архитектурам, поскольку запускаете несколько экземпляров как вычислительных, так и управляющих узлов. Для равномерного распределения нагрузки между всеми экземплярами `kube-apiserver` потребуется установить балансировщик нагрузки, что потребует чуть более тщательного планирования. Облачные провайдеры, такие как Amazon EKS и Google GKE, предоставляют кластеры Kubernetes с несколькими управляющими и несколькими вычислительными узлами. Если вы хотите пойти еще дальше, то можете разместить компоненты управляющего слоя на выделенных хостах. Этот вариант является предпочтительным, но не обязательным. Представленного на предыдущей схеме кластера будет вполне достаточно.

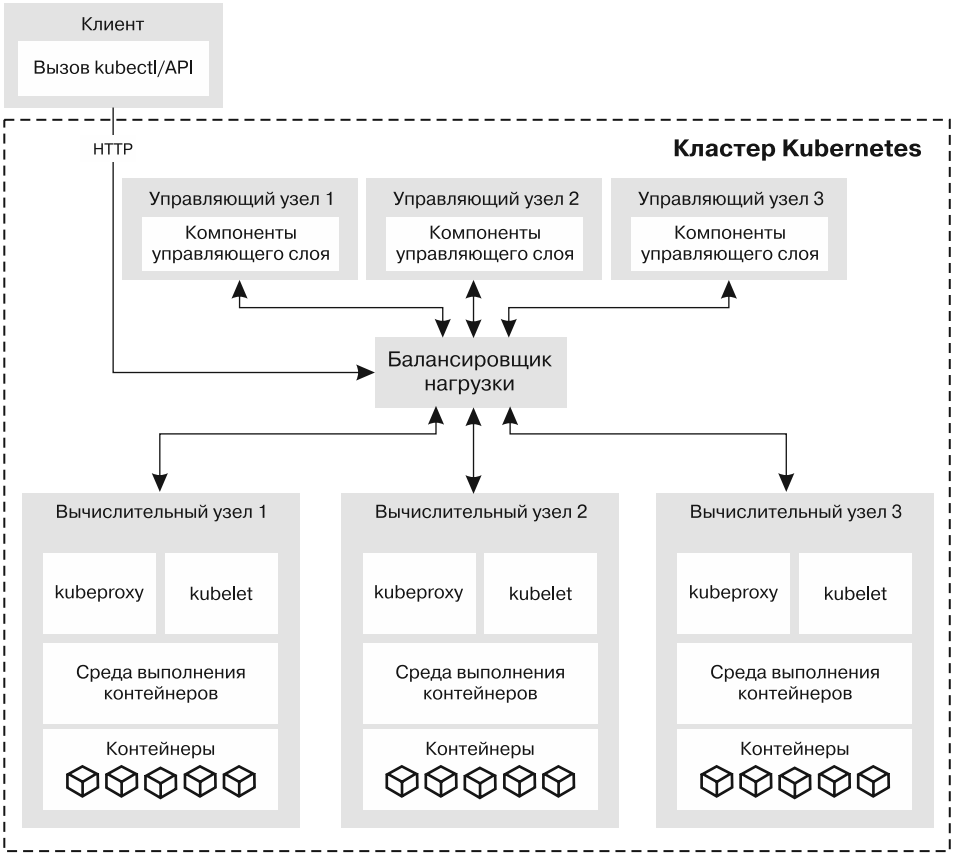

Рис. 2.11. Кластер Kubernetes с несколькими управляющими узлами

Управление компонентом etcd в кластере Kubernetes с несколькими управляющими узлами

В кластере с несколькими управляющими узлами на каждом таком узле запускается собственный экземпляр `etcd`. Это обеспечивает высокую доступность хранилища `etcd` даже в случае недоступности части управляющих узлов. Экземпляры `etcd` в таком кластере Kubernetes образуют собственный внутренний кластер. Это означает, что они обмениваются данными между собой для репликации состояния кластера и обеспечения согласованности данных во всех экземплярах хранилища. Кластер `etcd` использует алгоритм консенсуса Raft, который обеспечивает наличие единственного лидера в каждый момент времени. Лидер отвечает за прием запросов на изменение состояния кластера

и репликацию этих изменений в других экземплярах. Если лидер становится недоступен, оставшиеся экземпляры выбирают нового лидера.

Управление членами кластера `etcd`, а также механизмы резервного копирования и восстановления данных этого хранилища мы обсудим в последующих главах книги.

В завершение этой главы еще раз перечислим все компоненты системы Kubernetes. Таблица 2.4 поможет вам запомнить их назначение.

Таблица 2.4. Компоненты системы Kubernetes и их взаимодействие

Название компонента	С чем взаимодействует	Роль
`kube-apiserver`	Клиенты `kubectl`, `etcd`, `kube-scheduler`, `kube-controller-manager`, `kubelet`, `kube-proxy`	Выступает в качестве REST API, работающего по протоколу HTTP. Читает и записывает состояние, хранящееся в `etcd`. Единственный компонент, способный взаимодействовать с `etcd` напрямую
`etcd`	`kube-apiserver`	Хранит состояние кластера Kubernetes
`kube-scheduler`	`kube-apiserver`	Каждые 20 секунд опрашивает API, чтобы получить список незапланированных подов (подов с пустым свойством `nodeName`), выбирает вычислительный узел, а затем обновляет свойство `nodeName` в описании пода, вызывая `kube-apiserver`
`kube-controller-manager`	`kube-apiserver`	Опрашивает API и выполняет циклы согласования
`kubelet`	`kube-apiserver` и среда выполнения контейнеров	Каждые 20 секунд опрашивает API, чтобы получить поды, назначенные данному узлу, и преобразует их спецификации в запущенные контейнеры, инициируя операции в локальной среде выполнения контейнеров
`kube-proxy`	`kube-apiserver`	Обеспечивает работу сетевого слоя Kubernetes
Контейнерный движок	`kubelet`	Запускает контейнеры, получая инструкции от локального компонента `kubelet`

Эти компоненты являются стандартными и официально поддерживаются в рамках проекта Kubernetes. Помните, что другие дистрибутивы Kubernetes могут добавлять дополнительные компоненты или изменять поведение уже имеющихся.

Перечисленные компоненты составляют минимальный набор, необходимый для создания функционирующего кластера Kubernetes.

Резюме

Глава получилась довольно объемной, но теперь у вас есть представление обо всех компонентах Kubernetes. Все, что мы будем делать дальше, будет связано именно с ними, так как они составляют ядро данной системы. Несмотря на обилие технических подробностей, материал этой главы остался в большей степени теоретическим. Не переживайте, если что-то кажется непонятным, — понимание придет с практикой.

Хорошая новость заключается в том, что теперь вы готовы установить свой первый кластер Kubernetes в локальной среде и приступить к практической работе. Именно этим мы и займемся в следующей главе. После ее изучения на вашей рабочей станции будет развернут локальный кластер Kubernetes и вы сможете запустить в нем свои первые поды.

Дополнительные источники информации

Для более глубокого изучения тем, рассмотренных в этой главе, обратитесь к следующим ресурсам.

- Компоненты Kubernetes: https://kubernetes.io/docs/concepts/overview/components/.
- Компонент `cloud-controller-manager`: https://kubernetes.io/docs/concepts/architecture/cloud-controller/.
- Установка утилит Kubernetes (`kubectl`, `kind`, `kubeadm` и `minikube`): https://kubernetes.io/docs/tasks/tools/.
- Объявление об устаревших репозиториях пакетов Kubernetes: https://kubernetes.io/blog/2023/08/31/legacy-package-repository-deprecation/.
- Представление репозиториев пакетов Kubernetes `pkgs.k8s.io`: https://kubernetes.io/blog/2023/08/15/pkgs-k8s-io-introduction/.
- Эксплуатация `etcd`-кластеров в среде Kubernetes: https://kubernetes.io/docs/tasks/administer-cluster/configure-upgrade-etcd/.
- Функция автодополнения команд `kubectl`: https://kubernetes.io/docs/reference/kubectl/generated/kubectl_completion/.
- Ресурс RuntimeClass: https://kubernetes.io/docs/concepts/containers/runtime-class/.

3 Установка первого кластера Kubernetes

В предыдущей главе мы обсудили систему Kubernetes, ее распределенную архитектуру, устройство кластера и процесс управления контейнерами Docker, запущенными на нескольких машинах с Linux. Теперь пришло время перейти от теории к практике. В этой главе развернем рабочую установку Kubernetes, которая понадобится нам в последующих главах. В результате у вас появится собственный кластер для экспериментов, отработки навыков и изучения материала данной книги.

Установка Kubernetes предполагает настройку взаимодействия различных компонентов этой системы. Разумеется, мы не пойдем сложным путем с ручной настройкой каждого элемента кластера; вместо этого воспользуемся автоматизированными инструментами, позволяющими запустить и сконфигурировать все необходимые компоненты локально, без дополнительных усилий. Такой подход особенно актуален для команд DevOps, которым нужно быстро тестировать изменения в YAML-манифестах, для разработчиков, нуждающихся в локальной среде для проверки работоспособности приложений, а также для специалистов по безопасности, оперативно тестирующих различные YAML-определения объектов Kubernetes.

На тот случай, если вы не хотите устанавливать кластер Kubernetes на локальной машине, в последующих главах мы также создадим минималистичные, но полнофункциональные кластеры на базе *Google Kubernetes Engine (GKE)*, *Amazon Elastic Kubernetes* Service *(EKS)* и *Azure Kubernetes Service (AKS)*. Это облачные решения производственного уровня, позволяющие экспериментировать с реальным кластером Kubernetes, размещенным в облаке.

Выбор между локальным и облачным решением остается за вами. Взвесьте преимущества и недостатки каждого варианта и выберите тот, который лучше всего вам подходит. В любом случае для взаимодействия с кластером Kubernetes вам потребуется установленная на локальной машине утилита `kubectl`. Инструкции по ее установке можно найти в предыдущей главе.

Технические требования

Чтобы выполнять примеры из этой главы, вам потребуется следующее.

- Установленная на локальной машине утилита `kubectl`.
- Рабочая станция с не менее чем двумя процессорами, 2 Гбайт оперативной памяти и 20 Гбайт свободного места на диске (для работы с многоузловыми кластерами потребуется больше ресурсов).
- Установленный на рабочей станции менеджер контейнеров или виртуальных машин, например Docker, QEMU, Hyperkit, Hyper-V, KVM, Parallels, Podman, VirtualBox или VMware Fusion/Workstation.
- Надежное подключение к Интернету.

Примеры кода для этой главы можно загрузить из официального репозитория GitHub по адресу https://github.com/PacktPublishing/The-Kubernetes-Bible-Second-Edition/tree/main/Chapter03.

Установка кластера Kubernetes с помощью minikube

В этом разделе мы рассмотрим процесс установки локального кластера Kubernetes с помощью `minikube`, который является, пожалуй, простейшим способом его развертывания. К концу этого раздела на вашей локальной машине будет установлен одноузловой кластер Kubernetes, готовый к работе.

Инструмент `minikube` прост в использовании и полностью бесплатен. Он установит все необходимые компоненты Kubernetes на вашей локальной машине и настроит их. Удаление этих компонентов через `minikube` тоже не вызывает сложностей, поэтому вы легко сможете уничтожить локальный кластер, когда в нем отпадет потребность.

Главное преимущество `minikube` перед полноценными методами развертывания кластера производственного уровня заключается в том, что этот инструмент позволяет быстро тестировать различные сценарии работы Kubernetes. Если вы не хотите использовать `minikube`, то можете пропустить этот раздел и воспользоваться одним из других способов, описанных в этой главе.

Хотя `minikube` является популярным решением для локальной разработки под Kubernetes, он уступает полноценным кластерам производственного уровня в плане эффективности использования ресурсов и функциональности.

- *Высокая нагрузка на ресурсы.* Запуск `minikube` параллельно с другими процессами на локальной машине требует большого объема ресурсов процессора и памяти, особенно при создании более крупных кластеров Kubernetes, что может негативно повлиять на работу других приложений.

- *Отличия в сетевой конфигурации*. В отличие от кластера Kubernetes производственного уровня `minikube` по умолчанию использует упрощенную сетевую модель, которая не полностью воспроизводит реальные условия. Это может усложнить воспроизведение и диагностику сетевых проблем, возникающих в рабочей среде.
- *Ограничения совместимости*. Некоторые функции Kubernetes и сторонние инструменты могут требовать более полной конфигурации кластера, чем та, что предоставляет `minikube`, что может вызывать проблемы совместимости на этапе разработки.
- *Сложности с постоянным хранилищем*. По сравнению с полноценным кластером Kubernetes `minikube` имеет ограниченную поддержку постоянных томов, что усложняет управление постоянным хранилищем для приложений.

В следующем подразделе вы узнаете о том, как установить `minikube` и развернуть кластер Kubernetes.

Установка инструмента minikube

В этом подразделе мы рассмотрим способы установки инструмента `minikube` в Linux, macOS и Windows. Как вы увидите далее, установка `minikube` с помощью двоичного файла или пакетного менеджера осуществляется достаточно просто.

Вы можете установить minikube с помощью встроенного пакетного менеджера, такого как apt-get, yum, Zypper, Homebrew (для macOS) или Chocolatey (для Windows). Более подробную информацию можно найти в документации: https://minikube.sigs.k8s.io/docs/start.

Установка minikube в Linux

В Linux инструмент `minikube` можно установить с помощью пакета Debian, RPM-пакета или двоичного файла:

```
$ curl -LO https://storage.googleapis.com/minikube/releases/latest/
minikube-linux-amd64
$ sudo install minikube-linux-amd64 /usr/local/bin/minikube

# Проверка доступности команды minikube и корректности пути
$ which minikube
/usr/local/bin/minikube
```

Учтите, что путь может различаться в зависимости от операционной системы вашей рабочей станции. Чтобы команда `minikube` выполнялась без ошибок, необходимо убедиться в том, что этот путь добавлен в переменную окружения PATH.

Установка minikube в macOS

В macOS `minikube` можно установить с помощью двоичного файла[1]:

```
$ curl -LO https://storage.googleapis.com/minikube/releases/latest/
minikube-darwin-amd64
$ sudo install minikube-darwin-amd64 /usr/local/bin/minikube
# Проверка доступности команды minikube и корректности пути
$ which minikube
```

Установить `minikube` в macOS также можно с помощью пакетного менеджера Homebrew.

Установка minikube в Windows

Как и в случае с macOS и Linux, установить `minikube` в Windows можно несколькими способами:

```
# С помощью Windows Package Manager (если установлен)
$ winget install minikube
# С помощью Chocolatey
$ choco install minikube
# Скачав файл .exe и настроив переменную PATH
# 1. Скачайте minikube по ссылке:
# https://storage.googleapis.com/minikube/releases/latest/
minikube-installer.exe
# 2. Настройте переменную PATH
```

После установки и настройки `minikube` вы сможете создавать различные типы кластеров Kubernetes.

Конфигурации minikube

Утилита `minikube` предусматривает минимальный, но достаточный для полноценной разработки набор параметров.

Например, создаваемый с помощью этой утилиты кластер Kubernetes по умолчанию будет иметь два процессора и 2 Гбайт памяти. При необходимости эти значения можно изменить:

```
$ minikube config set cpus 4
! These changes will take effect upon a minikube delete and then a minikube start
$ minikube config set memory 16000
! These changes will take effect upon a minikube delete and then a minikube start
$ minikube config set container-runtime containerd
! These changes will take effect upon a minikube delete and then a minikube start
```

[1] Здесь и далее стоит обращать внимание на архитектуру процессора, заменяя amd64 на arm64 для macOS на Apple Silicon.

Как видите, для применения новых параметров требуется удалить и заново создать кластер `minikube`.

Драйверы для minikube

Инструмент `minikube` позволяет с легкостью запустить локальный кластер Kubernetes на машине для разработки. Для этого он использует *драйверы* — ключевой механизм, управляющий жизненным циклом кластера. Драйверы взаимодействуют с различными технологиями виртуализации и контейнеризации, позволяя `minikube` создавать, настраивать и контролировать инфраструктуру, лежащую в основе локальной среды Kubernetes. Гибкость драйверов `minikube` позволяет развернуть кластер в виде виртуальной машины и контейнера, а также непосредственно на «голом железе» рабочей станции, адаптируя установку под конкретные задачи и предпочтения.

- *Драйверы контейнеров.* При контейнерном развертывании `minikube` может использовать локальные установки Podman или Docker. Это позволяет запускать `minikube` непосредственно внутри контейнера на рабочей машине, что делает установку более легковесной и эффективной с точки зрения потребления ресурсов.
- *Драйверы виртуальных машин (ВМ).* Если вы предпочитаете вариант с виртуальными машинами, то можете использовать `minikube` для запуска ВМ с необходимыми компонентами Kubernetes на вашей рабочей станции. Этот подход обеспечивает более изолированное окружение для локального кластера.

Выбор между драйверами контейнеров и виртуальных машин зависит от ваших задач, предпочтений и возможностей среды разработки.

Подробнее о драйверах `minikube` и поддерживаемых операционных системах можно узнать в документации: https://minikube.sigs.k8s.io/docs/drivers/.

Для `minikube` можно задать драйвер по умолчанию с помощью следующей команды:

```
$ minikube config set driver docker
! These changes will take effect upon a minikube delete and then a minikube start
# или указать VirtualBox в качестве драйвера
$ minikube config set driver virtualbox
! These changes will take effect upon a minikube delete and then a minikube start
$ minikube config view driver
- driver: docker
```

Мы также можем указать драйвер непосредственно при создании кластера `minikube`:

```
$ minikube start --driver=docker
```

Предварительные требования зависят от конкретного драйвера `minikube` и должны быть выполнены заранее. К ним может относиться установка Docker, Podman или VirtualBox, а также предоставление необходимых прав в используемой операционной системе. Инструкции по установке и настройке приведены в документации, посвященной конкретным драйверам `minikube`: https://minikube.sigs.k8s.io/docs/drivers.

В следующем подразделе мы развернем первый кластер Kubernetes с помощью `minikube`.

Создание одноузлового кластера Kubernetes с помощью minikube

Основное назначение `minikube` — запуск компонентов Kubernetes в локальной системе и организация их взаимодействия. В следующих подразделах мы рассмотрим процесс развертывания кластеров `minikube` с помощью драйвера VirtualBox и Docker.

Настройка minikube с использованием виртуальных машин

Метод, основанный на использовании виртуальных машин, требует предварительной установки гипервизора на рабочую станцию:

- Linux — KVM2 (предпочтительно), VirtualBox, QEMU;
- Windows — Hyper-V (предпочтительно), VirtualBox, VMware Workstation, QEMU;
- macOS — Hyperkit, VirtualBox, Parallels, VMware Fusion, QEMU.

После этого `minikube` упакует все компоненты Kubernetes в виртуальную машину и запустит ее.

В следующем примере в качестве рабочей станции используется Fedora 39, а в качестве гипервизора — VirtualBox, так как он доступен для Linux, macOS и Windows.

Скачать VirtualBox для вашей рабочей станции можно по адресу https://www.virtualbox.org/wiki/Downloads. Вы также можете выбрать другое программное обеспечение для виртуализации, однако перед этим уточните поддерживаемые решения в документации: https://minikube.sigs.k8s.io/docs/drivers/.

Не путайте версию minikube и версию развернутой системы Kubernetes. Например, minikube 1.32 использует Kubernetes 1.28 по соображениям стабильности и совместимости. Это обеспечивает более тщательное тестирование, более широкую поддержку инструментов, контролируемое внедрение обновлений и более долгосрочную поддержку старых версий. При этом пользователи сохраняют возможность отдельно запускать другие версии Kubernetes. Такой баланс между стабильностью и гибкостью делает minikube надежной и универсальной платформой для разработчиков.

На рабочей станции, где установлены `minikube` и VirtualBox, выполните следующую команду:

```
$ minikube start --driver=virtualbox --memory=8000m --cpus=2
```

Если вы используете определенную версию `minikube`, но хотите установить другую версию Kubernetes, то можете указать ее явно, например, так:

```
$ minikube start --driver=virtualbox --memory=8000m --cpus=2 \
--kubernetes-version=1.29.0
```

После этого `minikube` запустит процесс развертывания Kubernetes, включая загрузку образа виртуальной машины:

```
😄  minikube v1.32.0 on Fedora 39
❗  Specified Kubernetes version 1.29.0 is newer than the newest supported
version: v1.28.3. Use `minikube config defaults kubernetes-version`
for details.
❗  Specified Kubernetes version 1.29.0 not found in Kubernetes version list
🤔  Searching the internet for Kubernetes version...
☑  Kubernetes version 1.29.0 found in GitHub version list
✨  Using the virtualbox driver based on user configuration
👍  Starting control plane node minikube in cluster minikube
🔥  Creating virtualbox VM (CPUs=2, Memory=8000MB, Disk=20000MB) ...
🤦  Preparing Kubernetes v1.29.0 on Docker 24.0.7 ...
    ▪ Generating certificates and keys ...
    ▪ Booting up control plane ...
    ▪ Configuring RBAC rules ...
🔗  Configuring bridge CNI (Container Networking Interface) ...
    ▪ Using image gcr.io/k8s-minikube/storage-provisioner:v5
```

В зависимости от операционной системы вашей рабочей станции и выбранного программного обеспечения для виртуализации вы также можете увидеть дополнительные рекомендации:

```
|    You have selected "virtualbox" driver, but there are better options!
|    For better performance and support consider using
|    a different driver:
|            - kvm2
|            - qemu2
|
|    To turn off this warning run:
|            $ minikube config set WantVirtualBoxDriverWarning false
|    To learn more about minikube drivers checkout
| https://minikube.sigs.k8s.io/docs/drivers/
| To see benchmarks checkout https://minikube.sigs.k8s.io/docs/benchmarks/
| cpuusage/
```

В конце концов вы получите сообщение об успешном завершении установки от `minikube`:

```
🔎  Verifying Kubernetes components...
🌟  Enabled addons: storage-provisioner, default-storageclass
🏄  Done! kubectl is now configured to use "minikube" cluster and "default"
namespace by default
```

Итак, всего за минуту вы развернули рабочий кластер Kubernetes, готовый к запуску приложений.

Теперь проверьте состояние кластера Kubernetes с помощью команды `minikube`:

```
$ minikube status
minikube
type: Control Plane
host: Running
kubelet: Running
apiserver: Running
kubeconfig: Configured
```

Новая виртуальная машина `minikube` должна отобразиться в интерфейсе VirtualBox (рис. 3.1).

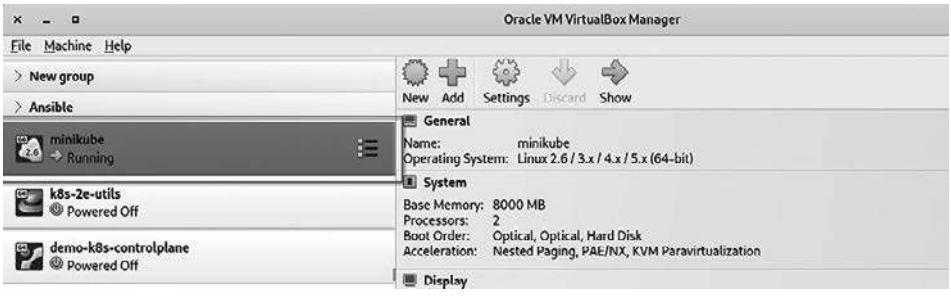

Рис. 3.1. Виртуальная машина minikube в интерфейсе VirtualBox

В следующем подразделе вы узнаете, как развернуть кластеры Kubernetes с использованием `minikube` и контейнеров.

Настройка minikube с использованием контейнера

Контейнерный метод является более простым: вместо виртуальной машины `minikube` использует локальный экземпляр Docker Engine или Podman для запуска компонентов Kubernetes внутри одного большого контейнера. Чтобы воспользоваться этим вариантом, установите Docker или Podman на свою рабочую станцию в соответствии с инструкциями для используемой операционной

системы. Учтите, что `minikube` не сможет установить их за вас. Если нужный драйвер отсутствует в системе или `minikube` его не находит, вы получите следующее сообщение об ошибке:

```
$ minikube start --driver=podman
😄  minikube v1.32.0 on Fedora 39 (hyperv/amd64)
✨  Using the podman driver based on user configuration
🤷  Exiting due to PROVIDER_PODMAN_NOT_FOUND: The 'podman' provider was not
found: exec: "podman": executable file not found in $PATH
```

Процесс установки Docker достаточно прост, однако его конкретные шаги зависят от используемой операционной системы. Подробные инструкции приведены в документации: https://docs.docker.com/engine/install/. Инструкции по установке Podman для различных операционных систем доступны по адресу https://podman.io/docs/installation.

Если вы выполняете упражнения на рабочей станции с Windows и используете виртуальную машину на базе Hyper-V, отключите функцию динамического выделения памяти (Dynamic Memory) для ВМ, на которой устанавливаются minikube и контейнерный движок.

При запуске с драйвером Podman minikube проверяет доступную память и отображает ее используемый объем (задаваемый динамически). Поэтому вам необходимо убедиться, что памяти достаточно, либо явно задать требования к памяти для узла Kubernetes.

В следующем примере в качестве рабочей станции используется Fedora 39, а в качестве контейнерного движка — Docker:

```
$ minikube start --driver=docker --kubernetes-version=1.29.0
😄  minikube v1.32.0 on Fedora 39
❗  Specified Kubernetes version 1.29.0 is newer than the newest supported
version: v1.28.3. Use `minikube config defaults kubernetes-version`
for details.
❗  Specified Kubernetes version 1.29.0 not found in Kubernetes version list
🤔  Searching the internet for Kubernetes version...
☑  Kubernetes version 1.29.0 found in GitHub version list
✨  Using the docker driver based on user configuration
📌  Using Docker driver with root privileges
👍  Starting control plane node minikube in cluster minikube
🚜  Pulling base image ...
🔥  Creating docker container (CPUs=2, Memory=8000MB) ...
🤦  Preparing Kubernetes v1.29.0 on Docker 24.0.7 ...
    ▪ Generating certificates and keys ...
    ▪ Booting up control plane ...
    ▪ Configuring RBAC rules ...
```

```
🔗 Configuring bridge CNI (Container Networking Interface) ...
   ▪ Using image gcr.io/k8s-minikube/storage-provisioner:v5
🔎 Verifying Kubernetes components...
🌟 Enabled addons: storage-provisioner, default-storageclass
🏄 Done! kubectl is now configured to use "minikube" cluster and "default"
namespace by default
```

Мы также можем использовать Podman в качестве контейнерного движка и создать кластер Kubernetes с помощью `minikube`, выполнив следующую команду:

```
$ minikube start --driver=podman
```

Теперь у нас есть кластер Kubernetes, созданный с использованием `minikube`. В следующем подразделе вы узнаете, как получать к нему доступ и управлять им с помощью утилиты `kubectl`.

Получение доступа к кластеру Kubernetes, созданному с помощью minikube

Теперь нам необходимо создать файл `kubeconfig`, чтобы локальная утилита `kubectl` могла взаимодействовать с новой установкой Kubernetes. Хорошая новость в том, что `minikube` сгенерировал его автоматически при выполнении команды `minikube start`. Этот файл `kubeconfig` указывает на локальную конечную точку `kube-apiserver`, и ваша локальная утилита `kubectl` настроена на работу с этим кластером по умолчанию. Иными словами, никаких дополнительных действий выполнять не нужно: файл `kubeconfig` уже имеет правильный формат и находится в нужном месте.

По умолчанию эта конфигурация хранится в файле `~/.kube/config`, и вы можете увидеть, что теперь там присутствует контекст `minikube`:

```
$ cat ~/.kube/config
...<фрагмент опущен для краткости>..
- context:
    cluster: minikube
    extensions:
      - extension:
          last-update: Mon, 03 Jun 2024 13:06:44 +08
          provider: minikube.sigs.k8s.io
          version: v1.33.1
        name: context_info
    namespace: default
    user: minikube
  name: minikube
... <фрагмент опущен для краткости>...
```

Используйте следующую команду, чтобы отобразить текущий файл `kubeconfig`. Вы увидите кластер с именем `minikube`, который указывает на локальный IP-адрес:

```
$ kubectl config view
```

Затем выполните следующую команду, чтобы увидеть кластер Kubernetes, на работу с которым настроена ваша утилита `kubectl`:

```
$ kubectl config current-context
minikube
```

Теперь попробуем выполнить настоящую команду `kubectl`, чтобы вывести список узлов, входящих в кластер `minikube`. Если все пройдет успешно, эта команда обратится к компоненту `kube-apiserver`, запущенному инструментом `minikube`, и возвратит единственный узел, поскольку по умолчанию `minikube` создает одноузловой кластер. Список узлов можно получить с помощью следующей команды:

```
$ kubectl get nodes
NAME       STATUS   ROLES           AGE     VERSION
minikube   Ready    control-plane   3m52s   v1.29.0
```

Если при выполнении этой команды вы не увидите сообщений об ошибках, значит, ваш кластер `minikube` готов к работе и функционирует корректно!

Итак, вы выполнили свою первую настоящую команду `kubectl`. В данном случае реальный компонент `kube-apiserver` получил ваш API-запрос и прислал HTTP-ответ с данными из реального хранилища `etcd`. В нашем примере это был список узлов кластера.

Поскольку minikube по умолчанию создает одноузловой кластер Kubernetes, эта команда возвратит данные только об одном узле, который одновременно является и управляющим, и вычислительным. Такая конфигурация подходит для локального тестирования, но не рекомендуется для развертывания в рабочей среде.

Теперь посмотрим статусы компонентов управляющего слоя, чтобы лучше познакомиться с утилитой `kubectl`:

```
$ kubectl get componentstatuses
Warning: v1 ComponentStatus is deprecated in v1.19+
NAME                 STATUS    MESSAGE   ERROR
controller-manager   Healthy   ok
scheduler            Healthy   ok
etcd-0               Healthy   ok
```

В результате выполнения этой команды вы должны увидеть следующее:

- работающее хранилище данных `etcd`;
- работающий компонент `kube-scheduler`;
- работающий компонент `kube-controller-manager`.

В следующем подразделе мы поговорим об управлении учебной средой Kubernetes через остановку и удаление кластера `minikube`.

Остановка и удаление локального кластера minikube

Если вам требуется остановить или удалить локальную установку `minikube`, не стоит завершать работу виртуальной машины или контейнера. Вместо этого лучше использовать утилиту командной строки `minikube`:

```
$ minikube stop
✋  Stopping node "minikube" ...
🛑  1 node stopped.
```

Данная команда останавливает кластер. Однако он продолжает существовать: его состояние сохраняется, и позднее вы сможете возобновить его работу, выполнив команду `minikube start`. Проверить его состояние можно с помощью команды `minikube status`:

```
$ minikube status
minikube
type: Control Plane
host: Stopped
kubelet: Stopped
apiserver: Stopped
kubeconfig: Stopped
```

Мы также можем приостановить работу кластера, чтобы иметь возможность быстро ее возобновить:

```
$ minikube pause
⏸      Pausing node minikube ...
⏯      Paused 14 containers in: kube-system, kubernetes-dashboard,
storagegluster, istio-operator

$ minikube status
minikube
type: Control Plane
host: Running
kubelet: Stopped
apiserver: Paused
kubeconfig: Configured
```

Для возобновления работы кластера можно воспользоваться следующей командой:

```
$ minikube unpause
```

Если вы хотите полностью уничтожить кластер, выполните команду:

```
$ minikube delete
🔥  Deleting "minikube" in docker ...
🔥  Deleting container "minikube" ...
🔥  Removing /home/gmadappa/.minikube/machines/minikube ...
💀  Removed all traces of the "minikube" cluster.
```

В результате ее выполнения кластер будет полностью удален. Его состояние будет потеряно, и восстановить его будет невозможно.

Теперь, когда ваш кластер `minikube` настроен и готов к работе, вы можете использовать его для прохождения следующих глав или выбрать другое решение.

Создание многоузлового кластера Kubernetes с использованием minikube

С помощью `minikube` также можно создавать многоузловые кластеры Kubernetes. Следующая команда позволяет развернуть кластер, состоящий из трех узлов:

```
$ minikube start --driver=podman --nodes=3
```

При создании многоузловых кластеров важно убедиться в том, что ваша рабочая станция имеет достаточно ресурсов для запуска нескольких узлов Kubernetes (будь то виртуальные машины или контейнеры). Учтите также, что minikube запускает узлы с теми же параметрами vCPU и памяти, которые вы указали в настройках или аргументах.

После создания кластера проверьте сведения об узлах следующим образом:

```
$ kubectl get nodes
NAME           STATUS   ROLES           AGE     VERSION
minikube       Ready    control-plane   93s     v1.29.0
minikube-m02   Ready    <none>          74s     v1.29.0
minikube-m03   Ready    <none>          54s     v1.29.0
```

В данном случае инструмент `minikube` создал кластер из трех узлов (`--nodes=3`), первый из которых выполняет функции управляющего (`control plane`), а два оставшихся выступают в роли вычислительных (соответствующие метки мы назначим в одной из следующих глав).

Создание кластера Kubernetes с несколькими управляющими узлами с помощью minikube

Иногда возникает необходимость развернуть и протестировать кластер Kubernetes с отказоустойчивым управляющим слоем, включающим несколько управляющих узлов. Это можно реализовать с помощью следующей команды `minikube`:

```
$ minikube start \
  --driver=virtualbox \
  --nodes=5 \
  --ha=true \
  --cni=calico \
  --cpus=2 \
  --memory=2g \
  --kubernetes-version=v1.30.0 \
  --container-runtime=containerd
$ kubectl get nodes
NAME           STATUS   ROLES           AGE     VERSION
minikube       Ready    control-plane   6m28s   v1.30.0
minikube-m02   Ready    control-plane   4m36s   v1.30.0
minikube-m03   Ready    control-plane   2m45s   v1.30.0
minikube-m04   Ready    <none>          112s    v1.30.0
minikube-m05   Ready    <none>          62s     v1.30.0
```

В результате `minikube` создаст кластер из пяти узлов (`--nodes 5`) и назначит первым трем роль управляющих (`--ha true`).

В данном случае также важно убедиться в том, что ваша рабочая станция имеет достаточно ресурсов для запуска таких многоузловых кластеров.

Создание нескольких кластеров Kubernetes с помощью minikube

Как мы уже выяснили, инструмент `minikube` предназначен для разработки и тестирования Kubernetes-сред. Иногда у вас может возникнуть необходимость смоделировать окружение с несколькими кластерами Kubernetes. В этом случае можно снова воспользоваться `minikube`, однако не забудьте присвоить этим кластерам разные имена с помощью параметра `--profile`, как показано ниже:

```
# Запуск кластера minikube с драйвером VirtualBox
$ minikube start --driver=virtualbox --kubernetes-version=1.30.0
--profile=cluster-vbox
# Запуск другого кластера minikube с драйвером Docker
$ minikube start --driver=docker --kubernetes-version=1.30.0
--profile=clusterdocker
```

Вы можете вывести список всех кластеров `minikube` и просмотреть их параметры (рис. 3.2).

```
$ minikube profile list
| Profile       | VM Driver  | Runtime    |            IP | Port | Version | Status  | Nodes | Active Profile | Active Kubectx |
|---------------|------------|------------|---------------|------|---------|---------|-------|----------------|----------------|
| cluster-docker| docker     | containerd|  192.168.49.2 | 8443 | v1.30.0 | Running |     1 |                |                |
| cluster-vbox  | virtualbox | containerd| 192.168.59.145| 8443 | v1.30.0 | Running |     1 |                | *              |
```

Рис. 3.2. Список профилей minikube, где мы видим несколько кластеров Kubernetes

```
# Остановка кластера по имени профиля
$ minikube stop --profile=cluster-docker

# Удаление кластера по имени профиля
$ minikube delete --profile=cluster-docker
```

Итак, теперь вы знаете, как создавать кластеры Kubernetes разного типа и размера. Далее мы рассмотрим еще один инструмент для развертывания локального кластера Kubernetes под названием `kind`.

Создание многоузлового кластера Kubernetes с помощью kind

В этом разделе мы рассмотрим инструмент `kind`, который, как и `minikube`, предназначен для запуска локальных кластеров Kubernetes.

Основная идея `kind` заключается в использовании контейнеров Docker или Podman в качестве узлов Kubernetes, которое является возможным благодаря модели *Docker-in-Docker* (DinD, Docker в Docker) или *Containers-in-Container* (Контейнеры в контейнере). Запуская контейнеры, содержащие контейнерный движок и `kubelet`, мы можем заставить их работать в качестве вычислительных узлов Kubernetes.

На следующей схеме представлена высокоуровневая архитектура кластера `kind` (рис. 3.3).

Таким образом, работа `kind` во многом аналогична использованию драйвера Docker с `minikube`, только в том случае задействовано несколько контейнеров вместо одного. Результатом будет локальный многоузловой кластер. Как и `minikube`, `kind` является бесплатным инструментом с открытым исходным кодом.

Как и minikube, инструмент kind предназначен исключительно для локальной разработки и тестирования. Не используйте его в рабочей среде.

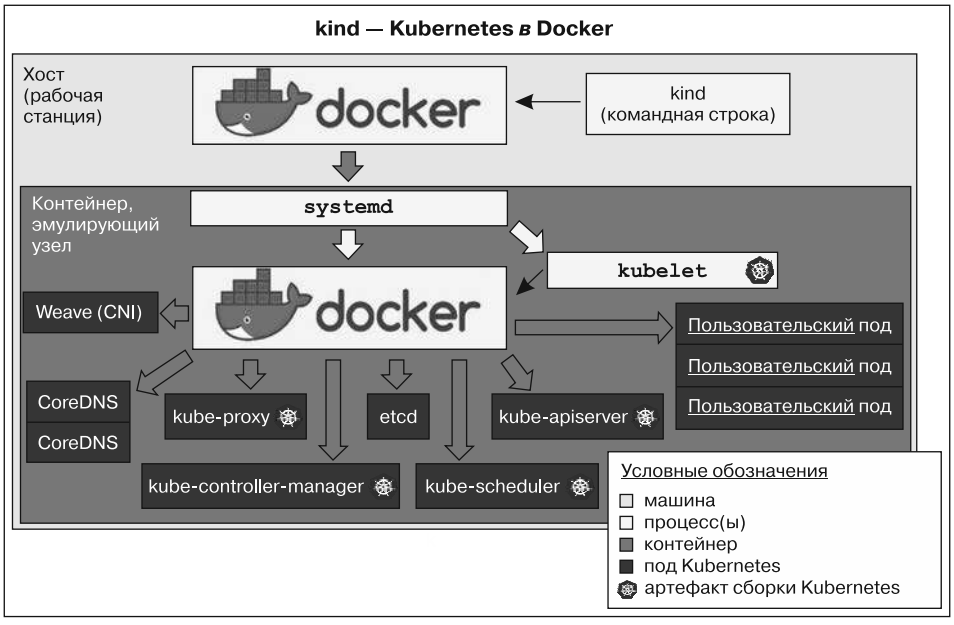

Рис. 3.3. Компоненты кластера kind
(источник изображения: https://kind.sigs.k8s.io/docs/design/initial)

Установка kind в локальную систему

Поскольку инструмент `kind` полностью завязан на Docker и Podman, вам необходимо установить и настроить один из этих контейнерных движков в своей системе.

Ссылки на инструкции по установке Docker и Podman приведены в пункте «Настройка minikube с использованием контейнера» ранее.

С информацией о версиях kind и доступных релизах можно ознакомиться на странице https://github.com/kubernetes-sigs/kind/releases.

Процесс установки инструмента `kind` зависит от используемой операционной системы.

- В Linux выполните следующие команды:

```
$ curl -Lo ./kind https://kind.sigs.k8s.io/dl/v0.23.0/kind-linux-amd64
$ chmod +x ./kind
```

```
$ sudo mv ./kind /usr/local/bin/kind
# Проверка версии
$ kind version
kind v0.23.0 go1.21.10 linux/amd64
```

- В macOS выполните команды:

```
$ curl -Lo ./kind https://kind.sigs.k8s.io/dl/v0.20.0/kind-darwin-amd64
$ chmod +x ./kind
$ mv ./kind /usr/local/bin/kind
```

- Данный инструмент также можно установить с помощью Homebrew:

```
$ brew install kind
```

- В Windows PowerShell выполните следующие команды:

```
$ curl.exe -Lo kind-windows-amd64.exe \
https://kind.sigs.k8s.io/dl/v0.23.0/kind-windows-amd64
$ Move-Item .\kind-windows-amd64.exe c:\some-dir-in-your-PATH\kind.exe
```

- Или воспользуйтесь утилитой Chocolatey:

```
$ choco install kind
```

- Если у вас установлен язык Go, то вы можете выполнить следующую команду:

```
$ go install sigs.k8s.io/kind@v0.22.0 && kind create cluster
```

Альтернативные методы установки описаны в документации: https://kind.sigs.k8s.io/docs/user/quick-start#installation.

В следующем разделе мы рассмотрим процесс создания кластера Kubernetes с помощью `kind`.

Создание кластера Kubernetes с помощью kind

После установки `kind` вы сможете сразу же развернуть кластер Kubernetes, запустив следующую команду:

```
$ kind create cluster --name test-kind
Creating cluster "test-kind" ...
```

При ее выполнении инструмент `kind` приступит к сборке локального кластера Kubernetes, загрузив образ контейнера, содержащий все компоненты

управляющего слоя. В результате будет создан одноузловой кластер Kubernetes, в котором роль *управляющего узла* играет контейнер Docker.

При желании в качестве провайдера для кластера `kind` можно использовать Podman:

```
$ KIND_EXPERIMENTAL_PROVIDER=podman kind create cluster
```

Однако такой вариант нам неинтересен, так как мы можем достичь нужного результата и с помощью `minikube`. В данном случае нужно создать многоузловой кластер на базе `kind`, узлы которого можно настраивать. Для этого необходимо написать небольшой конфигурационный файл и приказать инструменту `kind` использовать его в качестве шаблона при сборке локального кластера Kubernetes. Итак, удалим только что созданный одноузловой кластер `kind` и пересоберем его в виде многоузлового кластера.

1. Чтобы удалить кластер, выполните команду:

   ```
   $ kind delete cluster
   Deleting cluster "kind" ...
   ```

2. Создайте конфигурационный файл, который будет служить шаблоном для сборки кластера `kind`. Для этого скопируйте следующий фрагмент кода в локальный файл текущего каталога, например `~/.kube/kind_cluster`:

   ```
   kind: Cluster
   apiVersion: kind.x-k8s.io/v1alpha4
   nodes:
     - role: control-plane
     - role: worker
     - role: worker
     - role: worker
   ```

   Учтите, что данный файл имеет формат YAML. Важнейшей частью этого файла является массив `nodes`, где указывается нужное количество узлов в кластере. Ключ `role` может принимать два значения: `control-plane` и `worker`.

   В зависимости от выбранной роли будет создан управляющий или вычислительный узел.

3. Теперь снова выполните команду `kind create` с использованием этого конфигурационного файла, чтобы собрать многоузловой кластер. В данном случае результатом будет кластер, состоящий из одного управляющего и трех вычислительных узлов:

   ```
   $ kind create cluster --config ~/.kube/kind_cluster
   ```

Мы также можем использовать определенную версию Kubernetes, указав соответствующие параметры образа при создании кластера `kind`:

```
$ kind create cluster \
  --name=my-kind-cluster \
  --config=~/.kube/kind_cluster \
  --image=kindest/node:v1.29.0\
@sha256:eaa1450915475849a73a9227b8f201df25e55e268e5d619312131292e324d570
```

После выполнения всех этих шагов инструмент `kind` развернет и сконфигурирует новый кластер Kubernetes, а в конце вы получите сообщение о том, как к нему можно подключиться:

```
Creating cluster "my-kind-cluster" ...
✓ Ensuring node image (kindest/node:v1.29.0) 🖼
✓ Preparing nodes 📦 📦 📦
✓ Writing configuration 📜
✓ Starting control-plane 🕹
✓ Installing CNI 🔌
✓ Installing StorageClass 💾
✓ Joining worker nodes 🚜

Set kubectl context to "kind-my-kind-cluster"

You can now use your cluster with:
kubectl cluster-info --context kind-my-kind-cluster

Thanks for using kind! 😊
```

В итоге у вас появится четыре новых контейнера Docker: один будет выступать в роли управляющего узла кластера Kubernetes, а три других — в роли вычислительных узлов.

Теперь нам следовало бы создать файл `kubeconfig`, чтобы утилита `kubectl` могла взаимодействовать с новым кластером. Однако инструмент `kind` уже сгенерировал корректную конфигурацию и добавил ее в файл `~/.kube/config`, а также задал новый кластер в качестве текущего контекста. Таким образом, дополнительных действий от нас не требуется и мы можем сразу приступить к работе.

Выведем список узлов кластера, выполнив команду `kubectl get nodes`. Если все в порядке, вы должны увидеть четыре узла:

```
$ kubectl get nodes
NAME                           STATUS   ROLES           AGE     VERSION
my-kind-cluster-control-plane  Ready    control-plane   4m      v1.29.0
my-kind-cluster-worker         Ready    <none>          3m43s   v1.29.0
my-kind-cluster-worker2        Ready    <none>          3m42s   v1.29.0
my-kind-cluster-worker3        Ready    <none>          3m41s   v1.29.0
```

Судя по всему, кластер `kind` работает как надо.

Как и в случае с `minikube`, вы можете проверить состояние компонентов управляющего слоя с помощью следующей команды:

```
$ kubectl cluster-info
Kubernetes control plane is running at https://127.0.0.1:42547
CoreDNS is running at https://127.0.0.1:42547/api/v1/namespaces/kube-system/
services/kube-dns:dns/proxy
```

Для дополнительной отладки и диагностики проблем в кластере используйте команду `kubectl cluster-info dump`:

```
$ kubectl get --raw='/readyz?verbose'
[+]ping ok
[+]log ok
[+]etcd ok
... <фрагмент опущен для краткости>...
[+]poststarthook/apiservice-openapiv3-controller ok
[+]shutdown ok
readyz check passed
```

В следующем подразделе мы рассмотрим способы остановки и удаления кластеров Kubernetes, созданных с помощью инструмента `kind`.

Остановка и удаление локального кластера kind

Иногда вам может потребоваться остановить или удалить все, что было создано в вашей локальной системе с помощью инструмента `kind`. Для этого можно использовать следующую команду:

```
$ kind stop
```

Эта команда остановит контейнеры Docker, управляемые инструментом `kind`. Того же результата можно добиться, применив команду Docker `stop` к этим контейнерам вручную. В этом случае контейнеры будут остановлены, но состояние кластера сохранится. Это означает, что кластер не будет уничтожен и при повторном запуске вернется в то состояние, в котором находился до остановки.

Чтобы полностью удалить кластер из системы, используйте следующую команду. После ее выполнения кластер и его состояние будут удалены из системы без возможности восстановления:

```
$ kind delete cluster
Deleting cluster "kind" ...
```

Теперь, когда ваш кластер `kind` настроен и готов к работе, вы можете использовать его для прохождения следующих глав или выбрать другое решение, описанное далее в этой главе. Инструмент `kind` особенно удобен тем, что является бесплатным и позволяет развернуть многоузловой кластер. Однако он не предназначен для использования в рабочей среде и остается средством для разработки и тестирования. Инструмент `kind` использует контейнеры Docker для создания *узлов Kubernetes*, которые в реальной среде должны представлять собой машины под управлением Linux.

В следующем разделе мы рассмотрим альтернативные учебные и тестовые среды Kubernetes.

Альтернативные учебные среды для освоения Kubernetes

Вы также можете воспользоваться готовыми учебными средами, которые не требуют настройки и призваны сделать ваше знакомство с системой Kubernetes простым и удобным.

Killercoda

Killercoda (https://killercoda.com/playgrounds/scenario/kubernetes) — это среда, предоставляющая временный кластер Kubernetes, доступный прямо из браузера без необходимости в настройке. Эта платформа оперативно обновляется, предлагая новейшую версию Kubernetes на базе `kubeadm` спустя всего несколько недель после релиза.

Эта среда предлагает следующие возможности.

- Эфемерное окружение: используйте заранее настроенный кластер, который исчезает после завершения работы. Это идеальный вариант для проведения быстрых экспериментов.
- Пустой кластер `kubeadm` с двумя узлами: изучите основные возможности Kubernetes с помощью доступного двухузлового кластера.
- Управляющий узел с возможностью планирования: в отличие от некоторых других сред, эта площадка позволяет планировать запуск рабочих нагрузок прямо на управляющем узле, что обеспечивает дополнительную гибкость при тестировании.

В следующем разделе мы поговорим о кластерах Kubernetes, предназначенных для использования в рабочей среде.

Кластеры Kubernetes производственного уровня

До сих пор мы рассматривали Kubernetes-среды для разработки и обучения. Теперь пришло время обсудить способы создания рабочей среды Kubernetes, отвечающей конкретным требованиям.

В следующем подразделе мы рассмотрим управляемые сервисы Kubernetes, предоставляемые основными *облачными провайдерами*.

Готовые кластеры Kubernetes, предоставляемые облачными сервисами

На тот случай, если вы предпочитаете использовать управляемые сервисы Kubernetes, у вас есть несколько вариантов, среди которых можно выделить следующие.

- *Google Kubernetes Engine (GKE)*. Данный сервис Kubernetes, предоставляемый платформой *Google Cloud Platform (GCP)*, управляет всеми этапами жизненного цикла кластера — от развертывания и настройки до масштабирования и обслуживания. GKE органично интегрируется с другими сервисами GCP, что делает его отличным выбором для пользователей этой платформы.
- *Azure Kubernetes Service (AKS)*. Этот управляемый сервис Kubernetes входит в состав Microsoft Azure. Подобно GKE, он берет на себя все аспекты управления кластером, позволяя вам сосредоточиться на развертывании и эксплуатации контейнерных приложений. AKS тесно интегрируется с другими сервисами Azure, поэтому является естественным выбором для пользователей этой облачной платформы.
- *Amazon Elastic Kubernetes Service (EKS)*. Управляемый сервис Kubernetes, предоставляемый платформой *Amazon Web Services (AWS)*. Подобно GKE и AKS, он упрощает управление кластерами и позволяет сосредоточиться на приложениях. EKS интегрируется с экосистемой сервисов AWS, что делает его оптимальным вариантом для ее пользователей.

Эти управляемые сервисы Kubernetes обеспечивают удобный и масштабируемый способ развертывания и эксплуатации контейнерных приложений без необходимости в самостоятельном администрировании кластеров. Подробно мы поговорим о них в главах 15–17.

Если у вас нет локального кластера Kubernetes (см. предыдущие разделы этой главы), вы можете развернуть его с помощью управляемого сервиса на выбранной облачной платформе, обратившись к соответствующей главе. Имейте в виду, что для перехода ко второй половине книги, посвященной углубленному

изучению ключевых концепций Kubernetes, наличие рабочего кластера является обязательным условием.

В следующем подразделе мы обсудим дистрибутивы и платформы Kubernetes.

Дистрибутивы Kubernetes

Дистрибутивы Kubernetes представляют собой преднастроенные версии системы, которые включают дополнительные возможности, выходящие за рамки базового функционала Kubernetes. Их можно рассматривать как пакеты, ориентированные на конкретные потребности и упрощающие процесс развертывания приложений. Если вам требуется более насыщенный функционал, обратите внимание на следующие дистрибутивы Kubernetes.

- *Red Hat OpenShift* — дистрибутив корпоративного уровня, расширяющий возможности Kubernetes за счет инструментов для разработчиков (средств для сборки образов и CI/CD-конвейеров), функций управления несколькими кластерами, механизмов безопасности (RBAC и SCC), а также встроенной масштабируемости для осуществления сложных развертываний (https://www.redhat.com/en/technologies/cloud-computing/openshift).
- *Rancher* — полноценная платформа управления контейнерами, выходящая за рамки системы Kubernetes. Она позволяет управлять несколькими кластерами в различных средах, поддерживает рабочие нагрузки для разных платформ оркестрации, а также предусматривает маркетплейс для преднастроенных приложений (https://www.rancher.com/).
- *VMware Tanzu* — решение, разработанное для экосистемы VMware, которое органично интегрируется с процессами предоставления инфраструктуры, обеспечения безопасности и развертывания в гибридных облачных средах. Tanzu предоставляет средства для управления жизненным циклом контейнеризованных приложений в среде VMware (https://tanzu.vmware.com/platform).

Учтите, что некоторые из перечисленных выше дистрибутивов Kubernetes доступны только по подписке или лицензии и не являются бесплатными.

В следующем подразделе мы рассмотрим некоторые инструменты для развертывания системы Kubernetes.

Инструменты для установки Kubernetes

Перечисленные ниже инструменты обеспечивают гибкость и контроль над процессом развертывания кластера Kubernetes. Разумеется, для полноценного управления средой Kubernetes вам потребуется реализовать дополнительную автоматизацию с помощью сторонних инструментов и платформ.

- *kubeadm* — официальный инструмент Kubernetes, предлагающий удобный способ настройки кластеров. Он подходит как для тестовых, так и для

рабочих сред. Простота его использования позволяет быстро развернуть кластер, однако для обеспечения высокой доступности и других критически важных свойств рабочей среды может потребоваться дополнительная настройка (https://kubernetes.io/docs/reference/setup-tools/kubeadm/).

- *kops* — официальный проект Kubernetes для управления кластерами в рабочей среде. Этот CLI-инструмент упрощает создание, обновление и сопровождение отказоустойчивых кластеров, обеспечивая надежную работу контейнеризованных приложений (https://kops.sigs.k8s.io/).
- *Kubespray* — инструмент для развертывания Kubernetes на физических серверах или виртуальных машинах. Объединяет плейбуки Ansible с ресурсами Kubernetes, позволяя автоматизировать создание кластеров в выбранной инфраструктуре (https://github.com/kubespray).
- *Terraform* — инструмент, позволяющий описывать и управлять инфраструктурой кластеров Kubernetes в различных облачных средах. Подход, основанный на использовании кода, обеспечивает согласованность и воспроизводимость при развертывании кластеров в разных окружениях (https://www.terraform.io/).
- *Pulumi* — аналогичный Terraform инструмент для описания и управления инфраструктурой как кодом. С его помощью можно описывать и управлять инфраструктурой кластера Kubernetes, используя языки программирования вроде Python или Go. Такой подход обеспечивает большую гибкость и возможность более тонкой настройки по сравнению с использованием исключительно декларативных языков (https://www.pulumi.com/).

Если в вашей инфраструктуре используется множество кластеров Kubernetes, то вам стоит рассмотреть гибридные решения для управления мультикластерной средой, речь о которых пойдет в следующем подразделе.

Гибридные и мультиоблачные решения

Управление кластерами Kubernetes в разных средах требует мощных инструментов, среди которых можно выделить следующие решения.

- *Anthos* (Google) — гибридная и мультиоблачная платформа, упрощающая управление кластерами Kubernetes, развернутыми в различных средах. Она позволяет организациям применять единый подход к развертыванию и управлению контейнеризованными приложениями в облаке, локальной и периферийной инфраструктуре (https://cloud.google.com/anthos).
- *Red Hat Advanced Cluster Management (RHACM) for Kubernetes* — решение для управления кластерами Kubernetes в гибридных и мультиоблачных средах от компании Red Hat. Платформа Advanced Cluster Management предоставляет централизованный управляющий слой, обеспечивая согласованное развертывание, администрирование и контроль над контейнеризованны-

ми рабочими нагрузками (https://www.redhat.com/en/technologies/management/advanced-cluster-management).

- *VMware Tanzu Mission Control* — централизованный инструмент для управления кластерами Kubernetes в разных окружениях. С помощью единой консоли можно создавать, отслеживать и администрировать кластеры независимо от того, где они размещены — в локальной среде, в облаке или в гибридной инфраструктуре (https://docs.vmware.com/en/VMware-Tanzu-Mission-Control/index.html).

В следующем подразделе мы рассмотрим ключевые критерии выбора подходящей платформы Kubernetes и инструментов управления.

Выбор подходящей среды

Выбор оптимальной среды для развертывания кластеров Kubernetes производственного уровня зависит от нескольких факторов.

- *Уровень контроля*. Хотите ли вы полностью управлять конфигурацией кластера, или вам достаточно возможностей управляемых сервисов?
- *Существующая инфраструктура*. При выборе способа развертывания учитывайте уже используемую инфраструктуру (облачный провайдер, физические серверы).
- *Потребности в масштабировании*. Насколько легким должен быть процесс масштабирования кластера при изменении уровня нагрузки?
- *Экспертность команды*. Оцените опыт вашей команды в работе с Kubernetes и облачной инфраструктурой, чтобы выбрать решение, лучше всего соответствующее навыкам ее участников.

Взвесив эти факторы и изучив доступные варианты, вы сможете построить рабочую среду Kubernetes, обеспечивающую оптимальную производительность и масштабируемость ваших контейнеризованных приложений.

В следующем разделе мы рассмотрим особенности обслуживания кластеров.

Запуск Kubernetes в локальной инфраструктуре: вызовы и факторы, которые следует учитывать

Запуск Kubernetes в локальной среде (on-premises) обеспечивает больший контроль над инфраструктурой, но требует более тщательного управления. В отличие от облачных управляемых решений сопровождение кластера Kubernetes в локальной среде требует ручного выполнения всех задач — от развертывания до обновлений. Ниже мы рассмотрим ключевые аспекты и трудности,

возникающие при управлении кластерами Kubernetes в локальной инфраструктуре.

- *Выделение инфраструктуры*. Настройка инфраструктуры для локального кластера Kubernetes предполагает автоматизацию выделения узлов. Для упрощения этого процесса и обеспечения согласованности используются такие инструменты, как облачные контроллеры Rancher или Terraform. Дополнительно можно использовать инструмент Packer для создания образов виртуальных машин, что позволяет выполнять обновления более плавно, распространяя обновленные образы на все узлы.
- *Развертывание и обслуживание кластера*. Создание локального кластера Kubernetes часто выполняется с помощью инструмента `kubeadm` и оказывается более трудоемким, чем при использовании облачных решений. К задачам обслуживания, привносящим дополнительную сложность, относятся продление срока действия сертификатов, управление узлами и обеспечение высокой доступности.
- *Балансировка нагрузки и предоставление доступа*. Предоставление доступа к приложениям в локальной среде извне может оказаться трудной задачей. Стандартные сервисы Kubernetes (*NodePort* и *LoadBalancer*) не всегда позволяют ее решить. Инструмент *MetalLB* предоставляет решение для балансировки нагрузки в средах на базе физических серверов, однако он имеет ограничения, в частности не поддерживает балансировку нагрузки для API-сервера в окружениях, характеризующихся высокой доступностью.
- *Постоянное хранилище*. Для запуска рабочих нагрузок производственного уровня постоянное хранилище критически важно. Система Kubernetes использует ресурсы *PersistentVolumeClaim (PVC)* и *PersistentVolume (PV)*, которые должны быть интегрированы с физическими системами хранения. Инструменты вроде Longhorn позволяют динамически выделять тома и реплицировать данные между узлами, повышая гибкость локальных конфигураций.
- *Обновления и масштабируемость*. Система Kubernetes обновляется довольно часто, что усложняет управление версиями в локальной инфраструктуре. Важно тестировать новые версии перед их внедрением в рабочую среду. Для упрощения процессов добавления узлов и обновления можно использовать инструменты Packer и Terraform.
- *Сетевые взаимодействия*. Сетевые взаимодействия в локальной инфраструктуре Kubernetes зависят от конфигурации вашего дата-центра и требуют ручного управления DNS, балансировщиками нагрузки и сетевыми параметрами. В этом могут помочь такие инструменты мониторинга, как Prometheus, а также балансировщики нагрузки вроде MetalLB, однако они требуют интеграции и постоянного сопровождения.

- *Мониторинг и управление*. Мониторинг локальных кластеров имеет ключевое значение для поддержания работоспособности системы. Для отслеживания уровня потребления ресурсов можно использовать такие инструменты, как Prometheus и Grafana. Кроме того, необходимо настроить системы логирования и оповещений, чтобы оперативно выявлять и устранять проблемы, минимизируя время простоя.
- *Инструменты и автоматизация*. Автоматизация таких задач, как обновление и управление узлами, играет важнейшую роль в процессе эксплуатации локальных кластеров. Корпоративные платформы Kubernetes, вроде Rancher и OpenShift, снижают необходимость ручного вмешательства и делают процесс управления средой Kubernetes более удобным и прозрачным.
- *Безопасность и соответствие нормативным требованиям*. Безопасность имеет первостепенное значение для корпоративных установок Kubernetes. Реализация поддержки *FIPS* (Federal Information Processing Standards, федеральные стандарты обработки информации) на раннем этапе помогает обеспечить соответствие нормативным требованиям и поддерживать нужный уровень защищенности среды по мере ее развития.

Таким образом, управление Kubernetes в локальной инфраструктуре обеспечивает гибкость, но требует повышенного внимания к инфраструктуре, сети и системам хранения. Выбор подходящих инструментов и стратегий позволяет организациям эффективно масштабировать и поддерживать устойчивую среду Kubernetes в собственной инфраструктуре.

Резюме

Эта глава оказалась довольно насыщенной! Для дальнейшей работы с книгой вам необходим кластер Kubernetes, и мы рассмотрели пять способов его развертывания на разных платформах. Вы познакомились с `minikube` — самым распространенным инструментом для создания кластера на локальной машине. Мы также обсудили инструмент `kind`, позволяющий развернуть многоузловой локальный кластер.

Кроме того, вы познакомились с учебными средами для освоения Kubernetes, мы рассмотрели варианты создания рабочих сред с помощью трех крупных облачных сервисов: GKE, Amazon EKS и AKS. Эти сервисы позволяют создать кластер Kubernetes в облаке для практики и обучения. Более подробно мы рассмотрим их позднее. На данном этапе вам достаточно выбрать решение, которое лучше всего подходит именно вам.

В следующей главе приступим к детальному изучению Kubernetes, начав с понятия пода, который является ключевым объектом данной системы. Вы научитесь

создавать, обновлять и удалять поды, а также назначать их узлам, получать информацию о них и обновлять контейнеры, которые они запускают.

Мы развернем под Nginx в кластере Kubernetes, и вы узнаете, как получать к нему доступ извне. После изучения следующей главы вы сможете запускать собственные контейнеры в кластере Kubernetes с использованием подов. Кластер, который вы установили в ходе освоения этой главы, окажется весьма полезным при выполнении практических примеров, представленных далее.

Дополнительные источники информации

- Установка `minikube`: https://minikube.sigs.k8s.io/docs/start/.
- Драйверы `minikube`: https://minikube.sigs.k8s.io/docs/drivers/.
- Установка Docker: https://docs.docker.com/engine/install/.
- Установка Podman: https://podman.io/docs/installation.
- Создание многоузлового кластера Kubernetes с помощью `minikube`: https://minikube.sigs.k8s.io/docs/tutorials/multi_node/.
- Установка `kind`: https://kind.sigs.k8s.io/docs/user/quick-start#installation.

4 Запуск контейнеров в Kubernetes

Эта глава, пожалуй, самая важная во всей книге. В ней мы рассмотрим понятие *пода* — объекта, с помощью которого Kubernetes запускает контейнеры приложений. Поды являются центральной концепцией системы Kubernetes, поэтому очень важно понять, что они собой представляют.

В предыдущей главе мы говорили о том, что API Kubernetes определяет набор ресурсов, представляющих вычислительную единицу. Под — это ресурс, описанный в API Kubernetes, который представляет один или несколько контейнеров. С помощью Kubernetes мы никогда не создаем контейнеры напрямую, мы создаем поды, которые *преобразуются* в контейнеры на вычислительном узле кластера.

Поначалу уловить связь между подами Kubernetes и контейнерами может быть непросто, поэтому мы подробно разберем, что такое под и почему мы используем именно его, а не непосредственно контейнеры. Под Kubernetes может содержать один или несколько контейнеров приложений. В этой главе мы сосредоточимся на подах, содержащих только один контейнер, а в следующей познакомимся с подами, включающими несколько контейнеров.

Мы создадим, удалим и обновим поды, используя *BusyBox* — образ на базе Linux, включающий множество утилит, полезных для проведения тестов. Мы также запустим под на основе контейнерного образа Nginx с целью запуска HTTP-сервера. Кроме того, мы рассмотрим способ получения доступа к домашней странице Nginx с помощью функции переадресации (или проброса) портов, предоставляемой утилитой `kubectl`. Это позволит обращаться к подам, запущенным в кластере Kubernetes, и тестировать их прямо через браузер.

Затем вы научитесь снабжать поды метками и аннотациями для облегчения доступа к ним и поддержания порядка в кластере Kubernetes. Наконец, вы познакомитесь с двумя дополнительными ресурсами — *Job* и *CronJob*. После изучения этой главы вы сможете запускать собственные контейнеры под управлением Kubernetes, что станет первым шагом к освоению данной системы.

Технические требования

Чтобы выполнять примеры из этой главы, вам потребуется следующее.

- Правильно настроенный кластер Kubernetes для выполнения обсуждаемых команд. Этот кластер может быть создан с помощью `minikube`, `kind`, GKE, EKS или AKS.
- Установленная на локальной машине утилита `kubectl`. Ваш кластер может состоять из нескольких узлов, но для корректной работы Kubernetes необходим как минимум один узел в состоянии `Ready`.

Примеры кода для этой главы можно загрузить из официального репозитория GitHub по адресу https://github.com/PacktPublishing/The-Kubernetes-Bible-Second-Edition/tree/main/Chapter04.

Понятие пода

Для работы с подами необходимо понимать их особенности, поэтому сначала мы рассмотрим концепцию пода с теоретической точки зрения.

Что такое под

Создание, обновление или удаление контейнера с помощью Kubernetes всегда осуществляется через под (pod). Под — это группа из одного или нескольких контейнеров, которые должны быть запущены на одной машине, в одном пространстве имен Linux. Запомните, что поды могут включать один или несколько контейнеров, но все контейнеры внутри одного пода всегда запускаются на одном вычислительном узле. Абсолютное правило заключается в том, что под *не может* одновременно работать на нескольких вычислительных узлах.

Но почему мы вообще делегируем управление контейнерами этому промежуточному ресурсу? Почему система Kubernetes не предусматривает специальный ресурс для запуска отдельного контейнера? Дело в том, что сама технология контейнеризации заставляет мыслить категориями процессов Linux, а не виртуальных машин. Возможно, вы уже знаете, что один из самых распространенных антипаттернов контейнеризации заключается в использовании контейнеров в качестве заменителей виртуальных машин. До распространения технологий контейнеризации все процессы развертывались непосредственно на виртуальных машинах. Однако контейнеры не являются альтернативой ВМ и не предназначены для запуска множества процессов.

Согласно золотому правилу контейнерной технологии *между контейнером и процессом Linux должно существовать соотношение «один к одному»*. Однако

современные приложения часто предусматривают не один, а несколько процессов. Поэтому для работы полноценного микросервиса одного контейнера в большинстве случаев будет недостаточно. Из этого следует, что процессы, а значит, и контейнеры должны иметь возможность взаимодействовать друг с другом через общие файловые системы, сетевые интерфейсы и другие механизмы. Именно для логической группировки контейнеров в Kubernetes используется такой ресурс, как под. Все контейнеры и процессы, составляющие приложение, должны быть объединены в одном поде. Благодаря этому они будут запускаться вместе, задействуя все преимущества межпроцессной и межконтейнерной коммуникации (рис. 4.1).

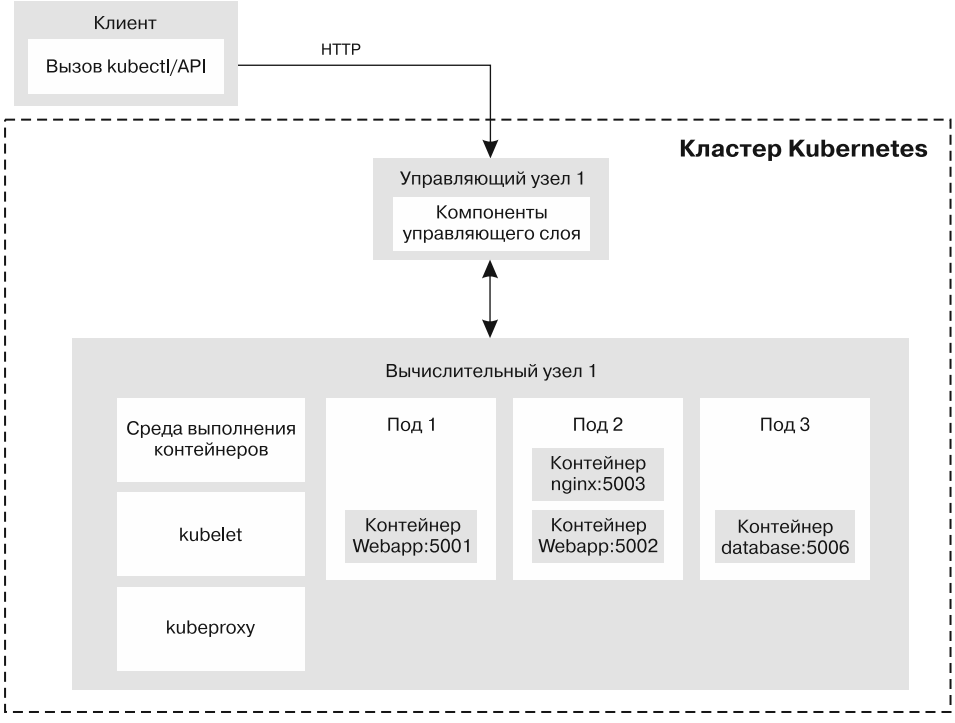

Рис. 4.1. Контейнеры и поды

Чтобы лучше понять данную концепцию, представьте, что у вас есть работающий блог WordPress на виртуальной машине и вы хотите преобразовать эту виртуальную машину в под WordPress, чтобы развернуть блог в кластере Kubernetes. WordPress — это одно из самых распространенных приложений, для корректной работы которого требуется несколько процессов. Это делает его отличной иллюстрацией того, зачем нужны поды.

WordPress представляет собой PHP-приложение, которому необходимы и веб-сервер, и интерпретатор PHP. Ниже перечислены процессы Linux, без которых WordPress не сможет функционировать.

- *HTTP-сервер Nginx*. Поскольку WordPress является веб-приложением, ему требуется HTTP-сервер, запущенный в качестве процесса, чтобы принимать запросы и предоставлять страницы блога. Nginx — это надежный HTTP-сервер, который прекрасно справляется с этой задачей.
- *Интерпретатор PHP-FPM* (FastCGI-Process-Manager — менеджер процессов FastCGI). Поскольку WordPress представляет собой движок для создания блогов, написанный на PHP, ему необходим интерпретатор соответствующего языка.

Nginx и PHP-FPM — это два отдельных процесса, то есть два исполняемых файла, которые должны запускаться по отдельности, но при этом взаимодействовать друг с другом. На виртуальной машине эта задача решается просто — путем установки Nginx и PHP-FPM и настройки их коммуникации через Unix-сокеты. Для этого достаточно указать в конфигурационном файле Nginx (`/etc/nginx/nginx.conf`) путь к сокету PHP-FPM.

В мире контейнеров задача усложняется, поскольку запуск двух процессов в одном контейнере считается антипаттерном. В связи с этим нам необходимо запустить два контейнера (по одному на каждый процесс) и обеспечить их взаимодействие, а также общий доступ к каталогу с кодом приложения. Для решения этой задачи приходится использовать сетевой слой Docker, позволяющий контейнерам Nginx и PHP-FPM обмениваться данными. Кроме того, требуется смонтировать том, чтобы оба контейнера могли совместно использовать код WordPress. Все это можно реализовать с помощью нескольких команд Docker, но представьте себе такой сценарий в рабочей среде со множеством машин. Реализовать межпроцессное взаимодействие только средствами Docker возможно, но при этом крайне сложно учесть все требования, предъявляемые к рабочей среде. При работе с десятками микросервисов, распределенными по множеству машин, управление всеми этими сетями Docker, монтирование томов и решение прочих сопутствующих задач превращается в сущий кошмар. Именно эту проблему призван решить ресурс Pod системы Kubernetes. Поды особенно полезны тем, что объединяют несколько контейнеров и упрощают межпроцессное взаимодействие. Вот ключевые преимущества, которые они обеспечивают.

- Все контейнеры, содержащиеся в одном поде, могут обращаться друг к другу через локальный хост, так как используют одно и то же сетевое пространство имен.
- Все контейнеры в одном поде имеют общее пространство портов.
- К поду можно подключить том и затем смонтировать его в контейнеры, предоставив им общий доступ к каталогам и файлам.

Благодаря этим возможностям Kubernetes развернуть блог WordPress оказывается очень просто. Для этого достаточно создать под, который будет запускать два контейнера — Nginx и PHP-FPM. Поскольку они могут взаимодействовать друг с другом через локальный хост, настройка обмена данными между ними не составляет труда. Затем с помощью тома мы можем предоставить обоим контейнерам доступ к коду WordPress.

Наиболее сложные приложения неизбежно требуют использования нескольких контейнеров, которые имеет смысл сгруппировать в один под, чтобы система Kubernetes запускала их вместе. Помните о том, что под призван упростить межконтейнерное (или межпроцессное) взаимодействие в условиях масштабной рабочей среды.

Тем не менее под нередко состоит всего из одного контейнера. Так или иначе, под представляет собой самый низкий уровень абстракции, предоставляемый API Kubernetes, и именно с ним вам предстоит взаимодействовать.

Наконец, учтите, что контейнер, запущенный вручную на машине, входящей в кластер Kubernetes, не будет рассматриваться системой как управляемый ею контейнер. Такой контейнер становится «осиротевшим», то есть находящимся вне зоны контроля оркестратора. Система Kubernetes управляет только теми контейнерами, которые она сама запустила через API подов.

Каждый под получает IP-адрес

Контейнеры внутри одного пода могут обмениваться данными через локальный хост, но и поды способны взаимодействовать друг с другом. При запуске каждому поду назначается частный IP-адрес. Каждый под может связываться с любым другим подом в кластере, используя его IP-адрес.

Система Kubernetes использует модель плоской сети, реализованную с помощью компонента *CNI* (Container Network Interface, сетевой интерфейс контейнера). CNI выступает в качестве стандартизированного моста между контейнеризованными приложениями и сетевой инфраструктурой, лежащей в основе кластеров Kubernetes. Это устраняет необходимость в создании отдельной сетевой конфигурации для каждого контейнера и упрощает процесс коммуникации и передачи данных.

Компонент CNI построен на основе гибкой архитектуры с плагинами. Эти плагины, написанные на разных языках программирования, взаимодействуют со средой выполнения контейнеров через стандартные механизмы ввода-вывода. Спецификация плагинов определяет четкий интерфейс для настройки сети, выделения IP-адресов и поддержания соединений между узлами. Среда выполнения контейнеров вызывает эти плагины, что обеспечивает динамическое управление и обновление сетей контейнеров в среде Kubernetes. Такой подход

гарантирует непрерывное и адаптивное сетевое взаимодействие контейнеризованных приложений.

Следующая схема демонстрирует высокоуровневый процесс взаимодействия между подами и контейнерами (рис. 4.2).

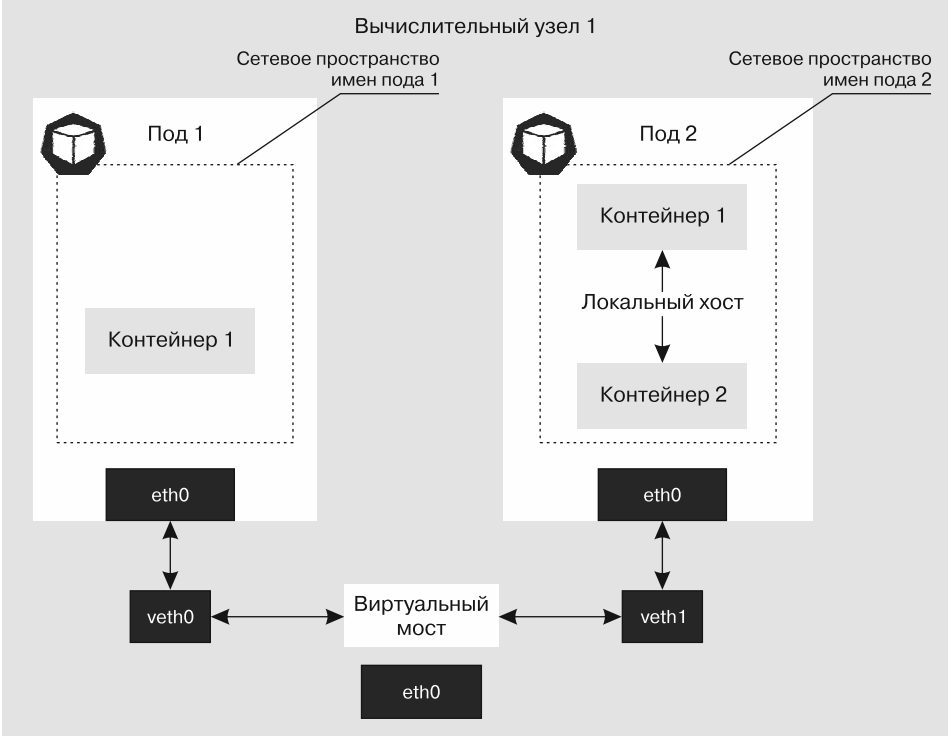

Рис. 4.2. Взаимодействие контейнеров и подов

Проектирование подов

Хотя понимание подов имеет ключевое значение, в реальной практике большинство команд используют более продвинутую конструкцию Kubernetes — объект Deployment (развертывание). Данный объект представляет собой абстракцию более высокого уровня для управления подами. Он автоматизирует такие задачи, как масштабирование и перезапуск подов при сбоях, что обеспечивает более надежную и управляемую работу приложений. Мы подробно рассмотрим объекты Deployment чуть позже, а пока обсудим API подов (pods API), чтобы углубить понимание этих базовых строительных блоков.

Второе золотое правило использования подов гласит, что *они должны легко уничтожаться и пересоздаваться*. Под может быть удален как намеренно, так

и в результате сбоя. Например, если рабочий узел, на котором запущены четыре пода, выйдет из строя, все контейнеры внутри этих подов станут недоступны. Поэтому необходимо проектировать поды так, чтобы их можно было безболезненно уничтожать и пересоздавать, не влияя на стабильность работы приложения. Для этого при создании подов следует соблюдать два простых правила:

- под должен содержать все необходимое для запуска приложения;
- под должен хранить любое состояние вне себя, используя внешнее хранилище (PersistentVolume).

Когда вы только начинаете проектировать поды в Kubernetes, вам может быть трудно определиться с его содержимым. Но правило достаточно простое: под должен включать в себя приложение или микросервис. Возьмем в качестве примера под WordPress, упомянутый ранее: он должен содержать контейнеры Nginx и PHP-FPM, необходимые для запуска WordPress. Если такой под выйдет из строя, сайт WordPress станет недоступен, но после пересоздания пода блог вновь заработает, так как под содержит все необходимое для запуска WordPress.

Каждое современное приложение хранит свое состояние во внешнем хранилище, в базе данных (например, Redis или MySQL) или передает его другому микросервису. WordPress не исключение: для сохранения и извлечения сообщений эта система использует MySQL (или MariaDB). Следовательно, нам также необходимо запустить контейнер MySQL. Здесь возможны два варианта:

- запуск контейнера MySQL в составе пода WordPress;
- запуск контейнера MySQL в выделенном поде.

Оба варианта допустимы, но второй является предпочтительным. Хорошей практикой является отделение приложения (в нашем случае WordPress, но это может быть и микросервис) от базы данных или логического уровня путем их размещения в отдельных подах. Помните, что поды способны взаимодействовать друг с другом. Этим можно воспользоваться, выделив отдельный под для MySQL и указав его адрес в настройках блога WordPress.

Разделение уровней базы данных и приложения повышает устойчивость системы. Благодаря этому сбой пода с приложением не затронет базу данных.

Итак, объединение всех слоев приложения в одном поде порождает проблемы:

- с надежностью хранения данных;
- доступностью;
- устойчивостью.

Именно поэтому рекомендуется хранить состояние приложений в отдельных подах. Рассматривая уровень данных как самостоятельное приложение с собственным циклом разработки и управления, мы минимизируем связность

архитектуры, что позволяет масштабировать, обновлять и тестировать уровень данных, не затрагивая сам код приложения.

Монолитные приложения с сохранением состояния

Если не считать редких исключений, запуск быстро развивающихся монолитных рабочих нагрузок с сохранением состояния в Kubernetes в целом не рекомендуется. Это связано со сложностью управления монолитами в контейнерах, потенциальной неэффективностью частых обновлений в динамичных средах и повышенными затратами на управление постоянным хранилищем по сравнению с традиционными способами развертывания.

Доступ к подам по IP-адресу

К подам можно обращаться по их IP-адресам, однако это не рекомендуется в качестве основного способа взаимодействия с работающими приложениями. В одной из следующих глав вы познакомитесь с ресурсом Service (сервис), который играет ключевую роль в сопоставлении IP-адресов и подов. Там мы подробно рассмотрим, как сервисы упрощают доступ к подам и процесс коммуникации приложений.

Теперь запустим наш первый под. Создание пода WordPress на данном этапе является слишком сложным, поэтому начнем с простого — запустим несколько подов Nginx и посмотрим, как система Kubernetes управляет контейнером.

Запуск подов

В этом разделе мы создадим первые поды в кластере Kubernetes. У подов есть определенные особенности, которые необходимо понимать, чтобы уверенно с ними работать.

Определившись с тем, что такое поды, далее мы приступим к их созданию.

Создание пода с помощью императивного синтаксиса

В этом подразделе мы создадим под на основе образа Nginx. Для этого нам понадобится два параметра:

- произвольное имя пода, задаваемое пользователем;
- контейнерный образ, на основе которого будут собраны контейнеры внутри пода.

Как и почти все в Kubernetes, под можно создать с использованием императивного или декларативного синтаксиса, о которых мы говорили во второй главе. Напомним, что императивный синтаксис предполагает выполнение команд `kubectl` непосредственно в терминале, а декларативный требует написания

YAML-файла с конфигурацией пода и его дальнейшего применения с помощью команды `kubectl apply -f`.

Для создания пода в кластере Kubernetes используется команда `kubectl run`. Это самый простой и быстрый способ запуска пода. Команда вызывается так:

```
$ kubectl run nginx-pod --image=nginx:latest
```

Здесь в качестве имени пода задается `nginx-pod`. Имя имеет большое значение, так как служит идентификатором пода: при обновлении (`update`) или удалении (`delete`) пода необходимо указывать именно его, чтобы система Kubernetes знала, над каким именно объектом должна быть выполнена указанная операция. Флаг `--image` указывает, какой контейнерный образ будет использован для запуска контейнера в этом поде. После создания пода его состояние можно проверить следующим образом:

```
$ kubectl get pods
NAME       READY   STATUS    RESTARTS   AGE
nginx-pod  1/1     Running   0          79s
```

Учтите, что под не запускается мгновенно. Системе Kubernetes может потребоваться время, чтобы загрузить образ контейнера из реестра (если он отсутствует в локальном хранилище) и настроить окружение пода. Чтобы отслеживать этот процесс в реальном времени, используйте команду kubectl get po -w, которая выводит информацию о подах и автоматически обновляет ее[1].

В данном примере мы создаем под на основе образа контейнера `nginx:latest` из реестра Docker Hub. Под `nginx-pod` будет содержать только один контейнер, собранный из этого образа. Указать несколько образов здесь невозможно из-за ограничений императивного синтаксиса.

Если вам требуется создать под, содержащий несколько контейнеров, собранных на основе разных образов, то вам придется использовать декларативный синтаксис и описать конфигурацию в YAML-файле.

Теги и дайджесты: обеспечение согласованности образов

При создании подов вы можете столкнуться с такими понятиями, как теги и дайджесты. И те и другие используются для идентификации контейнерных образов, но между ними есть важное различие.

- *Теги* — это человекочитаемые имена версий образов. Они могут указывать на разные версии одного и того же образа, что способно вызвать непредсказуемое поведение.

[1] В книге встречаются сокращенные имена ресурсов, как в этом примере po (pods). Их полный список можно посмотреть, введя команду kubectl api-resources.

- *Дайджесты* — это уникальные «отпечатки» образа (контрольные суммы), гарантирующие то, что вы всегда ссылаетесь именно на ту версию, которая вам нужна. Это особенно важно для обеспечения безопасности и воспроизводимости, учитывая риск атак на цепочку поставок программного обеспечения.

Например, вместо тега `nginx:latest` можно использовать дайджест `nginx@sha256:1445eb9c6dc5e9619346c836ef6fbd6a95092e4663f27dcfce116f051cdbd232`. Информацию о дайджесте контейнерного образа можно получить непосредственно в реестре или с помощью команды `podman manifest inspect nginx:latest` (рис. 4.3).

Рис. 4.3. Получение дайджеста образа из реестра контейнеров

Использование дайджестов гарантирует то, что вы разворачиваете конкретную версию образа с уникальной контрольной суммой. Данная практика приобретает все большее значение для обеспечения безопасности и надежности развертываний.

Теперь перейдем к созданию подов с помощью декларативного описания в формате YAML.

Создание пода с помощью декларативного синтаксиса

Под также можно создать с использованием декларативного синтаксиса. Для этого необходимо подготовить YAML-файл с описанием пода и применить его к кластеру Kubernetes с помощью команды `kubectl apply -f`.

Помните, что Kubernetes не допускает запуск двух подов с одинаковым именем в одном пространстве имен (в нашем случае это `default`). Имя пода является уникальным и используется для его однозначной идентификации внутри пространства имен, поэтому перед созданием одноименного пода в том же пространстве имен необходимо удалить под, ранее созданный:

```
$ kubectl delete pod nginx-pod
pod "nginx-pod" deleted
```

Ниже приведено содержимое файла `nginx-pod.yaml`, который вы можете создать на своей локальной машине:

```
# nginx-pod.yaml
apiVersion: v1
kind: Pod
metadata:
  name: nginx-pod
spec:
  containers:
    - name: nginx-container
      image: nginx:latest
```

Попробуйте внимательно прочитать этот файл и разобраться в его содержании. YAML-файлы представляют собой набор пар «ключ — значение». В данном случае именем (`name`) пода является `nginx-pod`, а в разделе `spec`: содержится массив контейнеров, включающий только один контейнер `nginx-container`, созданный на основе образа `nginx:latest`.

После сохранения файла `nginx-pod.yaml` вы можете создать под, выполнив команду:

```
$ kubectl apply -f nginx-pod.yaml
pod/nginx-pod created
```

Если в кластере уже существует под с именем `nginx-pod`, эта команда завершится с ошибкой. В этом случае измените имя пода в YAML-файле и повторите попытку.

Пространства имен в Kubernetes

Если не указать пространство имен при создании ресурса, по умолчанию будет использоваться `default`. Подробно о пространствах имен мы поговорим в главе 6.

Чтение информации и метаданных пода

На данном этапе у вас уже должен быть запущен под в кластере Kubernetes. Теперь попробуем получить информацию о нем. В любой момент времени нам необходимо иметь возможность извлекать и просматривать сведения о созданных ресурсах кластера Kubernetes, в том числе о подах. Чтение информации о кластере выполняется с помощью двух команд утилиты `kubectl`: `kubectl get` и `kubectl describe`.

- Команда `kubectl get` реализует операцию перечисления и используется для вывода списка объектов. Как вы помните, в предыдущей главе мы перечисляли узлы кластера после выполнения всех установочных процедур с помощью команды `kubectl get nodes`. При вызове этой команды необходимо указать

тип объектов, список которых вы хотите получить. В нашем случае это поды, поэтому мы используем команду `kubectl get pods`. В следующих главах мы познакомимся и с другими объектами, например `configmaps` и `secrets`. Для их отображения применяются команды `kubectl get configmaps` и `kubectl get secrets`. Чтобы вывести список, содержащий наш под `nginx-pod`, можно использовать команду:

```
$ kubectl get pods
```

- Команда `kubectl describe` работает иначе. Она предназначена для получения всей информации о конкретном объекте, который определяется типом и именем. Информацию о ранее созданном нами поде можно извлечь с помощью команды `kubectl describe pods nginx-pod`. В результате ее выполнения вы получите полный набор данных об этом поде, включая его IP-адрес. Чтобы просмотреть подробные сведения о поде `nginx-pod`, можно также использовать следующую команду:

```
$ kubectl describe pod nginx-pod
Name:             nginx-pod
Namespace:        default
... <фрагмент опущен для краткости>...
Containers:
  nginx-container:
    Container ID:  containerd://3afbbe30b51b77994df69f4c4dbefb02fc304efb2bf0\
f5bdb65a651154a8e311
    Image:          nginx:latest
... <фрагмент опущен для краткости>...
Conditions:
  Type                        Status
  PodReadyToStartContainers   True
  Initialized                 True
... <фрагмент опущен для краткости>...
Events:
  Type    Reason   Age    From            Message
  ----    ------   ----   ----            -------
... <фрагмент опущен для краткости>...
  Normal  Created  112s   kubelet         Created container nginx-container
  Normal  Started  112s   kubelet         Started container nginx-container
```

Из вывода предыдущей команды можно получить множество сведений, включая:

- *имя пода и пространство имен* (`Name` и `Namespace`) — идентифицируют конкретный под, по которому был сделан запрос (например, `nginx-pod`);
- *сведения о контейнерах* (`Containers`) — содержат информацию о контейнерах внутри пода, включая имя образа, сведения о выделенных ресурсах и текущее состояние;
- *состояния пода* (`Conditions`) — отражают текущее рабочее состояние пода (например, `Running`, `Pending`, `CrashLoopBackOff`);

- *события* (`Events`) — этот раздел содержит историю событий, связанных с жизненным циклом пода, таких как создание, перезапуски и ошибки.

Теперь рассмотрим некоторые дополнительные способы вывода информации о ресурсах Kubernetes.

Вывод списка объектов в формате JSON или YAML

Параметр `-o` или `--output` является одним из самых полезных в `kubectl`. Он имеет несколько преимуществ, о которых необходимо знать. С его помощью можно изменять формат вывода информации, возвращаемой в результате выполнения команд `kubectl`. По умолчанию команда `kubectl get pods` возвращает список подов в кластере Kubernetes в удобочитаемом виде. Однако, используя параметр `-o`, эти данные можно вывести в формате YAML или JSON:

```
$ kubectl get pods --output=yaml   # В формате YAML
$ kubectl get pods --output=json   # В формате JSON
```

Зная имя пода, вы можете запросить данные именно о нем:

```
$ kubectl get pods <ИМЯ_ПОДА> -o yaml
# или
$ kubectl get pods <ИМЯ_ПОДА> -o json
```

Таким образом, вы можете извлекать и экспортировать данные из кластера Kubernetes в удобном формате.

Резервное копирование ресурса с помощью операции перечисления

Упомянутые выше флаги можно использовать и для резервного копирования ресурсов Kubernetes. Допустим, вы создали под с помощью императивного подхода, то есть на вашей локальной машине отсутствует YAML-файл с его описанием. Если под выйдет из строя, его будет трудно воссоздать. Параметр `-o` позволяет извлечь YAML-файл с описанием ресурса Kubernetes, даже если он был создан императивным способом. Для этого выполните команду:

```
$ kubectl get pods/nginx-pod -o yaml > nginx-pod-output.yaml
```

Таким образом, вы получите резервную копию ресурса `nginx-pod` в формате YAML. Вы всегда можете сравнить полученный файл с исходной YAML-декларацией и проанализировать различия с помощью команды `diff` или других утилит:

```
$ diff nginx-pod.yaml nginx-pod-output.yaml
```

Существуют инструменты, позволяющие получить «чистый» YAML-файл, пригодный для использования в качестве декларации. Например, для удаления лишней информации можно использовать утилиту kube-neat. Подробнее о ней можно узнать здесь: https://github.com/itaysk/kubectl-neat.

Если что-то пойдет не так, вы сможете без труда воссоздать свой под. Обратите внимание на фрагмент команды с `nginx-pod`: чтобы получить YAML-декларацию, необходимо явно указать целевой ресурс. Перенаправление вывода этой команды в файл позволяет извлекать и сохранять конфигурацию объекта, входящего в состав кластера Kubernetes.

Получение дополнительных сведений с помощью операции перечисления

Нам также стоит упомянуть весьма полезный параметр `-o wide`, который расширяет стандартный вывод, включая в него дополнительные данные. Например, при применении этого параметра к объекту `pods` вы получите имя рабочего узла, на котором запущен данный под:

```
$ kubectl get pods -o wide
NAME       READY   STATUS    RESTARTS   AGE   IP           NODE       NOMINATED
NODE   READINESS GATES
nginx-pod  1/1     Running   0          15m   10.244.0.4   minikube   <none>
<none>
```

Учтите, что параметр `-o` принимает множество различных аргументов, в том числе весьма продвинутых. Например, `jsonpath` позволяет выполнять сортировку и выборку прямо из JSON-документа, благодаря чему можно извлечь только нужную информацию. Это похоже на работу с библиотекой `jq`, которая используется в Bash-сценариях для обработки данных JSON.

Получение доступа к поду из внешнего мира

К этому моменту в вашем кластере Kubernetes уже должен быть запущен под с HTTP-сервером Nginx. Теперь вам нужно получить к нему доступ из браузера. Однако сделать это не так просто.

По умолчанию кластер Kubernetes не позволяет получать доступ к подам извне. Для этого используется отдельный ресурс — Service (сервис), о котором мы подробно поговорим в главе 8. Тем не менее утилита `kubectl` предусматривает команду `kubectl port-forward` для быстрого доступа к запущенному в кластере контейнеру. Вот как ее можно использовать:

```
$ kubectl port-forward pod/nginx-pod 8080:80
Forwarding from 127.0.0.1:8080 -> 80
Forwarding from [::1]:8080 -> 80
```

Эта команда приказывает утилите `kubectl` перенаправить трафик с порта `8080` локальной машины (где запущена `kubectl`) на порт `80` пода с именем `nginx-pod`.

После выполнения этой команды утилита `kubectl` выведет сообщение о том, что локальный порт `8080` был перенаправлен на порт `80` пода. Если появится сообщение об ошибке, скорее всего, локальный порт `8080` уже занят. В этом случае попробуйте указать другой порт или просто уберите локальный порт из команды, чтобы утилита `kubectl` выбрала его случайным образом:

```
$ kubectl port-forward pod/nginx-pod :80
Forwarding from 127.0.0.1:xxxxx -> 80
Forwarding from [::1]:xxxxx -> 80
```

Теперь вы можете открыть браузер и перейти по адресу `http://localhost:<локальный_порт>`. В нашем случае это будет `http://localhost:8080` (рис. 4.4).

Рис. 4.4. Доступность страницы запущенного в поде сервера Nginx через локальный хост подтверждает успешную переадресацию портов

По окончании тестирования остановите переадресацию портов, нажав Ctrl+C.

Получение доступа к контейнеру внутри пода

После запуска пода мы можем получить доступ к контейнерам, которые он содержит. В Docker для выполнения команд в работающем контейнере используется команда `docker exec`. В Kubernetes этот подход реализован с помощью команды `kubectl exec`. Чтобы обратиться к запущенному ранее контейнеру Nginx внутри пода `nginx-pod`, выполните команду:

```
$ kubectl exec -it nginx-pod -- bash
root@nginx-pod:/# hostname
nginx-pod
```

В результате вы окажетесь внутри контейнера Nginx и сможете выполнять в нем любые действия. Приведенная выше команда предполагает, что в контейнере установлен интерпретатор `bash`. При его отсутствии в контейнерах обычно доступен интерпретатор `sh`, который можно использовать для получения доступа к ним. При необходимости вы можете указать полный путь к двоичному файлу, например, так:

```
$ kubectl exec -it nginx-pod -- /bin/bash
```

По окончании тестирования выйдите из оболочки контейнера с помощью команды `exit`:

```
root@nginx-pod:/# exit
exit
```

Безопасность и работа в контейнерах от имени непривилегированных пользователей

Чтобы ограничить потенциальный ущерб от эксплуатации уязвимостей, рекомендуется не запускать контейнеры от имени пользователя root. Непривилегированный пользователь имеет меньше прав доступа к системе, что снижает последствия в случае атаки. Кроме того, для минимизации поверхности атаки следует придерживаться принципа наименьших привилегий, то есть предоставлять контейнеру только те разрешения, которые необходимы для его работы. В главе 18 мы подробно обсудим такое понятие, как контекст безопасности.

Теперь посмотрим, как можно удалить под из кластера Kubernetes.

Удаление пода

Удалить под очень просто. Для этого достаточно использовать команду `kubectl delete` и указать имя пода, который требуется удалить. В нашем случае это `nginx-pod`:

```
$ kubectl delete pods nginx-pod
# или...
$ kubectl delete pods/nginx-pod
```

Если вы не знаете имя пода, сначала используйте команду `kubectl get pods`, чтобы вывести список подов и найти нужный.

Учтите также, что если под был создан декларативным способом и у вас сохранился его YAML-файл, то удалить этот под можно без явного указания его имени, так как оно содержится в YAML-декларации.

Для этого выполните следующую команду:

```
$ kubectl delete -f nginx-pod.yaml
```

Выполнение этой команды тоже приведет к удалению пода.

Помните, что при удалении пода удаляются и все контейнеры, входящие в его состав. Жизненный цикл контейнера привязан к жизненному циклу пода, в котором он был запущен. Когда под удаляется, то же самое происходит с контейнерами, которыми он управляет. Поэтому всегда следует взаимодействовать именно с подами, а не напрямую с контейнерами.

Итак, мы рассмотрели такие ключевые аспекты управления подами, как запуск подов императивным и декларативным способом, удаление подов, а также отображение их списка и описания. Теперь поговорим о маркировке и аннотировании подов.

Снабжение подов метками и аннотациями

В этом разделе мы обсудим такие важные понятия Kubernetes, как метки и аннотации. Метки — это произвольные пары «ключ — значение», прикрепляемые к объектам кластера Kubernetes. Благодаря такой маркировке вы можете формировать запросы для выборки нужных объектов на основании присвоенных им меток. В этом разделе вы научитесь взаимодействовать с метками с помощью утилиты `kubectl`.

Метки и их назначение

Вы можете снабжать объекты абсолютно любыми метками, никаких строгих правил здесь нет. Метки представляют собой атрибуты, позволяющие организовать объекты в кластере. Например, можно прикрепить к подам рабочей среды метку `environment=production`, а затем с помощью команды `kubectl get pods` перечислить все поды, относящиеся к этому окружению:

```
$ kubectl get pods --selector "environment=production"
```

Как видите, эту задачу можно решить с помощью флага `--selector` или его сокращенной версии `-l`:

```
$ kubectl get pods -l "environment=production"
```

Эта команда выведет все поды, для которых задана метка `environment` со значением `production`. Разумеется, в нашем случае таких подов не будет, так как мы еще не назначили ни одной метки. Работа с метками требует дисциплины, так что старайтесь задавать их при создании каждого нового пода или другого объекта. Мы затронули тему меток на данном этапе именно потому, что их можно присваивать не только подам, но и практически любому объекту Kubernetes. Метки позволяют организовать ресурсы кластера, и этой возможностью следует пользоваться как можно чаще.

Метки также можно использовать для установления связей между различными объектами Kubernetes. Вы заметите, что некоторые объекты этой системы считывают метки, присвоенные определенным подам, и выполняют над ними те или иные операции, исходя из значений этих меток. Если ваши поды не имеют меток или имеют некорректные метки, то часть таких механизмов может не работать ожидаемым образом.

С другой стороны, использование меток полностью произвольно: система Kubernetes не предусматривает никаких правил именования или соглашений. Поэтому вам придется сформулировать собственные правила применения меток. Если вы отвечаете за управление кластером Kubernetes, то вам следует внедрить правила использования обязательных меток и мониторинга для быстрого выявления непомеченных ресурсов.

Учтите, что длина метки ограничена 63 символами — они должны быть короткими. Вот несколько примеров возможных меток:

- `environment` (`prod`, `dev`, `uat` и т. д.);
- `stack` (`blue`, `green` и т. д.);
- `tier` (`frontend`, `backend`);
- `app_name` (`wordpress`, `magento`, `mysql` и т. д.);
- `team` (`business`, `developers`).

Метки не обязательно должны быть уникальными для разных объектов. Например, вам может понадобиться вывести список всех подов, относящихся к рабочему окружению. В этом случае в кластере может одновременно существовать несколько подов с одинаковой парой «ключ — значение» в качестве метки — это не только допустимо, но и является рекомендуемой практикой, если вы планируете делать выборку. Так, для получения списка всех ресурсов, относящихся к рабочему окружению, необходимо заранее задать для них метку `environment=prod`. Далее обсудим аннотации, которые также позволяют присваивать метаданные подам.

Аннотации и их отличие от меток

В Kubernetes также используется такой тип метаданных, как *аннотации*. По своей структуре они напоминают метки, так как тоже представляют собой пары «ключ — значение». Однако они имеют другое назначение. Метки предназначены для идентификации ресурсов и установления связей между ними, а аннотации — для добавления контекстной информации о ресурсе, к которому они относятся.

Например, создавая под, вы можете добавить аннотацию с адресом электронной почты команды поддержки, к которой следует обращаться в случае неполадок в работе приложения. Такая информация уместна в аннотации, но не в составе метки.

В отличие от меток, которые рекомендуется определять при каждой возможности, аннотации можно опускать, так как они менее важны для работы кластера. Однако учтите, что некоторые объекты Kubernetes или сторонние приложения часто считывают аннотации и используют их в качестве конфигурации. В таких случаях способ использования аннотаций будет явно описан в соответствующей документации.

Добавление метки

В этом подразделе вы узнаете, как добавлять и удалять метки и аннотации, а также как изменять метки уже существующих в кластере подов.

Возьмем под на основе образа Nginx, который мы использовали ранее, и пересоздадим его, снабдив меткой `tier` со значением `frontend`. Для этого выполните команду:

```
$ kubectl run nginx-pod --image nginx --labels "tier=frontend"
```

Как видите, метку можно задать с помощью параметра `--labels`. Мы также можем указать несколько меток, перечислив их через запятую:

```
$ kubectl run nginx-pod --image nginx --labels="app=myapp,env=dev,tier=frontend"
```

Данная команда создаст под `nginx` с тремя метками.

Флаг `--labels` предусматривает сокращенную форму `-l`, что позволяет упростить запись команды и сделать ее более читаемой. Метки также можно задать в YAML-файле с описанием пода. Вот конфигурация пода с двумя метками, описанная с помощью декларативного синтаксиса:

```
# labeled_pod.yaml
apiVersion: v1
kind: Pod
```

```
metadata:
  name: nginx-pod
  labels:
    environment: prod
    tier: frontend
spec:
  containers:
    - name: nginx-container
      image: nginx:latest
```

Если этот файл сохранен по адресу `~/labeled_pod.yaml`, то мы можем создать под с помощью следующей команды:

```
$ kubectl apply -f ~/labeled_pod.yaml
```

Теперь при выполнении фильтрации по метке мы получим как минимум один под — тот, который только что был создан:

```
$ kubectl get pod -l environment=prod
NAME        READY   STATUS    RESTARTS   AGE
nginx-pod   1/1     Running   0          31m
```

Далее выведем список меток, назначенных поду.

Вывод списка меток, присвоенных поду

В Kubernetes нет отдельной команды для отображения меток, присвоенных поду, но вы можете сделать вывод команды `kubectl get pods` более подробным. Если добавить параметр `--show-labels`, то в результате выполнения этой команды будут показаны все метки, назначенные подам:

```
$ kubectl get pods --show-labels
NAME        READY   STATUS    RESTARTS   AGE   LABELS
nginx-pod   1/1     Running   0          56s   environment=prod,tier=frontend
```

Эта команда не выполняет никаких запросов на основе меток, а просто выводит список меток.

Добавление или обновление метки запущенного пода

Теперь, когда вы научились создавать поды с метками, рассмотрим способ присвоения метки уже запущенному поду. Мы можем в любой момент добавить или изменить метку ресурса с помощью команды `kubectl label`. Например, присвоим поду `nginx` новую метку `stack` со значением `blue`:

```
$ kubectl label pod nginx-pod stack=blue
pod/nginx-pod labeled
```

Эта команда сработает только в том случае, если у пода еще нет метки с ключом `stack`. Она добавляет новую метку, но не обновляет существующую. В результате ее выполнения под должен быть дополнен меткой `stack=blue`. Давайте это проверим:

```
$ kubectl get pods nginx-pod --show-labels
NAME        READY   STATUS    RESTARTS   AGE   LABELS
nginx-pod   1/1     Running   0          38m   environment=prod,stack=blue,
                                                tier=frontend
```

Для обновления существующей метки нужно использовать параметр `--overwrite`. Например, изменим значение метки `stack` с `blue` на `green`:

```
$ kubectl label pod nginx-pod stack=green --overwrite
pod/nginx-pod labeled
```

В результате обновления метка с ключом `stack` изменит значение на `green`. Выполните следующую команду, чтобы это проверить:

```
$ kubectl get pods nginx-pod --show-labels
NAME        READY   STATUS    RESTARTS   AGE   LABELS
nginx-pod   1/1     Running   0          41m   environment=prod,stack=green,
                                                tier=frontend
```

Использование команды kubectl label для добавления или изменения меток может быть рискованным. Как мы уже отмечали, взаимосвязи между объектами Kubernetes формируются именно на основе меток. Изменение их значений может нарушить эти взаимосвязи, в результате чего ресурсы начнут вести себя не так, как от них ожидается. Поэтому рекомендуется задавать метки при создании пода и оставлять их неизменными. Всегда лучше удалить и создать ресурс заново, чем изменять конфигурацию уже работающего объекта.

Способ удаления метки запущенного пода мы рассмотрим в следующем разделе.

Удаление метки, присвоенной запущенному поду

Мы можем не только добавлять и обновлять метки, присвоенные запущенному поду, но и удалять их. Для этого после имени метки следует добавить знак минус (`-`). Например, для удаления метки с именем `stack` можно выполнить следующую команду:

```
$ kubectl label pod nginx-pod stack-
pod/nginx-pod unlabeled
```

Использование данного символа в конце команды может показаться странным, но при повторном выполнении команды `kubectl get pods --show-labels` мы увидим, что метка с ключом `stack` действительно исчезла:

```
$ kubectl get pods nginx-pod --show-labels
NAME        READY   STATUS    RESTARTS   AGE   LABELS
nginx-pod   1/1     Running   0          42m   environment=prod,tier=frontend
```

Теперь поговорим об аннотациях в Kubernetes.

Добавление аннотации

Аннотации в Kubernetes представляют собой пары «ключ — значение», которые можно прикреплять к различным объектам системы вроде Pod, Deployment и Service. Они позволяют добавлять к этим объектам дополнительную информацию без изменения их основной функциональности. В отличие от меток, которые используются для идентификации и выборки ресурсов, аннотации предназначены для хранения вспомогательных данных, повышающих удобство восприятия кода или используемых внешними инструментами. В аннотациях можно хранить, например, сведения о конфигурации или имя автора объекта.

Рассмотрим аннотированный под:

```
# annotated_pod.yaml
apiVersion: v1
kind: Pod
metadata:
  name: nginx-pod
  labels:
    environment: prod
    tier: frontend
  annotations:
    tier: webserver
spec:
  containers:
    - name: nginx-container
      image: nginx:latest
```

Здесь аннотация `tier: webserver` может указывать на то, что в поде запущен HTTP-сервер. Помните, что аннотации используются для дополнения объекта метаданными.

Для обновления существующей конфигурации пода можно воспользоваться командой `kubectl replace -f`.

Имя аннотации может иметь префикс в виде DNS-имени. Такой префикс используется, например, в системных компонентах Kubernetes (таких как `kube-scheduler`), чтобы показать пользователям кластера, что данный компонент

входит в ядро Kubernetes. При этом префикс можно опустить, как в приведенном выше примере.

Команда kubectl replace используется для обновления или замены существующих ресурсов Kubernetes на основании манифеста. В отличие от команды kubectl apply -f, которая пытается объединить изменения, команда kubectl replace полностью перезаписывает конфигурацию ресурса, заменяя ее новой версией из манифеста. Такой подход полезен в тех случаях, когда вы хотите применить определенную конфигурацию, независимо от текущего состояния объекта, или когда существующее определение ресурса повреждено и требует полной замены.

Просмотреть аннотации можно несколькими способами: с помощью команд `kubectl describe pod`, `kubectl get po -o yaml` или утилиты `jq`:

```
$ kubectl get pod nginx-pod -o json | jq '.metadata.annotations'
{
  "cni.projectcalico.org/containerID":
    "666d12cd2fb7d6ffe09add73d8466db218f01e7c7ef5315ef0187a675725b5ef",
  "cni.projectcalico.org/podIP":
    "10.244.151.1/32",
  "cni.projectcalico.org/podIPs":
    "10.244.151.1/32",
  "kubectl.kubernetes.io/last-applied-configuration":
    "{\"apiVersion\":\"v1\",\"kind\":\"Pod\",\"metadata\":{\"annotations\":{},
    \"name\":\"nginx-pod\",\"namespace\":\"default\"},\"spec\":{\"containers\":
    [{\"image\":\"nginx:latest\",\"name\":\"nginx-container\"}]}}\n"
}
```

Запуск объекта Job

Теперь познакомим вас еще с одним объектом Kubernetes, происходящим от подов, — ресурсом Job. В Kubernetes вычислительным ресурсом является под, а все остальные объекты представляют собой промежуточные ресурсы, управляющие подами.

Объект Job создает один или несколько подов для выполнения конкретной вычислительной задачи, например для запуска команды Linux.

Что такое Job

Job (задание) — это еще один ресурс, предоставляемый API Kubernetes, который создает один или несколько подов для выполнения определенной вами команды. Здесь важно понять взаимосвязь: объект Job не существует сам по себе, без подов он бесполезен. Его основная задача — запускать поды и управлять ими.

После выполнения определенной задачи объект Job должен завершить работу. Вот несколько типичных сценариев использования этого объекта:

- создание резервной копии базы данных;
- отправка электронного письма;
- обработка сообщений в очереди.

Подобные задачи не должны выполняться бесконечно. После их выполнения поды должны завершить работу. Именно для таких случаев и предназначен ресурс Job.

Но зачем использовать отдельный ресурс для выполнения команды, если можно напрямую создать под (или несколько), способный выполнить команду и завершить работу?

Действительно, вы можете запустить под на основе нужного контейнерного образа для выполнения поставленной задачи. Однако объект Job реализует механизмы управления подами, которые обеспечивают более широкие возможности. Например, этот объект может:

- запускать поды многократно;
- запускать поды параллельно;
- повторять запуск подов при возникновении ошибок;
- завершать работу пода после заданного количества секунд.

Кроме того, объект Job управляет метками создаваемых им подов, избавляя вас от необходимости делать это вручную.

Теоретически все это можно реализовать и без использования объекта Job, однако управлять подобным процессом было бы крайне сложно. Именно поэтому в Kubernetes существует соответствующий ресурс.

Создание объекта Job с параметром restartPolicy

Поскольку создание объекта Job может требовать довольно сложной настройки, мы воспользуемся декларативным подходом и создадим задание Kubernetes с помощью YAML-файла. Для простоты мы ограничим задание выводом строки `Hello world`:

```
# hello-world-job.yaml
apiVersion: batch/v1
kind: Job
metadata:
  name: hello-world-job
```

```
spec:
  template:
    metadata:
      name: hello-world-job
    spec:
      restartPolicy: OnFailure
      containers:
        - name: hello-world-container
          image: busybox
          command: ["/bin/sh", "-c"]
          args: ["echo 'Hello world'"]
```

Обратите внимание на значение поля `kind`, которое сообщает системе Kubernetes о том, что мы создаем задание, а не под. Заметьте также, что параметр `apiVersion` отличается от того, который использовался при создании пода (рис. 4.5).

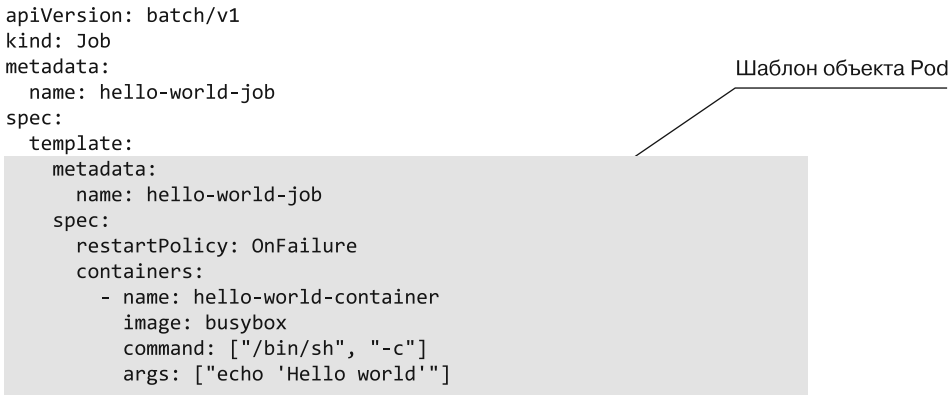

Рис. 4.5. Определение объекта Job и шаблон объекта Pod

Создать задание можно с помощью следующей команды:

```
$ kubectl apply -f hello-world-job.yaml
```

Как видите, этот объект Job предназначен для создания пода на основе контейнерного образа `busybox`:

```
$ kubectl get jobs
NAME             COMPLETIONS   DURATION   AGE
hello-world-job  1/1           9s         8m46s
```

В данном случае выполняется команда `echo 'Hello World'`. Параметр `restartPolicy` (политика перезапуска) установлен в значение `OnFailure`, обеспечивающее перезапуск пода или контейнера при сбое. Если сбой произойдет на уровне всего пода, будет создан новый под. Если же сбой произойдет на уровне

контейнера (например, из-за превышения лимита памяти или возврата ненулевого кода завершения), он будет перезапущен на том же узле, так как под при этом остается в рабочем состоянии и продолжает быть назначенным той же машине.

Параметр `restartPolicy` может принимать два значения:

- `Never`;
- `OnFailure`.

Если установить значение `Never`, то объект Job не будет перезапускать поды даже в случае их сбоя. Такой режим следует использовать при отладке нерабочего задания, в противном случае поды будут создаваться снова и снова, затрудняя диагностику проблемы.

В нашем случае вероятность сбоя минимальна, поскольку наше задание всего лишь запускает простую команду `Hello world`. Чтобы убедиться в успешном выполнении задания, можно просмотреть логи объекта Job:

```
$ kubectl logs jobs/hello-world-job
Hello world
```

Мы также можем получить имя пода, созданного заданием, с помощью команды `kubectl get pods`, а затем просмотреть его логи, выполнив команду `kubectl logs`:

```
$ kubectl logs pods/hello-world-job-2qh4d
Hello world
```

Отображение сообщения `Hello world` в журнале пода говорит о том, что задание было выполнено успешно. Но что произойдет в случае сбоя? Все зависит от значения параметра `restartPolicy`. При использовании значения `Never` система Kubernetes ничего не предпримет, то есть не станет запускать поды повторно.

Если же значением параметра `restartPolicy` является `OnFailure`, система попробует перезапустить задание через 10 секунд, а затем будет удваивать интервал между попытками: 10, 20, 40, 80 секунд и т. д. Через 6 минут система прекратит попытки перезапуска.

Параметр backoffLimit

По умолчанию объект Job будет пытаться пересоздать сбойный под шесть раз в течение следующих шести минут после его сбоя. Это ограничение можно изменить, установив параметр `backoffLimit`. Вот обновленный YAML-файл:

```
# hello-world-job-2.yaml
apiVersion: batch/v1
kind: Job
```

```
metadata:
  name: hello-world-job-2
spec:
  backoffLimit: 3
  template:
    metadata:
      name: hello-world-job-2
    spec:
      restartPolicy: OnFailure
      containers:
        - name: hello-world-container
          image: busybox
          command: ["/bin/sh", "-c"]
          args: ["echo 'Hello world'"]
```

Теперь в случае сбоя пода объект Job попытается пересоздать его лишь трижды.

Многократное выполнение задания с помощью параметра completions

Для многократного выполнения задания можно использовать параметр объекта Job — `completions`, задающий число повторных запусков команды. Значение `completions` в показанном ниже обновленном YAML-файле приведет к последовательному созданию десяти подов. Как только один под завершит выполнение, будет запущен следующий:

```
# hello-world-job-3.yaml
apiVersion: batch/v1
kind: Job
metadata:
  name: hello-world-job-3
spec:
  backoffLimit: 3
  completions: 10
  template:
    metadata:
      name: hello-world-job-3
    spec:
      restartPolicy: OnFailure
      containers:
        - name: hello-world-container
          image: busybox
          command: ["/bin/sh", "-c"]
          args: ["echo 'Hello world'; sleep 3"]
```

Обратите внимание на то, что, помимо параметра `completions`, мы добавили в раздел `args` параметр `sleep 3`, который задает паузу 3 секунды перед завершением задачи, позволяющую нам заметить создание следующего пода. После

применения этого конфигурационного файла к кластеру Kubernetes вы можете выполнить следующую команду:

```
$ kubectl get pods --watch
$ kubectl get jobs -w

NAME                STATUS    COMPLETIONS   DURATION   AGE
hello-world-job-2   Complete  1/1           10s        52s
hello-world-job-3   Running   5/10          49s        49s
hello-world-job-3   Running   5/10          51s        51s
hello-world-job-3   Running   5/10          52s        52s
hello-world-job-3   Running   6/10          52s        52s
hello-world-job-3   Running   6/10          56s        56s
<фрагмент опущен для краткости>
```

Механизм `watch` (`-w` или `--watch`) будет обновлять вывод `kubectl` при появлении новых объектов, например при создании подов, управляемых Kubernetes. Если вы решите дождаться завершения задания, то сможете наблюдать последовательный запуск десяти подов с паузами по 3 секунды.

Многократное выполнение задания в параллельном режиме

Параметр `completions` обеспечивает последовательное создание подов. Однако мы можем запускать их и в параллельном режиме с помощью параметра `parallelism`. В этом случае параметр `completions` становится ненужным. Объекты Job системы Kubernetes могут реализовать параллелизм для значительного ускорения выполнения. Запуская несколько подов одновременно, вы распределяете нагрузку по кластеру, что ведет к более быстрой обработке задач и лучшему использованию ресурсов, особенно при выполнении масштабных или сложных заданий. Рассмотрим обновленный YAML-файл:

```
# hello-world-job-4.yaml
apiVersion: batch/v1
kind: Job
metadata:
  name: hello-world-job-4
spec:
  backoffLimit: 3
  parallelism: 5
  template:
    metadata:
      name: hello-world-job-4
    spec:
      restartPolicy: OnFailure
      containers:
        - name: hello-world-container
          image: busybox
          command: ["/bin/sh", "-c"]
          args: ["echo 'Hello world'; sleep 3"]
```

Обратите внимание на то, что параметр `completions` был заменен параметром `parallelism`. Теперь задание будет запускать пять подов одновременно, выполняя их в параллельном режиме:

```
$ kubectl get pods -w
NAME                    READY   STATUS              RESTARTS   AGE
hello-world-job-4-9dspk   0/1     ContainerCreating   0          7s
hello-world-job-4-n6qv9   0/1     Completed           0          7s
hello-world-job-4-pv754   0/1     ContainerCreating   0          7s
hello-world-job-4-ss4g8   1/1     Running             0          7s
hello-world-job-4-v78cj   1/1     Running             0          7s
<фрагмент опущен для краткости>
```

В следующем подразделе вы узнаете, как автоматически завершить задание по истечении заданного промежутка времени.

Завершение задания по истечении определенного времени

Мы также можем завершить работу пода по истечении заданного промежутка времени. Это полезно, например, при работе с заданиями, которые обрабатывают очередь сообщений. В таком случае можно проверять очередь в течение одной минуты, а затем автоматически завершать этот процесс. Для этого используется параметр `activeDeadlineSeconds`. Вот обновленный YAML-файл:

```
# hello-world-job-5.yaml
apiVersion: batch/v1
kind: Job
metadata:
  name: hello-world-job-5
spec:
  backoffLimit: 3
  activeDeadlineSeconds: 60
  template:
    metadata:
      name: hello-world-job-5
    spec:
      restartPolicy: OnFailure
      containers:
        - name: hello-world-container
          image: busybox
          command: ["/bin/sh", "-c"]
          args: ["echo 'Hello world'"]
```

В данном примере выполнение задания завершится через 60 секунд независимо от обстоятельств. Этот механизм целесообразно применять в ситуациях, когда вам требуется поддерживать процесс в активном состоянии в течение строго определенного временного интервала.

Что происходит при успешном завершении задания

После выполнения задание по умолчанию сохраняется в кластере Kubernetes и не удаляется автоматически. Это дает возможность просмотреть его логи даже спустя долгое время после завершения. Однако хранение выполненных заданий в кластере может оказаться нежелательным. В этом случае можно настроить автоматическое удаление заданий и созданных ими подов с помощью параметра `ttlSecondsAfterFinished`. Вот обновленный YAML-файл с этой конфигурацией:

```
# hello-world-job-6.yaml
apiVersion: batch/v1
kind: Job
metadata:
  name: hello-world-job-6
spec:
  backoffLimit: 3
  ttlSecondsAfterFinished: 30
  template:
    metadata:
      name: hello-world-job-6
    spec:
      restartPolicy: OnFailure
      containers:
        - name: hello-world-container
          image: busybox
          command: ["/bin/sh", "-c"]
          args: ["echo 'Hello world'"]
```

Теперь задания будут удаляться спустя 30 секунд после завершения.

Удаление задания

Помните, что поды, создаваемые в рамках выполнения задания, связаны с его жизненным циклом. Удаление задания приведет к удалению управляемых им подов.

Сначала получите имя задания, которое требуется удалить. В нашем случае это `hello-world-job`. Если имя вам неизвестно, используйте команду `kubectl get jobs`, чтобы его уточнить. Затем выполните следующую команду:

```
$ kubectl delete jobs hello-world-job
```

Если вы хотите удалить задание, но сохранить созданные им поды, добавьте параметр `--cascade=false` к команде `delete`:

```
$ kubectl delete jobs hello-world-job --cascade=false
```

Эта команда позволяет вам избавляться от всех завершенных заданий, которые в противном случае сохранялись бы в вашем кластере Kubernetes. Теперь перейдем к запуску объектов CronJob.

Запуск объекта CronJob

В завершение вашего знакомства с подами рассмотрим еще один ресурс Kubernetes под названием *CronJob*.

Что такое CronJob

Название *CronJob* может обозначать два разных понятия, которые важно не путать:

- планировщик задач `cron` в Unix;
- ресурс CronJob в системе Kubernetes.

Изначально объекты CronJob представляли собой команды, запускаемые по расписанию с помощью механизма `cron` в ОС семейства Unix, который считается наиболее надежным способом планирования выполнения команд в таких системах. Позднее эта идея была реализована и в Kubernetes.

Однако учтите, что, несмотря на схожесть этих концепций, они работают поразному. В Unix-подобных системах планирование выполнения команд осуществляется путем редактирования файла `crontab`, который обычно расположен по адресу `/etc/crontab` или в других связанных директориях. В Kubernetes все иначе: вы планируете не выполнение команд, а выполнение ресурсов Job, которые, в свою очередь, создают поды. Имейте в виду, что созданный вами объект CronJob будет порождать объекты Job.

По сути, CronJob представляет собой обертку вокруг ресурса Job, которая в Kubernetes называется контроллером. Объект CronJob может делать все то же, что и объект Job, поскольку он лишь оборачивает его, используя заданное cron-выражение.

Хорошая новость в том, что ресурс CronJob в Kubernetes использует тот же формат `cron`, что и Unix. Поэтому если вы уже писали cron-выражения в Linux, то без труда освоите ресурс CronJob системы Kubernetes.

Вот несколько типичных ситуаций, в которых выполнение пода по расписанию может оказаться уместным:

- создание резервных копий базы данных каждое воскресенье в 01:00;
- очистка кэша каждый понедельник в 16:00;
- отправка электронных писем каждые 5 минут;
- регулярное выполнение операций, связанных с обслуживанием системы.

Объекты CronJob в Kubernetes решают примерно те же задачи, что и их аналоги в Unix. Главное преимущество заключается в том, что вы можете использовать уже настроенный кластер Kubernetes, чтобы регулярно выполнять задания, опираясь на собственные контейнерные образы и существующую инфраструктуру Kubernetes.

Подготовка к созданию объекта CronJob

Для создания первого объекта CronJob воспользуемся декларативным подходом. Для начала создадим файл `cronjob.yaml` и поместим в него следующий код:

```
# hello-world-cronjob.yaml
apiVersion: batch/v1
kind: CronJob
metadata:
  name: hello-world-cronjob
spec:
  schedule: "* * * * *"
  jobTemplate:
    spec:
      template:
        metadata:
          name: hello-world-cronjob
        spec:
          restartPolicy: OnFailure
          containers:
            - name: hello-world-container
              image: busybox
              imagePullPolicy: IfNotPresent
              command: ["/bin/sh", "-c"]
              args: ["echo 'Hello world'"]
```

Прежде чем применить этот файл к кластеру Kubernetes, обсудим два важных элемента:

- ключ `schedule`, позволяющий задать cron-выражение;
- раздел `jobTemplate`, содержимое которого полностью соответствует тому, что вы указали бы в YAML-манифесте объекта Job.

Разберем эти два элемента подробнее.

Ключ schedule

Ключ `schedule` позволяет задать выражение в формате `cron`, используемом в Linux. Посмотрим, как устроены эти выражения (если вы уже знакомы с ними, можете пропустить этот раздел):

```
# ┌──────────────── минута (0–59)
# │ ┌────────────── час (0–23)
# │ │ ┌──────────── день месяца (1–31)
# │ │ │ ┌────────── месяц (1–12)
# │ │ │ │ ┌──────── день недели (0–6) (воскресенье – суббота;
# │ │ │ │ │           в некоторых системах 7 может означать воскресенье)
# │ │ │ │ │           ИЛИ sun, mon, tue, wed, thu, fri, sat
# │ │ │ │ │
# * * * * *
```

Cron-выражение состоит из пяти полей, разделенных пробелами. Слева направо эти поля означают:

- минуты;
- часы;
- день месяца;
- месяц;
- день недели.

Любое поле может быть заполнено символом *, который означает «каждый». В одном поле можно указать несколько значений, разделив их запятыми. Можно также указать диапазоны значений с помощью дефиса (-). Вот несколько примеров.

- `"10 11 * * *"` означает «в 11:10 каждый день каждого месяца».
- `"10 11 * 12 *"` означает «в 11:10 каждый день декабря».
- `"10 11 * 12 1"` означает «в 11:10 каждый понедельник декабря».
- `"10 11 * * 1,2"` означает «в 11:10 каждый понедельник и вторник каждого месяца».
- `"10 11 * 2-5 *"` означает «в 11:10 каждый день с февраля по май».

Эти примеры помогут вам лучше понять принцип работы cron-выражений. Разумеется, запоминать синтаксис не обязательно: большинство пользователей опираются на документацию или специальные генераторы cron-выражений, такие как `crontab.cronhub.io` или `crontab.guru`. Если синтаксис кажется слишком сложным, смело используйте такие инструменты, чтобы проверить правильность выражения перед развертыванием объекта в Kubernetes.

Роль раздела jobTemplate

Если вы внимательно следили за структурой YAML-файла, то, вероятно, заметили, что раздел `jobTemplate` содержит определение объекта Job. Используя объект CronJob, мы просто делегируем ему создание объекта Job (рис. 4.6).

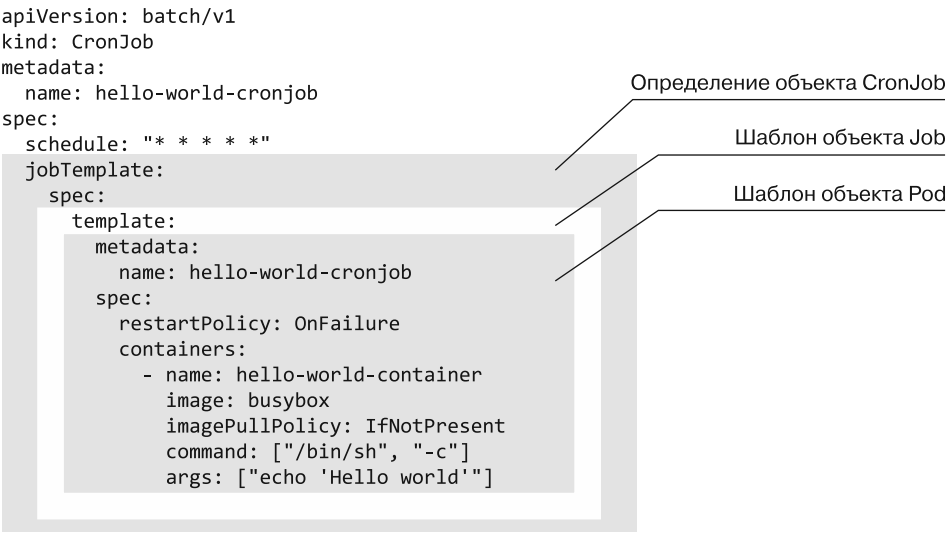

Рис. 4.6. Структура объекта CronJob

Таким образом, объект CronJob — это ресурс, который управляет другим ресурсом.

Позднее мы познакомимся со многими объектами, позволяющими создавать поды и освобождающими нас от необходимости делать это вручную. Эти специальные объекты называются *контроллерами*, потому что управляют ресурсами Kubernetes, руководствуясь собственной логикой. Более того, объект Job тоже является контроллером, поскольку в конечном счете он лишь управляет подами, дополняя их собственными возможностями, в частности возможностью запуска подов в параллельном режиме.

В реальных сценариях всегда следует стремиться создавать поды через такие промежуточные объекты, так как они предоставляют дополнительные и более совершенные механизмы управления.

Запомните правило: базовой единицей в Kubernetes является под, но его создание можно делегировать множеству других объектов. В оставшейся части этого раздела мы продолжим рассмотрение «голых» подов, а в следующих главах обсудим способы управления их созданием и жизненным циклом с помощью контроллеров.

Управление сроком выполнения объекта CronJob

Объект CronJob может не выполниться по тем или иным причинам. В этом случае Kubernetes не сможет запустить задание в запланированное время. Задания, превысившие установленный срок выполнения, считаются неуспешными.

Необязательное поле `.spec.startingDeadlineSeconds` задает предельное время (в секундах) для запуска задания на случай, если запланированный момент

его запуска был по какой-то причине пропущен. При превышении этого срока объект CronJob пропускает данный конкретный запуск задания, но не отменяет последующие запланированные запуски.

Управление историей выполнения заданий

Кластер Kubernetes сохраняет историю выполнения всех заданий CronJob независимо от их результата. Политику хранения этой истории можно настроить индивидуально для каждого объекта CronJob, указав количество сохраняемых записей об успешных и неуспешных запусках. Для этого используются необязательные параметры `.spec.successfulJobsHistoryLimit` и `.spec.failedJobsHistoryLimit`.

Создание объекта CronJob

При наличии YAML-манифеста создание объекта CronJob осуществляется очень просто:

```
# hello-world-cronjob.yaml
apiVersion: batch/v1
kind: CronJob
metadata:
  name: hello-world-cronjob
spec:
  schedule: "*/1 * * * *"   # запуск каждую минуту
  successfulJobsHistoryLimit: 5
  startingDeadlineSeconds: 30
  jobTemplate:
    spec:
      template:
        metadata:
          name: hello-world-cronjob
        spec:
          restartPolicy: OnFailure
          containers:
            - name: hello-world-container
              image: busybox
              imagePullPolicy: IfNotPresent
              command: ["/bin/sh", "-c"]
              args: ["echo 'Hello world'"]
```

В данном случае параметр `successfulJobsHistoryLimit: 5` указывает контроллеру CronJob хранить сведения о последних пяти успешных запусках задания. История более ранних успешных заданий будет удаляться автоматически.

Объект CronJob можно создать с помощью команды `kubectl apply -f`:

```
$ kubectl apply -f hello-world-cronjob.yaml
cronjob.batch/hello-world-cronjob created
```

После этого вновь созданный объект CronJob будет запускать под по расписанию, указанному в YAML-файле, — в данном случае каждую минуту:

```
$ kubectl get cronjobs
NAME                 SCHEDULE     TIMEZONE SUSPEND  ACTIVE  LAST SCHEDULE  AGE
hello-world-cronjob  */1 * * * *  <none>   False    0       37s            11m

$ kubectl get jobs
NAME                             COMPLETIONS   DURATION   AGE
hello-world-cronjob-28390196   1/1           3s         4m47s
hello-world-cronjob-28390197   1/1           3s         3m47s
hello-world-cronjob-28390198   1/1           3s         2m47s
hello-world-cronjob-28390199   1/1           3s         107s
hello-world-cronjob-28390200   1/1           4s         47s

$ kubectl get pods
NAME                                 READY   STATUS      RESTARTS   AGE
hello-world-cronjob-28390196-fpmc6   0/1     Completed   0          4m52s
hello-world-cronjob-28390197-vkzw2   0/1     Completed   0          3m52s
hello-world-cronjob-28390198-tj6qv   0/1     Completed   0          2m52s
hello-world-cronjob-28390199-dd666   0/1     Completed   0          112s
hello-world-cronjob-28390200-kn89r   0/1     Completed   0          52s
```

Поскольку мы задали значение `successfulJobsHistoryLimit: 5`, нам будут видны только последние пять заданий или запущенных подов.

Удаление объекта CronJob

Как и любой другой ресурс Kubernetes, объект CronJob можно удалить с помощью команды `kubectl delete`. При наличии YAML-манифеста это делается очень просто:

```
$ kubectl delete -f hello-world-cronjob.yaml
cronjob/hello-world-cronjob deleted
```

После выполнения этой команды объект CronJob будет удален из вашего кластера Kubernetes и задания больше не будут запускаться по расписанию.

Резюме

Итак, мы закончили главу, посвященную подам, самым важным объектам Kubernetes, и способам их использования.

Материал этой главы составляет основу для дальнейшего освоения Kubernetes: все, что вы будете делать в этой системе, сводится к манипулированию подами, их маркировке и получению доступа к ним. Однако помните, что в реальных

средах Kubernetes вы не будете создавать или изменять ресурсы напрямую, поскольку для этого обычно используются другие методы развертывания подов приложений и других объектов. Кроме того, теперь вы знаете, что система Kubernetes ведет себя как традиционный API, выполняющий CRUD-операции для взаимодействия с ресурсами кластера. В этой главе вы научились запускать контейнеры, подключаться к ним с помощью проброса портов, добавлять метки и аннотации к подам, удалять поды, а также запускать и планировать задания с помощью ресурса CronJob.

Запомните, что в Kubernetes любой контейнер запускается внутри пода. Поэтому для освоения этой системы нужно в первую очередь освоить работу с подами, так как все остальные ресурсы Kubernetes служат лишь для автоматизации процесса управления ими. Например, единственная функция объекта CronJob сводится к созданию объектов Job, которые, в свою очередь, создают поды. Если вы поняли, что некоторые объекты могут управлять другими, но в конечном счете все контейнеры управляются подами, значит, вы постигли философию Kubernetes и дальнейшее освоение этого оркестратора не вызовет затруднений.

Мы также рекомендуем вам снабжать свои поды метками и аннотациями, даже если необходимость в этом пока неочевидна. Корректное использование меток и аннотаций является важным условием поддержания кластера в структурированном и хорошо организованном состоянии.

В дальнейшем вам предстоит еще многое узнать об управлении подами. До этого момента мы рассматривали только поды, содержащие один Docker-контейнер. Однако настоящая сила подов заключается в возможности управления несколькими контейнерами одновременно. Чтобы делать это правильно, применяются специальные паттерны проектирования, облегчающие работу с многоконтейнерными подами.

В следующей главе вы узнаете, как управлять подами, состоящими из нескольких контейнеров. Хотя в целом они очень похожи на те поды, с которыми вы уже познакомились, у них есть определенные особенности, достойные обсуждения. Сначала вы узнаете, как запускать многоконтейнерные поды с помощью утилиты `kubectl` (спойлер: это не сработает), а затем — как организовать взаимодействие контейнеров внутри одного пода. Далее мы рассмотрим способ обращения к конкретному контейнеру в многоконтейнерном поде и получения его логов. А в конце вы узнаете, как организовывать общий доступ к томам для контейнеров одного пода.

Таким образом, следующая глава познакомит вас с оставшимися ключевыми концепциями работы с подами, а также с другими объектами Kubernetes, которые могут пригодиться для развертывания приложений в ваших кластерах.

Дополнительные источники информации

- Управление подами: https://kubernetes.io/docs/concepts/workloads/pods/.
- Объекты Job в Kubernetes: https://kubernetes.io/docs/concepts/workloads/controllers/job/.
- Стандартные метки, аннотации и ограничения: https://kubernetes.io/docs/reference/labels-annotations-taints/.
- Объект CronJob в Kubernetes: https://kubernetes.io/docs/concepts/workloads/controllers/cron-jobs/.

5 Использование многоконтейнерных подов и паттерны проектирования

Запуск сложных приложений в Kubernetes требует использования не одного, а сразу нескольких контейнеров внутри одного пода. Мощь данного оркестратора проявляется в том числе в возможности создания подов, состоящих из нескольких контейнеров. В этой главе мы сосредоточимся именно на таких подах, изучим разные варианты их организации, а также способы налаживания взаимодействия между составляющими их контейнерами.

До этого момента мы создавали поды с одним контейнером. Эти простейшие формы подов применяются для управления простыми приложениями. Мы также обсудили способы обновления и удаления таких подов путем выполнения стандартных *CRUD-операций* (`create` (создание), `read` (чтение), `update` (обновление) и `delete` (удаление)) с помощью утилиты `kubectl`.

Помимо освоения базовых CRUD-операций, вы научились получать доступ к подам, работающим внутри кластера Kubernetes.

Хотя поды с одним контейнером встречаются чаще всего, в некоторых ситуациях целесообразным является использование нескольких контейнеров в одном поде. Например, в дополнение к основному контейнеру приложения в под можно добавить контейнер для сбора логов или контейнер-прокси для организации сетевого взаимодействия между сервисами. В этой главе мы поговорим об управлении подами, содержащими несколько контейнеров. Хорошая новость заключается в том, что все, чему вы научились ранее, будет справедливо и для многоконтейнерных подов, так как управление подами, в том числе их обновление и удаление, осуществляется одинаково, независимо от числа содержащихся в них контейнеров.

Кроме этих базовых операций, мы обсудим способ обращения к конкретному контейнеру в поде и просмотра его логов. Если под содержит несколько контейнеров, то для доступа к ним придется использовать специальные команды с дополнительными аргументами, которые мы тоже обсудим в данной главе.

Мы также рассмотрим такие важные паттерны проектирования, как Ambassador (Посредник), Sidecar (Прицеп) и Adapter (Адаптер), освоение которых

необходимо для эффективного управления многоконтейнерными подами. Кроме того, вы узнаете, как работать с томами в Kubernetes. Docker тоже позволяет использовать тома, но в Kubernetes они применяются для обмена данными между контейнерами одного пода. После изучения этой главы вы сможете запускать сложные приложения внутри подов Kubernetes.

Технические требования

Для работы с этой главой вам потребуется:

- установленная утилита `kubectl`;
- локальный или облачный кластер Kubernetes.

Примеры кода для этой главы можно загрузить из официального репозитория GitHub по адресу https://github.com/PacktPublishing/The-Kubernetes-Bible-Second-Edition/tree/main/Chapter05.

Многоконтейнерные поды

Многоконтейнерные поды — это вариант объединения тесно связанных приложений в Kubernetes. Такой подход позволяет нескольким контейнерам совместно использовать ресурсы и легко обмениваться данными, что особенно полезно в сценариях с использованием контейнеров-прицепов и сервисных сеток. В этом разделе мы разберем основные концепции одновременного управления несколькими контейнерами на конкретных примерах многоконтейнерных подов.

Сценарии использования многоконтейнерных подов

В один под следует объединять те контейнеры, которые должны быть тесно связаны между собой. В общем случае под должен соответствовать приложению или процессу, выполняющемуся в кластере Kubernetes. Если для корректной работы вашего приложения требуется несколько контейнеров, то они должны запускаться и управляться в рамках одного пода.

Если контейнеры предполагается использовать совместно, их следует группировать в один под. Помните, что под не может работать на нескольких вычислительных узлах. Следовательно, если вы создаете под, содержащий несколько контейнеров, то все они будут запущены на одном узле. Чтобы понять, где и когда уместно применять многоконтейнерные поды, рассмотрим два простых примера.

- *Пересылка логов*. Допустим, вы развернули веб-сервер (например, Nginx), который сохраняет свои логи в отдельной папке. Если вы решите пересылать эти логи, то сможете запустить контейнер с инструментом для пересылки логов (например, Splunk forwarder) в поде с сервером Nginx.

Инструменты для пересылки логов используются для передачи содержимого журналов от источника к месту назначения. Обычно для этого разворачиваются такие агенты, как Splunk, Fluentd или Filebeat, которые забирают логи из контейнера и отправляют их в центральное хранилище, например в кластер Elasticsearch. В мире Kubernetes это обычно реализуется путем запуска многоконтейнерного пода, в котором один контейнер выполняет приложение, а другой отвечает за сбор логов и их пересылку. Управление обоими контейнерами в рамках одного пода гарантирует их одновременный запуск на одном узле.

- *Прокси-сервер*. Представьте контейнер с обратным прокси Nginx, находящийся в одном поде с основным приложением, который эффективно маршрутизирует трафик и обеспечивает безопасность с помощью пользовательских правил. Эта же концепция лежит в основе *сервисной сетки*, где рядом с контейнером приложения разворачивается специализированный прокси-сервер, например Envoy, что позволяет реализовать такие функции, как балансировка нагрузки и обнаружение сервисов в рамках микросервисной архитектуры. (Подробнее о сервисной сетке мы поговорим в главе 8.) При объединении двух контейнеров в один под эти контейнеры будут запущены на одном узле. Более того, в этот же под можно добавить третий контейнер, который будет пересылать логи от первых двух в центральное хранилище. В Kubernetes нет ограничения на количество контейнеров в поде — важно лишь, чтобы ресурсов узла хватало на их выполнение (рис. 5.1).

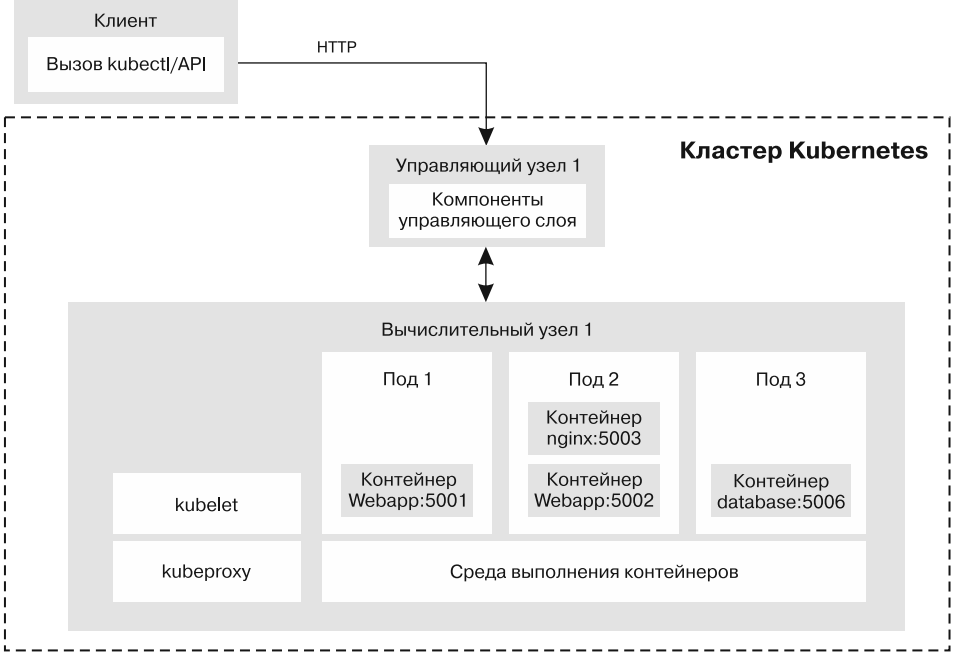

Рис. 5.1. Пример многоконтейнерного пода

Итак, если несколько контейнеров должны работать совместно и быть тесно связаны между собой, то их следует объединять в многоконтейнерный под.

Теперь обсудим процесс создания таких подов.

Создание пода, состоящего из двух контейнеров

В предыдущей главе мы выяснили, что для управления объектами в Kubernetes можно использовать:

- императивный синтаксис;
- декларативный синтаксис.

Большинство объектов Kubernetes, с которыми мы будем работать в этой книге, можно создавать и обновлять обоими способами. Однако многоконтейнерные поды являются исключением.

Для их создания необходимо использовать декларативный синтаксис. Это означает, что вы должны создать YAML-файл с описанием пода и всех контейнеров, которые он будет содержать, а затем применить его с помощью команды `kubectl apply -f file.yaml`.

Рассмотрим следующий YAML-манифест, сохраненный по адресу `~/multi-container-pod.yaml`:

```
# multi-container-pod.yaml
apiVersion: v1
kind: Pod
metadata:
  name: multi-container-pod
spec:
  restartPolicy: Never
  containers:
    - name: nginx-container
      image: nginx:latest
    - name: debian-container
      image: debian
      command: ["/bin/sh"]
      args: ["-c", "while true; do date; echo debian-container; sleep 5; done"]
```

Этот файл предназначен для создания пода Kubernetes, который содержит два контейнера: один на основе образа `nginx:latest`, а другой — на основе образа `debian`.

Для его создания используйте команду:

```
$ kubectl apply -f multi-container-pod.yaml
pod/multi-container-pod created
```

В результате будет создан под. Агент `kubelet` на выбранном узле поручит среде выполнения контейнеров (например, `containerd`, CRI-O или демону Docker) скачать оба образа и создать на их основе два контейнера.

Чтобы удостовериться в том, что под действительно был создан, выполните команду `kubectl get pods`:

```
$ kubectl get pods
NAME                READY   STATUS    RESTARTS   AGE
multi-container-pod  2/2     Running   0          2m7s
```

Напомним, что компонент `kubelet` (описанный в главе 2) работает на каждом узле кластера Kubernetes и отвечает за преобразование манифестов подов, полученных от `kube-apiserver`, в реальные контейнеры.

Все контейнеры, объявленные в составе одного пода, будут планироваться и запускаться на одном узле, а поды не могут работать на нескольких машинах.

Контейнеры внутри пода предназначены для совместной работы: при завершении работы пода все его контейнеры будут остановлены одновременно, а при создании пода агент kubelet как минимум попытается запустить все контейнеры сразу.

Высокая доступность в Kubernetes обычно обеспечивается путем репликации подов на разных узлах, о чем мы поговорим далее в книге.

С точки зрения Kubernetes применение такого манифеста создает полностью работоспособный многоконтейнерный под. Убедиться в том, что под запущен и содержит два контейнера, можно с помощью стандартной команды `kubectl get pods`, которая запрашивает список подов у компонента `kube-apiserver`.

Обратите внимание на столбец со значением `2/2` в предыдущем выводе. Оно отражает количество контейнеров внутри пода. В данном случае оно говорит об успешном запуске обоих контейнеров, входящих в состав пода. Логи из разных контейнеров можно просмотреть следующим образом:

```
$ kubectl logs multi-container-pod -c debian-container
Mon Jan  8 01:33:23 UTC 2024
debian-container
Mon Jan  8 01:33:28 UTC 2024
debian-container
...<фрагмент опущен для краткости>...

$ kubectl logs multi-container-pod -c nginx-container
/docker-entrypoint.sh: /docker-entrypoint.d/ is not empty,
will attempt to perform configuration
...<фрагмент опущен для краткости>...
2024/01/08 01:33:20 [notice] 1#1: start worker process 39
2024/01/08 01:33:20 [notice] 1#1: start worker process 40
```

Итак, теперь вы умеете создавать и управлять многоконтейнерными подами. В следующем подразделе мы поговорим о том, как отлаживать поды в случае их неудачного запуска.

Что происходит при неудачном запуске контейнера внутри пода

Система Kubernetes отслеживает все контейнеры, запускаемые в составе одного пода. Однако бывает, что какой-то контейнер запустить не удается. Давайте специально сделаем ошибку в YAML-манифесте, чтобы посмотреть, как Kubernetes реагирует на эту ситуацию.

В следующем файле мы указали несуществующий образ Nginx-контейнера `nginx:i-do-not-exist`:

```
# failed-multi-container-pod.yaml
apiVersion: v1
kind: Pod
metadata:
  name: multi-container-pod
spec:
  restartPolicy: Never
  containers:
    - name: nginx-container
      image: nginx:i-do-not-exist
    - name: debian-container
      image: debian
      command: ["/bin/sh"]
      args: ["-c", "while true; do date; echo debian-container; sleep 5; done"]
```

Теперь применим этот манифест с помощью команды `kubectl apply -f failed-multi-container pod.yaml`:

```
$ kubectl apply -f failed-multi-container-pod.yaml
pod/failed-multi-container-pod created
```

Как видите, под все-таки был создан. Это объясняется тем, что, даже если образ не существует, YAML-манифест остается действительным с точки зрения Kubernetes. Поэтому система создает под и вносит запись о нем в хранилище `etcd`, однако во время запуска контейнера агент `kubelet` сталкивается с ошибкой при попытке получить образ из реестра контейнеров (например, Docker Hub).

Проверим статус пода с помощью команды `kubectl get pod`:

```
$ kubectl get pod
NAME                        READY   STATUS             RESTARTS   AGE
failed-multi-container-pod  1/2     ImagePullBackOff   0          93s
```

Статус `ImagePullBackOff` означает, что Kubernetes пытается запустить под, но сталкивается с ошибкой доступа к образу. Чтобы узнать причину ошибки, нужно выполнить команду `kubectl describe pod failed-multi-container-pod`:

```
$ kubectl describe pod failed-multi-container-pod
Name:         failed-multi-container-pod
Namespace:    default
...<фрагмент опущен для краткости>...
Events:
  Type     Reason   Age                    From     Message
  ----     ------   ----                   ----     -------
  ...<фрагмент опущен для краткости>...
  Warning  Failed   5m23s (x3 over 6m13s)  kubelet  Error: ErrImagePull
  Warning  Failed   4m55s (x5 over 6m10s)  kubelet  Error: ImagePullBackOff
  Normal   Pulling  4m42s (x4 over 6m17s)  kubelet  Pulling image
                                                     "nginx:i-do-not-exist"
  Warning  Failed   4m37s (x4 over 6m13s)   kubelet  Failed to pull image
                                                     "nginx:i-do-not-exist":
                                                     Error response from daemon:
                                                     manifest for nginx:i-do-not-
                                                     exist not found: manifest
                                                     unknown: manifest unknown
  Normal   BackOff  75s (x19 over 6m10s)   kubelet  Back-off pulling image
                                                      "nginx:i-do-not-exist"
```

Проанализировав этот лог, можно сделать вывод о том, что с контейнером `debian-container` все в порядке — агент `kubelet` успешно создал его, на что указывает последняя строка предыдущего вывода. Однако с контейнером `nginx-container` возникла проблема.

Указанная в выводе ошибка `ErrImagePull` говорит о невозможности запуска контейнера из-за неудачной попытки загрузки образа `nginx:i-do-not-exist`.

Итак, система Kubernetes делает следующее.

1. Создает запись в хранилище `etcd`, если YAML-файл с описанием пода является действительным.
2. Пытается запустить контейнер.
3. При возникновении ошибки многократно повторяет попытку запуска проблемного контейнера.

Даже если остальные контейнеры работают корректно, под не перейдет в состояние выполнения (`Running`) до тех пор, пока хотя бы один из них продолжает сбоить. И это логично: ваше приложение наверняка нуждается в этом контейнере для полноценной работы, иначе его просто не следовало включать в под.

Теперь обсудим способ удаления многоконтейнерного пода.

Удаление многоконтейнерного пода

Для удаления пода с несколькими контейнерами используется та же команда `kubectl delete`, что и для удаления одноконтейнерного пода.

Здесь есть два варианта.

- Указать путь к YAML-манифесту с помощью параметра `-f`.
- Удалить под по его имени, не используя путь к YAML-файлу.

Путь к YAML-манифесту указывается так:

```
$ kubectl delete -f multi-container-pod.yaml
```

А если вы знаете имя пода, то можете сделать следующее:

```
$ kubectl delete pods/multi-container-pod
# или эквивалент
$ kubectl delete pods multi-container-pod
```

Чтобы определить имя пода, можно воспользоваться командой `kubectl get`:

```
$ kubectl get pod
NAME                         READY   STATUS             RESTARTS   AGE
failed-multi-container-pod   1/2     ImagePullBackOff   0          13m
```

В нашем случае в кластере был создан только под `failed-multi-container-pod`, поэтому в выводе отображается лишь одна строка.

Удалить его можно императивным способом, без указания пути к YAML-файлу:

```
$ kubectl delete -f failed-multi-container-pod.yaml
pod "failed-multi-container-pod" deleted
```

Через несколько секунд под будет удален из кластера Kubernetes, а все его контейнеры — из контейнерного демона и узла кластера.

Временной интервал между запуском команды и фактическим удалением пода называется *периодом корректного завершения работы* (grace period). О том, как с ним работать, мы поговорим далее.

Период корректного завершения работы пода

Одним из ключевых понятий, связанных с удалением подов, является так называемый период корректного завершения работы, который предусмотрен как для одноконтейнерных, так и для многоконтейнерных подов. Этот период можно проигнорировать, передав параметр `--grace-period=0 --force` команде `kubectl delete`.

Во время удаления пода некоторые команды `kubectl` отображают статус `Terminating`, который не относится к стандартным фазам жизненного цикла пода. Каждому поду выделяется определенный период времени для корректного завершения работы, обычно равный 30 секундам. Для принудительного

завершения работы пода используется флаг `--force`. В случае применения параметров `--grace-period=0` и `--force` имя пода немедленно освобождается и становится доступным для создания нового пода. При обычном удалении система учитывает период корректного завершения работы, и имя пода освобождается только после фактического завершения этой операции.

```
$ kubectl delete pod failed-multi-container-pod --grace-period=0 --force
Warning: Immediate deletion does not wait for confirmation that the running
resource has been terminated. The resource may continue to run on the cluster
indefinitely.
pod "failed-multi-container-pod" force deleted
```

Эту команду следует применять с осторожностью. Принудительное удаление пода не является нормальной практикой, поскольку, как указано в выводе команды, мы не можем удостовериться в фактическом удалении пода. Если по какой-то причине это не удастся сделать, он может продолжать работу неопределенно долго. Поэтому не запускайте эту команду, если не уверены в своих действиях.

Теперь рассмотрим, как получать доступ к конкретному контейнеру, работающему внутри многоконтейнерного пода.

Доступ к конкретному контейнеру внутри многоконтейнерного пода

Когда в одном поде работает несколько контейнеров, мы можем получать доступ к каждому из них по отдельности. В этом примере мы обратимся к контейнеру Nginx внутри многоконтейнерного пода. Поскольку в предыдущем примере мы его удалили, сначала воссоздадим этот под:

```
$ kubectl apply -f multi-container-pod.yaml
pod/multi-container-pod created
```

Для получения доступа к работающему контейнеру используется команда `kubectl exec`. Это аналогично использованию `docker exec` для выполнения команд в уже созданном контейнере при работе с Docker напрямую (то есть без Kubernetes).

Эта команда принимает два параметра:

- имя пода, в котором находится нужный контейнер;
- имя контейнера, заданное в YAML-манифесте.

Имя пода мы уже знаем: его легко получить с помощью команды `kubectl get`. В нашем случае под называется `multi-container-pod`.

Однако нам неизвестно имя контейнера, так как в Kubernetes нет команды `kubectl get containers`, позволяющей вывести список запущенных контейнеров. Поэтому мы воспользуемся командой `kubectl describe pods/multi-container-pod`, чтобы выяснить, какие контейнеры входят в состав целевого пода:

```
$ kubectl describe pods/multi-container-pod
```

Эта команда выведет имена всех контейнеров, работающих в указанном поде. В нашем случае под содержит два контейнера: `debian-container` и `nginx-container`.

Существует и другой способ вывода имен всех контейнеров, входящих в состав пода:

```
$ kubectl get pod/multi-container-pod -o jsonpath="{.spec.containers[*].name}"
nginx-container debian-container
```

Команда избавляет от необходимости использовать `describe`, но задействует `jsonpath`, продвинутую функцию `kubectl`. Этот непривычный синтаксис означает применение фильтра к результату выполнения команды. Выражение `{.spec.containers[*].name}` в связке с `kubectl get pod` возвращает имена всех контейнеров внутри указанного пода. Точка (`.`) обозначает весь объект ответа, `spec.containers` указывает на раздел спецификации пода с контейнерами, оператор `[*]` означает перебор всех элементов в списке контейнеров, а `.name` извлекает значение свойства `name` для каждого контейнера. В итоге получается список имен контейнеров, разделенных запятыми.

Фильтры `jsonpath` сложно составить правильно, поэтому имеет смысл сохранить эту команду, например, в качестве псевдонима в Bash или записать отдельно, поскольку она действительно очень полезна.

В любом случае теперь мы знаем, что внутри пода `multi-container-pod` работают два контейнера:

- `nginx-container`
- `debian-container`

Подключимся к контейнеру `nginx-container`. У нас уже есть имя целевого контейнера в нужном поде, поэтому используем следующую команду:

```
$ kubectl exec -it multi-container-pod --container nginx-container -- /bin/bash
root@multi-container-pod:/# hostname
multi-container-pod
root@multi-container-pod:/#
```

После выполнения этой команды вы окажетесь внутри контейнера `nginx-container`. Разберем ее подробнее. Команда `kubectl exec` выполняет ту же функцию, что и `docker exec`.

- `kubectl exec` — основная команда для запуска процессов внутри контейнера Kubernetes.
- `it`-параметры: `-t` выделяет псевдо-TTY, а `-i` позволяет взаимодействовать с контейнером.
- `multi-container-pod` — имя пода, внутри которого мы хотим выполнить команду.
- `--container nginx-container` — указывает, в каком контейнере пода будет выполняться команда. В случае с многоконтейнерными подами этот параметр обязателен, иначе Kubernetes не будет знать, к какому контейнеру подключиться.
- `-- /bin/bash` — команда, запускаемая в контейнере. В данном случае запускается оболочка Bash в интерактивном режиме (`/bin/bash`), позволяющая взаимодействовать с командной строкой контейнера.

Выполнение этой команды предоставляет вам доступ к оболочке конкретного контейнера внутри пода, благодаря чему вы можете выполнять команды именно в нем.

Главное отличие от работы с одноконтейнерным подом состоит в необходимости указания параметра `--container` (или его сокращенной версии `-c`), сообщающего команде `kubectl` о том, к какому контейнеру в составе пода следует подключиться.

Теперь посмотрим, как можно запускать команды в контейнерах, работающих внутри подов.

Выполнение команд в контейнерах

Одна из сильных сторон Kubernetes заключается в том, что вы можете в любой момент получить доступ к контейнерам, работающим внутри подов, и выполнять в них команды. Ранее мы это уже делали, но важно отметить, что с помощью утилиты `kubectl` можно запустить любую команду прямо из командной строки.

Для начала воссоздадим многоконтейнерный под:

```
$ kubectl apply -f multi-container-pod.yaml
pod/multi-container-pod created
```

Для выполнения команды в контейнере, как и раньше, используется `kubectl exec`. Но на этот раз параметр `-it` нужно опустить, чтобы утилита `kubectl` не привязывалась к текущему терминальному сеансу.

В следующем примере запускается команда `ls`, которая выводит список файлов в контейнере `nginx-container`, работающем внутри пода `multi-container-pod`:

```
$ kubectl exec pods/multi-container-pod -c nginx-container -- ls
bin
boot
dev
docker-entrypoint.d
docker-entrypoint.sh
...<фрагмент опущен для краткости>...
```

Если не указать имя контейнера, утилита `kubectl` по умолчанию выберет первый контейнер из списка.

Далее мы обсудим способ переопределения команд, выполняемых контейнерами.

Переопределение команд, выполняемых контейнерами по умолчанию

Переопределение команд по умолчанию позволяет управлять поведением каждого отдельного контейнера внутри многоконтейнерных подов. Например, контейнер с веб-сервером обычно запускает команду `start server`, но вы можете переопределить его поведение, чтобы он отвечал за логирование. Этот подход также помогает в управлении ресурсами: если контейнер по умолчанию запускает ресурсоемкий процесс, вы можете заменить его более легковесным, чтобы остальные контейнеры в поде имели достаточно ресурсов. Наконец, это упрощает управление зависимостями: например, контейнер приложения может запуститься быстро, но вы можете переопределить его поведение так, чтобы он дожидался готовности связанного с ним контейнера базы данных.

В Docker для сборки образа контейнера используются специальные файлы Dockerfile, в которых есть два ключевых параметра, `ENTRYPOINT` и `CMD`, определяющие то, какие команды и аргументы будут выполняться контейнером при его запуске с помощью команды `docker run`:

- `ENTRYPOINT` определяет основную команду, которая будет запущена внутри контейнера;
- `CMD` задает параметры по умолчанию для команды `ENTRYPOINT`, которые пользователь может переопределить при запуске контейнера.

Например, классический Dockerfile для запуска команды, приостанавливающей выполнение текущего процесса на 30 секунд, выглядел бы так:

```
# ~/Dockerfile
FROM busybox:latest
ENTRYPOINT ["sleep"]
CMD ["30"]
```

Файл Containerfile и Podman

Файл Containerfile служит рецептом для сборки контейнерного образа, как и Dockerfile. В нем последовательно указываются инструкции по выбору ОС, установке зависимостей, копированию кода приложения и настройке параметров. Podman, инструмент, похожий на Docker, способен интерпретировать файлы Containerfile и собирать образы на основе содержащихся в них инструкций.

Если вы работали с Docker, то предыдущий фрагмент кода должен показаться вам знакомым. Инструкция `CMD` задает аргументы по умолчанию, которые можно переопределить при запуске контейнера с помощью команды `docker run`. Если собрать образ на основе этого файла Dockerfile с помощью команды `docker build`, то получится образ `busybox`, который при запуске контейнера будет выполнять команду `sleep` (заданную в `ENTRYPOINT`) в течение 30 секунд (значение `CMD` по умолчанию).

С помощью инструкции `CMD` мы можем переопределить значение по умолчанию:

```
$ docker run my-custom-ubuntu:latest 60
$ docker run my-custom-ubuntu:latest  # Приостанавливает выполнение на 30 секунд
```

Система Kubernetes, в свою очередь, позволяет переопределять как `ENTRYPOINT`, так и `CMD` с помощью YAML-манифестов пода. Для этого необходимо добавить в YAML-файл конфигурации два необязательных ключа: `command` и `args`.

Kubernetes предоставляет в этом плане значительное преимущество: вы можете либо добавить аргументы к команде, указанной в Dockerfile контейнера (аналогично тому, как это работает с инструкцией `CMD` в Docker), либо полностью переопределить инструкцию `ENTRYPOINT`.

Создадим новый манифест и зададим в нем значения `ENTRYPOINT` и `CMD` для образа `busybox`, чтобы контейнер, созданный на его основе, приостанавливал выполнение на 60 секунд:

```
# nginx-debian-with-custom-command-and-args.yaml
apiVersion: v1
kind: Pod
metadata:
  name: nginx-debian-with-custom-command-and-args
```

```
spec:
  restartPolicy: Never
  containers:
    - name: nginx-container
      image: nginx:latest
    - name: debian-container
      image: debian
      command: ["sleep"]  # соответствует ENTRYPOINT
      args: ["60"]        # соответствует CMD
```

Здесь есть небольшой нюанс: то, что в Dockerfile называется `ENTRYPOINT`, в YAML-манифесте соответствует ключу `command`, а то, что в Dockerfile называется `CMD`, соответствует ключу `args`.

Если мы опустим один из этих параметров, система Kubernetes использует настройки по умолчанию из образа контейнера. То есть при отсутствии ключа `args` в YAML-файле Kubernetes возьмет значение `CMD` из Dockerfile, а при отсутствии ключа `command` будет использовано значение `ENTRYPOINT` из Dockerfile. В большинстве случаев (если вас устраивает значение `ENTRYPOINT` контейнера) вы будете переопределять только ключ `args` (`CMD` в Dockerfile).

После создания пода мы можем проверить результат следующим образом:

```
$ kubectl apply -f nginx-debian-with-custom-command-and-args.yaml
pod/nginx-debian-with-custom-command-and-args created

$ kubectl get po -w
NAME                                        READY STATUS             RESTARTS  AGE
nginx-debian-with-custom-command-and-args  0/2   ContainerCreating  0         2s
nginx-debian-with-custom-command-and-args  2/2   Running            0         6s
nginx-debian-with-custom-command-and-args  1/2   NotReady           0         66s
```

Итак, переопределение команд по умолчанию позволяет контролировать поведение отдельных контейнеров в многоконтейнерных подах, а также оптимизировать использование ресурсов и управлять зависимостями. Как было показано в этом разделе, Kubernetes позволяет переопределять настройки контейнеров с помощью ключей `command` и `args` в YAML-манифестах пода.

В следующем подразделе вы познакомитесь с контейнерами инициализации.

Контейнеры инициализации

Контейнеры инициализации, или init-контейнеры (`initContainers`), позволяют выполнять подготовительные действия перед запуском основных контейнеров пода. Их можно рассматривать как дополнительные контейнеры, которые задаются в YAML-манифесте пода и выполняются первыми при его создании. После завершения их работы система Kubernetes запускает основные контейнеры.

В одном поде можно использовать не один, а несколько контейнеров инициализации. Однако следует помнить, что они выполняются последовательно, а не параллельно: после завершения одного контейнера запускается следующий и т. д. Обычно init-контейнеры применяются для решения таких подготовительных задач.

- *Инициализация базы данных* — настройка и подготовка базы данных перед запуском основного приложения.
- *Загрузка файлов конфигурации* — получение необходимых конфигурационных файлов из удаленных источников.
- *Установка пакетов* — установка зависимостей, требующихся основному приложению.
- *Ожидание ответа внешнего сервиса* — проверка доступности внешних сервисов перед запуском основного контейнера.
- *Проведение предварительных проверок* — выполнение необходимых проверок или валидации перед запуском приложения.
- *Управление секретами* — безопасная загрузка и внедрение секретов в окружение основного контейнера.
- *Миграция данных* — перенос данных в базу или хранилище перед запуском основного приложения.
- *Настройка прав доступа к файлам* — задание разрешений для основного приложения.

Поскольку контейнеры инициализации могут использовать собственные образы, в них можно вынести часть настроек, сделав образы основных контейнеров максимально компактными. Это повышает безопасность всей системы за счет исключения ненужных инструментов из основных контейнерных образов. Ниже приведен YAML-манифест с init-контейнером:

```
# nginx-with-init-container.yaml
---
apiVersion: v1
kind: Pod
metadata:
  name: nginx-with-init-container
  labels:
    environment: prod
    tier: frontend
spec:
  restartPolicy: Never
  volumes:
    - name: website-volume
      emptyDir: {}
```

```
initContainers:
  - name: download-website
    image: busybox
    command:
      - sh
      - -c
    args:
      - |
        wget \
        https://github.com/iamgini/website-demo-one-page/archive/refs/
        heads/main.zip -O /tmp/website.zip && \
        mkdir /tmp/website && \
        unzip /tmp/website.zip -d /tmp/website && \
        cp -r /tmp/website/website-demo-one-page-main/* /usr/share/nginx/html
    volumeMounts:
      - name: website-volume
        mountPath: /usr/share/nginx/html
containers:
  - name: nginx-container
    image: nginx:latest
    volumeMounts:
      - name: website-volume
        mountPath: /usr/share/nginx/html
```

Как видите, init-контейнер запускает образ `busybox`, который загружает приложение (здесь это содержимое простого сайта со страницы https://github.com/iamgini/website-demo-one-page) и копирует его в общий том `website-volume`. (О томах и постоянном хранилище мы поговорим в главе 9.) Этот же том монтируется в контейнер Nginx, чтобы тот использовал его в качестве источника веб-контента по умолчанию. По завершении работы init-контейнера Kubernetes создаст контейнер `nginx-container`.

Учтите, что в случае сбоя при выполнении init-контейнера система Kubernetes не сможет запустить основные контейнеры. Контейнеры инициализации не стоит рассматривать в качестве второстепенных компонентов, сбой которых допустим. Если они определены в YAML-манифесте, значит, их успешное выполнение служит обязательным условием для запуска основных контейнеров.

Создадим под, а затем выполним команду `kubectl get Pods -w`, вывод которой будет регулярно обновляться, отражая изменения в статусе пода. Обратите внимание на столбец `STATUS`, в котором указано, что init-контейнер перешел в состояние выполнения (`Running`):

```
$ kubectl apply -f nginx-with-init-container.yaml
pod/nginx-with-init-container created

$ kubectl get po -w
```

```
NAME                          READY   STATUS            RESTARTS   AGE
nginx-with-init-container     0/1     Init:0/1          0          3s
nginx-with-init-container     0/1     Init:0/1          0          4s
nginx-with-init-container     0/1     PodInitializing   0          19s
nginx-with-init-container     1/1     Running           0          22s
```

Значение `Init:0/1` указывает на запуск init-контейнера. Затем префикс `Init:` исчезает из последующих статусов, что говорит об успешном выполнении контейнера инициализации и о том, что система Kubernetes приступила к созданию основного контейнера — в нашем случае Nginx.

Если вы хотите изучить этот процесс подробнее, то можете открыть доступ к поду с помощью сервиса типа `NodePort`:

```
$ kubectl expose pod nginx-with-init-container --port=80 --type=NodePort
service/nginx-with-init-container exposed
```

Затем настройте переадресацию портов с помощью команды `kubectl port-forward`, чтобы получить доступ к сервису в кластере извне:

```
$ kubectl port-forward pod/nginx-with-init-container 8080:80
Forwarding from 127.0.0.1:8080 -> 80
Forwarding from [::1]:8080 -> 80
```

Теперь, если вы перейдете по адресу `http://localhost:8080`, увидите одностраничный сайт с содержимым, скопированным со страницы https://github.com/iamgini/website-demo-one-page. Подробнее о предоставлении доступа к сервисам мы поговорим в главе 8. Прежде чем переходить к следующему разделу, не забудьте остановить переадресацию портов, нажав в терминале Ctrl+C.

Используйте контейнеры инициализации обдуманно! Их применение не является обязательным, но они могут пригодиться для выполнения конфигурационных сценариев или загрузки данных с внешних серверов перед запуском основных контейнеров.

Теперь посмотрим, как можно получить логи конкретного контейнера внутри работающего пода.

Получение логов конкретного контейнера

Для получения логов отдельного контейнера, входящего в состав многоконтейнерного пода, предназначена команда `kubectl logs`.

Чаще всего контейнеризованные приложения публикуют свои логи путем их отправки в стандартный поток вывода (`stdout`). Команда `kubectl logs` может в потоковом режиме выводить содержимое потока `stdout` конкретного

контейнера внутри пода и тем самым извлекать логи приложения. Чтобы это работало, нужно знать имя контейнера и имя пода, в котором он работает (точно так же, как при использовании команды `kubectl exec` для получения доступа к отдельному контейнеру).

Способ получения этой информации был описан в подразделе «Доступ к конкретному контейнеру внутри многоконтейнерного пода» ранее в главе.

```
$ kubectl logs -f pods/multi-container-pod --container nginx-container
```

Параметр `--container` (сокращенная версия — `-c`) указывает контейнер, логи которого требуется получить. Этот параметр работает и для контейнера инициализации: чтобы просмотреть его журналы, достаточно передать его имя.

Если не указать параметр --container, система Kubernetes выведет логи контейнера по умолчанию из тех, что запущены внутри пода (по аналогии с kubectl exec). Его можно смело опустить в случае одноконтейнерного пода, но при работе с многоконтейнерными подами этот параметр стоит использовать всегда.

Для работы с логами контейнеров предусмотрены и другие полезные параметры. Например, получить записи журналов, сделанные за последние два часа, можно следующим образом:

```
$ kubectl logs --since=2h pods/multi-container-pod --container nginx-container
```

Вы также можете использовать параметр `--tail`, чтобы вывести последние строки лога:

```
$ kubectl logs --tail=30 pods/multi-container-pod --container nginx-container
```

Команда извлекает 30 последних строк лога контейнера `nginx-container`.

Теперь вы умеете читать и извлекать логи из одноконтейнерных и многоконтейнерных подов Kubernetes.

В этом разделе вы научились создавать, обновлять и удалять многоконтейнерные поды. Мы также рассмотрели способы принудительного удаления пода, обращения к конкретному контейнеру в поде и получения логов отдельного контейнера. В примерах мы использовали контейнеры Nginx и Debian, но они практически не взаимодействовали друг с другом. Далее обсудим работу с томами, которые позволяют контейнерам обмениваться файлами.

Совместное использование томов контейнерами одного пода

В этом разделе мы обсудим концепцию томов в Kubernetes и способы их использования. В Docker тоже есть тома, но они отличаются от томов Kubernetes, хоть и служат тем же целям.

Далее мы разберем, что собой представляют тома в Kubernetes и чем они могут быть нам полезны.

Тома в Kubernetes

Начнем с решения простой задачи. В настоящее время наш многоконтейнерный под состоит из двух контейнеров: Nginx и Debian. Мы попробуем примонтировать каталог логов Nginx к каталогу внутри контейнера Debian. Таким образом мы создадим связь между двумя контейнерами, позволив им использовать одну и ту же папку.

В системе Kubernetes существует два типа томов:

- обычные тома, о которых мы будем говорить здесь;
- постоянные тома (PersistentVolume) — более продвинутая возможность, которую мы рассмотрим в главе 9.

Учтите, что это не одно и то же. PersistentVolume представляет собой самостоятельный ресурс, а тома (`volumes`) — часть конфигурации пода. Как следует из названия, PersistentVolume является постоянным хранилищем, в то время как тома обычно не предназначены для постоянного хранения данных. Но имейте в виду, что из этого правила есть исключения.

Проще говоря, тома в Kubernetes тесно связаны с жизненным циклом пода. При создании пода можно определить и подключить тома к его контейнерам. По сути, тома представляют собой хранилища, существующие ровно столько, сколько существует под. При удалении пода все связанные с ним тома тоже удаляются.

Хотя у томов есть и другое предназначение, в первую очередь их удобно применять для организации совместного использования файлов и каталогов контейнерами одного пода.

Следует помнить, что тома связаны с жизненным циклом пода, а не контейнера. Если контейнер аварийно завершит работу, том сохранится, так как сбой контейнера не приводит к сбою пода. Пока под работает, работают и его тома.

Тома играют ключевую роль при управлении данными в контейнеризированных приложениях. Они позволяют хранить данные независимо от жизненного цикла контейнера. Система Kubernetes поддерживает различные типы томов, включая тома, смонтированные из файловой системы хоста, облачной платформы и сетевых хранилищ.

Помимо этого, Kubernetes поддерживает разные драйверы, позволяющие интегрировать тома подов с внешними системами хранения. Например, том AWS EBS (Elastic Block Store) может использоваться в качестве тома Kubernetes. Среди наиболее распространенных решений можно выделить:

- `hostPath`;
- `emptyDir`;
- `nfs`;
- `persistentVolumeClaim` (используется вместе с ресурсом `PersistentVolume`, обсуждаемым в главе 9).

Имейте в виду, что некоторые типы томов были удалены или признаны устаревшими в последних версиях Kubernetes. Подробнее об этом можно узнать в документации по адресу https://kubernetes.io/docs/concepts/storage/volumes/. Вот некоторые примеры:

- `azureDisk` (удален);
- `gcePersistentDisk` (удален);
- `glusterfs` (удален);
- `azureFile` (устарел).

При использовании внешних решений для работы с томами Kubernetes необходимо соблюдать их требования. Например, если в качестве тома Kubernetes используется AWS EBS, то под должен запускаться на рабочем узле Kubernetes, являющемся экземпляром EC2, поскольку тома AWS EBS могут подключаться только к этим виртуальным машинам. Подробнее об этом можно узнать в документации: https://kubernetes.io/docs/concepts/storage/volumes/.

Ниже показана общая схема, иллюстрирующая использование томов `hostPath` и `emptyDir` (рис. 5.2).

Далее мы познакомимся с двумя базовыми драйверами томов: `emptyDir` и `hostPath`. Мы также немного поговорим о ресурсе PersistentVolumeClaim, так как он немного отличается от остальных и будет подробно рассмотрен в главе 9.

Итак, выясним, как можно организовать совместное использование файлов контейнерами одного пода с помощью тома типа `emptyDir`.

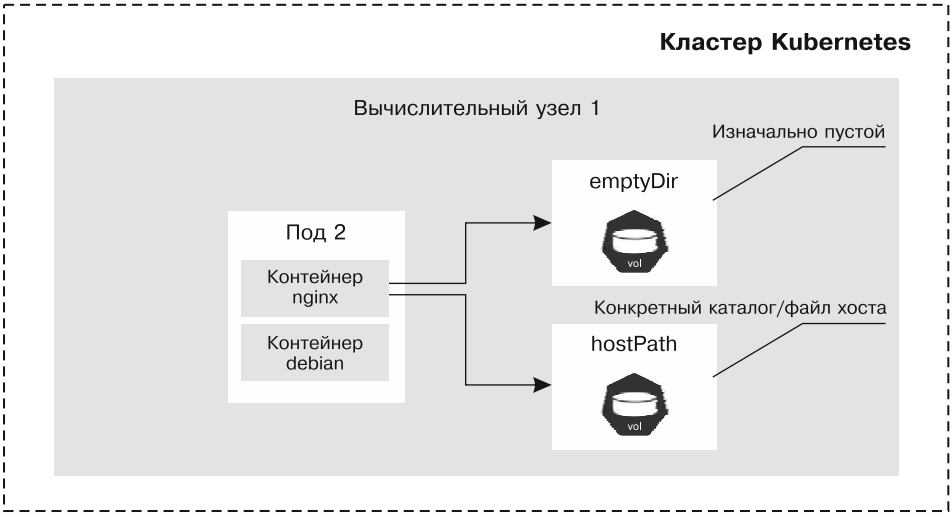

Рис. 5.2. Тома hostPath и emptyDir

Создание и монтирование тома emptyDir

Как следует из названия, том `emptyDir` представляет собой пустой каталог, который инициализируется при создании пода и может быть смонтирован в каждый контейнер, работающий в поде.

Это самый простой способ организации совместного использования данных несколькими контейнерами.

Создадим под, который, как и в предыдущих примерах, будет запускать два контейнера: Nginx и Debian. Мы переопределим команду, выполняемую при запуске контейнера Debian, чтобы предотвратить его немедленное завершение. Таким образом, он будет работать непрерывно, и мы сможем выполнить дополнительные команды, чтобы проверить корректность инициализации нашего тома `emptyDir`.

Оба контейнера будут иметь общий том `emptyDir`, смонтированный в каталог `/var/i-am-empty-dir-volume/`. Вот YAML-манифест для создания такого пода:

```
# multi-container-with-emptydir-pod.yaml
---
apiVersion: v1
kind: Pod
metadata:
  name: multi-container-with-emptydir-pod
spec:
  containers:
    - name: nginx-container
      image: nginx:latest
```

```
    volumeMounts:
      - mountPath: /var/i-am-empty-dir-volume
        name: empty-dir-volume
  - name: debian-container
    image: debian
    command: ["/bin/sh"]
    args: ["-c", "while true; do sleep 30; done;"]  # Предотвращает завершение
                                                    # работы контейнера
    volumeMounts:
    - mountPath: /var/i-am-empty-dir-volume
      name: empty-dir-volume
volumes:
- name: empty-dir-volume # имя тома
  emptyDir: {}           # создает временный каталог на узле,
                         # общий для всех контейнеров пода
```

По мере продвижения наш кластер Kubernetes будет усложняться. Как вы понимаете, большинство сложных сценариев невозможно реализовать с помощью одних лишь императивных команд. Мы все чаще будем использовать YAML-манифесты, поэтому вам стоит научиться их читать и разбираться в их содержимом.

Применим приведенный выше манифест с помощью команды `kubectl apply -f`:

```
$ kubectl apply -f multi-container-with-emptydir-pod.yaml
pod/multi-container-with-emptydir-pod created
```

А теперь проверим результат запуска пода, выполнив команду `kubectl get Pods`:

```
$ kubectl get po
NAME                                READY   STATUS    RESTARTS   AGE
multi-container-with-emptydir-pod   2/2     Running   0          25s
```

Убедившись, что под работает, а контейнеры Nginx и Debian запущены, мы можем проверить доступность общего каталога для обоих контейнеров с помощью команды `ls`.

Эту команду можно выполнить внутри контейнеров через команду `kubectl exec`, которая, как вы помните, принимает в качестве аргументов имя пода и имя контейнера. Мы запустим ее дважды, чтобы убедиться, что наш том смонтирован в оба контейнера:

```
$ kubectl exec multi-container-with-emptydir-pod -c debian-container -- ls /var
backups
cache
i-am-empty-dir-volume
lib
local
lock
log
mail
```

```
opt
run
spool
tmp
$ kubectl exec multi-container-with-emptydir-pod -c nginx-container -- ls /var
backups
cache
i-am-empty-dir-volume
lib
local
lock
log
mail
opt
run
spool
tmp
```

Как видите, команда `ls /var` выводит имя каталога в обоих контейнерах! Это подтверждает, что том `emptyDir` был инициализирован и успешно смонтирован в оба контейнера.

Теперь создадим файл в одном из контейнеров. Если этот файл мгновенно станет видимым во втором контейнере, это подтвердит корректность монтирования тома.

Следующая команда создает текстовый файл `hello-world.txt` в смонтированном каталоге:

```
$ kubectl exec multi-container-with-emptydir-pod \
  -c debian-container -- /bin/sh -c \
  "echo 'hello world' >> /var/i-am-empty-dir-volume/hello-world.txt"
$ kubectl exec multi-container-with-emptydir-pod \
  -c nginx-container -- cat \
  /var/i-am-empty-dir-volume/hello-world.txt
hello world
$ kubectl exec multi-container-with-emptydir-pod \
  -c debian-container -- cat \
  /var/i-am-empty-dir-volume/hello-world.txt
hello world
```

Здесь мы использовали контейнер `debian-container` для создания файла `/var/i-am-empty-dir-volume/hello-world.txt`, содержащего строку `hello-world`. Затем с помощью команды `cat` мы получили доступ к его содержимому из обоих контейнеров и убедились в том, что файл доступен каждому из них. Напомним, что тома `emptyDir` связаны с жизненным циклом пода. Если под будет удален, том и все его данные также будут уничтожены и восстановить их будет невозможно.

Теперь рассмотрим другой тип тома — `hostPath`. Как подсказывает название, он представляет собой каталог на хостовой машине (узле Kubernetes, на котором запущен под), который можно смонтировать в контейнеры.

Создание и монтирование тома hostPath

Как следует из названия, том `hostPath` позволяет смонтировать каталог хостовой машины в контейнеры внутри пода. Под хостовой машиной в данном случае понимается вычислительный (или управляющий) узел Kubernetes, на котором выполняется под. Вот несколько примеров.

- Если ваш (одноузловой) кластер создан с помощью `minikube`, то хостом является ваш локальный компьютер.
- В Amazon EKS хостовая машина представляет собой экземпляр EC2.
- В кластере, развернутом с помощью `kubeadm`, хостовая машина — это, как правило, обычный сервер под управлением Linux.

Иными словами, хостовая машина — та, на которой работает под Kubernetes, и каталог из ее файловой системы можно смонтировать в этот под.

В следующем примере мы будем работать с кластером Kubernetes на основе `minikube`, поэтому томом `hostPath` будет каталог, созданный на вашем локальном компьютере и смонтированный в под Kubernetes.

Том типа hostPath может быть полезным, но в мире Kubernetes его использование считается антипаттерном. Несмотря на его удобство, том hostPath не рекомендуется применять из-за снижения портируемости приложений и потенциальных рисков безопасности. Кроме того, он может быть несовместим с современными механизмами защиты, в частности с механизмом SELinux MCS (multi-category security, мультикатегорийная безопасность), поддерживаемым во многих дистрибутивах Kubernetes. Более надежным, безопасным и перспективным решением является использование постоянных томов (PersistentVolume, PV) и заявок на их выделение (PersistentVolumeClaim, PVC) для управления данными, постоянно хранящимися в контейнеризованных приложениях.

Поды должны легко удаляться и без проблем перезапускаться на другом рабочем узле. Использование тома `hostPath` привязывает под к конкретному узлу, что может привести к серьезным проблемам, если под выйдет из строя и будет перезапущен на другой хостовой машине, где требуемый путь отсутствует.

Теперь создадим том типа `hostPath`.

Допустим, на рабочем узле есть файл `worker-node/nginx.conf` и мы хотим смонтировать его в контейнер Nginx по пути `/var/config/nginx.conf`.

Ниже показан YAML-манифест для создания такой конфигурации. В нижней части файла мы объявляем том `hostPath`, указывающий путь, который должен существовать на хостовой машине. Затем этот том можно смонтировать в любой контейнер в блоке `containers`:

```
# multi-container-with-hostpath.yaml
---
apiVersion: v1
kind: Pod
metadata:
  name: multi-container-with-hostpath
spec:
  containers:
    - name: nginx-container
      image: nginx:latest
      volumeMounts:
      - mountPath: /foo
        name: my-host-path-volume
    - name: debian-container
      image: debian
      command: ["/bin/sh"]
      args: ["-c", "while true; do sleep 30; done"]  # Предотвращает завершение
                                                     # работы контейнера
  volumes:
    - name: my-host-path-volume
      hostPath:
        path: /tmp   # путь на рабочем узле
        type: Directory
```

Как видите, том `hostPath` монтируется так же, как и том `emptyDir`, рассмотренный ранее. Этот процесс состоит из двух этапов: объявления тома на уровне пода и его подключения через `volumeMounts` на уровне контейнера.

В предыдущем фрагменте YAML-манифеста мы использовали параметр type: Directory, который означает, что каталог уже существует на хостовой машине. Если вам требуется создать каталог или файл на хосте, используйте параметры DirectoryOrCreate и FileOrCreate соответственно.

Мы также можем смонтировать каталог в контейнер Debian, чтобы предоставить ему доступ к данным, хранящимся на хостовой машине.

Перед запуском YAML-манифеста необходимо создать путь и сам файл на хосте:

```
$ echo "Hello World" >> /tmp/hello-world.txt
```

При использовании кластера `minikube` это действие нужно выполнить внутри соответствующей виртуальной машины:

```
$ minikube ssh
docker@minikube:~$ echo "Hello World" > /tmp/hello-world.txt
docker@minikube:~$ exit
Logout
```

Если кластер `minikube` создан с использованием контейнеров Podman (например, с помощью команды `minikube start --profile cluster2-podman --driver=podman`), нужно войти в под `minikube` и создать файл там:

```
$ sudo podman exec -it minikube /bin/bash
root@minikube:/# cat /tmp/hello-world.txt
```

После создания пути на хосте можно применить YAML-манифест к кластеру Kubernetes, а затем выполнить команду `kubectl get pod`, чтобы убедиться в том, что под был успешно создан:

```
$ kubectl apply -f multi-container-with-hostpath.yaml
pod/multi-container-with-hostpath created
$ kubectl get pod
NAME                            READY   STATUS    RESTARTS   AGE
multi-container-with-hostpath   2/2     Running   0          11s
```

Кажется, все в порядке! Теперь выведем содержимое файла, который должен быть смонтирован по пути `/foo/hello-world.txt`:

```
$ kubectl exec multi-container-with-hostpath -c nginx-container -- cat /foo/
hello-world.txt
Hello World
```

Как видите, локальный файл (на узле Kubernetes) стал доступен внутри контейнера благодаря монтированию тома `hostPath`.

В начале этой главы мы рассмотрели основы работы с многоконтейнерными подами: как создавать, обновлять и удалять такие поды, использовать контейнеры инициализации, просматривать логи, переопределять команды контейнеров непосредственно в конфигурации пода, а также организовывать совместное использование каталогов несколькими контейнерами с помощью двух базовых типов томов.

Далее обсудим несколько паттернов проектирования, которые можно использовать при создании многоконтейнерных подов.

Паттерн проектирования Ambassador

При проектировании многоконтейнерного пода можно опираться на архитектурные принципы, помогающие решать типовые задачи. Одним из таких принципов является использование паттерна Ambassador (Посредник).

В этом разделе мы разберем суть этого паттерна, обсудим способ создания контейнера-посредника в подах Kubernetes и рассмотрим конкретный пример его применения.

Как устроен паттерн Ambassador

Паттерн Ambassador (Посредник) представляет собой подход к проектированию многоконтейнерных подов, в рамках которого определяются два контейнера:

- первый контейнер называется основным или главным (main container);
- второй называется контейнером-посредником (ambassador container).

Данный паттерн предполагает, что основному контейнеру может потребоваться доступ к внешним сервисам. Например, вашему приложению может быть необходимо извлекать информацию из базы данных SQL, находящейся за пределами пода (рис. 5.3).

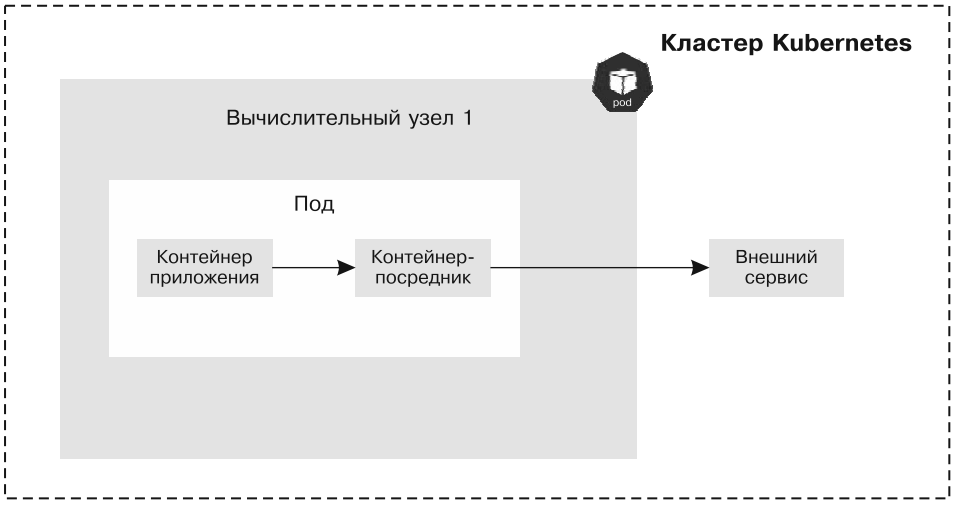

Рис. 5.3. Реализация паттерна Ambassador в Kubernetes

В подобном сценарии рядом с основным контейнером в составе одного пода можно развернуть контейнер-посредник, который будет перенаправлять запросы основного контейнера к серверу базы данных.

В этом случае контейнер-посредник будет представлять собой SQL-прокси. Каждый раз, когда основному контейнеру нужно будет обратиться к базе данных, он не станет делать это напрямую, а установит соединение с посредником, играющим роль прокси.

Использовать контейнер-посредник имеет смысл лишь в том случае, если внешний API не находится в том же кластере Kubernetes. Для обращения к другому поду в пределах кластера система Kubernetes предоставляет такой мощный механизм, как сервисы, с которыми мы познакомимся в главе 8.

Но зачем вообще нужен посредник для доступа к внешним базам данных? Вот основные преимущества, которые предоставляет этот паттерн проектирования:

- разделение логики приложения и конфигурации SQL;
- управление сертификатами *SSL/TLS* (Secure Sockets Layer/Transport Layer Security — уровень защищенных сокетов/безопасность транспортного уровня).

Важно отметить, что паттерн Ambassador (Посредник) применяется не только для создания SQL-прокси — этот пример лишь иллюстрирует его возможности. Учтите, что такой посредник предназначен исключительно для организации исходящих подключений от основного контейнера к внешним ресурсам (например, к хранилищам данных или API). То есть он не является точкой входа в кластер! Теперь посмотрим, как можно описать такой SQL-прокси в YAML-файле.

Пример пода с контейнером-посредником

Итак, вы познакомились с контейнерами-посредниками, а теперь рассмотрим способ их создания в Kubernetes. Следующий YAML-манифест описывает под, включающий два контейнера:

- `nginx-app`, созданный на основе образа `nginx:latest`;
- `sql-ambassador-proxy`, созданный на основе образа `mysql-proxy:latest`.

Приведенный пример предназначен исключительно для демонстрации концепции SQL-прокси. Для полноценного тестирования его работы вам необходим действующий экземпляр AWS *RDS* (Relational Database Service), доступный из кластера Kubernetes, а также подходящее приложение для проверки операций с базой данных.

```
# nginx-with-ambassador.yaml
apiVersion: v1
kind: Pod
metadata:
  name: nginx-with-ambassador
spec:
  containers:
    - name: mysql-proxy-ambassador-container
      image: mysql-proxy:latest
      ports:
        - containerPort: 3306
      env:
      - name: DB_HOST
        value: mysql.xxx.us-east-1.rds.amazonaws.com
    - name: nginx-container
      image: nginx:latest
```

Разработчику необходимо самому настроить приложение в контейнере Nginx так, чтобы оно выполняло запросы к контейнеру-посреднику, а не напрямую

к Amazon RDS. Поскольку контейнер-посредник можно конфигурировать с помощью переменных окружения, для решения этой задачи достаточно передать ему необходимые параметры.

Не обращайте внимания на порядок определения контейнеров в YAML-манифесте. Тот факт, что контейнер-посредник указан первым, не делает его основным контейнером пода. С точки зрения Kubernetes такого понятия, как основной контейнер, вообще не существует — оба контейнера запускаются параллельно и не имеют между собой никаких иерархических отношений. В данном примере мы обращаемся к поду через контейнер Nginx, что делает его более значимым в этом конкретном контексте.

Важно помнить, что контейнер-посредник, работающий в том же поде, что и контейнер Nginx, становится доступным для него по адресу `localhost:3306`.

В следующем разделе вы познакомитесь еще с одним важным принципом организации многоконтейнерных подов — паттерном Sidecar (Прицеп).

Паттерн проектирования Sidecar

Паттерн Sidecar (Прицеп) применяется для дополнения основного контейнера функциями, которые он сам по себе реализовать не может.

Далее мы подробно рассмотрим этот паттерн на конкретных примерах.

Как устроен паттерн Sidecar

Контейнер-прицеп можно рассматривать как расширение для основного контейнера. Его задача — дополнить основной контейнер новыми функциями, не изменяя его самого. При этом основной контейнер может даже не знать о существовании контейнера-прицепа.

Как и паттерн Ambassador (Посредник), паттерн Sidecar (Прицеп) предполагает использование как минимум двух контейнеров:

- основного — того, в котором работает приложение;
- контейнера-прицепа — того, который добавляет дополнительный функционал.

Вы, вероятно, уже догадались, что этот паттерн особенно полезен для запуска агентов мониторинга или систем пересылки логов. На рис. 5.4 показан простой пример его реализации, состоящей из основного контейнера приложения и контейнера-прицепа, собирающего его логи.

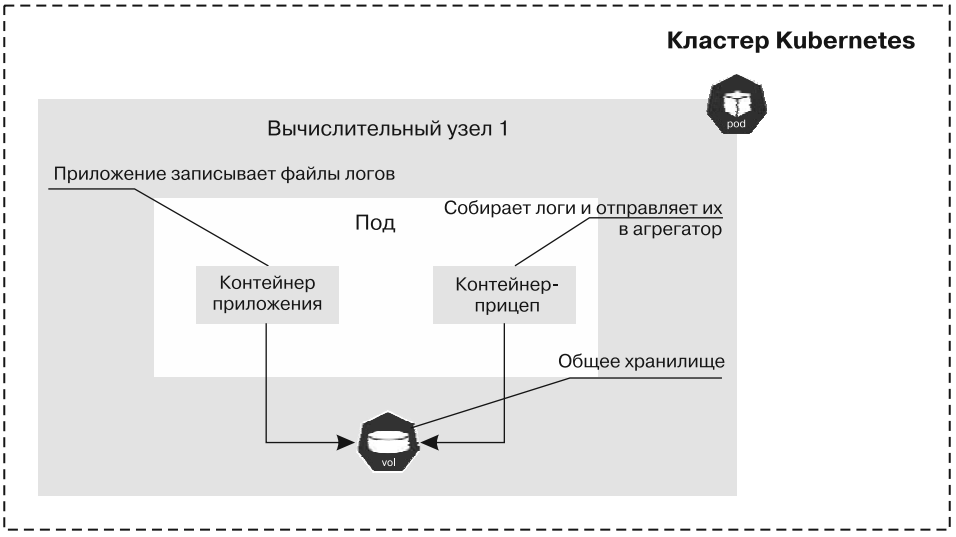

Рис. 5.4. Реализация паттерна Sidecar в Kubernetes

Для создания контейнера-прицепа, пересылающего логи в другое место, необходимо сделать следующее.

1. Определить каталог, в который основной контейнер записывает свои данные (например, логи).
2. Создать том, который сделает этот каталог доступным для контейнера-прицепа (например, того, который отвечает за пересылку логов).
3. Запустить контейнер-прицеп с правильной конфигурацией.

При таком подходе основной контейнер остается неизменным. Даже если контейнер-прицеп выйдет из строя, это не повлияет на работу основного контейнера.

Когда следует использовать паттерн Sidecar

Контейнеры-прицепы особенно полезны в следующих сценариях.

- *Сетевые прокси*. Сетевые прокси можно настроить так, чтобы они инициализировались раньше других контейнеров в поде, обеспечивая немедленную доступность своих сервисов. Ярким примером такого контейнера-прицепа является прокси Envoy в составе Istio.
- *Логирование*. Контейнеры-прицепы для сбора логов могут запускаться первыми и работать вплоть до завершения пода, что гарантирует корректный сбор логов даже в случае сбоев.

- *Задания.* Контейнеры-прицепы могут развертываться вместе с объектами Job — они не препятствуют успешному завершению задания. Для запуска такого контейнера в составе объекта Job не требуется дополнительной настройки.
- *Управление учетными данными.* Многие сторонние платформы управления учетными данными используют контейнер-прицеп для безопасного внедрения и управления секретами в рабочих нагрузках. Они также позволяют безопасно обновлять и отзывать учетные данные.

Пример пода с контейнером-прицепом

Как и паттерн Ambassador (Посредник), паттерн Sidecar (Прицеп) предполагает использование нескольких контейнеров в одном поде. В этом примере мы определяем два контейнера: контейнер Nginx, выступающий в качестве основного приложения, и контейнер-прицеп Fluentd, собирающий логи вебсервера Nginx:

```
# nginx-with-fluentd-sidecar.yaml
apiVersion: v1
kind: Pod
metadata:
  name: nginx-with-sidecar
spec:
  containers:
    - name: nginx-container
      image: nginx:latest
      ports:
        - containerPort: 80
      volumeMounts:
        - name: log-volume
          mountPath: /var/log/nginx
    - name: fluentd-sidecar
      image: fluent/fluentd:v1.17
      volumeMounts:
        - name: log-volume
          mountPath: /var/log/nginx
  volumes:
    - name: log-volume
      emptyDir: {}
```

Чтобы контейнер Fluentd работал корректно, необходимо передать ему конфигурацию через объект ConfigMap. Вот типовой пример такой конфигурации (подробнее о ConfigMap мы поговорим в главе 7):

```
---
apiVersion: v1
kind: ConfigMap
metadata:
  name: fluentd-config-map
  namespace: default
```

```
data:
  fluentd.conf: |
    <source>
      @type tail
      path /var/log/nginx/*.log
      pos_file /var/log/nginx/nginx.log.pos
      tag nginx
      <parse>
        @type nginx
      </parse>
    </source>

    <match nginx.**>
      @type elasticsearch
      host elastic.lab.example.com
      port 9200
      logstash_format true
      logstash_prefix fluentd
      logstash_dateformat %Y.%m.%d
    </match>
```

Fluentd — это популярный агент с открытым исходным кодом для сбора и пересылки логов, часто используемый в качестве контейнера-прицепа в подах Kubernetes. Он эффективно собирает логи из разных источников, структурирует их и перенаправляет в такие централизованные системы логирования, как Elasticsearch, Google Cloud Logging или Amazon CloudWatch Logs. Это упрощает управление логами, повышает наблюдаемость и облегчает анализ состояния и производительности приложений. В данном примере логи отправляются на демонстрационный сервер Elasticsearch (например, elastic.lab.example.com), однако Fluentd можно интегрировать с различными внешними решениями для логирования в зависимости от ваших потребностей.

В следующем разделе мы рассмотрим паттерн проектирования Adapter (Адаптер).

Паттерн проектирования Adapter

Как следует из названия, паттерн Adapter (Адаптер) предназначен для преобразования данных из одного формата (исходного) в другой (целевой).

Как и два предыдущих паттерна, он предполагает наличие в поде как минимум двух контейнеров:

- основного контейнера;
- контейнера-адаптера.

Паттерн полезен в ситуациях, когда основной контейнер генерирует данные в формате А, а другое приложение ожидает получить данные в формате Б. Контейнер-адаптер в данном случае выполняет функцию преобразования.

Этот паттерн особенно уместен в контексте логирования и мониторинга. Представьте кластер Kubernetes, в котором работают десятки приложений, и все они записывают логи в формате Apache. Если вам необходимо преобразовать эти логи в формат JSON для последующей индексации поисковым движком, то паттерн Adapter (Адаптер) — это как раз то, что вам нужно. Запуск контейнера-адаптера рядом с приложениями позволяет преобразовать данные в нужный формат перед их отправкой куда-либо.

Как и паттерн Sidecar (Прицеп), этот вариант будет работать только при предоставлении контейнерам внутри пода доступа к одному и тому же каталогу через тома.

Пример пода с контейнером-адаптером

В этом примере мы создадим под, который использует контейнер-адаптер с общим каталогом, смонтированным в виде тома Kubernetes (рис. 5.5).

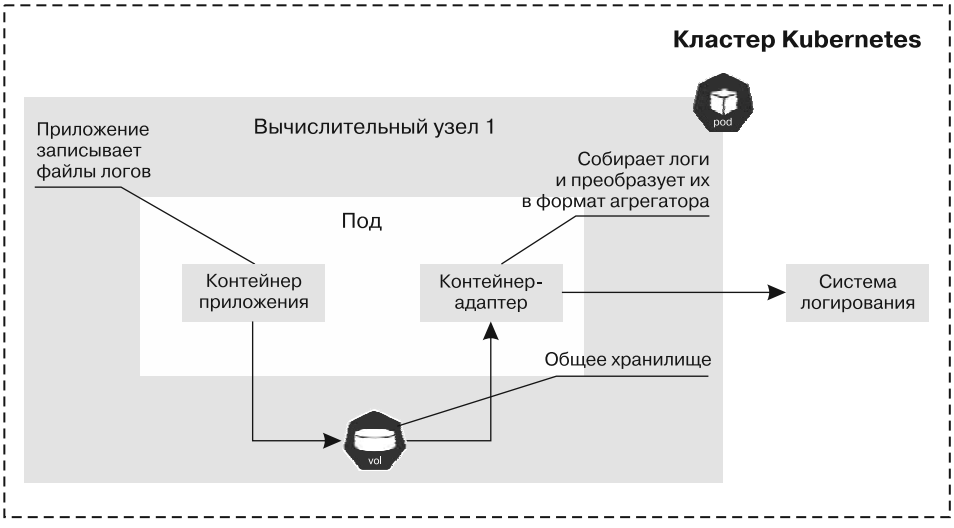

Рис. 5.5. Реализация паттерна проектирования Adapter в Kubernetes

Этот под будет запускать два контейнера:

- `alpine-writer` — основной контейнер приложения, записывающий логи в каталог `/var/log/app`;

- `log-adapter` — контейнер-адаптер, который будет читать эти логи и преобразовывать их в другой формат (например, добавлять строку `PROCESSED` в конец каждой записи).

Приведенный ниже YAML-манифест определяет многоконтейнерный под с контейнером-адаптером:

```
# alpine-with-adapter.yaml
apiVersion: v1
kind: Pod
metadata:
  name: pod-with-adapter
spec:
  containers:
    - name: alpine-writer
      image: alpine:latest
      command:
        - "sh"
        - "-c"
        - |
          i=1; while true;
          do
            echo "$(date) - log $i" >> /var/log/app/app.log;
            i=$((i+1));
            sleep 5;
          done
      volumeMounts:
        - name: log-volume
          mountPath: /var/log/app

      # контейнер-адаптер
    - name: log-adapter
      image: alpine:latest
      command:
        - "sh"
        - "-c"
        - |
          while true;
          do
            cat /logs/app.log | sed 's/$/ PROCESSED/' > \
              /logs/processed_app.log;
            cat /logs/processed_app.log;
            sleep 10;
          done
      volumeMounts:
        - name: log-volume
          mountPath: /logs
  volumes:
    - name: log-volume
      emptyDir: {}
```

После применения YAML-файла можно проверить статус пода с помощью команды:

```
$ kubectl apply -f nginx-with-adapter.yaml
pod/pod-with-adapter created
```

Как только под будет создан, начнется генерация логов и мы сможем проверить вывод обоих контейнеров. Вот команда для просмотра логов, созданных контейнером `alpine-writer`:

```
$ kubectl exec -it pod-with-adapter -c alpine-writer -- head -5 /var/log/app/
app.log
Sun Jun 30 15:05:26 UTC 2024 - log 1
Sun Jun 30 15:05:31 UTC 2024 - log 2
Sun Jun 30 15:05:36 UTC 2024 - log 3
Sun Jun 30 15:05:41 UTC 2024 - log 4
Sun Jun 30 15:05:46 UTC 2024 - log 5
```

А вот как можно проверить логи, преобразованные контейнером `log-adapter`:

```
$ kubectl exec -it pod-with-adapter -c log-adapter -- head -5 /logs/processed_
app.log
Sun Jun 30 15:05:26 UTC 2024 - log 1 PROCESSED
Sun Jun 30 15:05:31 UTC 2024 - log 2 PROCESSED
Sun Jun 30 15:05:36 UTC 2024 - log 3 PROCESSED
Sun Jun 30 15:05:41 UTC 2024 - log 4 PROCESSED
Sun Jun 30 15:05:46 UTC 2024 - log 5 PROCESSED
```

Использование контейнеров-адаптеров позволяет выполнять сложные операции над данными, не изменяя исходные контейнеры приложений.

Прежде чем завершить эту главу, рассмотрим еще одну возможность Kubernetes, связанную с многоконтейнерными подами.

Контейнеры-прицепы и их нативная реализация в Kubernetes

Традиционно контейнеры-прицепы в Kubernetes представляли собой обычные контейнеры, запускаемые вместе с основным приложением в составе пода, как вы уже видели в предыдущих разделах. Такой подход позволяет расширять функциональность, но имеет ограничения. Например, контейнер-прицеп может продолжать работать даже после завершения работы основного приложения, что приведет к неэффективному расходу ресурсов. Кроме того, сама система Kubernetes изначально не знает о контейнерах-прицепах и их связи с основным приложением.

Чтобы устранить эти ограничения, в версию Kubernetes v1.28 были добавлены так называемые нативные контейнеры-прицепы (native sidecars), использующие контейнеры инициализации с особыми настройками. Это позволяет задать параметр `restartPolicy` для контейнеров внутри раздела `initContainers`. Такие контейнеры-прицепы можно запускать, останавливать и перезапускать, не влияя на основное приложение и другие контейнеры инициализации, что обеспечивает более тонкий контроль над их жизненным циклом.

В следующем описании объекта Deployment показано, как можно настроить нативные контейнеры-прицепы в Kubernetes:

```
... <фрагмент опущен для краткости>...
  spec:
    containers:
      - name: myapp
        image: alpine:latest
        command:
          - 'sh'
          - '-c'
          - 'while true; do echo "logging" >> /opt/logs.txt; sleep 1; done'
        volumeMounts:
          - name: data
            mountPath: /opt
    initContainers:
      - name: logshipper
        image: alpine:latest
        restartPolicy: Always
        command:
          - 'sh'
          - '-c'
          - 'tail -F /opt/logs.txt'
        volumeMounts:
          - name: data
            mountPath: /opt
    volumes:
      - name: data
        emptyDir: {}
```

При таком подходе контейнер-прицеп запускается и завершает работу вместе с основным контейнером, что оптимизирует использование ресурсов. Более того, система Kubernetes узнает о роли контейнера-прицепа в поде, что в будущем может открыть возможности для более тесной интеграции.

Использование многоконтейнерных подов Kubernetes с контейнерами инициализации и контейнерами-прицепами, а также паттернов Adapter (Адаптер) и Ambassador (Посредник) позволяет создавать сложные приложения в виде модульных компонентов. Это упрощает развертывание и способствует более эффективному использованию ресурсов в контейнерной среде.

Резюме

Глава получилась довольно объемной, но теперь вы знаете, что такое поды, умеете их использовать и управлять несколькими контейнерами внутри одного пода.

Рекомендуем вам освоить декларативный способ создания ресурсов Kubernetes. Как вы уже поняли, ключ к решению самых сложных задач в Kubernetes заключается в умении писать YAML-файлы. Например, без написания такого файла невозможно создать многоконтейнерный под.

Эта глава дополняет предыдущую, в которой мы говорили о запуске контейнеров. Важно понять, что все, что мы будем делать в Kubernetes, сводится к управлению подами, так как именно они являются центральным объектом этой системы. Помните о том, что контейнеры никогда не создаются напрямую — они всегда запускаются через объект Pod. Кроме того, все контейнеры, входящие в состав пода, размещаются на одном узле Kubernetes. Если это понимание закрепилось, можете смело переходить к следующей главе, в которой речь пойдет о пространствах имен.

Дополнительные источники информации

- Управление несколькими контейнерами в подах: https://kubernetes.io/docs/concepts/workloads/pods/#how-pods-manage-multiple-containers.
- Тома Kubernetes: https://kubernetes.io/docs/concepts/storage/volumes/.
- Контейнеры-прицепы: https://kubernetes.io/docs/concepts/workloads/pods/sidecar-containers/.

6 Пространства имен, квоты и лимиты для реализации мультитенантности в Kubernetes

До сих пор вы знакомились с ключевыми концепциями Kubernetes, создавая объекты в кластере и наблюдая за их поведением. Возможно, вы уже заметили, что со временем поддерживать кластер в упорядоченном состоянии становится все труднее. По мере его роста количество ресурсов, которыми необходимо управлять, увеличивается, что усложняет контроль над ними. Именно здесь на помощь приходят пространства имен Kubernetes.

В этой главе вы узнаете, что такое *пространства имен* (namespaces) и как они помогают поддерживать порядок в кластере за счет группировки ресурсов по приложениям или средам.

Технические требования

Для работы с этой главой вам потребуется:

- локальный или облачный кластер Kubernetes;
- рабочий интерфейс командной строки `kubectl`, настроенный для взаимодействия с кластером.

Если эти требования не выполнены, вернитесь к главам 2 и 3.

Примеры кода для этой главы можно загрузить из официального репозитория GitHub по адресу https://github.com/PacktPublishing/The-Kubernetes-Bible-Second-Edition/tree/main/Chapter06.

Пространства имен в Kubernetes

Чем больше приложений вы развертываете в своих кластерах Kubernetes, тем выше ваша потребность в упорядочении их ресурсов. Для управления объектами внутри кластера можно использовать метки и аннотации, однако еще

более высокий уровень организации достигается с помощью *пространств имен*. Пространства имен в Kubernetes позволяют логически изолировать части кластера, что облегчает управление ресурсами. Однако для распределения ресурсов и ограничения их использования требуются дополнительные объекты, такие как ResourceQuota. После создания пространства имен вы можете запускать объекты Kubernetes, например поды, которые будут существовать только в пределах этого пространства. Все операции с кластером, выполняемые с помощью утилиты `kubectl`, будут ограничены выбранным пространством имен, благодаря чему вы сможете выполнять любые действия, не затрагивая ресурсы, принадлежащие другому пространству.

Для начала разберемся, что именно представляют собой пространства имен и зачем они были созданы.

Для более сложных сценариев, предполагающих использование нескольких кластеров или реализацию мультитенантности, существуют проекты, предоставляющие расширенные возможности. Например, Capsule (https://capsule.clastix.io/) помогает создавать безопасные мультитенантные кластеры Kubernetes, позволяя различным командам или арендаторам управлять собственными изолированными пространствами имен. A HyperShift (https://github.com/openshift/hypershift) упрощает управление несколькими кластерами, предоставляя легковесный и масштабируемый способ безопасной изоляции и управления ресурсами Kubernetes в разных средах.

В следующем подразделе мы поговорим о важности пространств имен в Kubernetes.

Важность пространств имен в Kubernetes

Как уже было сказано, пространства имен в Kubernetes помогают администратору кластера поддерживать порядок и обеспечивают изоляцию ресурсов.

В крупных кластерах Kubernetes могут запускаться сотни и даже тысячи приложений. Если все они развертываются в одном пространстве имен, становится крайне сложно определить, к какому приложению относится тот или иной ресурс.

Обновление или изменение неправильного ресурса может нарушить работу приложения, запущенного в кластере. Для решения этой проблемы можно использовать метки и селекторы, но по мере роста числа ресурсов управление кластером может превратиться в настоящий хаос, если не начать применять пространства имен.

В этой главе мы поговорим о том, как именно пространства имен используются для поддержания порядка и изоляции ресурсов.

Использование пространств имен для разделения ресурсов

При создании кластера Kubernetes в нем изначально предусматривается несколько пространств имен для компонентов. То есть вы уже используете пространства имен, даже не задумываясь об этом:

```
$ kubectl get namespaces
NAME              STATUS   AGE
default           Active   8d
kube-node-lease   Active   8d
kube-public       Active   8d
kube-system       Active   8d
```

Развертывание подов и других объектов Kubernetes в выбранном пространстве имен помогает упорядочивать ресурсы кластера. Важно отметить, что в Kubernetes существует пространство имен по умолчанию, которое используется в тех случаях, когда вы не указали его явно.

Следующая схема иллюстрирует принцип работы пространств имен и изоляции ресурсов (рис. 6.1).

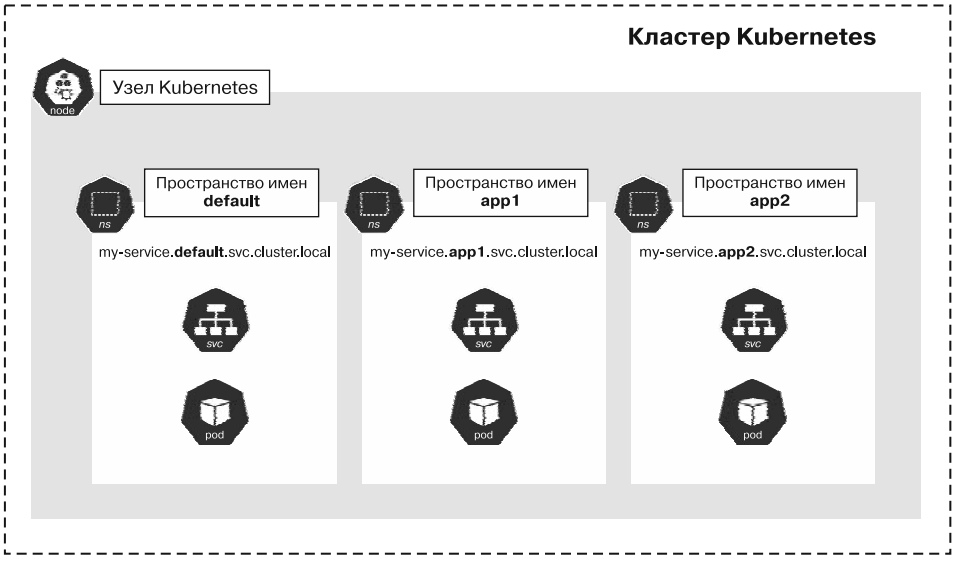

Рис. 6.1. Пространства имен и изоляция ресурсов в Kubernetes

В более широком смысле пространства имен Kubernetes служат сразу нескольким целям, среди которых:

- изоляция ресурсов;
- ограничение области видимости имен ресурсов;
- распределение и ограничение использования аппаратных ресурсов;
- управление правами доступа с помощью механизма RBAC (Role-Based Access Control — управление доступом на основе ролей).

Мы рекомендуем создавать отдельное пространство имен для каждого микросервиса или приложения и развертывать в нем все связанные с ним ресурсы. Однако учтите, что система Kubernetes не предусматривает жестких правил. Пространства имен можно использовать и другими способами.

- *Разделение по средам* — одно пространство можно использовать для рабочей среды, а другое — для среды разработки.
- *Разделение по уровням* — одно пространство можно использовать для баз данных, другое — для подов приложений, а третье — для промежуточного ПО.
- *Использование пространства по умолчанию* — для создания кластеров с небольшим количеством ресурсов можно использовать одно общее пространство по умолчанию, помещая в него все компоненты.

При этом важно помнить, что, даже если два пода развернуты в разных пространствах имен и опубликованы через сервисы, они все равно могут взаимодействовать и обмениваться данными. Хотя сервис Kubernetes создается в конкретном пространстве имен, он получает *полное доменное имя* (FQDN), доступное во всем кластере. Таким образом, если приложению из пространства А необходимо взаимодействовать с приложением из пространства Б, оно будет обращаться к сервису, публикующему приложение Б, по его FQDN. Взаимодействие между пространствами имен разрешено по умолчанию и может контролироваться с помощью сетевых политик.

Теперь перейдем к рассмотрению стандартных пространств имен.

Пространства имен по умолчанию

Большинство кластеров Kubernetes создаются с несколькими пространствами имен по умолчанию. Просмотреть их можно с помощью команды `kubectl get namespaces` (или ее сокращенной формы `kubectl get ns`), как показано ниже:

```
$ kubectl get namespaces
NAME              STATUS   AGE
default           Active   8d
kube-node-lease   Active   8d
kube-public       Active   8d
kube-system       Active   8d
```

В качестве примера возьмем кластер `minikube`. Из вывода команды следует, что этот кластер изначально предусматривает такие пространства имен.

- `default` — автоматически предоставляется системой Kubernetes, чтобы вы могли начать пользоваться новым кластером, не создавая его вручную. До настоящего момента все ваши ресурсы создавались именно в этом пространстве, оно же используется как пространство имен по умолчанию, если явно не указано другое.
- `kube-public` — доступно всем клиентам, включая тех, кто не прошел аутентификацию. Оно предназначено для решения общекластерных задач и обеспечивает публичную видимость и доступность определенных ресурсов остальным компонентам кластера. Однако важно отметить, что публичный характер этого пространства имен — скорее условность, чем строгое требование. Поскольку оно в данный момент не используется, его можно спокойно оставить без изменений.
- `kube-system` — зарезервировано для объектов, создаваемых самой системой Kubernetes. Здесь развертываются объекты, необходимые для работы кластера. В типичных конфигурациях Kubernetes такие важные компоненты, как `kube-scheduler` и `kube-apiserver`, запускаются в виде подов именно в этом пространстве.

  Эти компоненты жизненно важны для функционирования кластера, поэтому, чтобы не нарушить его работу, это пространство не рекомендуется изменять.
- `kube-node-lease` — используется для хранения объектов Lease, связанных с отдельными узлами. Эти объекты позволяют `kubelet` передавать контрольные сигналы, помогающие управляющему слою обнаруживать сбои узлов.

В зависимости от используемого дистрибутива Kubernetes набор предустановленных пространств имен может быть другим. Но в большинстве случаев перечисленные пространства будут созданы по умолчанию.

Далее мы рассмотрим способ создания пространств имен, вы узнаете, как влияют на поды, в частности на разрешение DNS-имен сервисов.

Влияние пространств имен на ресурсы и сервисы

В этом разделе мы обсудим, как создавать, обновлять и удалять пространства имен, а также какое влияние они оказывают на сервисы и поды.

Вы также научитесь создавать ресурсы, указывая собственное пространство имен вместо стандартного.

Просмотр пространств имен в кластере

Как было показано в предыдущем разделе, вывести список пространств имен, созданных в кластере Kubernetes, можно следующим образом:

```
$ kubectl get namespaces
NAME              STATUS   AGE
default           Active   8d
kube-node-lease   Active   8d
kube-public       Active   8d
kube-system       Active   8d
```

Учтите, что для всех команд, работающих с объектом `namespaces`, также можно использовать сокращенный псевдоним `ns`.

Получение данных о конкретном пространстве имен

Данные о конкретном пространстве имен можно получить с помощью команды `kubectl describe`:

```
$ kubectl describe namespaces default
Name:         default
Labels:       kubernetes.io/metadata.name=default
Annotations:  <none>
Status:       Active

No resource quota.

No LimitRange resource.
```

Мы также можем воспользоваться командой `kubectl get` и перенаправить вывод в формате YAML в файл, чтобы получить сведения о выбранном пространстве имен:

```
$ kubectl get namespaces default -o yaml > default-ns.yaml
```

Обратите внимание, что пространство имен может находиться в одном из двух состояний:

- `Active` — пространство имен активно и может использоваться для размещения новых объектов;
- `Terminating` — пространство имен находится в процессе удаления вместе со всем его содержимым и не может использоваться для размещения новых объектов.

В следующем подразделе вы научитесь создавать новое пространство имен императивным способом.

Создание пространства имен с помощью императивного синтаксиса

Для создания пространства имен императивным способом предназначена команда `kubectl create namespace` с указанием имени создаваемого пространства. В этом примере мы создадим новое пространство `custom-ns`. Напомним, что при выполнении операций с пространствами имен через `kubectl` можно использовать сокращение `ns`:

```
$ kubectl create ns custom-ns
namespace/custom-ns created
```

В результате выполнения этой команды в кластере будет создано новое пространство `custom-ns`. Проверить это можно с помощью команды `kubectl get`:

```
$ kubectl get ns custom-ns
NAME        STATUS   AGE
custom-ns   Active   35s
```

Как видите, пространство имен успешно создано и находится в состоянии `Active`. Теперь в нем можно размещать ресурсы.

Не используйте в имени пространств имен префикс kube-, так как он зарезервирован для системных пространств имен и объектов Kubernetes.

Теперь посмотрим, как можно создать пространство имен с помощью декларативного синтаксиса.

Создание пространства имен с помощью декларативного синтаксиса

Для создания пространства имен декларативным способом необходимо использовать YAML- (или JSON-) файл. Ниже приведен простой YAML-файл для создания нового пространства имен в кластере. Обратите внимание на поле `kind: Namespace`:

```
# custom-ns-2.yaml
apiVersion: v1
kind: Namespace
metadata:
  name: custom-ns-2
```

Примените это определение с помощью команды `kubectl create`, указав путь к YAML-файлу:

```
$ kubectl apply -f custom-ns-2.yaml
namespace/custom-ns-2 created
```

Итак, мы создали два пользовательских пространства имен. Первое, `custom-ns`, было создано императивным способом, а второе, `custom-ns-2`, — декларативным.

Теперь рассмотрим процедуру удаления этих двух пространств имен с помощью `kubectl`.

Удаление пространства имен

Удалить пространство имен можно с помощью команды `kubectl delete`:

```
$ kubectl delete namespaces custom-ns
namespace "custom-ns" deleted
```

То же самое можно сделать и декларативным способом. Например, удалим пространство `custom-ns-2`, созданное в предыдущем YAML-файле:

```
$ kubectl delete -f custom-ns-2.yaml
namespace "custom-ns-2" deleted
```

В результате выполнения этой команды пространство имен перейдет из состояния `Active` в состояние `Terminating`. После запуска команды в этом пространстве нельзя будет размещать новые объекты, а через некоторое время оно будет полностью удалено из кластера.

Имейте в виду, что команда kubectl delete namespace крайне опасна. Удаление пространства имен является необратимой операцией. Все ресурсы, созданные в этом пространстве, будут уничтожены. Перед тем как использовать эту команду, убедитесь в наличии YAML-файлов для восстановления удаленных ресурсов и самого пространства имен.

Теперь рассмотрим процесс создания ресурсов внутри заданного пространства имен.

Создание ресурса внутри пространства имен

Следующий фрагмент кода показывает, как можно создать под `nginx` с указанием пользовательского пространства имен. В этом примере мы заново создадим пространство `custom-ns` и запустим в нем под `nginx`:

```
$ kubectl create ns custom-ns
$ kubectl run nginx --image nginx:latest -n custom-ns
pod/nginx created
```

Обратите внимание на параметр `-n`, полной формой которого является `--namespace`. Он используется для указания имени пространства имен, в котором требуется создать ресурс (или из которого нужно получить информацию). Этот параметр поддерживается всеми видами ресурсов, область видимости которых может быть ограничена пространством имен.

В качестве еще одного примера рассмотрим следующую команду для создания `configmap` в пространстве имен `custom-ns`:

```
$ kubectl create configmap configmap-custom-ns --from-literal=Lorem=Ipsum -n
custom-ns
configmap/configmap-custom-ns created
```

Пространство имен также можно указать при использовании декларативного синтаксиса. Вот пример создания пода в конкретном пространстве имен декларативным способом:

```
# pod-in-namespace.yaml
apiVersion: v1
kind: Pod
metadata:
  name: nginx2
  namespace: custom-ns
spec:
  containers:
    - name: nginx
      image: nginx:latest
```

Обратите внимание на ключ `namespace` в разделе `metadata`, расположенный сразу под именем пода. Он указывает на то, что под нужно создать в пространстве `custom-ns`. Теперь мы можем применить этот файл с помощью следующей команды `kubectl`:

```
$ kubectl apply -f pod-in-namespace.yaml
pod/nginx2 created
```

На этом этапе у нас должно быть пространство `custom-ns`, содержащее два пода `nginx` и один объект ConfigMap с именем `configmap-custom-ns`.

При работе с пространствами имен всегда используйте флаг `-n`, чтобы указать нужное пространство. В противном случае есть риск выполнения операций не с тем пространством имен.

Теперь выведем список ресурсов, находящихся внутри конкретных пространств имен.

Вывод списка ресурсов внутри конкретного пространства имен

Чтобы вывести список ресурсов в определенном пространстве имен, необходимо добавить параметр `-n`, так же как при создании ресурса. Например, чтобы отобразить поды в пространстве `custom-ns`, выполните команду:

```
$ kubectl get pods -n custom-ns
NAME     READY   STATUS    RESTARTS   AGE
nginx    1/1     Running   0          9m23s
nginx2   1/1     Running   0          94s
```

Как видите, в этом пространстве присутствует созданный ранее под `nginx`. С этого момента все команды, относящиеся к данному поду, должны содержать параметр `-n custom-ns`.

Причина в том, что под не находится в пространстве имен по умолчанию. Если не указать параметр `-n`, команда будет выполнена в отношении пространства `default`. Убрав параметр `-n custom-ns` из команды `get`, мы увидим, что под `nginx` больше не отображается:

```
$ kubectl get pods
No resources found in default namespace.
```

Аналогичным образом можно выполнить команду `kubectl get configmap`, чтобы проверить наличие объекта ConfigMap в пространстве имен. Как и в случае со списком подов, если не указать параметр `-n`, операция будет выполнена в отношении пространства имен `default`:

```
$ kubectl get cm
NAME               DATA   AGE
kube-root-ca.crt   1      9d

$ kubectl get cm -n custom-ns
NAME                  DATA   AGE
configmap-custom-ns   1      70m
kube-root-ca.crt      1      76m
```

Итак, при работе с кластером, предусматривающим несколько пространств имен, не забывайте добавлять параметр `-n`. Такое, казалось бы, мелкое упущение может привести к напрасной трате времени, так как без нее все действия будут выполняться в пространстве `default`.

Чтобы каждый раз не передавать имя пространства имен в командной строке, можно задать его в контексте `kubeconfig`. В следующем подразделе вы узнаете, как это делается.

Задание текущего пространства имен с помощью kubectl config set-context

В некоторых случаях можно задать текущее пространство имен для работы. Например, если в рамках проекта вы используете определенное пространство имен для приложения и связанных с ним ресурсов, то можете указать его в качестве текущего контекста:

```
$ kubectl config set-context --current --namespace=custom-ns
Context "minikube" modified.
```

Чтобы проверить, какое пространство имен задано в качестве контекста, можно использовать команду:

```
$ kubectl config view --minify --output 'jsonpath={..namespace}'
custom-ns
```

Теперь можно получать информацию о приложении или применять конфигурацию без указания параметра `-n <пространство_имен>`:

```
$ kubectl get pods
NAME     READY   STATUS    RESTARTS   AGE
nginx    1/1     Running   0          79m
nginx2   1/1     Running   0          71m
```

Команда kubectl config и ее подкоманды выполняют только операции чтения или изменения конфигурационного файла ~/.kube/config, используемого утилитой kubectl.

Выполняя команду kubectl config set-context, вы фактически обновляете этот файл, задавая в нем другое пространство имен.

Умение переключаться между пространствами имен с помощью `kubectl` весьма полезно, однако перед выполнением таких операций записи, как `kubectl delete` или `kubectl create`, следует убедиться в том, что вы находитесь в правильном пространстве. В противном случае рекомендуется продолжать использовать параметр `-n`. Поскольку операция переключения контекста может выполняться довольно часто, для упрощения работы пользователи Kubernetes обычно создают псевдонимы в Linux. Вы тоже можете определить такой псевдоним, если считаете, что так вам будет удобнее.

Например, если вы используете Linux или macOS, то можете задать псевдоним в файле `~/.bashrc` следующим образом:

```
alias kubens='kubectl config set-context --current --namespace'
```

А затем воспользоваться им:

```
$ kubens custom-ns
Context "minikube" modified.
$ kubectl config view --minify --output 'jsonpath={..namespace}'
custom-ns
# Переключение на другое пространство имен
$ kubens default
Context "minikube" modified.
$ kubectl config view --minify --output 'jsonpath={..namespace}'
default
```

Тем не менее во избежание ошибок при выполнении операций в Kubernetes настоятельно рекомендуется использовать параметр `-n <пространство_имен>`.

Прежде чем двигаться дальше, восстановим исходный контекст, выполнив команду:

```
$ kubectl config set-context --current --namespace=default
Context "minikube" modified.
```

А теперь посмотрим, как можно перечислить все ресурсы, находящиеся внутри конкретного пространства имен.

Перечисление ресурсов, находящихся в конкретном пространстве имен

Для вывода списка всех ресурсов, находящихся в определенном пространстве имен, можно воспользоваться командой `kubectl get all -n custom-ns`:

```
$ kubectl get all -n custom-ns
```

Она перечисляет все ресурсы, созданные в пространстве имен, указанном с помощью флага `-n`.

Область видимости и пространство имен

Еще одно преимущество пространств имен заключается в том, что они определяют область видимости для имен ресурсов, которые в них находятся.

Возьмем, к примеру, имена подов. Если вы не указываете конкретное пространство имен, то взаимодействуете с пространством по умолчанию. В этом случае при создании двух одноименных подов вы столкнетесь с ошибкой, так как система Kubernetes использует имя пода в качестве его уникального идентификатора.

Попробуем создать два пода с именем `nginx` в пространстве `default`. Для этого можно два раза подряд выполнить одну и ту же команду:

```
$ kubectl run nginx --image nginx:latest
pod/nginx created
$ kubectl run nginx --image nginx:latest
Error from server (AlreadyExists): pods "nginx" already exists
```

Вторая команда завершится с ошибкой, и система Kubernetes сообщит, что под с таким именем уже существует. Выполнив команду `kubectl get pods`, мы увидим, что в пространстве имен по умолчанию был создан только один под:

```
$ kubectl get pods
NAME    READY   STATUS    RESTARTS   AGE
nginx   1/1     Running   0          64s
```

Теперь попробуем снова вывести список подов, но на этот раз в пространстве имен `custom-ns`:

```
$ kubectl get pods --namespace custom-ns
NAME    READY   STATUS    RESTARTS   AGE
nginx   1/1     Running   0          89m
nginx2  1/1     Running   0          23m
```

Как видим, в этом пространстве тоже есть под `nginx`, и это не тот же самый под, что находится в пространстве `default`. В этом и заключается одно из главных преимуществ пространств имен: они позволяют определить несколько одноименных ресурсов в кластере Kubernetes, распределив их по разным пространствам. С их помощью мы можем легко создавать независимые копии микросервисов или приложений.

Мы также можем переопределить пространство имен для ресурсов, создаваемых декларативным способом. Если при выполнении команды `kubectl apply` добавить параметр `-n`, то утилита `kubectl` выполнит операцию в отношении пространства, заданного с помощью переданного параметра, а не того, которое указано в YAML-файле. Такой подход упрощает дублирование ресурсов между разными пространствами имен, например между рабочей (`production`) и тестовой (`test`) средами.

Не все ресурсы принадлежат пространству имен

В Kubernetes не все объекты принадлежат пространству имен. Например, узлы, представленные на уровне кластера объектами типа `Node`, не относятся ни к какому конкретному пространству имен. Вывести список ресурсов, не принадлежащих никакому пространству имен, можно с помощью следующей команды:

```
$ kubectl api-resources --namespaced=false
NAME                SHORTNAMES   APIVERSION   NAMESPACED   KIND
componentstatuses   cs           v1           false        ComponentStatus
```

```
namespaces          ns           v1          false       Namespace
nodes               no           v1          false       Node
persistentvolumes   pv           v1          false       PersistentVolume
... <фрагмент опущен для краткости>...
```

Чтобы отобразить все ресурсы, относящиеся к пространствам имен, достаточно передать параметр `--namespaced=true`:

```
$ kubectl api-resources --namespaced=true
```

Теперь обсудим влияние пространств имен на DNS-имена сервисов.

Разрешение DNS-имен сервисов с помощью пространств имен

Как будет показано в главе 8, поды могут быть опубликованы через объект Service. При создании сервису назначается внутренняя DNS-запись, позволяющая подам в кластере обращаться к нему.

Однако при обращении к сервису по DNS под может достичь его только в том случае, если сервис находится в том же пространстве имен, что и сам под. Пространства имен позволяют обойти это ограничение. Когда сервис создается в определенном пространстве имен, его имя добавляется в DNS-адрес следующим образом:

```
<имя_сервиса>.<имя_пространства>.svc.cluster.local
```

Обращаясь к этому доменному имени, мы можем получить доступ к любому сервису в любом пространстве имен кластера Kubernetes. Таким образом, взаимодействие между подами не ограничивается одним пространством имен: они могут обмениваться данными даже в том случае, если находятся в разных пространствах.

В следующем подразделе мы рассмотрим лучшие практики работы с пространствами имен в Kubernetes.

Лучшие практики работы с пространствами имен в Kubernetes

Хотя строгих правил создания и управления пространствами имен не существует, рассмотрим несколько признанных лучших практик.

- *Организация и разделение.*
  - *Логическая группировка* — объединяйте приложения, сервисы и ресурсы по их назначению, стадии разработки (разработка, тестирование, эксплуатация) или владельцу (разные команды). Это помогает поддерживать порядок и упрощает управление ресурсами.

  - *Изоляция* — используйте пространства имен для разделения развертываемых приложений. Это гарантирует то, что приложения в одном пространстве имен не будут мешать объектам в другом, то есть снижает вероятность конфликтов. Для повышения безопасности также можно использовать механизмы RBAC и сетевые политики.

- *Правила именования.*
  - *Ясные и описательные названия* — давайте пространствам имен такие названия, которые четко указывают на их назначение. Это упрощает их отслеживание, особенно в крупных кластерах. Хорошей практикой считается использование префиксов вроде `dev-`, `test-` или `prod-` для обозначения разных сред.
  - *Последовательность* — придерживайтесь единого стиля именования во всем кластере. Это упростит взаимодействие в команде и понимание происходящего.

- *Управление ресурсами.*
  - *Ограничение потребления ресурсов* — задавайте предельный объем ресурсов, которые может использовать пространство имен. Это предотвращает ситуацию, когда одно приложение потребляет все, а другим ничего не достается. Для более точного контроля потребление ресурсов можно ограничивать и на уровне подов.
  - *Задание лимитов* — определяйте правила для подов и контейнеров в пространстве имен, устанавливая ограничения на то, сколько ресурсов они могут запросить или использовать. Это позволяет точнее контролировать распределение ресурсов.

- *Контроль доступа.*
  - *Управление доступом на основе ролей* — применяйте механизм RBAC для задания разрешений на выполнение тех или иных действий в каждом пространстве имен. Предоставляйте пользователям и сервисам только те права, которые им действительно необходимы.

- *Наблюдение.*
  - *Мониторинг* — отслеживайте уровень потребления ресурсов и состояние приложений в каждом пространстве имен. Это помогает быстрее выявлять проблемы и эффективнее использовать ресурсы.
  - *Управление жизненным циклом пространства имен* — регулярно проверяйте актуальность пространств имен. Удаляйте те, которые больше не используются, для поддержания порядка и снижения рисков безопасности. Рассмотрите возможность автоматизации процессов создания и удаления пространств имен.

- *Прочие соображения.*
  - *Неидеальная изоляция* — хотя пространства имен и помогают изолировать объекты, они не обеспечивают абсолютной защиты. Для дополнительной безопасности могут потребоваться сетевые правила.
  - *Кластеры или пространства имен* — если ваша архитектура очень сложна и требует более надежного разделения ресурсов, подумайте об использовании нескольких кластеров Kubernetes вместо пространств имен.

Следуя этим практикам, вы сможете поддерживать инфраструктуру Kubernetes в организованном, безопасном и управляемом состоянии. Чтобы добиться наилучших результатов, адаптируйте их под особенности своей среды. На этом мы завершаем рассмотрение основ использования пространств имен в Kubernetes. Мы узнали, что они собой представляют, как их создавать и удалять, как использовать для упорядочения кластера и как настраивать контекст `kubeconfig` для выполнения операций над конкретным пространством имен.

А теперь обсудим более продвинутые возможности, связанные с пространствами имен. В следующем разделе вы познакомитесь с объектами ResourceQuota и Limit, которые позволяют ограничивать объем вычислительных ресурсов, доступный приложениям, развернутым в Kubernetes.

Настройка квот и лимитов на уровне пространства имен

В этом разделе вы узнаете о том, что пространства имен можно использовать не только для упорядочения ресурсов в кластере, но и для ограничения вычислительных ресурсов, доступных подам.

С помощью объектов ResourceQuota и Limit можно задавать предельный объем вычислительных ресурсов, которые могут задействовать ваши поды. Далее мы обсудим, как именно это делается. В целом задание квот и лимитов для рабочих кластеров считается хорошей практикой, поэтому данный навык имеет смысл освоить.

Зачем задавать квоты

Так же как приложения и системы, поды Kubernetes требуют определенного количества вычислительных ресурсов для корректной работы. В Kubernetes можно задать предельный объем для двух типов таких ресурсов:

- для ресурсов центрального процессора (ЦП);
- памяти.

Узлы кластера (вычислительные и управляющие) совместно обеспечивают поды процессорными ресурсами и памятью. В системе Kubernetes увеличение ресурсов ЦП и памяти сводится к добавлению новых вычислительных узлов, позволяющих запускать дополнительные поды. В зависимости от того, где размещен кластер Kubernetes — локально или в облаке, добавление вычислительных узлов может означать либо покупку и настройку дополнительного оборудования, либо простой вызов облачного API для создания новых виртуальных машин.

Как поды потребляют ресурсы

Когда вы запускаете под в Kubernetes, компонент управляющего слоя под названием `kube-scheduler` выбирает вычислительный узел и назначает ему этот под. Затем агент `kubelet` на выбранном узле пытается запустить контейнеры, определенные в спецификации пода.

Процесс выбора вычислительного узла в Kubernetes называется *планированием пода* (pod scheduling).

После назначения и запуска на вычислительном узле под по умолчанию получает доступ ко всем ресурсам этого узла. Ничто не мешает ему потреблять все больше ресурсов ЦП и памяти по мере роста нагрузки на приложение. Если подам перестает хватать памяти или процессорных ресурсов для нормальной работы, приложение аварийно завершается.

Это чревато серьезными проблемами, поскольку один вычислительный узел может одновременно использоваться для запуска нескольких приложений, а значит, и нескольких подов. Например, если на одном узле запущено десять подов и один из них начинает потреблять все доступные ресурсы, это неминуемо скажется на работе остальных девяти подов, запущенных на том же узле.

В связи с этим:

- каждый под должен иметь возможность запрашивать необходимый для работы объем ресурсов;
- кластер должен иметь возможность ограничивать потребление ресурсов конкретным подом, чтобы он не отобрал все ресурсы у остальных подов.

В системе Kubernetes это реализуется с помощью двух параметров, доступных в объекте Pod. Первый называется `requests` (запросы) и позволяет указать необходимый поду объем вычислительных ресурсов. Второй называется `limits` (лимиты) и задает максимальный объем ресурсов, который под может потребить. Разберем эти параметры подробнее.

Как поды могут запрашивать вычислительные ресурсы

Запросы и лимиты задаются в YAML-файле с определением пода. Кроме того, их можно применить и к работающему развертыванию с помощью команды `kubectl set resource`. В этом подразделе мы сосредоточимся на запросах.

Запрос определяет минимальное количество вычислительных ресурсов, которое необходимо поду для корректной работы. Хорошей практикой считается задание параметра `requests` для подов, особенно для тех, которые запускаются в рабочей среде.

Предположим, вы хотите запустить под `nginx` в кластере Kubernetes. Заполнив поле `requests`, вы можете сообщить системе, что для корректной работы этому поду потребуется как минимум 512 МиБ памяти и 25 % ресурсов одного процессорного ядра.

Ниже приведен YAML-файл для создания такого пода:

```
# pod-in-namespace-with-request.yaml
apiVersion: v1
kind: Pod
metadata:
  name: nginx-with-request
  namespace: custom-ns
spec:
  containers:
    - name: nginx
      image: nginx:latest
      resources:
        requests:
          memory: "512Mi"
          cpu: "250m"
```

Как видите, запросы можно делать на уровне контейнеров, задавая разные значения для каждого из них.

В данном случае под:

- создается в пространстве имен `custom-ns`;
- требует `512 Mi` памяти;
- требует `250m`, то есть 1/4 процессорного ядра.

Что означают эти метрики

Объем памяти выражается в байтах (один МиБ равен 1 048 576 байтам), а объем ресурсов ЦП — в *миллиядрах* и может задаваться дробными значениями. Если вы хотите, чтобы под потреблял ресурсы всего процессорного ядра, задайте для ключа `cpu` значение `1000m`. Два ядра соответствуют `2000m`, половина ядра — `500m`,

или `0.5`, и т. д. Однако при запросе целого ядра принято указывать целое число, например `2` вместо `2000m`. Таким образом, предыдущий YAML-файл означает, что создаваемому поду `nginx` потребуется 512 МиБ памяти и четверть процессорного ядра вычислительного узла. При этом в нем ничего не говорится о частоте ЦП или о памяти.

Когда вы применяете этот YAML-файл к своему кластеру, планировщик ищет вычислительный узел, способный запустить описанный в этом файле под, то есть имеющий достаточно доступных ресурсов ЦП и памяти для удовлетворения его запросов.

Если ни один вычислительный узел не сможет удовлетворить запросы пода, то этот под не запустится до тех пор, пока вы не освободите ресурсы, удалив какие-то из работающих подов, или не добавите новый вычислительный узел, способный его разместить.

Помните, что поды не могут быть распределены по нескольким узлам. Например, если вы зададите значение 8000m (соответствующее восьми ядрам), а ваш кластер состоит из двух узлов, каждый из которых предусматривает четыре ядра, то ни один узел не сможет удовлетворить запрос и под не будет назначен ни одному из них.

В связи с этим параметр `requests` следует использовать с осторожностью. Помните, что он задает минимальное количество вычислительных ресурсов, необходимых поду для работы. Слишком большое его значение может привести к тому, что под никогда не будет назначен узлу, но, с другой стороны, в случае успешного запуска указанный объем ресурсов будет ему гарантирован.

Теперь рассмотрим способ ограничения объема потребляемых ресурсов.

Как можно ограничить потребление ресурсов

При написании YAML-файла вы также можете задать лимиты (`limits`), определяющие максимальный объем ресурсов, доступный поду.

Указания запрашиваемого объема ресурсов недостаточно для корректной работы. Каждый раз при выделении ресурсов следует устанавливать лимит, указывающий системе Kubernetes максимальный объем ресурсов, которые под может использовать. Таким образом вы гарантируете, что под не потребит все ресурсы вычислительного узла.

Однако имейте в виду, что поведение Kubernetes зависит от того, какой именно лимит был исчерпан. При исчерпании выделенного ресурса ЦП произойдет снижение производительности пода. А при исчерпании выделенной памяти работа пода может быть завершена. Это связано с тем, что память нельзя ограничить

так же, как ресурсы ЦП, а система Kubernetes должна обеспечивать стабильность работы остальных приложений. Поэтому это важно учитывать.

Без задания лимитов под может потребить все ресурсы вычислительного узла. Ниже приведен обновленный YAML-файл для пода `nginx`, в который были добавлены ограничения на использование памяти и ресурсов ЦП.

Данный под сможет потреблять до 1 ГиБ памяти и до одного полного процессорного ядра вычислительного узла:

```
# pod-in-namespace-with-request-and-limit.yaml
apiVersion: v1
kind: Pod
metadata:
  name: nginx-with-request-and-limit
  namespace: quota-ns
spec:
  containers:
    - name: nginx
      image: nginx:latest
      resources:
        requests:
          memory: "512Mi"
          cpu: "250m"
        limits:
          memory: "1Gi"
          cpu: "1000m"
```

Таким образом, при задании запроса всегда следует указывать и лимит. В следующем примере мы сделаем и то и другое.

Для начала проверим доступность системных ресурсов. Поскольку для демонстрации используется кластер `minikube`, активизируем компонент `metrics-server` для сбора подробной информации о потреблении ресурсов. Подробнее об использовании метрик мы поговорим в главе 10.

```
$ minikube addons enable metrics-server
```

Дождитесь перехода подов сервера метрик в состояние `Running`, прежде чем продолжить:

```
$ kubectl get po -n kube-system | grep metrics
metrics-server-7c66d45ddc-82ngt   1/1     Running   0   113m
```

Теперь выведите информацию об использовании ресурсов кластера:

```
$ kubectl top node
NAME       CPU(cores)   CPU%   MEMORY(bytes)   MEMORY%
minikube   181m         1%     806Mi           2%
```

В данном случае в нашем кластере Kubernetes используется примерно 800 МиБ памяти и 180 миллиядер ЦП.

При использовании minikube с драйвером Podman или Docker будут отображены доступные ресурсы (ЦП и память) хостовой машины, а не узла кластера Kubernetes. В этом случае для использования ресурсов ВМ minikube рекомендуется развернуть дополнительный кластер с помощью драйвера VirtualBox: minikube start --profile cluster2-vb --driver=virtualbox.

Создадим новое пространство имен для демонстрации принципа работы с запросами и лимитами ресурсов:

```
$ kubectl create ns quota-ns
namespace/quota-ns created
```

Теперь создадим YAML-файл с нереалистичным запросом на выделение памяти, например, в объеме 100 ГиБ:

```
# pod-with-request-and-limit-1.yaml
apiVersion: v1
kind: Pod
metadata:
  name: nginx-with-request-and-limit-1
  namespace: quota-ns
spec:
  containers:
    - name: nginx
      image: nginx:latest
      resources:
        requests:
          memory: "100Gi"
          cpu: "100m"
        limits:
          memory: "100Gi"
          cpu: "500m"
```

Создайте под с помощью команды `kubectl apply`:

```
$ kubectl apply -f pod-with-request-and-limit-1.yaml
pod/nginx-with-request-and-limit-1 created
```

Проверьте статус пода:

```
$ kubectl get pod -n quota-ns
NAME                            READY   STATUS    RESTARTS   AGE
nginx-with-request-and-limit-1   0/1     Pending   0          45s
```

Статус `Pending` говорит о том, что под еще не запущен. Выведем подробные сведения о поде с помощью команды `kubectl describe`:

```
$ kubectl describe po nginx-with-request-and-limit-1 -n quota-ns
Name:         nginx-with-request-and-limit-1
Namespace:    quota-ns
...<фрагмент опущен для краткости>...
Status:       Pending
IP:
IPs:          <none>
Containers:
  nginx:
    ...<фрагмент опущен для краткости>...
    Limits:
      cpu:     500m
      memory:  100Gi
    Requests:
      cpu:     100m
      memory:  100Gi
    Environment:  <none>
... <фрагмент опущен для краткости>...
  Warning  FailedScheduling  105s  default-scheduler  0/1 nodes are available:
1 Insufficient memory.
preemption: 0/1 nodes are available: 1 No preemption victims found
for incoming pod.
```

В выводе вы увидите сообщение о том, что планировщик не смог найти ни одного узла в кластере, способного предоставить запрошенный объем памяти. Это означает, что под не будет запущен до тех пор, пока в кластере не появится подходящий узел.

Теперь изменим запрос в YAML-файле на более реалистичное значение 1 ГиБ:

```
# pod-with-request-and-limit-2.yaml
apiVersion: v1
kind: Pod
metadata:
  name: nginx-with-request-and-limit-2
  namespace: quota-ns
spec:
  containers:
    - name: nginx
      image: nginx:latest
      resources:
        requests:
          memory: "1Gi"
          cpu: "100m"
        limits:
          memory: "2Gi"
          cpu: "500m"
```

И снова создадим под:

```
$ kubectl apply -f pod-with-request-and-limit-2.yaml
pod/nginx-with-request-and-limit-2 created
$ kubectl get po -n quota-ns
NAME                              READY   STATUS    RESTARTS   AGE
nginx-with-request-and-limit-1    0/1     Pending   0          8m24s
nginx-with-request-and-limit-2    1/1     Running   0          20s
```

Как видите, в этот раз планировщик Kubernetes смог подобрать подходящий узел и под успешно запустился.

Теперь, когда вы знаете о запросах и лимитах, не забывайте задавать их для своих подов.

Зачем нужны объекты ResourceQuota

Вы можете управлять потреблением ресурсов с помощью одних лишь запросов и лимитов. Все приложения в Kubernetes представляют собой поды, и последовательное задание этих двух параметров позволяет контролировать использование ресурсов в кластере.

Однако забыть про эти параметры довольно легко. Велика вероятность, что вы или кто-то из членов вашей команды однажды развернете в кластере под без задания запросов и лимитов. Это создаст риск нестабильной работы приложений, поскольку под без лимита способен потребить все ресурсы вычислительного узла, на котором он был запущен.

Для решения этой проблемы в системе Kubernetes предусмотрено два полезных объекта: ResourceQuota и LimitRange, позволяющих задавать подобные ограничения на уровне пространства имен.

ResourceQuota (квота ресурсов) представляет собой такой же ресурс Kubernetes, как Pod или ConfigMap. Процесс работы с ним очень прост и предусматривает всего два шага.

1. Создание нового пространства имен.
2. Создание объектов ResourceQuota и LimitRange внутри этого пространства.

После этого все поды, запускаемые в данном пространстве имен, будут ограничены этими объектами.

Квоты позволяют, например, гарантировать, что все контейнеры, работающие в пространстве имен, не смогут потребить более 4 ГиБ оперативной памяти.

Таким образом, ограничивать выполнение процессов внутри подов не только можно, но и желательно. Настоятельно рекомендуется создавать объекты ResourceQuota и LimitRange для каждого пространства имен в кластере!

Без этих квот развернутые объекты смогут потребить любое количество ресурсов ЦП или памяти, что неизбежно приведет к нестабильности кластера и всех работающих в нем приложений, если, конечно, в конфигурациях подов не будут заданы параметры `requests` и `limits`.

В общем случае объект ResourceQuota используется для ограничения:

- потребления ресурсов ЦП в пределах пространства имен;
- потребления памяти в пределах пространства имен;
- абсолютного количества объектов (Pod, Service, ReplicationController, ReplicaSet, Deployment и т. д.), работающих в пространстве имен;
- использования ресурсов хранилища в зависимости от его класса.

Сценариев применения этого объекта достаточно много, и все они подробно описаны в документации по Kubernetes. Теперь рассмотрим способ определения объекта ResourceQuota в пространстве имен.

Создание объекта ResourceQuota

Чтобы продемонстрировать полезность объекта ResourceQuota, создадим один такой объект для пространства имен `quota-ns`. Он будет задавать ограничения на объем ресурсов, который смогут потреблять все поды в данном пространстве имен. Ниже приведен YAML-файл для создания этого объекта (обратите внимание на поле `kind`):

```
# resourcequota.yaml
apiVersion: v1
kind: ResourceQuota
metadata:
  name: resourcequota
  namespace: quota-ns
spec:
  hard:
    requests.cpu: "1000m"
    requests.memory: "1Gi"
    limits.cpu: "2000m"
    limits.memory: "2Gi"
```

Помните, что объект ResourceQuota относится только к одному пространству имен.

В данном примере задаются следующие ограничения для пространства имен `quota-ns`. Все поды в совокупности не смогут:

- запросить более одного ядра ЦП;
- запросить более 1 ГиБ памяти;
- задействовать более двух ядер ЦП;
- потребить более 2 ГиБ памяти.

Для создания объекта ResourceQuota достаточно применить YAML-конфигурацию:

```
$ kubectl apply -f resourcequota.yaml
resourcequota/resourcequota created

$ kubectl get quota -n quota-ns
NAME           AGE   REQUEST                                          LIMIT
resourcequota  4s    requests.cpu: 100m/1, requests.memory: 1Gi/1Gi   limits.
cpu: 500m/2, limits.memory: 2Gi/2Gi
```

Теперь проверим ресурсы, доступные в пространстве имен `quota-ns`:

```
$ kubectl get po -n quota-ns
NAME                           READY   STATUS    RESTARTS         AGE
nginx-with-request-and-limit-2 1/1     Running   1 (5h41m ago)    7h18m

$ kubectl top pod -n quota-ns
NAME                           CPU(cores)   MEMORY
nginx-with-request-and-limit-2 50m          120Mi
```

Если вы не удалили предыдущий демонстрационный под, то в пространстве будет находиться под `nginx`, а уровень потребления ресурсов будет очень низким.

Теперь у нас есть новый YAML-файл пода, в котором мы запрашиваем 3 ГиБ памяти:

```
# pod-with-request-and-limit-3.yaml
apiVersion: v1
kind: Pod
metadata:
  name: nginx-with-request-and-limit-3
  namespace: quota-ns
spec:
  containers:
    - name: nginx
      image: nginx:latest
      resources:
        requests:
          memory: "3Gi"
          cpu: "100m"
        limits:
          memory: "4Gi"
          cpu: "500m"
```

Создадим этот под и посмотрим на результат:

```
$ kubectl apply -f pod-with-request-and-limit-3.yaml
Error from server (Forbidden): error when creating "pod-with-request-andlimit-3.
yaml": pods "nginx-with-request-and-limit-3" is forbidden: exceeded quota:
resourcequota, requested: limits.memory=4Gi,requests.memory=3Gi, used: limits.
memory=2Gi,requests.memory=1Gi, limited: limits.memory=2Gi,requests.memory=1Gi
```

Причина ошибки вполне очевидна: количество запрошенных ресурсов превышает квоту, поэтому система Kubernetes не позволила создать новый под.

В пространство имен можно поместить сколько угодно подов и контейнеров при условии соблюдения ограничений. Чаще всего объект ResourceQuota применяется для управления запросами и лимитами, но он также может использоваться для задания этих ограничений на уровне пространства имен.

При настройке квот ресурсов на уровне пространства имен важно не допустить потребления всех ресурсов кластера одним пространством. Кроме того, важно задавать запросы и лимиты на уровне объектов Pod или Deployment. Такой двухуровневый подход гарантирует, что избыточное потребление ресурсов будет сдерживаться как в пределах отдельных пространств имен, так и на уровне приложений. Благодаря этим ограничениям можно создать более предсказуемую и стабильную среду, в которой ни один компонент не способен нарушить работу всей системы.

Ниже приведен обновленный YAML-файл для объекта ResourceQuota, указывающий на то, что пространство имен, в котором он создан, не может содержать более десяти объектов ConfigMap и более пяти сервисов. Это ограничение само по себе бессмысленно, но хорошо демонстрирует возможные варианты использования объекта ResourceQuota:

```
# resourcequota-with-object-count.yaml
apiVersion: v1
kind: ResourceQuota
metadata:
  name: my-resourcequota
  namespace: quota-ns
spec:
  hard:
    requests.cpu: "1000m"
    requests.memory: "1Gi"
    limits.cpu: "2000m"
    limits.memory: "2Gi"
    configmaps: "10"
    services: "5"
```

При использовании YAML-определения ResourceQuota убедитесь, что объект назначен правильному пространству имен. Если пространство не задано в YAML-файле, добавьте флаг --namespace, чтобы его указать.

Создайте объект ResourceQuota с помощью такой команды:

```
$ kubectl apply -f resourcequota-with-object-count.yaml
```

В следующем подразделе мы обсудим квоты на использование ресурсов хранилища в Kubernetes.

Квоты на использование ресурсов хранилища

В системе Kubernetes квоты также позволяют контролировать суммарный объем ресурсов хранилища, запрашиваемый в пределах пространства имен. Мы можем задавать ограничения как на общий объем запрошенных ресурсов хранилища для всех объектов PersistentVolumeClaim (PVC), так и на количество самих заявок PVC. Кроме того, квоты можно определять для конкретных классов хранилища, устанавливая отдельные лимиты для их разных типов. Например, можно задать разные квоты для классов `gold` и `bronze`. Начиная с версии 1.8, квоты также можно применять для ограничения совокупного объема запросов и лимитов на использование ресурсов локального эфемерного хранилища всеми подами в пространстве имен.

Теперь посмотрим, как можно вывести список всех существующих объектов ResourceQuota.

Вывод списка объектов ResourceQuota

Список объектов ResourceQuota можно вывести с помощью команды `kubectl get` с указанием параметра `resourcequotas`:

```
$ kubectl get resourcequotas -n quota-ns
NAME           AGE   REQUEST                                              LIMIT
resourcequota  15m   requests.cpu: 100m/1, requests.memory: 1Gi/1Gi   limits.
cpu: 500m/2, limits.memory: 2Gi/2Gi
```

Теперь удалим объект ResourceQuota из кластера Kubernetes.

Удаление объекта ResourceQuota

Чтобы удалить объект ResourceQuota из кластера, используйте команду `kubectl delete`:

```
$ kubectl delete -f resourcequota-with-object-count.yaml
resourcequota "my-resourcequota" deleted
```

А теперь познакомимся с объектом LimitRange.

Знакомство с объектом LimitRange

LimitRange (диапазон ограничений) — это еще один объект, создаваемый на уровне пространства имен. Он используется для задания значений запросов и лимитов по умолчанию для отдельных контейнеров. Даже при наличии в пространстве имен объекта ResourceQuota вы можете создать контейнер, способный

потребить все доступные ресурсы. Объект LimitRange как раз и предотвращает создание слишком маленьких или чрезмерно больших контейнеров в пространстве имен.

Ниже приведен YAML-файл для создания объекта LimitRange:

```
# limitrange.yaml
apiVersion: v1
kind: LimitRange
metadata:
  name: my-limitrange
  namespace: quota-ns
spec:
  limits:
    - default:             # задает значение лимитов по умолчанию
        cpu: 500m
        memory: 256Mi
      defaultRequest:      # задает значение запросов по умолчанию
        cpu: 500m
        memory: 128Mi
      max:                 # значения max и min определяют диапазоны ограничений
        cpu: "1"
        memory: 1000Mi
      min:
        cpu: 100m
        memory: 128Mi
      type: Container
```

Как видите, YAML-файл для создания объекта LimitRange содержит четыре ключевых раздела, в каждом из которых задается объем памяти и ресурсов ЦП.

- `default` — задает лимиты памяти и ресурсов ЦП по умолчанию для контейнеров, если они не указаны на уровне пода. Все контейнеры без явно заданных лимитов наследуют эти значения объекта LimitRange.
- `defaultRequest` — задает запросы ресурсов ЦП и памяти по умолчанию. Если для контейнера в поде не указан параметр `requests`, то к нему автоматически применяются значения из этого раздела.
- `max` — данное значение определяет максимальный лимит (не запрос), который может быть установлен для контейнера в поде. Нельзя настроить под со значением лимита, превышающим этот. Как и значение `default`, данный лимит не может превысить указанный здесь предел.
- `min` — это значение аналогично `max`, но применяется к запросам (`requests`). Это минимальный объем ресурсов, который может запросить под; параметр `defaultRequest` не может быть ниже этого значения.

Учтите, что если ключи `default` и `defaultRequest` не заданы, то ключ `max` используется как значение по умолчанию для `default`, а ключ `min` — как значение по умолчанию для `defaultRequest`.

Задание диапазона ограничений служит страховкой на тот случай, если вы забудете указать значения `requests` и `limits` для подов. Благодаря объекту LimitRange у них всегда будут значения по умолчанию.

```
$ kubectl apply -f limitrange.yaml
```

Теперь посмотрим, как можно вывести список объектов LimitRange.

Вывод списка объектов LimitRange

Вывести список объектов LimitRange можно с помощью утилиты `kubectl`. Не забудьте добавить флаг `-n`, чтобы ограничить запрос определенным пространством имен:

```
$ kubectl get limitranges -n quota-ns
NAME           CREATED AT
my-limitrange  2024-03-10T16:13:00Z

$ kubectl describe limitranges my-limitrange -n quota-ns
Name:       my-limitrange
Namespace:  quota-ns
Type        Resource  Min    Max     Default Request  Default Limit  Max Limit/
Request Ratio
----        --------  ---    ---     ---------------  -------------  ----------
-------------
Container   memory    128Mi  1000Mi  128Mi            256Mi          -
Container   cpu       100m   1       500m             500m           -
```

А теперь удалим объект LimitRange из пространства имен.

Удаление объекта LimitRange

Удалить объект LimitRange также можно с помощью утилиты `kubectl`:

```
$ kubectl delete limitranges my-limitrange -n quota-ns
limitrange "my-limitrange" deleted
```

Как и всегда, не забывайте добавлять флаг `-n`, чтобы указать конкретное пространство имен, иначе вы рискуете затронуть не то, которое вам нужно!

Резюме

В этой главе вы познакомились с таким важным механизмом системы Kubernetes, как пространства имен. Они обеспечивают логическую изоляцию ресурсов, поэтому необходимы для эффективного управления кластером. Чаще всего пространства имен создаются для отдельных сред или приложений. Нередко в кластерах создаются сотни таких пространств.

Мы выяснили, что большинство ресурсов Kubernetes привязаны к конкретным пространствам имен, хотя есть и исключения. Помните, что по умолчанию в Kubernetes предусмотрено несколько предопределенных пространств имен, таких как `kube-system`. Изменять объекты в этих пространствах не рекомендуется, особенно если вы не до конца понимаете, что делаете.

Вы также узнали, что пространства имен можно использовать для задания квот и лимитов, ограничивающих объем ресурсов, доступных подам. Для предотвращения чрезмерного потребления вычислительных ресурсов подами рекомендуется задавать квоты и лимиты на уровне пространства имен с помощью объектов ResourceQuota и LimitRange. Эти меры позволяют эффективно управлять ресурсами, а значит, поддерживать стабильность и производительность всех приложений, работающих в кластере.

В следующей главе мы обсудим способы управления конфигурационными и чувствительными данными в Kubernetes с помощью объектов ConfigMap и Secret.

Дополнительные источники информации

- Предопределенные пространства имен в Kubernetes: https://kubernetes.io/docs/concepts/overview/working-with-objects/namespaces/#initial-namespaces.
- Управление ресурсами подов и контейнеров: https://kubernetes.io/docs/concepts/configuration/manage-resources-containers/.
- Квоты ресурсов: https://kubernetes.io/docs/concepts/policy/resource-quotas/.
- Диапазоны ограничений: https://kubernetes.io/docs/concepts/policy/limit-range/.
- Механизм для сбора, агрегации и предоставления метрик использования ресурсов: https://kubernetes.io/docs/tasks/debug/debug-cluster/resource-metrics-pipeline/.
- Настройка запросов и лимитов памяти по умолчанию для пространства имен: https://kubernetes.io/docs/tasks/administer-cluster/manage-resources/memory-default-namespace/.

7 Настройка подов с помощью объектов ConfigMap и Secret

В предыдущих главах вы научились запускать контейнеры приложений. Теперь вы знаете, что в Kubernetes это делается через поды. Понимание этой концепции является ключевым условием освоения данного оркестратора. Kubernetes — это сложная система, управляемая через RESTful API. Центральным компонентом, который отвечает за этот процесс, является API-сервер Kubernetes, предоставляющий основной интерфейс для взаимодействия с кластером. Когда пользователи создают объекты Kubernetes через этот API, система реагирует выделением необходимых ресурсов на узлах кластера. Среди этих объектов особенно выделяется под, так как его создание на узле Kubernetes приводит к запуску контейнеров приложения.

В этой главе мы познакомимся с двумя новыми объектами Kubernetes: *ConfigMap* и *Secret*, которые используются для отделения конфигурации приложения от его кода. Эти объекты позволяют управлять параметрами конфигурации, не затрагивая код, что повышает переносимость и безопасность приложений. Объект ConfigMap хранит неконфиденциальные данные конфигурации в формате пар «ключ — значение», тогда как объект Secret предназначен для работы с такой секретной информацией, как пароли или ключи API. Оба объекта можно подключать к подам в виде переменных окружения или монтировать в виде томов, позволяя приложениям динамически получать доступ к параметрам конфигурации и обходясь без жестко заданных в коде значений. Отделяя конфигурацию от приложения, мы создаем более гибкие, устойчивые и безопасные развертывания в экосистеме Kubernetes.

Технические требования

Для работы с этой главой вам потребуется:

- локальный или облачный кластер Kubernetes;
- рабочий интерфейс командной строки `kubectl`, настроенный для взаимодействия с кластером.

Инструкции по созданию кластера Kubernetes и настройке клиента `kubectl` приведены в главах 2 и 3.

Примеры кода для этой главы можно загрузить из официального репозитория GitHub по адресу https://github.com/PacktPublishing/The-Kubernetes-Bible-Second-Edition/tree/main/Chapter07.

Объекты ConfigMap и Secret

Среды Kubernetes динамичны и постоянно меняются, что затрудняет управление конфигурацией приложений, поскольку традиционные методы часто не соответствуют быстрому темпу облачно-нативной разработки.

Для решения этой проблемы система Kubernetes предоставляет объекты ConfigMap (словарь конфигурации) и Secret (секрет) — специализированные средства управления разными типами конфигурационных данных. Разделяя конфигурацию и код приложения, эти ресурсы существенно повышают его переносимость, безопасность и управляемость.

В следующих разделах мы подробно рассмотрим принцип работы объектов ConfigMap и Secret, а также способы их эффективного использования в развертываниях Kubernetes.

Отделение приложения от его конфигурации

Контейнеры по своей природе неизменяемы, то есть после создания их нельзя модифицировать. Это усложняет управление конфигурацией. Если встроить конфигурацию непосредственно в образ контейнера, то каждое ее изменение будет требовать повторной сборки всего образа, а это процесс трудоемкий и неэффективный. Использование только переменных окружения или аргументов командной строки также не особенно удобно, особенно в сложных сценариях, и не гарантирует сохранения конфигурации при перезапуске контейнера. Эти ограничения подчеркивают потребность в более эффективной стратегии управления конфигурацией.

Работая в Kubernetes, мы стремимся сделать приложения максимально переносимыми. Один из самых надежных способов достижения этой цели сводится к отделению приложения от его конфигурации. Раньше, когда приложения проектировались для работы лишь в одной среде, конфигурационные параметры часто включались прямо в программный код, в результате чего конфигурация и код оказывались тесно связанными.

Однако такое объединение снижает переносимость приложения. В современных условиях нам требуется способ обновления конфигурации приложения, обеспечивающий его переносимость и возможность развертывания в нескольких средах.

Рассмотрим сценарий.

1. Вы развертываете Java-приложение в среде разработки для проведения тестирования.
2. После завершения тестов приложение готово к развертыванию в рабочей среде. Однако конечная точка MySQL в этой среде отличается от используемой в среде разработки.

В этой ситуации возможны два варианта.

- Если конфигурация и код приложения не разделены и адрес конечной точки MySQL жестко прописан в коде, то вам придется пересобрать все приложение после правки его исходного кода.
- Если конфигурация и код приложения разделены, то вам будет достаточно переопределить конечную точку MySQL при развертывании приложения в рабочей среде.

В этом и заключается ключевая идея переносимости ПО: код приложения не должен зависеть от инфраструктуры, в которой оно работает. Лучший способ добиться этого — отделить код приложения от его конфигурации.

На рис. 7.1 показан общий образ контейнера приложения и разные конфигурации для разных сред.

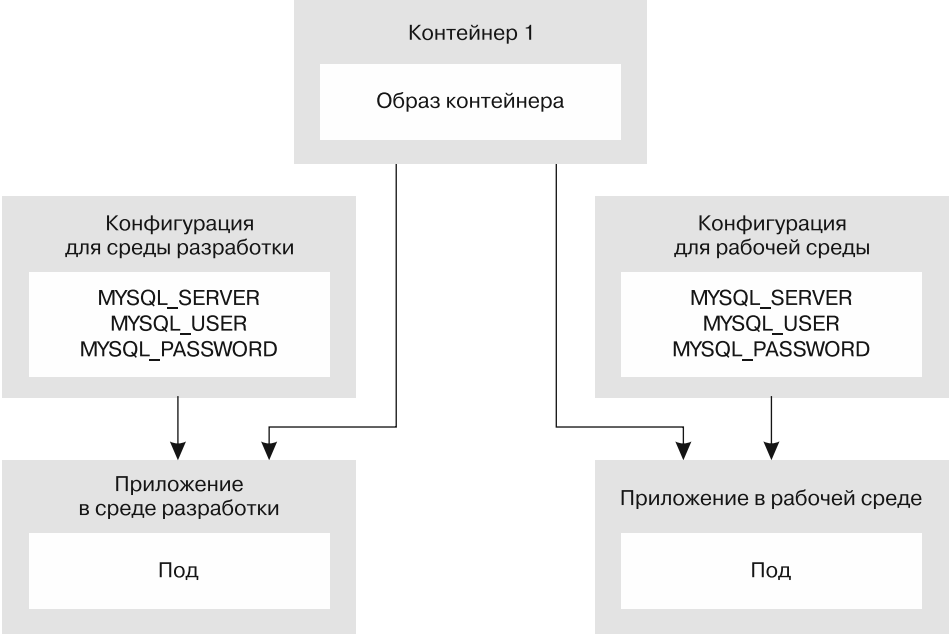

Рис. 7.1. Конфигурация приложения отделена от кода

Вот несколько типичных примеров конфигурационных параметров, которые следует отделять от кода приложения.

- Ключи API для доступа к хранилищу Amazon S3.
- Пароль от сервера MySQL, используемый приложением.
- Адрес конечной точки кластера Redis, используемый приложением.
- Заранее вычисленные значения, например закрытые ключи для подписи JWT-токенов.

Все эти значения, как правило, меняются в зависимости от среды. Приложения, запускаемые в подах, должны иметь возможность загружать разные конфигурации в зависимости от окружения, в котором они выполняются. Поэтому мы стремимся систематически поддерживать границу между приложениями и используемыми ими конфигурациями. Это позволяет нам рассматривать их в качестве двух совершенно разных сущностей в кластере Kubernetes.

Для хранения конфигурационных данных в этой системе предусмотрены объекты ConfigMap и Secret, которые необходимо подключать при создании подов.

Не включайте конфигурационные параметры в контейнерные образы (например, в Docker-образы). Ваш файл Dockerfile (или Containerfile в случае Podman) должен собирать приложение, но не конфигурировать его. Задействуя конфигурацию контейнера на этапе сборки, вы создаете жесткую зависимость между приложением и его настройками, что снижает переносимость контейнера между разными средами.

Объект ConfigMap предназначен для хранения неконфиденциальных конфигурационных параметров, а объект Secret, по сути, выполняет ту же роль, но используется для хранения конфиденциальных данных, таких как пароли баз данных, ключи API и т. п.

Таким образом, для каждой среды и для каждого приложения можно создать собственные объекты ConfigMap и Secret, которые будут содержать параметры, необходимые приложению для работы в конкретном контексте и окружении. Важный момент заключается в том, что эти объекты служат механизмами хранения данных конфигурации в формате пар «ключ — значение» внутри кластера Kubernetes. Эти пары могут содержать как простые значения (литералы), так и целые конфигурационные файлы, например, в формате YAML, TOML и т. д. При создании пода можно указать имя объекта ConfigMap или Secret и связать его с подом, чтобы сделать конфигурационные значения доступными контейнерам, работающим внутри него.

Данная процедура всегда выполняется в следующем порядке.

1. Создание объекта ConfigMap или Secret с конфигурационными параметрами.
2. Создание пода, ссылающегося на этот объект ConfigMap или Secret.

Придерживаясь такого подхода, вы повышаете переносимость и удобство сопровождения приложений, следуя лучшим практикам DevOps.

Теперь, когда вы знаете, зачем отделять код приложения от его конфигурации, обсудим способ реализации этого подхода в Kubernetes.

Как поды потребляют объекты ConfigMap и Secret

Перед рассмотрением объектов ConfigMap и Secret вспомним традиционные методы управления конфигурацией в контейнеризованных средах. Вне Kubernetes современные контейнерные приложения получают данные конфигурации несколькими способами:

- в виде переменных окружения ОС;
- в виде конфигурационных файлов;
- через аргументы командной строки;
- путем обращения к API.

Это объясняется простотой переопределения переменной окружения с помощью Docker или Podman, а также наличием функций для чтения таких переменных во всех языках программирования. К конфигурационным файлам тоже можно легко предоставить общий доступ нескольким контейнерам, смонтировав их в качестве томов.

Важно учитывать, что, хотя объекты ConfigMap и Secret являются основными средствами управления конфигурацией в Kubernetes, существуют и альтернативные подходы. Передавать параметры конфигурации непосредственно в контейнеры можно с помощью аргументов командной строки, однако этот способ является менее гибким и безопасным по сравнению с применением ConfigMap и Secret. Прямое обращение к API Kubernetes для получения конфигурации, как правило, не рекомендуется из-за рисков безопасности и дополнительной сложности.

После создания в кластере Kubernetes объекты ConfigMap и Secret можно задействовать следующим образом:

- включить их в качестве переменных окружения в контейнер, запущенный в поде;
- смонтировать их в качестве обычных томов Kubernetes.

В виде переменной окружения в под можно внедрить как одно конкретное значение из ConfigMap или Secret, так и все сразу.

Использование переменных окружения для передачи данных из объекта Secret настоятельно не рекомендуется. Это объясняется потенциальными рисками безопасности и вероятностью усечения значений. Объекты Secret рекомендуется монтировать в виде файловых томов, так как это позволяет задействовать механизм кеширования kubelet для динамического обновления и повышения безопасности.

Объекты ConfigMap и Secret также могут использоваться в качестве монтируемых томов. Монтируя ConfigMap как том, вы можете внедрить все его значения в каталог внутри контейнера. Если в ConfigMap хранятся полноценные конфигурационные файлы, то этот способ существенно упрощает переопределение конфигурационного каталога.

Итак, теперь вы понимаете важность объектов ConfigMap и Secret для настройки приложений. Их освоение является необходимым условием для грамотной и уверенной работы с Kubernetes. Как уже отмечалось, ConfigMap используется для хранения неконфиденциальных конфигурационных параметров, тогда как Secret предназначен для хранения *более чувствительных* данных, например хешей или паролей от баз данных. Основные различия между этими объектами перечислены в табл. 7.1.

Таблица 7.1. Различия между объектами ConfigMap и Secret

Характеристика	ConfigMap	Secret
Назначение	Хранение неконфиденциальных конфигурационных данных	Хранение конфиденциальной информации
Формат данных	Обычный текст	Кодировка base64
Безопасность	Менее защищенный	Более защищенный
Типичные сценарии использования	Конфигурация приложений, переменные окружения и аргументы командной строки	Пароли, ключи API, SSH-ключи и сертификаты TLS
Работа с конфиденциальными данными	Не рекомендуется	Настоятельно рекомендуется

Поскольку объекты ConfigMap и Secret ведут себя по-разному, мы рассмотрим их по отдельности. Начнем с ConfigMap, а затем перейдем к Secret.

Настройка подов с помощью ConfigMap

В этом разделе вы научитесь перечислять, создавать, удалять и читать содержимое объектов ConfigMap. Кроме того, вы узнаете, как подключать их к подам в виде переменных окружения или томов.

Итак, рассмотрим способы перечисления и создания объектов ConfigMap в Kubernetes, а также управления ими.

Перечисление объектов ConfigMap

Как и в случае с любыми другими объектами Kubernetes, просмотреть все объекты ConfigMap, созданные в кластере, можно с помощью утилиты `kubectl`. Для этого укажите полное имя ресурса `configmaps`:

```
$ kubectl get configmaps
```

Или воспользуйтесь сокращенным псевдонимом `cm`:

```
$ kubectl get cm
```

Обе команды решают одну задачу. В результате их выполнения `kubectl` выводит список предустановленных объектов ConfigMap или сообщение о том, что они не найдены. Эта разница объясняется тем, что одни облачные сервисы создают объекты ConfigMap по умолчанию для внутренних процессов, а другие — нет. Наличие или отсутствие таких предустановленных объектов зависит от среды, в которой развернут кластер Kubernetes.

```
$ kubectl get configmaps -A
NAMESPACE         NAME               DATA   AGE
default           kube-root-ca.crt   1      23d
kube-node-lease   kube-root-ca.crt   1      23d
kube-public       cluster-info       1      23d
kube-public       kube-root-ca.crt   1      23d
...               ...                ...    ...
... <фрагмент опущен для краткости>...
```

Как видите, в пространстве имен `default` и в служебных пространствах, управляемых Kubernetes, присутствует несколько объектов ConfigMap, созданных в процессе развертывания кластера. Теперь обсудим процесс создания таких объектов.

Создание объекта ConfigMap

Как и другие объекты Kubernetes, объект ConfigMap можно создать императивным или декларативным способом. Иначе говоря, вы можете либо создать пустой объект ConfigMap и затем добавить в него значения, либо задать начальные значения непосредственно при его создании. Следующая команда создает пустой объект ConfigMap с именем `my-first-configmap` императивным способом:

```
$ kubectl create configmap my-first-configmap
configmap/my-first-configmap created
```

После выполнения этой команды можно снова ввести команду `kubectl get cm`, чтобы увидеть созданный объект:

```
$ kubectl get cm
NAME                DATA   AGE
my-first-configmap   0      42s
```

Теперь создадим пустой объект ConfigMap декларативным способом. Для этого необходимо подготовить YAML-файл и применить его с помощью утилиты `kubectl`.

Приведенный ниже код необходимо поместить в файл `~/my-second-configmap.yaml`:

```
# my-second-configmap.yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: my-second-configmap
```

После создания файла примените его к вашему кластеру Kubernetes с помощью команды `kubectl apply -f`:

```
$ kubectl apply -f my-second-configmap.yaml
configmap/my-second-configmap created
```

Затем снова выполните команду `kubectl get cm`, чтобы убедиться в том, что новый объект ConfigMap появился рядом с созданным ранее:

```
$ kubectl get cm
NAME                DATA   AGE
kube-root-ca.crt    1      23d
my-first-configmap   0      5s
my-second-configmap  0      2s
```

Обратите внимание, что в выводе команды `kubectl get cm` в столбце `DATA` отображается количество ключей, содержащихся в каждом объекте ConfigMap. Сейчас оно равно нулю, но в следующих примерах мы обсудим способ заполнения этого объекта при его создании, и тогда столбец `DATA` будет отражать количество ключей, добавленных в ConfigMap.

Создание объекта ConfigMap на основе литеральных значений

Поскольку пустой объект ConfigMap практически бесполезен, создадим такой объект с изначально заданными значениями. Это можно сделать императивным способом, добавив флаг `--from-literal` к команде `kubectl create cm`.

В следующем примере мы создаем ConfigMap с именем `my-third-configmap`, в котором ключу `color` присвоено значение `blue`:

```
$ kubectl create cm my-third-configmap --from-literal=color=blue
configmap/my-third-configmap created
```

Мы также можем создать ConfigMap сразу с несколькими параметрами, добавив в команду нужное количество флагов `--from-literal` и значений:

```
$ kubectl create cm my-fourth-configmap \
    --from-literal=color=blue \
    --from-literal=version=1 \
    --from-literal=environment=prod
configmap/my-fourth-configmap created
```

В данном примере создается объект ConfigMap, содержащий три конфигурационных параметра. Теперь, если вы выполните команду `kubectl get cm`, увидите в списке вновь созданные объекты.

Обратите внимание, что в столбце `DATA` теперь отражается количество конфигурационных значений, содержащихся в каждом объекте ConfigMap:

```
$ kubectl get cm
NAME                 DATA   AGE
my-first-configmap   0      9m30s
my-fourth-configmap  3      6m23s
my-second-configmap  0      8m2s
my-third-configmap   1      7m9s
```

Чтобы просмотреть подробные сведения о ConfigMap (или любом другом объекте), их можно вывести в удобном для восприятия формате YAML или JSON:

```
$ kubectl get cm my-fourth-configmap -o yaml
apiVersion: v1
data:
  color: blue
  environment: prod
  version: "1"
kind: ConfigMap
metadata:
  creationTimestamp: "2024-08-10T06:20:49Z"
  name: my-fourth-configmap
  namespace: default
  resourceVersion: "25647"
  uid: 3c8477dc-f3fe-4d69-b66a-403679a88450
```

Данный подход также упрощает резервное копирование объектов в формате YAML или JSON, что соответствует лучшим практикам *CaC* (Configuration as Code — конфигурация как код) и *IaC* (Infrastructure as Code — инфраструктура как код).

Тот же объект ConfigMap можно создать и декларативным способом. Для этого используется YAML-файл, в котором присутствует ключ `data`, содержащий все конфигурационные значения:

```
# my-fifth-configmap.yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: my-fifth-configmap
data:
  color: "blue"
  version: "1"
  environment: "prod"
```

После создания файла примените его к кластеру с помощью команды:

```
$ kubectl apply -f my-fifth-configmap.yaml
configmap/my-fifth-configmap created
```

В результате на основе файла `my-fifth-configmap.yaml` будет создан новый объект ConfigMap, включающий указанные данные.

Теперь посмотрим, как можно хранить целые конфигурационные файлы внутри ConfigMap.

Хранение целых конфигурационных файлов в объекте ConfigMap

Как уже было сказано, в объекте ConfigMap можно хранить не только литеральные значения, но и целые файлы. Для этого достаточно передать путь к файлу в файловой системе в команду `kubectl`. В результате содержимое файла будет считано и добавлено в ConfigMap в виде параметров.

Возможность сохранять содержимое конфигурационного файла в ConfigMap весьма полезна, так как этот объект можно монтировать в поды точно так же, как и тома.

Более того, внутри одного объекта ConfigMap можно хранить и литеральные значения, и файлы. Литералы представляют собой короткие строки, а файлы рассматриваются как длинные строки, так что они не воспринимаются системой как два разных типа данных.

Мы уже рассматривали подобный пример при обсуждении пода с контейнером-прицепом в главе 5, в котором конфигурация Fluentd сохранялась внутри объекта ConfigMap в виде файла.

Сейчас мы создадим шестой объект ConfigMap, который, помимо литерального значения, будет содержать еще и файл.

Сначала создадим файл `configfile.txt` с произвольным содержимым по адресу `$HOME/configfile.txt`:

```
$ echo "I'm just a dummy config file" >> $HOME/configfile.txt
```

Файл имеет расширение `.txt`, однако конфигурационные данные могут храниться в любом формате, поддерживаемом приложением, включая `.yaml`, `.toml`, `.rb` и т. д.

Теперь необходимо импортировать этот файл в ConfigMap. Для этого создадим новый объект с помощью флага `--from-file`, который можно комбинировать с флагом `--from-literal` в одной команде:

```
$ kubectl create cm my-sixth-configmap \
    --from-literal=color=yellow \
    --from-file=$HOME/configfile.txt
configmap/my-sixth-configmap created
```

Выполнив команду `kubectl get cm`, мы можем убедиться, что объект `my-sixth-configmap` успешно создан и содержит два значения:

```
$ kubectl get cm my-sixth-configmap
NAME                 DATA   AGE
my-sixth-configmap   2      38s
```

Таким образом, объект `my-sixth-configmap` включает в себя два элемента данных: литеральное значение и файл.

Теперь создадим еще один объект ConfigMap, содержащий литерал и файл, но на этот раз декларативным способом.

Формат YAML позволяет записывать многострочные значения с помощью символа `|`. Мы воспользуемся этой возможностью при создании декларативного файла:

```
# my-seventh-configmap.yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: my-seventh-configmap
data:
  color: "green"
  configfile.txt: |
    I'm another configuration file.
```

После создания файла применим его с помощью команды `kubectl apply -f`:

```
$ kubectl apply -f my-seventh-configmap.yaml
configmap/my-seventh-configmap created
```

Теперь снова перечислим объекты ConfigMap с помощью команды `kubectl get cm`, чтобы убедиться в том, что объект `my-seventh-configmap` создан и содержит два значения:

```
$ kubectl get configmap/my-seventh-configmap
NAME                  DATA   AGE
my-seventh-configmap   2      36s
```

В следующем разделе мы рассмотрим способ создания ConfigMap на основе ENV-файла.

Создание объекта ConfigMap на основе env-файла

Объект ConfigMap также можно создать императивным способом на основе ENV-файла, используя флаг `--from-env-file`.

Файл `.env` содержит пары «ключ — значение», каждая из которых располагается на отдельной строке. Такой формат конфигурации используется рядом приложений, поэтому в Kubernetes предусмотрен способ создания объекта ConfigMap из существующего ENV-файла. Это особенно удобно, если у вас уже есть работающее приложение, которое вы хотите перенести в Kubernetes.

Вот как выглядит типичный ENV-файл:

```
# my-env-file.env
hello=world
color=blue
release=1.0
production=true
```

Как правило, такие файлы называют `.env`, но это не обязательно. Если файл имеет корректный формат, система Kubernetes сможет создать объект ConfigMap на основе его содержимого.

Импортировать конфигурацию из ENV-файла в кластер Kubernetes можно с помощью следующей команды:

```
$ kubectl create cm my-eight-configmap --from-env-file=my-env-file.env
configmap/my-eight-configmap created
```

Выведем список объектов ConfigMap в кластере, чтобы убедиться в том, что новый объект был успешно создан и содержит четыре конфигурационных значения:

```
$ kubectl get cm my-eight-configmap
NAME                DATA   AGE
my-eight-configmap   4      7s
```

Как видите, в кластере появился новый объект ConfigMap, содержащий четыре параметра, которые были указаны в ENV-файле. Теперь вы знаете, как можно импортировать содержимое ENV-файлов в объекты Kubernetes ConfigMap.

Помните, что объекты ConfigMap не предназначены для хранения конфиденциальных данных. Информация в них не закодирована, поэтому ее можно просмотреть с помощью команды kubectl describe cm. Для хранения конфиденциальных данных следует использовать объект Secret. Хотя объекты Secret по умолчанию не шифруются, они хранятся в формате base64, что обеспечивает базовый уровень обфускации. Начиная с версии Kubernetes 1.27, для шифрования содержимого объектов Secret и ConfigMap можно использовать плагины KMS (Key Management System — система управления ключами), обеспечивающие дополнительную защиту конфиденциальной информации.

В следующем подразделе мы обсудим способ чтения содержимого объекта ConfigMap.

Чтение значений, содержащихся в объекте ConfigMap

До сих пор мы выводили список объектов ConfigMap, просто чтобы увидеть количество содержащихся в них ключей. Однако, помимо перечисления таких объектов, мы также можем просматривать хранящиеся в них данные. Это бывает полезно при отладке и в тех случаях, когда вы не уверены в их содержимом.

Поскольку данные в ConfigMap не являются конфиденциальными, их можно легко прочитать с помощью утилиты `kubectl`, которая выводит их прямо в терминал.

Для этого используется команда `kubectl describe`. Мы применим ее к объекту `my-fourth-configmap`, содержащему наибольшее количество данных. Вывод команды получается довольно объемным, но при этом все конфигурационные значения отображаются в удобочитаемом формате:

```
$ kubectl describe cm my-fourth-configmap
Name:         my-fourth-configmap
Namespace:    default
Labels:       <none>
Annotations:  <none>

Data
====
color:
----
```

```
blue
environment:
----
prod
version:
----
1

BinaryData
====

Events:      <none>
```

Команда `kubectl describe cm` возвращает результаты именно в таком виде, а не в ориентированном на компьютер формате вроде JSON или YAML.

Поскольку данные отображаются в терминале явно, важно помнить, что любой пользователь кластера Kubernetes, имеющий доступ к этому объекту ConfigMap, сможет просмотреть его содержимое, просто выполнив команду `kubectl describe cm`. Поэтому не храните в нем конфиденциальные данные.

Теперь посмотрим, как можно внедрить данные из ConfigMap в работающие поды в виде переменных окружения.

Подключение данных ConfigMap в виде переменных окружения

В этом разделе мы «оживим» наши объекты ConfigMap, связав их с подами. Сначала внедрим содержащиеся в них значения в контейнер в виде переменных окружения.

Это можно сделать двумя способами.

- *Внедрение одного или нескольких значений из ConfigMap*. В этом случае переменные окружения задаются на основе конкретных параметров объекта ConfigMap.
- *Внедрение всех значений из ConfigMap*. В этом случае в окружение контейнера внедряются все значения из ConfigMap. Такой способ особенно удобен, если вы создаете один такой объект для каждой спецификации пода или приложения.

Учтите, что связать объект ConfigMap с подом императивным способом невозможно, поскольку команда kubectl run не позволяет создать под, ссылающийся на ConfigMap. Использование таких объектов в подах требует написания декларативных YAML-файлов.

Ранее в этой главе мы создали объект `my-third-configmap`, содержащий параметр `color` со значением `blue`. В следующем примере мы создадим под с образом `quay.io/iamgini/my-flask-app:1.0` и свяжем с ним объект `my-third-configmap` так, чтобы контейнер Flask-приложения получил переменную окружения `COLOR` со значением `blue`. Вот YAML-манифест для решения этой задачи. Обратите внимание на ключ `env:` в спецификации контейнера:

```
# flask-pod-with-configmap.yaml
apiVersion: v1
kind: Pod
metadata:
  name: flask-pod-with-configmap
  labels:
    app: my-flask-app
spec:
  containers:
    - name: flask-with-configmap
      image: quay.io/iamgini/my-flask-app:1.0
      env:
        - name: COLOR   # Здесь можно указать любое другое имя.
          valueFrom:
            configMapKeyRef:
              name: my-third-configmap
              key: color
```

Теперь создадим под с помощью команды `kubectl apply -f`:

```
$ kubectl apply -f flask-pod-with-configmap.yaml
pod/flask-pod-with-configmap created
$ kubectl get pod
NAME                       READY   STATUS    RESTARTS   AGE
flask-pod-with-configmap   1/1     Running   0          5s
```

После создания пода приложения мы можем выполнить команду `env` внутри контейнера, чтобы вывести список всех доступных переменных окружения. Для этого мы воспользуемся командой `kubectl exec`. Ее ожидаемый вывод представлен ниже:

```
$ kubectl exec pod/flask-pod-with-configmap -- env
PATH=/usr/local/bin:/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/
bin
HOSTNAME=flask-pod-with-configmap
COLOR=blue
KUBERNETES_PORT_443_TCP=tcp://10.96.0.1:443
... <фрагмент опущен для краткости>...
```

Если объект ConfigMap был корректно связан с подом, в выводе появится переменная окружения `COLOR`.

Вы также можете проверить развернутое приложение, опубликовав под через сервис:

```
$ kubectl expose pod flask-pod-with-configmap \
    --port=8081 \
    --target-port=5000 \
    --type=NodePort
```

А затем протестировать его в браузере, выполнив проброс портов:

```
$ kubectl port-forward flask-pod-with-configmap 20000:5000
Forwarding from 127.0.0.1:20000 -> 5000
Forwarding from [::1]:20000 -> 5000
```

Использование команды kubectl port-forward

Команда `kubectl port-forward` создает защищенный туннель между вашей локальной машиной и конкретным подом внутри кластера Kubernetes. Это позволяет получать доступ к приложениям, работающим в поде, и взаимодействовать с ними так, как будто они запущены локально. Такой механизм особенно полезен для отладки, тестирования и разработки, так как предоставляет прямой доступ к сервисам без их публикации вовне.

В предыдущем примере мы обеспечили временное перенаправление трафика с локального порта `20000` на целевой порт `5000`. Теперь приложение доступно в браузере по адресу `127.0.0.1:20000`.

Хотя сервисы типа NodePort предназначены для предоставления внешнего доступа, их также можно использовать в средах minikube. Для получения доступа к приложению извне используется команда minikube service, возвращающая NodePort и соответствующий IP-адрес. Например, команда minikube service --url flask-pod-with-configmap может вывести результат вида http://192.168.49.2:31997, что позволит получить доступ к Flask-приложению по указанному URL-адресу.

Теперь если вы откроете Flask-приложение в браузере, то увидите синий фон, соответствующий параметру `COLOR=blue`, заданному в объекте ConfigMap. Если изменить значение переменной `COLOR` в ConfigMap и пересоздать под `flask-pod-with-configmap`, то эти изменения отразятся в интерфейсе приложения.

Мы подробно обсудим сервисы Kubernetes и работу DNS в следующей главе.

Учтите, что команда `kubectl port-forward` будет осуществлять переадресацию портов до тех пор, пока вы не завершите этот процесс вручную (например, нажав Ctrl+C).

Далее рассмотрим второй способ внедрения содержимого ConfigMap в виде переменных окружения.

В этом примере мы используем объект `my-fourth-configmap`. На этот раз нам нужно внедрить не одно, а сразу все значения, содержащиеся в ConfigMap. Ниже

приведен обновленный YAML-манифест с описанием пода. Теперь в спецификации контейнера вместо отдельных ключей `env` используется ключ `envFrom`:

```
# flask-pod-with-configmap-all.yaml
apiVersion: v1
kind: Pod
metadata:
  name: flask-pod-with-configmap-all
spec:
  containers:
    - name: flask-with-configmap
      image: quay.io/iamgini/my-flask-app:1.0
      envFrom:
        - configMapRef:
            name: my-fourth-configmap
```

После подготовки манифеста создадим под для Flask-приложения:

```
$ kubectl apply -f flask-pod-with-configmap-all.yaml
pod/flask-pod-with-configmap-all created
```

Теперь снова выполним команду `env` внутри Flask-контейнера с помощью `kubectl exec`, чтобы отобразить все переменные окружения:

```
$ kubectl exec pod/flask-pod-with-configmap-all -- env
PATH=/usr/local/bin:/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
HOSTNAME=flask-pod-with-configmap-all
environment=prod
version=1
color=blue
KUBERNETES_PORT_443_TCP_PORT=443
KUBERNETES_PORT_443_TCP_ADDR=10.96.0.1
... <фрагмент опущен для краткости>...
```

Как видите, три параметра (`color`, `version` и `environment`), объявленные в объекте `my-fourth-configmap`, были установлены в контейнере в качестве переменных окружения. Только теперь вы не контролируете их имена, поскольку они наследуются от названий ключей в ConfigMap.

Далее мы обсудим способ монтирования объекта ConfigMap в контейнер в виде тома.

Монтирование объекта ConfigMap в виде тома

Ранее в этой главе мы создали два объекта ConfigMap с фиктивными конфигурационными файлами. Утилита `kubectl` позволяет смонтировать ConfigMap в под в качестве тома. Это особенно полезно, если вам необходимо внедрить в файловую систему контейнера содержимое файла, хранящегося в ConfigMap.

Как и в случае с внедрением переменных окружения, сделать это можно только декларативным способом — через YAML-манифест. В следующем примере мы

смонтируем объект `my-sixth-configmap` в виде тома в новый под `flask-pod-with-configmap-volume`:

```
# flask-pod-with-configmap-volume.yaml
apiVersion: v1
kind: Pod
metadata:
  name: flask-pod-with-configmap-volume
spec:
  containers:
    - name: flask-with-configmap-volume
      image: quay.io/iamgini/my-flask-app:1.0
      volumeMounts:
        - name: configuration-volume   # должно совпадать с именем тома
          mountPath: /etc/conf
  volumes:
    - name: configuration-volume
      configMap:
        name: my-sixth-configmap       # здесь указывается имя объекта ConfigMap
```

В этом манифесте мы объявили том `configuration-volume` на уровне контейнеров и сообщили системе Kubernetes о том, что он создается на основе ConfigMap. Указанный объект (`my-sixth-configmap`) должен присутствовать в кластере в момент применения манифеста. Далее, на уровне контейнера, мы смонтировали объявленный том в каталог `/etc/conf`. Параметры из ConfigMap будут доступны по этому пути.

Применим этот манифест, чтобы создать под с объектом ConfigMap, подключенным в виде тома:

```
$ kubectl apply -f flask-pod-with-configmap-volume.yaml
pod/flask-pod-with-configmap-volume created
```

Теперь выполним внутри контейнера команду `ls`, чтобы проверить результат монтирования каталога:

```
$ kubectl exec pod/flask-pod-with-configmap-volume -- ls /etc/conf
color
configfile.txt
```

Как видите, каталог был успешно смонтирован и оба параметра, созданные в ConfigMap, доступны в нем в виде обычных файлов.

Теперь выполним команду `cat`, чтобы удостовериться в том, что оба файла содержат правильные значения:

```
$ kubectl exec pod/flask-pod-with-configmap-volume -- cat /etc/conf/color
yellow
$ kubectl exec pod/flask-pod-with-configmap-volume -- cat /etc/conf/configfile.txt
I'm just a dummy config file
```

Отлично! Файлы действительно содержат значения, которые мы задали при создании объекта ConfigMap. Итак, теперь вам доступен мощный и единообразный интерфейс для управления и настройки контейнеров, работающих в Kubernetes. С его помощью вы можете, например, сохранить конфигурационный файл виртуального хоста Nginx в объекте ConfigMap и смонтировать его в нужный каталог, чтобы сервер Nginx обслуживал сайт на основе этой конфигурации. Такой способ позволяет переопределять настройки по умолчанию и аккуратно управлять приложением в системе Kubernetes.

Далее вы научитесь удалять и обновлять объекты ConfigMap.

Удаление объекта ConfigMap

Удалить объект ConfigMap очень просто. Однако учтите, что это можно сделать даже в том случае, если значения этого объекта используются контейнером. После запуска под перестает зависеть от объекта ConfigMap:

```
$ kubectl delete cm my-first-configmap
configmap "my-first-configmap" deleted
```

Независимо от того, используется ли содержимое ConfigMap контейнером или нет, этот объект будет удален сразу после выполнения этой команды. Его невозможно будет восстановить, поэтому подумайте дважды, прежде чем удалять объект ConfigMap, созданный императивным способом. Вы не сможете пересоздать его, поскольку его содержимое не хранится в YAML-файле в отличие от содержимого декларативно созданных объектов. Однако, как уже было показано, мы можем выгрузить содержимое такого ресурса в формате YAML с помощью команды `kubectl get`, например: `kubectl get cm <имя_configmap> -o YAML > my-first-configmap.yaml`.

Мы также рекомендуем проявлять осторожность при удалении объектов ConfigMap, используемых работающими подами. Если под аварийно завершит работу, его нельзя будет перезапустить без обновления манифеста, так как под будет искать удаленный объект ConfigMap.

Проверим этот сценарий на примере объекта `my-sixth-configmap`. Для начала удалим его:

```
$ kubectl delete cm my-sixth-configmap
configmap "my-sixth-configmap" deleted
```

Теперь при попытке пересоздать под `flask-pod-with-configmap-volume` мы столкнемся с проблемой:

```
$ kubectl apply -f flask-pod-with-configmap-volume.yaml
pod/flask-pod-with-configmap-volume created
```

```
$ kubectl get pod
NAME                              READY   STATUS              RESTARTS   AGE
flask-pod-with-configmap-volume   0/1     ContainerCreating   0          61s
```

Как видите, под находится в состоянии `ContainerCreating`, а не `Running`. Выведем подробные сведения об этом поде:

```
$ kubectl describe pod flask-pod-with-configmap-volume
Name:             flask-pod-with-configmap-volume
Namespace:        default
... <фрагмент опущен для краткости>...

Events:
  Type     Reason       Age                  From               Message
  ----     ------       ----                 ----               -------
  Normal   Scheduled    71s                  default-scheduler  Successfully
assigned default/flask-pod-with-configmap-volume to minikube-m03
  Warning  FailedMount  7s (x8 over 71s)     kubelet            MountVolume.
SetUp failed for volume "configuration-volume" : configmap "my-sixth-configmap"
not found
```

Ошибка в выводе прямо указывает на отсутствие нужного объекта ConfigMap и невозможность монтирования тома. Именно поэтому под остается в состоянии `ContainerCreating`.

Обновление объекта ConfigMap

Система Kubernetes предусматривает два основных способа обновления объекта ConfigMap. Первый оптимален для контроля версий и совместной работы. В рамках этого подхода достаточно внести необходимые изменения в YAML-файл с описанием объекта ConfigMap и применить их с помощью команды `kubectl apply`.

Альтернативный способ сводится к непосредственному редактированию существующего объекта ConfigMap с помощью команды `kubectl edit`. Этот вариант предоставляет интерактивный способ изменения содержимого ConfigMap, однако при его использовании следует проявлять осторожность, так как он не предусматривает контроля версий.

Неизменяемые объекты ConfigMap

Система Kubernetes позволяет сделать объект ConfigMap неизменяемым, чтобы предотвратить случайное или преднамеренное изменение его содержимого.

Эта возможность особенно полезна в кластерах, в которых подобные объекты используются очень активно, так как защищает от ошибок конфигурации, способных нарушить работу приложений. Кроме того, она повышает производительность кластера за счет снижения нагрузки на компонент `kube-apiserver`.

Чтобы сделать объект ConfigMap неизменяемым, достаточно задать значение `true` для поля `immutable` в его определении:

```
# immutable-configmap.yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: immutable-configmap
data:
  color: "blue"
  version: "1.0"
  environment: "prod"
immutable: true
```

После этого изменить его данные будет невозможно. Для обновления ConfigMap вам придется удалить существующий объект и создать новый. Чтобы сохранить корректную конфигурацию, вам также потребуется пересоздать все поды, ссылающиеся на удаленный объект.

В следующем разделе мы поговорим об объекте Secret.

Управление конфиденциальной конфигурацией с помощью объекта Secret

Secret (секрет) — это ресурс, который позволяет настраивать приложения, работающие в Kubernetes. Секреты очень похожи на объекты ConfigMap и могут использоваться совместно с ними. Главное отличие заключается в том, что объекты Secret кодируются и предназначены для хранения конфиденциальной информации — паролей, токенов, секретных ключей API, а объекты ConfigMap предназначены для хранения неконфиденциальных данных. В остальном эти объекты ведут себя практически одинаково.

Чтобы обеспечить защиту чувствительных данных, хранящихся в объектах Secret, рекомендуется придерживаться следующих практик.

- *Ограничение доступа*. Используйте механизм *управления доступом на основе ролей* (RBAC), предоставляя пользователям и сервисам только необходимые привилегии.
- *Избегайте жесткого кодирования*. Никогда не встраивайте секреты напрямую в код приложения или конфигурационные файлы.
- *Регулярная ротация*. Обеспечьте регулярную ротацию секретов, чтобы снизить риск несанкционированного доступа.
- *Использование внешних систем управления секретами*. При повышенных требованиях к безопасности рассмотрите возможность использования та-

ких специализированных решений, как HashiCorp Vault или AWS Secrets Manager.

- *Шифрование*. Применяйте плагины KMS для шифрования содержимого объектов Secret в состоянии покоя, чтобы обеспечить его дополнительную защиту.
- *Мониторинг и аудит*. Регулярно проверяйте журналы доступа и аудита для выявления подозрительной активности и потенциальных нарушений безопасности.
- *Обучение команды*. Формируйте культуру безопасного обращения с секретами, проводя обучение лучшим практикам их хранения и управления.

Хотя секреты используются для хранения конфиденциальной информации и их данные преобразуются в формат base64, этого недостаточно для обеспечения надежной защиты. Кодирование в base64 является обратимой операцией, а значит, исходные данные можно легко восстановить. Для более надежной защиты конфиденциальной информации следует реализовать дополнительные меры безопасности, например шифрование с помощью плагинов KMS или использование внешних систем управления секретами.

Помимо объектов Secret, система Kubernetes предусматривает дополнительные средства защиты конфиденциальных данных. С помощью плагинов *KMS* (Key Management System — система управления ключами) теперь можно шифровать содержимое объектов Secret и ConfigMap. Это обеспечивает дополнительный уровень защиты чувствительных данных, хранящихся в объектах ConfigMap.

Шифрование содержимого ConfigMap с помощью KMS-плагинов позволяет защитить чувствительные конфигурационные данные, не прибегая к использованию объектов Secret. Такой подход упрощает управление конфигурацией при сохранении высокого уровня безопасности.

Тем не менее даже при использовании KMS-шифрования следует тщательно оценивать уровень конфиденциальности содержимого ConfigMap. Для хранения действительно секретной информации рекомендуется использовать объект Secret.

Подробнее о KMS-плагинах для шифрования данных можно узнать в официальной документации: https://kubernetes.io/docs/tasks/administer-cluster/kms-provider/.

Теперь посмотрим, как можно вывести список объектов Secret, присутствующих в кластере Kubernetes.

Перечисление объектов Secret

Как и любые другие объекты Kubernetes, секреты можно перечислить с помощью команды `kubectl get`. В данном случае идентификатором ресурса будет `secret`:

```
$ kubectl get secret -A
```

В столбце `DATA` содержится количество конфиденциальных параметров, которые были хешированы и сохранены в объекте Secret. При выполнении команды утилита `kubectl` выводит список предустановленных объектов Secret или сообщение о том, что они не найдены. Эта разница объясняется тем, что одни облачные сервисы создают объекты Secret по умолчанию для внутренних процессов, а другие — нет. Наличие или отсутствие таких предустановленных объектов зависит от среды, в которой развернут кластер Kubernetes.

Создание объекта Secret императивным способом

Система Kubernetes позволяет создавать объекты Secret как императивным, так и декларативным способом. Начнем с рассмотрения императивного подхода. В этом примере мы сохраним пароль от базы данных `my-db-password` в объекте Secret в нашем кластере. Это можно сделать с помощью команды `kubectl create secret` с флагом `--from-literal`:

```
$ kubectl create secret generic my-first-secret \
    --from-literal='db_password=my-db-password'
```

Теперь выведем список секретов в кластере, выполнив команду `kubectl get secrets`. В выводе должен отобразиться новый объект Secret:

```
$ kubectl get secrets
NAME              TYPE     DATA   AGE
my-first-secret   Opaque   1      37s
```

Выведем его содержимое в формате YAML:

```
$ kubectl get secret my-first-secret -o yaml
apiVersion: v1
data:
  db_password: bXktZGItcGFzc3dvcmQ=
kind: Secret
metadata:
  creationTimestamp: "2024-08-10T13:13:32Z"
  name: my-first-secret
  namespace: default
  resourceVersion: "36719"
  uid: 7ccf5120-d1c5-4874-ba4b-894274fd27e6
type: Opaque
```

В выводе будет присутствовать строка `bXktZGItcGFzc3dvcmQ=`, содержащая пароль в закодированном виде.

Далее мы обсудим декларативный способ создания объекта Secret.

Создание объекта Secret декларативным способом

Секреты также можно создавать декларативно с помощью YAML-файлов. Хотя значения для поля `data` можно вручную закодировать в формат base64, система Kubernetes предусматривает более удобный способ. Поле `stringData` позволяет указать секретные значения в виде обычных текстовых строк, которые Kubernetes автоматически преобразует в формат base64 при создании объекта Secret. Этот метод упрощает рабочий процесс и предотвращает случайное раскрытие конфиденциальных данных, содержащихся в конфигурационных файлах с открытым текстом.

Преобразование данных в формат base64 обеспечивает лишь базовую обфускацию и не является надежным методом шифрования. Для обеспечения дополнительной защиты рекомендуется использовать плагины KMS или внешние системы управления секретами.

При использовании флага `--from-literal` система Kubernetes сама преобразует строки в формат base64. Однако при создании секрета на основе YAML-манифеста этот шаг придется выполнить вам.

Для начала преобразуем строку `my-db-password` в формат base64:

```
$ echo -n 'my-db-password' | base64
bXktZGItcGFzc3dvcmQ=
```

Поскольку `bXktZGItcGFzc3dvcmQ=` — это представление строки `my-db-password` в кодировке base64, именно это значение следует записать в YAML-файл. Вот как выглядит содержимое файла, предназначенного для создания объекта Secret:

```
# secret-from-file.yaml
apiVersion: v1
kind: Secret
metadata:
  name: my-second-secret
type: Opaque
data:
    db_password: bXktZGItcGFzc3dvcmQ=
```

После сохранения этого файла в системе мы можем создать секрет с помощью команды `kubectl apply -f`:

```
$ kubectl apply -f secret-from-file.yaml
```

Чтобы убедиться в успешном создании секрета, выведите список соответствующих объектов в кластере:

```
$ kubectl get secret my-second-secret -o yaml
apiVersion: v1
kind: Secret
metadata:
  name: my-second-secret
  namespace: default
... <фрагмент опущен для краткости>...
data:
  db_password: bXktZGItcGFzc3dvcmQ=
type: Opaque
```

Теперь создадим тот же секрет, но с использованием поля `stringData`, чтобы не кодировать значение вручную. Новый YAML-файл будет выглядеть так:

```
# secret-from-file-stringData.yaml
apiVersion: v1
kind: Secret
metadata:
  name: my-secret-stringdata
type: Opaque
stringData:
  db_password: my-db-password
```

Обратите внимание на то, что пароль записан в виде открытого текста, а не закодированного значения.

Создадим объект Secret на основе этого файла:

```
$ kubectl apply -f secret-from-file-stringData.yaml
secret/my-secret-stringdata created
```

Выведите содержимое объекта Secret, чтобы просмотреть закодированные данные:

```
$ kubectl get secret my-secret-stringdata -o yaml
apiVersion: v1
kind: Secret
metadata:
  name: my-secret-stringdata
  namespace: default
  resourceVersion: "37229"
  uid: 5078124b-7318-44df-a2da-a30ea5088e3f
... <фрагмент опущен для краткости>...
data:
  db_password: bXktZGItcGFzc3dvcmQ=
type: Opaque
```

Как видите, значение, указанное в поле `stringData` в виде открытого текста, было автоматически закодировано и сохранено в разделе `data`.

В следующем подразделе вы научитесь создавать объекты Secret на основе значений из файла.

Создание объекта Secret на основе содержимого файла

Как и в случае с ConfigMap, объект Secret можно создать на основе хранящихся в файле значений. Начнем с создания файла, предназначенного для хранения пароля, который нам предстоит импортировать в систему Kubernetes в качестве секрета:

```
$ echo -n 'mypassword' > ./password.txt
```

После выполнения этой команды у нас появится файл `password.txt`, содержащий строку `mypassword`, которая и будет секретным значением. Флаг `-n` предотвращает появление в конце файла лишней пустой строки.

Далее выполним команду `kubectl create secret`, передав путь к файлу `password.txt` через флаг `--from-file`. В результате будет создан новый объект Secret со строкой `mypassword`, преобразованной в формат base64:

```
$ kubectl create secret generic mypassword \
    --from-file=./password.txt
secret/mypassword created
```

Теперь этот новый объект доступен в вашем кластере Kubernetes! Далее мы обсудим способ чтения данных, содержащихся в объекте Secret.

Чтение значений, содержащихся в объекте Secret

Как мы уже отметили, объекты Secret предназначены для хранения конфиденциальных данных, и поэтому вывод утилиты `kubectl` не отображает их в расшифрованном виде. Чтобы просмотреть секретное содержимое, вам необходимо декодировать его самостоятельно. Такая конфиденциальность сохраняется для того, чтобы:

- предотвратить случайное раскрытие секрета тем, кто не имеет соответствующих прав доступа;
- исключить отображение секрета в выводе терминала, который может быть записан в лог.

Хотя преобразование данных в формат base64 обеспечивает базовую обфускацию, эта кодировка не является надежным методом шифрования. Любой пользователь с доступом к API кластера Kubernetes может получить объект Secret и декодировать его содержимое. Для защиты чувствительной информации необходимо реализовать механизм RBAC, ограничивающий доступ к секретам на основе ролей и прав пользователей. Задание правил предоставления разрешений на просмотр, изменение и удаление секретов повышает общую безопасность кластера.

Таким образом, извлечь содержимое секрета напрямую нельзя, но мы можем получить информацию о размере объекта и других его характеристиках.

Это можно сделать с помощью команды `kubectl describe`, как было показано ранее в примере с объектом ConfigMap. Как уже было сказано, объекты ConfigMap и Secret во многом похожи и ведут себя почти одинаково:

```
$ kubectl describe secret/mypassword
Name:         mypassword
Namespace:    default
Labels:       <none>
Annotations:  <none>

Type:         Opaque

Data
====
password.txt:  10 bytes
```

Не удивляйтесь, если в вашем случае вывод будет немного отличаться от показанного здесь. Если вы получили похожий результат, значит, новый объект Secret доступен в вашем кластере Kubernetes, а его данные были успешно извлечены.

Однако помните, что закодированные данные можно просмотреть в YAML-формате с помощью такой команды:

```
$ kubectl get secret my-second-secret -o yaml
apiVersion: v1
data:
  db_password: bXktZGItcGFzc3dvcmQ=
kind: Secret
metadata:
  creationTimestamp: "2024-02-13T04:15:56Z"
  name: my-second-secret
  namespace: default
  resourceVersion: "90372"
  uid: 94b7b529-baed-4844-8097-f6c2a001fa7b
type: Opaque
```

И любой пользователь, имеющий доступ к данному объекту Secret, может легко декодировать эти данные следующим образом:

```
$ echo 'bXktZGItcGFzc3dvcmQ=' | base64 --decode
my-db-password
```

Важно иметь в виду, что преобразование в формат base64 является не шифрованием, а лишь способом представления двоичных данных в виде ASCII-символов. Этот формат делает данные менее читаемыми, но не защищает их. Любой человек, имеющий соответствующие знания и инструменты, сможет легко восстановить исходный текст. Для надежной защиты секретов используйте кодировку base64 в сочетании с дополнительными средствами обеспечения безопасности, такими как механизм RBAC и алгоритмы шифрования, или задействуйте внешние системы управления секретами.

Теперь, когда вы научились создавать и управлять секретами, посмотрим, как можно сделать эту конфиденциальную информацию доступной для приложений внутри подов.

Использование значения из объекта Secret в виде переменной окружения

Мы уже видели, как значения из объекта ConfigMap можно внедрять в под в виде переменных окружения. То же самое можно делать и с содержимым объекта Secret. Вернемся к примеру с Flask-контейнером и извлечем значение `db_password` из объекта `my-first-secret`, а затем внедрим его в под в качестве переменной окружения. Вот YAML-манифест. Как и прежде, все значения задаются в разделе `env`:

```
# flask-pod-with-secret.yaml
apiVersion: v1
kind: Pod
metadata:
  name: flask-pod-with-secret
  labels:
    app: flask-with-secret
spec:
  containers:
    - name: flask-with-secret
      image: quay.io/iamgini/my-flask-app:1.0
      env:
        - name: PASSWORD_ENV_VAR      # Имя переменной окружения
          valueFrom:
            secretKeyRef:
              name: my-first-secret   # Имя объекта Secret
              key: db_password        # Имя ключа в объекте Secret
```

Создадим под с помощью команды `kubectl apply -f`:

```
$ kubectl apply -f flask-pod-with-secret.yaml
```

Теперь выполним команду `env` внутри контейнера, чтобы вывести список переменных окружения:

```
$ kubectl exec pod/flask-pod-with-secret -- env
PATH=/usr/local/bin:/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
HOSTNAME=flask-pod-with-secret
PASSWORD_ENV_VAR=my-db-password
KUBERNETES_SERVICE_PORT=443
... <фрагмент опущен для краткости>...
```

Как видите, строка `my-db-password` доступна в качестве переменной окружения `PASSWORD_ENV_VAR`.

Альтернативный способ получения значений из объекта Secret сводится к использованию ключа `envFrom` в YAML-манифесте. В этом случае в под будут внедрены все значения из объекта Secret в качестве переменных окружения. Этот механизм работает так же, как и в случае с ConfigMap.

Чтобы под успешно запустился, объект Secret, на который он ссылается, должен существовать, если только в определении пода явно не указано, что он является необязательным. Убедитесь в существовании объекта Secret перед развертыванием подов, которые от него зависят.

Сначала создадим объект Secret с помощью YAML-файла:

```
# secret-from-file-database.yaml
apiVersion: v1
kind: Secret
metadata:
  name: appdb-secret
type: Opaque
stringData:
  db_user: appadmin
  db_password: appdbpassword
```

Обратите внимание на пароль в открытом виде, а не в закодированном.

Создайте Secret из файла следующим образом:

```
$ kubectl apply -f secret-from-file-database.yaml
secret/appdb-secret created
```

Теперь обновим YAML-манифест с описанием пода, добавив ключ `envFrom`:

```
# flask-pod-with-secret-all.yaml
apiVersion: v1
kind: Pod
metadata:
  name: flask-pod-with-secret-all
  labels:
    app: flask-with-secret
spec:
  containers:
    - name: flask-with-secret
      image: quay.io/iamgini/my-flask-app:1.0
      envFrom:
        - secretRef:
            name: appdb-secret   # Имя объекта Secret
```

Создадим под с помощью команды `kubectl apply -f`:

```
$ kubectl apply -f flask-pod-with-secret-all.yaml
pod/flask-pod-with-secret-all created
```

При таком подходе все ключи из объекта Secret будут доступны внутри пода в качестве переменных окружения. Проверим корректность загрузки этих переменных:

```
$ kubectl exec pod/flask-pod-with-secret-all -- env | grep -i app
db_user=appadmin
db_password=appdbpassword
```

При невозможности использования имени ключа в объекте Secret в качестве имени переменной окружения оно будет просто проигнорировано.

Теперь обсудим способ монтирования объекта Secret в виде тома.

Монтирование объекта Secret в виде тома

Объекты Secret можно монтировать в поды в виде томов, но сделать это можно только декларативным способом, то есть через YAML-манифест.

Начнем с написания YAML-файла, предназначенного для создания пода. В данном примере объект `mypassword` монтируется в каталог `/etc/passwords-mounted-path`:

```
# flask-pod-with-secret-volume.yaml
apiVersion: v1
kind: Pod
```

```
metadata:
  name: flask-pod-with-secret-volume
  labels:
    app: flask-with-secret-volume
spec:
  containers:
    - name: flask-with-secret
      image: quay.io/iamgini/my-flask-app:1.0
      volumeMounts:
        - name: mysecret-volume
          mountPath: /etc/password-mounted-path
          readOnly: true   # Значение true параметра readOnly запрещает
                           # запись в объект Secret
  volumes:
    - name: mysecret-volume
      secret:
        secretName: my-second-secret   # Здесь указывается имя объекта Secret
```

После сохранения файла примените его с помощью команды:

```
$ kubectl apply -f flask-pod-with-secret-volume.yaml
```

Убедитесь в существовании объекта `my-second-secret` перед созданием пода, который на него ссылается.

Наконец, выполните команду `cat` внутри контейнера `flask-with-secret` через `kubectl exec`, чтобы проверить корректность монтирования тома с секретом:

```
$ kubectl exec pod/flask-pod-with-secret-volume -- cat /etc/password-mounted-
path/db_password
my-db-password
```

Корректное отображение строки `my-db-password` говорит о том, что объект Secret был успешно смонтирован в виде тома.

Теперь, когда вы научились создавать объекты Secret и использовать их в подах различными способами, рассмотрим, как их удалять и обновлять.

Удаление объекта Secret

Для удаления объекта Secret достаточно выполнить команду `kubectl delete`:

```
$ kubectl delete secret my-first-secret
secret "my-first-secret" deleted
```

В следующем разделе вы узнаете, как обновить секрет, существующий в кластере Kubernetes.

Обновление объекта Secret

Для обновления объекта Secret в Kubernetes можно использовать команду `kubectl apply` с измененным YAML-описанием или команду `kubectl edit`.

Объекты Secret обеспечивают безопасный способ хранения и управления такой конфиденциальной информацией, как пароли, ключи API и сертификаты. В отличие от объектов ConfigMap объекты Secret предусматривают кодирование данных для защиты их конфиденциальности. В этом разделе мы рассмотрели императивный и декларативный способы создания объектов Secret. Мы также выяснили, как их можно внедрять в поды в виде переменных окружения и монтируемых томов, предоставляя приложениям доступ к конфиденциальным данным без их отображения в открытом виде.

Резюме

В этой главе мы рассмотрели фундаментальные концепции, связанные с управлением конфигурацией в Kubernetes. Мы обсудили ключевое различие между объектами ConfigMap и Secret, а также их роль в работе с публичными и конфиденциальными данными. Эффективное использование этих ресурсов Kubernetes позволяет существенно повысить переносимость и безопасность приложений.

Мы рассмотрели императивный и декларативный подходы к созданию и управлению объектами ConfigMap и Secret. А также обсудили способ внедрения конфигурационных данных в поды в виде переменных окружения и монтируемых томов, который обеспечивает беспрепятственный доступ приложений к необходимым настройкам.

Кроме того, мы подчеркнули необходимость применения надежных средств защиты данных в дополнение к их преобразованию в формат base64. Используя механизмы RBAC, шифрование и внешние системы управления секретами, вы сможете значительно повысить безопасность среды Kubernetes.

Освоение изложенных в этой главе концепций позволит вам создавать устойчивые и безопасные приложения с отделенной от кода конфигурацией. Такой подход обеспечивает гибкость и упрощает сопровождение ПО.

В следующей главе мы продолжим знакомство с Kubernetes и рассмотрим еще одну ключевую концепцию — сервисы. Объекты Service позволяют сделать поды доступными не только друг для друга, но и для внешней сети. Это важнейшее сетевое понятие в Kubernetes, и его освоение необходимо для правильного использования этого оркестратора. К счастью, разобраться с сервисами несложно.

В следующей главе вы узнаете, как сопоставлять порты контейнера с портами рабочего узла, на котором он выполняется, и как назначать подам статический IP-адрес, чтобы они всегда были доступны по одному и тому же адресу для других подов в кластере.

Дополнительные источники информации

- Объекты ConfigMap: https://kubernetes.io/docs/concepts/configuration/configmap/.
- Настройка пода для использования объекта ConfigMap: https://kubernetes.io/docs/tasks/configure-pod-container/configure-pod-configmap/.
- Управление объектами Secret с помощью утилиты `kubectl`: https://kubernetes.io/docs/tasks/configmap-secret/managing-secret-using-kubectl/.
- Управление объектами Secret с помощью конфигурационного файла: https://kubernetes.io/docs/tasks/configmap-secret/managing-secret-using-config-file/.
- Использование KMS-плагинов для шифрования данных: https://kubernetes.io/docs/tasks/administer-cluster/kms-provider/.
- Книга, посвященная безопасности объекта Secret, вопросам соответствия нормативным требованиям и DevOps-практикам: *Kubernetes Secrets Handbook: Design, implement and maintain production-grade Kubernetes Secrets management solutions*, https://www.amazon.com/Kubernetes-Secrets-Handbook-production-grade-management/dp/180512322X/.

8 Публикация подов с помощью сервисов

В предыдущих главах вы научились развертывать приложения в Kubernetes, создавая поды, содержащие один или несколько контейнеров. Вы узнали о том, что приложение можно отделить от его конфигурации с помощью объектов *ConfigMap*, а также о том, что Kubernetes позволяет хранить конфиденциальные данные в объектах *Secret*.

Этих трех ресурсов вам уже достаточно для корректного развертывания и запуска своего первого приложения. Однако вам также необходимо научиться делать поды доступными для конечных пользователей или других подов внутри кластера Kubernetes. В этом вам могут помочь *сервисы*, о которых мы и поговорим далее.

В этой главе вы познакомитесь с новым типом ресурсов Kubernetes — объектом Service. Поскольку тема сервисов достаточно объемна и включает множество аспектов, эта глава будет довольно насыщенной. Однако, усвоив содержащийся в ней материал, вы сможете публиковать свои поды и предоставлять пользователям доступ к своим приложениям.

Кроме того, сервисы крайне важны для обеспечения высокой доступности и отказоустойчивости среды Kubernetes. Короче говоря, чтобы эффективно использовать этот оркестратор, необходимо научиться с ними работать.

Технические требования

Для работы с этой главой вам потребуется:

- локальный или облачный кластер Kubernetes;
- рабочий *интерфейс командной строки* `kubectl`, настроенный для взаимодействия с кластером.

Примеры кода для этой главы можно загрузить из официального репозитория GitHub по адресу https://github.com/PacktPublishing/The-Kubernetes-Bible-Second-Edition/tree/main/Chapter08.

Причины для публикации подов

В предыдущих главах мы обсуждали микросервисную архитектуру, в рамках которой доступ к функциональности предоставляется через *API* (application programming interfaces — интерфейсы прикладного программирования), реализованные на базе *REST* (Representational State Transfer — передача состояния представления). Данные API опираются на протокол *HTTP* (HyperText Transfer Protocol — протокол передачи гипертекста). Это означает, что микросервисы должны быть доступны через сеть, то есть по *IP*-адресу (Internet Protocol — межсетевой протокол).

Хотя REST API широко применяются для организации взаимодействия в микросервисных архитектурах, важно рассматривать и другие протоколы обмена данными, например gRPC. Это особенно актуально в сценариях, в которых производительность и эффективность взаимодействия между сервисами являются критическими. Протокол gRPC основан на HTTP/2 и использует двоичную сериализацию (Protocol Buffers), что дает значительные преимущества в распределенных системах, такие как более высокая скорость взаимодействия, меньшая задержка и поддержка потоковой передачи данных. Поэтому, прежде чем делать выбор в пользу REST, стоит проверить, не является ли gRPC более подходящим вариантом для вашей системы.

В следующем разделе мы обсудим сетевое взаимодействие в Kubernetes.

Сетевое взаимодействие в Kubernetes

Сетевое взаимодействие является фундаментальным аспектом Kubernetes, обеспечивающим связь между контейнерами, подами, сервисами и внешними клиентами. Понимание того, как Kubernetes управляет этим взаимодействием, помогает обеспечить бесперебойную работу приложений в распределенной среде. Сетевая подсистема Kubernetes решает четыре ключевые задачи.

- *Обмен данными между контейнерами.* Эта задача решается с помощью подов: контейнеры в одном поде могут взаимодействовать через локальный хост (внутренняя коммуникация).
- *Обмен данными между подами.* Система Kubernetes обеспечивает взаимодействие подов на разных узлах благодаря собственной сетевой модели.
- *Обмен данными между подом и сервисом.* Сервисы абстрагируют набор подов и предоставляют стабильные конечные точки для связи.
- *Обмен данными между внешним клиентом и сервисом.* Система Kubernetes предоставляет внешнему трафику доступ к таким сервисам, как веб-приложения или API.

В кластере Kubernetes на одном и том же наборе узлов могут одновременно выполняться несколько приложений. Это создает определенные трудности, например необходимость предотвращать конфликты, возникающие при использовании одних и тех же сетевых портов разными приложениями.

Управление IP-адресами в Kubernetes

Кластеры Kubernetes используют непересекающиеся диапазоны IP-адресов для подов, сервисов и узлов. Сетевая модель реализуется с помощью следующих механизмов.

- *Поды*. Сетевой плагин, обычно реализованный как *CNI* (Container Network Interface — сетевой интерфейс контейнера), назначает IP-адреса подам.
- *Сервисы*. Компонент `kube-apiserver` назначает IP-адреса сервисам.
- *Узлы*. IP-адресами узлов управляет либо агент `kubelet`, либо компонент `cloud-controller-manager`.

Подробнее о сетевом взаимодействии в Kubernetes можно узнать на официальном сайте: https://kubernetes.io/docs/concepts/cluster-administration/networking/.

Прежде чем перейти к обсуждению сетевого взаимодействия подов и сервисов, рассмотрим базовые технологии, используемые сетевой подсистемой Kubernetes.

Сетевые плагины

CNI (Container Network Interface — сетевой интерфейс контейнера) — это спецификация и набор библиотек, разработанные организацией *Cloud Native Computing Foundation (CNCF)*. Данная спецификация стандартизирует конфигурацию сетевых интерфейсов Linux-контейнеров для обеспечения их беспрепятственного взаимодействия друг с другом и с внешней средой.

Эти плагины играют ключевую роль в реализации сетевой модели Kubernetes, гарантируя связность и обмен данными внутри кластера. Важно подобрать CNI-плагин, совместимый с вашим окружением и соответствующий его требованиям. В экосистеме Kubernetes доступно множество плагинов — как открытых, так и проприетарных, и выбор подходящего критически важен для стабильной работы кластера.

Далее приведен список CNI-плагинов с их кратким описанием.

Плагины с открытым исходным кодом

- *Calico*. Обеспечивает сетевое взаимодействие и безопасность, уделяя основное внимание связности на уровне 3 и детализированному управлению политиками.

- *Flannel*. Простой CNI-плагин, предоставляющий базовую сетевую функциональность для кластеров Kubernetes и хорошо подходящий для оверлейных сетей.
- *Weave Net*. Ориентирован на простую установку и поддержку шифрования, подходит как для облачных, так и для локальных сред.
- *Cilium*. Использует технологию *eBPF* для обеспечения повышенной безопасности и наблюдения за сетью. Оптимальное решение для микросервисных архитектур.

Проприетарные плагины

- *AWS VPC CNI*. Интегрирует поды Kubernetes напрямую с сетью AWS VPC, позволяя управлять IP-адресами и обеспечивая сетевую связность.
- *Azure CNI*. Позволяет подам использовать IP-адреса из Azure VNet, обеспечивая интеграцию с сетевой инфраструктурой платформы Azure.
- *VMware NSX-T*. Предоставляет расширенные сетевые функции, включая микросегментацию и защиту Kubernetes в средах VMware.

При выполнении некоторых упражнений мы будем использовать плагин Calico. Теперь разберемся с понятием сервисной сетки.

Что такое сервисная сетка

Сервисная сетка (service mesh) — это важный инфраструктурный слой, интегрированный в микросервисную архитектуру и обеспечивающий взаимодействие между сервисами. Она выполняет такие функции, как обнаружение сервисов, балансировка нагрузки, шифрование, аутентификация и мониторинг — все в рамках сетевой инфраструктуры. Обычно сервисная сетка реализуется с помощью легковесных прокси, которые размещаются рядом с каждым сервисом. Такая архитектура предоставляет точный контроль над маршрутизацией трафика и позволяет применять такие политики безопасности, как ограничение частоты запросов и предотвращение каскадных сбоев. Благодаря абстрагированию сложных сетевых задач сервисная сетка упрощает разработку, развертывание и эксплуатацию микросервисных приложений, одновременно повышая их надежность, безопасность и наблюдаемость. К наиболее известным реализациям сервисной сетки относятся *Istio*, *Linkerd* и *Consul Connect*. Однако стоит отметить, что подробное рассмотрение этой темы выходит за рамки данной книги.

Далее вы узнаете, для чего используются сервисные сетки и как они могут помочь в публикации микросервисов, запускаемых в подах.

Назначение IP-адреса поду

Чтобы разобраться в том, что такое сервисы, нужно снова вернуться к подам. В Kubernetes все строится вокруг подов: они запускают ваши приложения и обладают особым свойством. При создании пода в кластере система Kubernetes назначает ему приватный IP-адрес. Это крайне важно запомнить: каждый под в вашем кластере получает уникальный IP-адрес, выделенный Kubernetes.

В качестве иллюстрации создадим под `nginx`. В этом примере используется контейнерный образ `nginx`, однако результат будет тем же самым при использовании любого другого образа.

Мы создадим под декларативным способом с помощью следующего YAML-манифеста:

```
# new-nginx-pod.yaml
apiVersion: v1
kind: Pod
metadata:
  name: new-nginx-pod
spec:
  containers:
    - image: nginx:1.17
      name: new-nginx-container
      ports:
        - containerPort: 80
```

Под `new-nginx-pod` не содержит ничего особенного: он просто запускает контейнер `new-nginx-container` на основе образа `nginx:1.17`.

После сохранения этого YAML-файла мы можем применить его с помощью команды:

```
$ kubectl apply -f new-nginx-pod.yaml
pod/new-nginx-pod created
```

В результате ее выполнения в кластере появится под, которому Kubernetes назначит уникальный IP-адрес.

Чтобы просмотреть назначенный поду IP-адрес, мы используем команду `kubectl get pods` с параметром `-o wide`, который выводит дополнительную информацию о поде:

```
$ kubectl get po new-nginx-pod -o wide
NAME           READY   STATUS    RESTARTS   AGE   IP            NODE
NOMINATED NODE   READINESS GATES
new-nginx-pod  1/1     Running   0          99s   10.244.0.109  minikube
<none>           <none>
```

В этом примере под получил IP-адрес `10.244.0.109`. Этот адрес уникален внутри данного кластера Kubernetes и принадлежит только этому поду.

Если вы повторите эти шаги в своем кластере, то получите другой результат. Этот IP-адрес является приватным адресом версии IPv4 и существует только внутри кластера Kubernetes. То есть он недоступен во внешней сети Интернет, поэтому если ввести его в браузере, страница не откроется.

Независимо от того, какую облачную платформу вы используете — *Amazon Web Services (AWS)*, *Google Cloud Platform (GCP)* или *Azure*, кластер Kubernetes задействует сетевой сегмент, предоставляемый облачным провайдером. Этот сегмент, обычно называемый *виртуальным частным облаком* (VPC, virtual private cloud), представляет собой приватную и изолированную сеть, аналогичную приватному диапазону IP-адресов локальной сети частного дата-центра. Во всех случаях CNI-плагин, используемый Kubernetes, гарантирует назначение уникального IP-адреса каждому поду, обеспечивая их изоляцию. Это справедливо и для облачных, и для локальных сред.

Далее вы узнаете о том, что назначение IP-адресов подам является динамическим процессом и может создавать проблемы при масштабировании.

Динамическое назначение IP-адресов подам в Kubernetes

IP-адреса, назначаемые подам, не являются статичными. Если удалить под и создать его заново, он получит совершенно другой IP-адрес, даже если для создания используется тот же самый YAML-манифест. Чтобы продемонстрировать это, удалим под и создадим его заново, используя тот же YAML-файл:

```
$ kubectl delete -f new-nginx-pod.yaml
pod "new-nginx-pod" deleted
$ kubectl apply -f new-nginx-pod.yaml
pod/new-nginx-pod created
```

Теперь снова выполним команду `kubectl get pods -o wide`, чтобы убедиться в отличии нового IP-адреса от предыдущего:

```
$ kubectl get po new-nginx-pod -o wide
NAME            READY   STATUS    RESTARTS   AGE   IP            NODE
NOMINATED NODE   READINESS GATES
new-nginx-pod   1/1     Running   0          97s   10.244.0.110  minikube
<none>           <none>
```

Как видите, адрес изменился с `10.244.0.109` на `10.244.0.110`.

Итак, при удалении и повторном создании пода, даже с тем же именем и конфигурацией, он получает новый IP-адрес.

Причина в том, что технически это уже не тот же под, а два разных экземпляра, поэтому система Kubernetes назначает им разные IP-адреса.

Теперь представьте, что у вас есть приложение, которое обращается к этому поду `nginx` по его IP-адресу. Если под будет удален и создан заново, приложение перестанет работать, потому что прежний IP-адрес окажется недействительным.

В следующем разделе мы обсудим, почему не рекомендуется жестко прописывать IP-адреса подов в коде приложения, какие трудности это создает в рабочей среде, а также рассмотрим более надежные способы обеспечения стабильного взаимодействия между микросервисами в Kubernetes.

Почему не стоит прописывать IP-адреса подов в коде приложений

В рабочих средах использование IP-адресов подов для организации взаимодействия между приложениями создает серьезные трудности. Микросервисы, работающие по протоколу HTTP и использующие TCP/IP, нуждаются в надежном способе обнаружения и подключения друг к другу.

В связи с этим крайне важно создать устойчивый механизм получения информации о подах, а не только об их IP-адресах. Это обеспечит стабильную коммуникацию даже в тех случаях, когда поды пересоздаются или назначаются другим рабочим узлам.

Особенно важно избегать прописывания IP-адресов подов в коде приложений, поскольку они назначаются динамически. Эфемерная природа подов, выражающаяся в возможности их удаления, пересоздания и переноса, делает практику указания IP-адресов в YAML-манифестах ненадежной. Если под с жестко прописанным IP-адресом будет пересоздан, зависящие от него приложения потеряют к нему доступ, так как указанный IP-адрес перестанет существовать.

Эта проблема может возникнуть, например, в следующих ситуациях.

- Под, в котором работает микросервис *А*, имеет зависимость и вызывает микросервис *Б*, работающий в другом поде в том же кластере Kubernetes.
- Приложению, работающему в поде, требуется получить данные с сервера *MySQL*, запущенного в поде того же кластера.
- Приложение использует кластер *Redis* в качестве системы кеширования, развернутой в нескольких подах одного кластера.
- Конечные пользователи обращаются к приложению по IP-адресу, который изменяется из-за сбоя в работе пода.

Проблема актуальна для любого сетевого взаимодействия. Решение заключается в использовании ресурса *Service* системы Kubernetes.

Объект Service выступает в качестве промежуточного элемента, постоянно существующего в кластере. Даже в случае удаления сервиса его можно пересоздать без последствий, так как компоненты системы используют имя сервиса, а не IP-адрес. Более того, сервисы могут существовать в кластере длительное время, не создавая каких-либо проблем. Объекты Service обеспечивают уровень абстракции, позволяющий предоставить доступ к приложению, работающему в поде (или нескольких подах), на сетевом уровне без внесения изменений в его код или конфигурацию.

Как сервисы направляют трафик к подам

Сервисы Kubernetes существуют как ресурсы внутри кластера и выступают в роли абстрактного слоя, управляющего сетевым трафиком. Они используют CNI-плагины для организации взаимодействия между клиентами и подами. Это достигается путем создания конечных точек сервиса, представляющих группы подов. Такой подход обеспечивает балансировку нагрузки и гарантирует доставку трафика только к работоспособным экземплярам.

Сервисы Kubernetes предоставляют статичный и надежный способ доступа к подам внутри кластера. Они предоставляют внутреннее DNS-имя, которое остается постоянным, даже если поды изменяются в результате обновлений, масштабирования или перезапуска. Сервисы используют механизмы *обнаружения* и встроенную *балансировку нагрузки* для эффективной маршрутизации трафика к исправным подам (рис. 8.1).

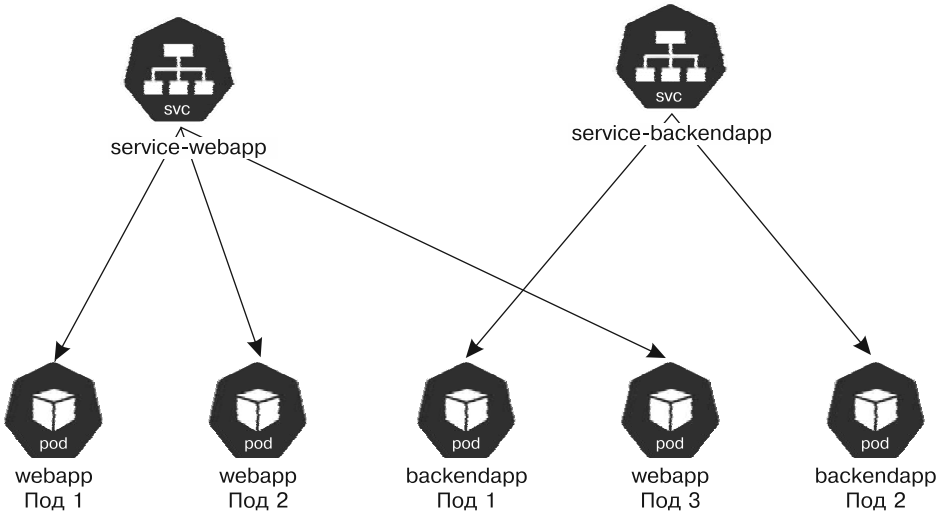

Рис. 8.1. Сервис service-webapp открывает доступ к подам webapp 1, 2 и 3, а сервис service-backendapp — к подам backendapp 1 и 2

Фактически сервисы создаются в Kubernetes как отдельный тип ресурсов. Как и большинство других объектов этой системы, их можно развернуть в кластере либо с помощью интерактивных команд, либо декларативным способом с использованием YAML-манифестов.

Как и в случае с другими ресурсами Kubernetes, при создании сервиса необходимо присвоить ему имя. Это имя система использует для формирования DNS-записи, к которой смогут обращаться все поды в кластере. Данная DNS-запись будет указывать на ваш сервис, который должен оставаться внутри кластера. От вас требуется лишь указать список подов, которые необходимо опубликовать через сервис. Как будет показано далее, это делается с помощью простой конфигурации на основе *меток* и *селекторов*.

После завершения настройки доступ к подам можно будет получить путем простого обращения к сервису, который принимает запросы и перенаправляет их в поды.

Циклический алгоритм балансировки нагрузки в Kubernetes

При условии корректной настройки сервисы Kubernetes могут предоставлять доступ к одному или нескольким подам. Когда один сервис обслуживает несколько подов, входящие запросы равномерно распределяются между ними с помощью циклического алгоритма (round-robin), как показано на рис. 8.2.

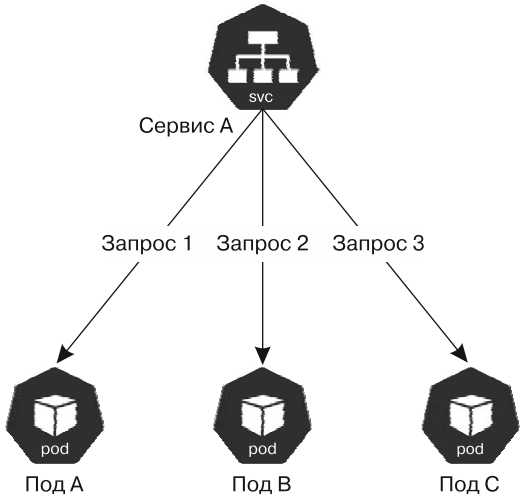

Рис. 8.2. Сервис A перенаправляет три запроса к трем подам, находящимся за ним. При масштабировании каждый под получает примерно 33 % запросов, адресованных сервису

Масштабирование приложений в этом случае сводится к добавлению реплик подов, работающих за сервисом. Подробнее о репликах и объекте Deployment мы поговорим в главе 11. Благодаря встроенной реализации циклического алгоритма сервис Kubernetes способен равномерно распределять запросы между подами.

Возможности сервисов Kubernetes не ограничиваются циклической балансировкой нагрузки. Хотя этот алгоритм обычно применяется в режиме IPVS компонента kube-proxy, стоит учитывать, что в режиме iptables (используемом по умолчанию во многих дистрибутивах) трафик часто распределяется случайным образом или на основе алгоритмов хеширования.

Система Kubernetes поддерживает и другие алгоритмы балансировки нагрузки для решения различных задач, включая метод наименьшего числа активных соединений для равномерной загрузки ресурсов, использование IP-адреса источника для обеспечения согласованной маршрутизации и даже реализацию пользовательской логики для более сложных сценариев. Пользователям также следует учитывать, что режим IPVS поддерживает такие расширенные возможности управления трафиком, как привязка сессий и формирование трафика, которые могут быть недоступны в режиме iptables.

Таким образом, понимание того, какой режим используется в вашем кластере (iptables или IPVS), помогает настраивать поведение сервисов с учетом требований к масштабируемости и распределению трафика. Подробнее об этом можно узнать в документации: https://kubernetes.io/docs/reference/networking/virtual-ips/.

Если бы в предыдущем примере у пода было четыре реплики, то каждая из них обрабатывала бы примерно 25 % всех запросов, получаемых сервисом. А если бы за сервисом находилось 50 подов, то каждый из них получал бы около 2 % запросов. Таким образом, сервисы в Kubernetes выполняют роль балансировщиков нагрузки, применяя различные алгоритмы распределения трафика.

Теперь посмотрим, как можно обратиться к сервису из другого пода.

Обращение к сервису в Kubernetes

В момент создания сервис Kubernetes получает два важных атрибута:

- уникальный IP-адрес;
- автоматически сгенерированное неизменяемое внутреннее DNS-имя.

Используя любой из этих атрибутов, можно обратиться к сервису, который перенаправит запрос на обслуживаемый им под. Однако чаще всего сервис вызывается по его DNS-имени, так как оно довольно легко определяется. Посмотрим, как Kubernetes назначает сервисам DNS-имена.

Процесс генерации DNS-имен сервисов

DNS-имя сервиса генерируется на основе его имени. Например, если вы создаете сервис с именем `my-app-service`, то его DNS-имя будет выглядеть так: `my-app-service.default.svc.cluster.local`.

Структура этого имени показана на рис. 8.3.

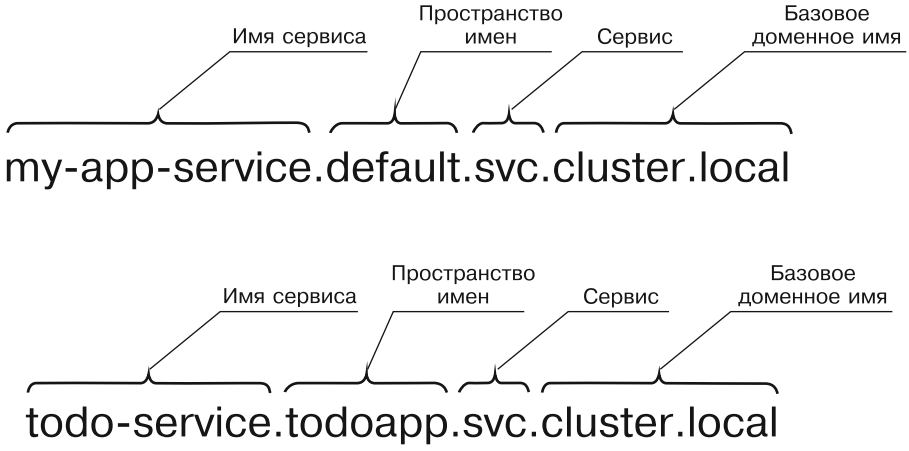

Рис. 8.3. Полное доменное имя сервиса

Первые два фрагмента — это имя сервиса и пространство имен, в котором он существует. Последним фрагментом DNS-имени обычно является `.svc.cluster.local`.

Таким образом, вы можете быть уверены, что запрос, направленный по адресу `my-app.default.svc.cluster.local` с помощью `curl` или `wget` из любого места в кластере достигнет соответствующего сервиса.

Это имя будет разрешаться в адрес сервиса при отправке запроса из любого пода в кластере. Однако по умолчанию сервис не будет перенаправлять трафик, пока мы не укажем ему, к каким именно подам необходимо проксировать запросы. Далее вы узнаете, как это можно сделать.

Обнаружение и направление трафика к подам Kubernetes

При работе с сервисами Kubernetes вы часто будете сталкиваться с понятием *публикации* подов (или предоставления доступа к ним). Именно так в Kubernetes называется процесс перенаправления сетевого трафика, осуществляемого сервисом. Помните об этом, когда коллеги спросят: «Какой сервис

предоставляет доступ к данному поду?» Идея публикации подов проиллюстрирована на рис. 8.4.

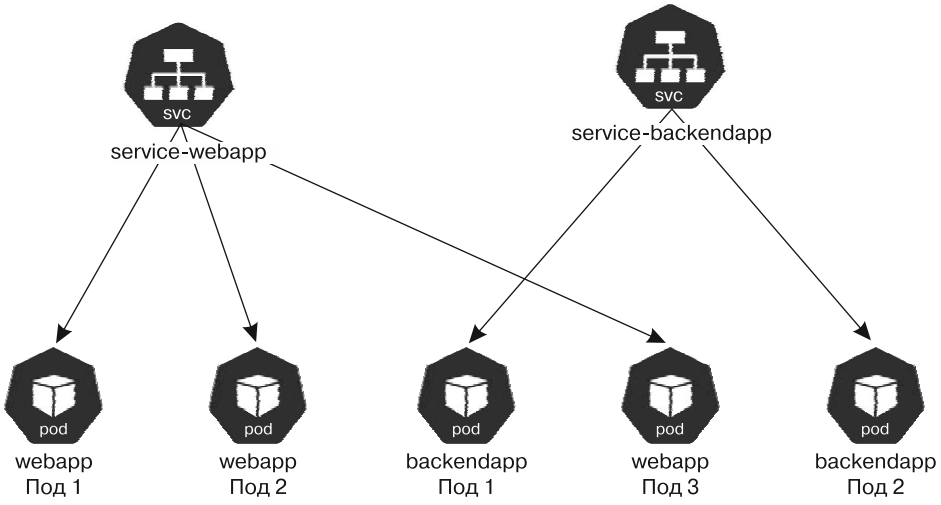

Рис. 8.4. Сервис service-webapp открывает доступ к подам webapp 1, 2 и 3, а сервис service-backendapp — к подам backendapp 1 и 2

Создать под и сервис, который предоставляет к нему доступ, можно с помощью одной команды `kubectl` с параметром `--expose`. В качестве примера создадим под `nginx` и публикующий его сервис.

В этой команде также необходимо указать порт, на котором данный сервис будет доступен:

```
$ kubectl run nginx --image nginx --expose=true --port=80
service/nginx created
pod/nginx created
```

Теперь выведем список подов и сервисов с помощью `kubectl`, чтобы убедиться в создании обоих объектов:

```
$ kubectl get po,svc nginx
NAME        READY   STATUS    RESTARTS   AGE
pod/nginx   1/1     Running   0          24s

NAME           TYPE        CLUSTER-IP      EXTERNAL-IP   PORT(S)   AGE
service/nginx  ClusterIP   10.111.12.100   <none>        80/TCP    24s
```

Как видите, и под, и сервис были успешно созданы. Ранее мы уже говорили, что сервисы определяют поды, которые они должны публиковать, на основе меток. Под `nginx`, который мы только что создали, тоже предусматривает несколько меток. Для их просмотра можно выполнить команду `kubectl get pods nginx --show-labels`.

Обратите внимание на параметр `--show-labels`: именно он отвечает за вывод меток:

```
$ kubectl get po nginx --show-labels
NAME   READY   STATUS    RESTARTS   AGE   LABELS
nginx  1/1     Running   0          51s   run=nginx
```

Как видите, под `nginx` был создан с меткой `run=nginx`. Теперь посмотрим описание сервиса `nginx`. В нем должен присутствовать селектор, совпадающий с этой меткой:

```
$ kubectl describe svc nginx
Name:              nginx
Namespace:         default
Labels:            <none>
Annotations:       <none>
Selector:          run=nginx
Type:              ClusterIP
IP Family Policy:  SingleStack
IP Families:       IPv4
IP:                10.111.12.100
IPs:               10.111.12.100
Port:              <unset>  80/TCP
TargetPort:        80/TCP
Endpoints:         10.244.0.9:80
Session Affinity:  None
Events:            <none>
```

В описании сервиса действительно присутствует строка `Selector`, значение в которой совпадает с меткой, назначенной поду `nginx`. Таким образом устанавливается связь между двумя объектами. Теперь мы можем быть уверены в том, что сервис «видит» под и может корректно перенаправлять к нему запросы.

Учтите, что при использовании `minikube` в лабораторных условиях вы не сможете получить доступ к сервису извне кластера, так как сервисы типа ClusterIP доступны только изнутри. В подобных случаях можно использовать такие методы, как `kubectl port-forward` или `kubectl proxy`. О тестировании сервисов типа ClusterIP мы подробно поговорим далее в главе.

Чтобы протестировать доступ к сервису, настроим временное перенаправление портов:

```
$ kubectl port-forward service/nginx 8080:80
Forwarding from 127.0.0.1:8080 -> 80
Forwarding from [::1]:8080 -> 80
```

Теперь откройте другую консоль и обратитесь к следующему URL-адресу:

```
$ curl 127.0.0.1:8080
<!DOCTYPE html>
<html>
... <фрагмент опущен для краткости>...
```

В этом примере `8080` — это порт локального хоста, а `80` — это порт `nginx`, предоставляющий доступ к веб-сервису.

Учтите, что команда `kubectl port-forward` будет выполняться до тех пор, пока вы не нажмете Ctrl+C.

Хотя этот способ работает, его не рекомендуется применять в рабочей среде. Сервисы — это очень гибкие объекты, а параметр `--expose` скрывает многие из возможностей для их настройки. Вместо этого следует применять декларативный синтаксис и адаптировать YAML-манифест под свои потребности.

Продемонстрируем этот подход на примере контейнерного образа `dnsutils`.

Использование пода k8sutils для отладки сервисов

Как мы уже отмечали, доступ к сервисам, развернутым внутри кластера, часто бывает затруднен, особенно в тех случаях, когда поды должны быть доступны исключительно внутри кластера или если кластер не имеет выхода в Интернет.

В таких случаях целесообразно развернуть в кластере отладочный под, в котором установлены базовые утилиты для сетевой диагностики, например `wget`, `nslookup` и т. п. Сделаем это, используя подготовленный контейнерный образ `quay.io/iamgini/k8sutils:debian12`.

При необходимости вы можете добавить дополнительные инструменты или утилиты внутрь этого контейнерного образа; исходный файл находится по адресу https://github.com/PacktPublishing/The-Kubernetes-Bible-Second-Edition/blob/main/Chapter08/Containerfile.

В данном примере мы попробуем получить главную страницу пода `nginx`, обратившись к сервису, который предоставляет к нему доступ. Именем этого сер-

виса является nginx. Поэтому нам нет необходимости использовать его полное DNS-имя nginx.default.svc.cluster.local, назначенное Kubernetes.

При обращении к этому *URL-адресу* (Uniform Resource Locator — унифицированный указатель ресурса) из любого пода в кластере[1] вы должны получить доступ к стартовой странице nginx.

Ниже показан YAML-файл, позволяющий создать отладочный под с образом k8sutils:

```
# k8sutils.yaml
apiVersion: v1
kind: Pod
metadata:
  name: k8sutils
  namespace: default
spec:
  containers:
    - name: k8sutils
      image: quay.io/iamgini/k8sutils:debian12
      command:
        - sleep
        - "infinity"
  restartPolicy: Always
```

Выполним следующую команду для запуска пода k8sutils:

```
$ kubectl apply -f k8sutils.yaml
pod/k8sutils created
```

И удостоверимся в том, что под был успешно создан:

```
$ kubectl get po k8sutils
NAME       READY   STATUS    RESTARTS   AGE
k8sutils   1/1     Running   0          13m
```

Отлично! Теперь выполним команду nslookup внутри пода k8sutils для проверки DNS-записи сервиса:

```
$ kubectl exec -it k8sutils -- nslookup nginx.default.svc.cluster.local
Server:         10.96.0.10
Address:        10.96.0.10#53
Name:  nginx.default.svc.cluster.local
Address: 10.106.124.200
```

[1] В пределах одного пространства имен.

В выводе приведены следующие сведения:

- `Server: 10.96.0.10` — это IP-адрес сервиса `kube-dns` (`kubectl get svc kube-dns -n kube-system -o wide`). Если вы используете другой DNS-сервис, проверьте его параметры;
- `nginx.default.svc.cluster.local` — DNS-запись, разрешающаяся в IP-адрес сервиса `nginx` (`kubectl get svc nginx -o wide`), например `10.106.124.xxx`.

Теперь запустим команду `curl`, чтобы проверить возможность получения главной страницы `nginx`:

```
$ kubectl exec -it k8sutils -- curl nginx.default.svc.cluster.local
... <фрагмент опущен для краткости>...
<body>
<h1>Welcome to nginx!</h1>
<p>If you see this page, the nginx web server is successfully installed and
working. Further configuration is required.</p>

... <фрагмент опущен для краткости>...
```

Как видите, нам удалось обратиться к сервису `nginx` с помощью отладочного пода `k8sutils`, как показано на рис. 8.5.

Рис. 8.5. Под k8sutils используется для отправки curl-запроса к сервису nginx, предоставляющему доступ к одноименному поду

Помните, что для отладки сервиса под `k8sutils` необходимо развернуть именно внутри кластера, так как DNS-имя `nginx.default.svc.cluster.local` не является публичным и доступно только в кластере.

В следующем подразделе вы узнаете, почему императивный способ публикации подов не является оптимальным.

Недостатки использования команды kubectl expose в Kubernetes

Использовать команду `kubectl expose` не рекомендуется, поскольку она не позволяет контролировать процесс создания сервиса. По умолчанию эта команда создает сервис типа `ClusterIP`, в то время как вам может потребоваться сервис типа `NodePort`.

Хотя задать тип сервиса можно и при использовании императивного синтаксиса, итоговая команда получится слишком длинной и трудной для понимания. Именно поэтому мы советуем не использовать параметр `expose` и придерживаться декларативного подхода при работе с такими сложными объектами, как сервисы.

Теперь обсудим процесс генерации DNS-имен сервисов в Kubernetes.

Принцип взаимодействия сервисов и подов

Вы уже знаете, что взаимодействие сервиса с подом целиком опирается на метки пода и селекторы сервиса.

Если эти параметры заданы некорректно, связь между этими объектами не будет установлена.

Процесс налаживания этого взаимодействия состоит из следующих этапов.

1. Вы создаете несколько подов и назначаете им произвольные метки.
2. Вы создаете сервис и настраиваете его селектор в соответствии с метками подов.
3. Сервис запускается и ищет поды с метками, соответствующими селектору.
4. Вы обращаетесь к сервису по его DNS-имени или IP-адресу (обычно проще использовать DNS).
5. Сервис перенаправляет трафик в один из подов, метки которого совпадают с селектором.

Если вы посмотрите на предыдущий пример, в котором использовались императивный синтаксис и команда `kubectl expose`, то заметите, что под и сервис были настроены корректно, поэтому под был успешно опубликован. Учтите, что в реальных сценариях подам следует назначать собственные метки вместо меток по умолчанию.

Кроме того, важно понимать, что в Kubernetes существует не один, а несколько типов сервисов, которые мы обсудим далее.

Типы сервисов

В Kubernetes поддерживается несколько типов сервисов. Хотя формально существует только один объект Service, его можно настраивать по-разному для достижения разных результатов.

Вне зависимости от выбранного типа сервиса его цель всегда заключается в предоставлении доступа к подам через единый статичный интерфейс.

Каждый тип сервиса имеет собственное назначение и область применения. Один сервис не может одновременно принадлежать к нескольким типам, но вы можете опубликовать один и тот же под с помощью двух сервисов разных типов, при условии, что они будут называться по-разному (чтобы система Kubernetes смогла присвоить им разные DNS-имена).

В этой главе мы рассмотрим три основных типа сервисов.

- `NodePort`. Этот тип сервиса связывает порт из диапазона эфемерных портов хост-машины (рабочего узла) с портом пода, делая его доступным извне. Обращаясь к порту узла, вы попадете в соответствующий под Kubernetes. Именно так осуществляется доступ к подам для перенаправления трафика, поступающего извне кластера.
- `ClusterIP`. Используется для организации приватного взаимодействия между подами внутри кластера Kubernetes. Мы уже сталкивались с ним в этой главе, так как именно такой сервис по умолчанию создает команда `kubectl expose`. Это наиболее распространенный тип сервисов, так как он обеспечивает взаимодействие между подами. Как следует из названия, данный сервис имеет статический IP-адрес, закрепленный на уровне кластера. При обращении к этому IP-адресу трафик перенаправляется к подам, развернутым за сервисом. Если за ним находится несколько подов, этот сервис обеспечивает балансировку нагрузки, используя циклический или другой алгоритм.
- `LoadBalancer`. Сервис этого типа в Kubernetes упрощает процесс перенаправления к подам внешнего трафика. При работе на поддерживаемых облачных платформах он автоматически создает соответствующий балансировщик нагрузки, например AWS ELB, что избавляет нас от необходимости вручную настраивать внешние балансировщики. Однако учтите, что без ручной настройки такой сервис несовместим с кластерами, развернутыми на «голом железе», и с кластерами, не управляемыми облаком. Хотя для управления облачной инфраструктурой существуют альтернативные решения, вроде Terraform, сервис типа `LoadBalancer` предоставляет удобный способ бесшовной интеграции облачных балансировщиков нагрузки с развертываниями Kubernetes. Следует также учитывать, что применимость этого подхода зависит от конкретных требований и конфигурации инфраструктуры.

Теперь подробно рассмотрим первый тип сервисов — `NodePort`.

Как уже упоминалось, такой сервис особенно полезен для предоставления доступа к подам извне кластера в средах разработки, так как он привязывает поды к портам узлов Kubernetes.

Сервис типа NodePort

`NodePort` — это тип сервиса Kubernetes, делающий поды доступными через порт хост-машины (рабочего узла). В этом разделе мы сосредоточимся исключительно на сервисах этого типа.

Назначение сервисов типа NodePort

Прежде всего важно понимать, что сервисы `NodePort` позволяют обращаться к поду, работающему на узле Kubernetes, через порт самого узла. После публикации подов с помощью сервиса типа `NodePort` доступ к ним можно будет получить по схеме `<ip-адрес_узла>:<порт_узла>`.

Порт может быть явно указан в YAML-манифесте или назначен автоматически системой Kubernetes. Проиллюстрируем это на конкретном примере.

Чаще всего сервисы `NodePort` используются в качестве входной точки в кластер Kubernetes. В следующем примере мы создадим два пода на основе контейнерного образа `containous/whoami`, доступного в реестре Docker Hub и выводящего имя хоста контейнера.

Мы создадим два пода, чтобы получить два контейнера с разными именами хостов, а затем опубликуем их с помощью сервиса типа `NodePort`.

Создание двух подов на основе образа containous/whoami

Начнем с создания двух подов с одной или несколькими метками. Эти метки позволят сервису «понять», к каким подам он должен предоставить доступ.

Нам также нужно открыть порт на стороне пода. Само по себе это не сделает под доступным извне, но позволит сервису обращаться к этому порту. Код для решения этой задачи приведен ниже:

```
$ kubectl run whoami1 --image=containous/whoami --port 80 --labels="app=whoami"
pod/whoami1 created
$ kubectl run whoami2 --image=containous/whoami --port 80 --labels="app=whoami"
pod/whoami2 created
```

Теперь выполним команду `kubectl get pods`, чтобы убедиться в успешном запуске обоих подов. Для отображения меток можно добавить параметр `--show-labels`:

```
$ kubectl get pods --show-labels
NAME      READY   STATUS    RESTARTS   AGE     LABELS
whoami1   1/1     Running   0          3m5s    app=whoami
whoami2   1/1     Running   0          3m      app=whoami
```

Как видите, все в порядке. Теперь у нас есть два пода с назначенной меткой и мы можем опубликовать их с помощью сервиса типа `NodePort`. Далее мы рассмотрим YAML-манифест для создания такого сервиса.

YAML-манифест для создания сервиса NodePort

Поскольку сервисы являются достаточно сложными объектами, их лучше создавать с помощью YAML-манифеста, а не команд.

Ниже приведен YAML-файл для публикации подов `whoami1` и `whoami2` с помощью сервиса типа `NodePort`:

```
# nodeport-whoami.yaml
apiVersion: v1
kind: Service
metadata:
  name: nodeport-whoami
spec:
  type: NodePort
  selector:
    app: whoami
  ports:
    - nodePort: 30001
      port: 80
      targetPort: 80
```

На первый взгляд этот YAML-манифест может показаться непростым, так как он ссылается на три разных порта и содержит блок `selector`.

Прежде чем подробно его разбирать, применим манифест и удостоверимся в создании сервиса:

```
$ kubectl apply -f nodeport-whoami.yaml
service/nodeport-whoami created

$ kubectl get service nodeport-whoami
NAME              TYPE       CLUSTER-IP     EXTERNAL-IP   PORT(S)        AGE
nodeport-whoami   NodePort   10.98.160.98   <none>        80:30001/TCP   14s
```

Вывод команды `kubectl get services` подтверждает, что сервис был успешно создан.

Блок `selector` имеет ключевое значение для сервисов `NodePort`. Он выполняет фильтрацию по меткам, определяя поды, к которым сервис будет открывать

доступ. По сути, он указывает сервису, куда именно следует перенаправлять трафик. В отсутствии селектора сервис останется неактивным. В данном примере селектор ищет поды с меткой `app=whoami`. Таким образом, сервис предоставляет доступ к обоим подам — `whoami1` и `whoami2`.

В разделе `spec` указывается тип сервиса (`type`). При создании сервисов других типов (`ClusterIP` или `LoadBalancer`) значение этого ключа будет иным. В данном случае мы задаем значение `NodePort`.

Последним и, пожалуй, самым непростым для понимания является блок `ports`, в котором задаются три разных порта.

- `nodePort` — порт хост-машины (рабочего узла), через который сервис `NodePort` будет доступен извне. Значение `30001` говорит о том, что доступ к данному сервису и развернутым за ним подам можно получить по адресу `<IP_адрес_рабочего_узла>:30001`. Учтите, что этот параметр нельзя задать произвольно: в стандартной установке Kubernetes допустимым диапазоном портов является `30000-32767`.
- `port` — это отдельный внутренний порт, который принадлежит сервису `NodePort`. В этом поле можно указать любое допустимое значение.
- `targetPort` — это порт целевых подов, на котором фактически работает приложение. Именно на него сервис будет перенаправлять трафик.

На рис. 8.6 представлена схема.

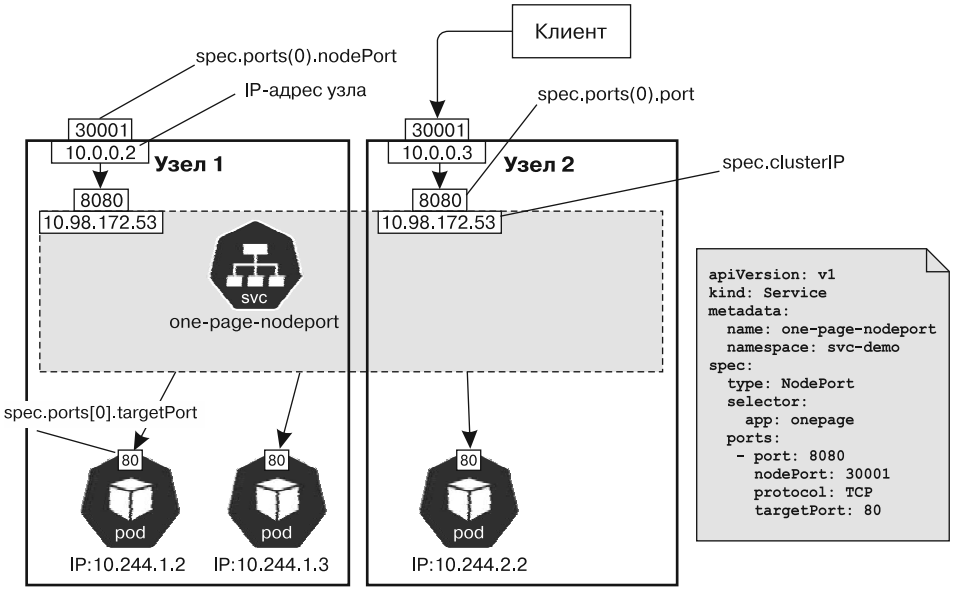

Рис. 8.6. Сервис NodePort. На схеме показано три порта: nodePort относится к рабочему узлу, port — к самому сервису, а targetPort — к поду

В данном случае TCP-порт `30001` используется в качестве внешнего порта на каждом узле. Если значение `nodePort` не указано, оно будет динамически выбираться из допустимого диапазона. При взаимодействии внутри кластера такой сервис будет вести себя как обычный сервис типа `ClusterIP`, поэтому вы сможете обращаться к нему по его `ClusterIP`-адресу.

Для большего удобства в качестве параметров `port` и `targetPort` сервиса `NodePort` часто используются одинаковые значения.

Проверка работоспособности сервиса NodePort

Чтобы протестировать сервис `NodePort`, первым делом нужно получить публичный IP-адрес машины, на которой он работает. В нашем примере используется локальный Kubernetes-кластер `minikube`, развернутый на одной машине. На такой облачной платформе, как AWS, GCP или Azure, ваш узел может иметь публичный IP-адрес либо приватный, если вы подключаетесь к нему через *виртуальную частную сеть* (virtual private network, VPN).

В случае с `minikube` IP-адрес проще всего получить с помощью следующей команды:

```
$ minikube ip
192.168.64.2
Or you can access the full URL as follows.
$ minikube service --url nodeport-whoami
http://192.168.49.2:30001
```

Теперь, когда у нас есть вся необходимая информация, можно открыть веб-браузер и ввести URL-адрес, чтобы получить доступ к сервису `NodePort` и запущенным подам. Если все настроено корректно, вы увидите работу циклического алгоритма, последовательно направляющего запросы к подам `whoami1` и `whoami2`.

Оценка готовности сервиса к развертыванию в рабочей среде

Готовность сервиса к развертыванию в рабочей среде зависит от конфигурации.

Сервис `NodePort` открывает доступ к подам извне, привязывая их к порту узла. Однако текущая конфигурация не предусматривает механизма обеспечения высокой доступности. Если оба пода выйдут из строя, они не будут автоматически пересозданы и сервис не сможет корректно перенаправлять трафик, что негативно скажется на пользовательском опыте.

При создании подов с помощью объектов Deployment и ReplicaSet система Kubernetes автоматически пересоздает новые поды на доступных узлах. Мы подробно рассмотрим эти объекты в главе 11.

Еще одна проблема связана с ограничением диапазона портов. По умолчанию вы вынуждены использовать порты в диапазоне `30000–32767`, что не очень удобно. Например, если вы хотите открыть доступ к HTTP-приложению, то для этого предпочтительнее использовать порты `80` или `443` на фронтальном узле, а не какой-то порт из диапазона `30000–32767`, поскольку все веб-браузеры по умолчанию настроены на работу с портами `80` (HTTP) и `443` (HTTPS).

В данном случае решение заключается в использовании многоуровневой архитектуры. Многие архитекторы Kubernetes предпочитают не использовать сервис `NodePort` в качестве точки входа и размещать кластер за обратным прокси, таким как AWS Application Load Balancer. Кроме того, в Kubernetes существуют объекты Ingress и IngressController, которые позволяют настраивать обратный прокси (например, `nginx` или HAProxy) непосредственно средствами Kubernetes и обеспечивать публичный доступ к приложению, выступая в роли точки входа в кластер. Однако рассмотрение этих концепций выходит за рамки обсуждения сервисов Kubernetes.

В следующих подразделах вы узнаете, как можно вывести список сервисов `NodePort` и добавить в них поды.

Просмотр списка сервисов NodePort

Посмотреть список созданных в кластере сервисов `NodePort` можно с помощью команды `kubectl get services`:

```
$ kubectl get service
NAME              TYPE        CLUSTER-IP       EXTERNAL-IP   PORT(S)        AGE
example-service   ClusterIP   10.106.224.122   <none>        80/TCP         26d
kubernetes        ClusterIP   10.96.0.1        <none>        443/TCP        26d
nodeport-whoami   NodePort    10.100.85.171    <none>        80:30001/TCP   21s
```

Теперь обсудим способ обновления сервисов типа `NodePort`.

Добавление подов в сервис NodePort

Для добавления пода в пул, обслуживаемый сервисом, достаточно создать новый под с меткой, совпадающей с селектором, определенным в сервисе. Все остальное сделает система Kubernetes. При удалении пода он будет исключен из пула, как только перейдет в состояние `Terminating`.

Kubernetes перенаправляет трафик сервисов, исходя из доступности подов. Например, если у вас есть три реплики веб-сервера и одна из них выходит из строя, достаточно создать еще одну реплику с меткой, совпадающей с селектором сервиса. В главе 11 вы увидите, что этот процесс можно полностью автоматизировать.

Просмотр подробных сведений о сервисе NodePort

Как и в случае с любыми другими объектами Kubernetes, для просмотра подробной информации о сервисе `NodePort` используется команда `kubectl describe`. Выведем сведения о сервисе `nodeport-whoami`:

```
$ kubectl describe service nodeport-whoami
Name:                     nodeport-whoami
... <фрагмент опущен для краткости> ...
Selector:                 app=whoami
Type:                     NodePort
IP Family Policy:         SingleStack
IP Families:              IPv4
IP:                       10.98.160.98
IPs:                      10.98.160.98
Port:                     <unset>  80/TCP
TargetPort:               80/TCP
NodePort:                 <unset>  30001/TCP
Endpoints:                10.244.0.16:80,10.244.0.17:80
... <фрагмент опущен для краткости> ...
```

В этом выводе содержится несколько ключевых элементов.

- `IP: 10.98.160.98` — это `ClusterIP`, назначенный сервису, то есть внутренний IP-адрес, по которому другие сервисы в кластере могут обращаться к данному сервису.
- `Port: <unset> 80/TCP` — данный сервис прослушивает порт `80` и использует протокол TCP. Запись `<unset>` означает, что имя для порта не было задано.
- `NodePort: <unset> 30001/TCP` — это порт на каждом узле кластера, через который внешний трафик получает доступ к сервису. В данном случае установлено значение `30001`, что позволяет обращаться к сервису по IP-адресу любого узла на порте `30001`.
- `Endpoints: 10.244.0.16:80, 10.244.0.17:80` — фактические IP-адреса и порты подов, развернутых за сервисом. В данном случае сервис обслуживают два пода, доступные по адресам `10.244.0.16:80` и `10.244.0.17:80`.

В следующем подразделе вы узнаете, как удалить сервис с помощью команды `kubectl delete svc`.

Удаление сервисов

Удалять сервисы не следует слишком часто. В отличие от подов, которые предполагается легко удалять и пересоздавать, сервисы предназначены для долгосрочного использования. Они обеспечивают стабильный доступ к подам, и их удаление напрямую влияет на доступность приложений.

Поэтому при удалении сервисов следует действовать осторожно. Поды, развернутые за ними, удалены не будут, но доступ к ним извне кластера станет невозможным.

Ниже приведена команда для удаления сервиса, созданного для публикации подов `whoami1` и `whoami2`:

```
$ kubectl delete svc/nodeport-whoami
service "nodeport-whoami" deleted
```

Чтобы убедиться в успешном удалении сервиса, можно выполнить команду `kubectl get svc`. При повторной попытке обращения к приложению через веб-браузер вы заметите, что оно стало недоступно, хотя поды по-прежнему присутствуют в кластере. Помните, что жизненные циклы подов и сервисов не зависят друг от друга, то есть удалять поды придется отдельно.

Сервис NodePort и команда kubectl port-forward

Ранее мы использовали команду `kubectl port-forward` для тестирования пода `nginx` и отображения его домашней страницы. У вас может возникнуть впечатление, что сервисы `NodePort` аналогичны команде `kubectl port-forward`, так как до этого момента и то и другое использовалось для получения доступа к работающему поду в кластере через браузер.

Однако делать этого не стоит. Команда `kubectl port-forward` — это инструмент для тестирования, тогда как сервисы `NodePort` предназначены для использования в рабочей среде.

Помните, что сеанс `kubectl port-forward` должен оставаться активным в вашем терминале. Как только работа команды будет прервана, перенаправление портов прекратится и ваше приложение снова станет недоступным извне кластера. Эта команда предназначена исключительно для тестирования, используется напрямую через `kubectl` и является одной из многих полезных утилит, входящих в состав этого инструмента командной строки.

В свою очередь, сервис `NodePort` предназначен для длительной эксплуатации в рабочей среде. Он не зависит от `kubectl` и делает приложение доступным для любого клиента, обращающегося к сервису, при условии, что этот сервис корректно настроен, а поды имеют правильные метки.

Иными словами, если вам нужно быстро протестировать приложение, используйте команду `kubectl port-forward`. Если же вы хотите сделать под постоянно доступным извне, используйте сервис `NodePort`. Не создавайте сервисы `NodePort` для проведения тестов и не пытайтесь использовать команду `kubectl`

port-forward в рабочей среде. У каждого из этих инструментов своя область применения!

Теперь познакомьтесь с другим типом сервиса Kubernetes — ClusterIP, который используется даже чаще, чем NodePort.

Сервис типа ClusterIP

В этом разделе речь пойдет о ClusterIP — самом простом типе сервиса, доступном в Kubernetes. Сервис ClusterIP позволяет опубликовать под так, чтобы другие поды могли обращаться к нему по его IP-адресу или DNS-имени.

Назначение сервисов типа ClusterIP

Сервис ClusterIP во многом похож на сервис NodePort, но между ними есть одно ключевое различие: сервис NodePort используется для обеспечения доступа к подам извне, а сервис ClusterIP делает поды доступными другим подам в кластере Kubernetes.

Именно сервисы ClusterIP обеспечивают взаимодействие подов внутри кластера через статичный интерфейс, которым является сам объект ClusterIP Service.

Как и NodePort, сервис ClusterIP предоставляет статический IP-адрес или DNS-имя, устраняя проблему смены IP-адресов подов при их сбое или перезапуске. Но в отличие от NodePort сервисы типа ClusterIP предоставляют внутреннее DNS-имя, доступное только внутри кластера и разрешающееся в поды, определяемые селектором меток.

Как следует из названия, сервис ClusterIP назначает статический IP-адрес в пределах кластера. Далее мы обсудим способ публикации подов с помощью сервиса типа ClusterIP. Помните, что такие сервисы недоступны извне кластера и предназначены только для организации взаимодействия между подами.

Выбор между сервисами NodePort и ClusterIP

Выбор между этими двумя типами сервисов предельно прост, так как они изначально предназначены для решения разных задач.

Если вам нужно сделать приложение доступным извне кластера, используйте сервис NodePort (или другие сервисы, которые мы рассмотрим далее в этой главе). Чтобы сделать приложение доступным только внутри кластера, используйте сервис ClusterIP. Сервисы ClusterIP также хорошо подходят для при-

ложений без сохранения состояния, которые можно масштабировать, удалять и пересоздавать, поскольку обеспечивают статичную точку входа для целого пула подов и, в отличие от сервисов `NodePort`, не привязываются к конкретному порту рабочего узла.

Сервис типа `ClusterIP` публикует поды с помощью виртуальных IP-адресов, видимых только внутри кластера и управляемых компонентом `kube-proxy` на каждом узле. Это означает, что к сервису можно получить доступ исключительно изнутри кластера. Принцип работы сервиса `ClusterIP` проиллюстрирован на рис. 8.7.

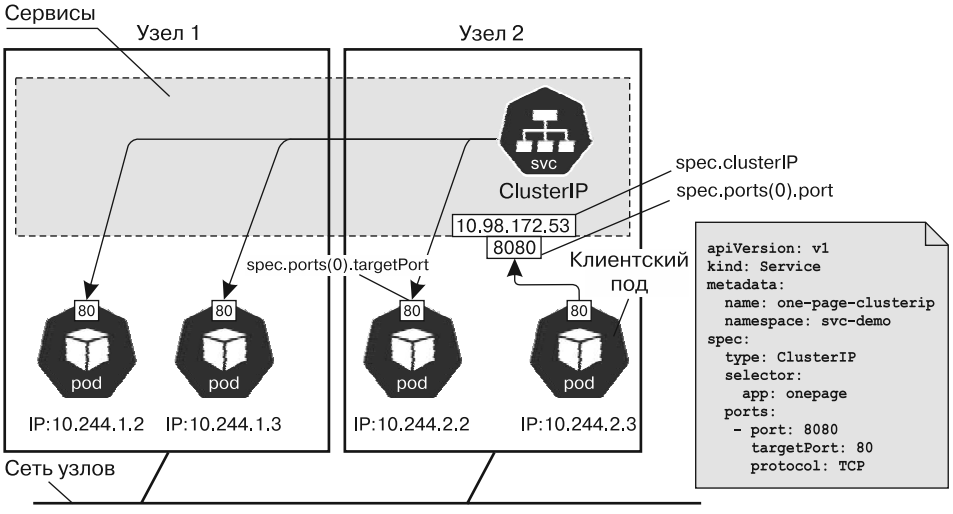

Рис. 8.7. Сервис типа ClusterIP

Изображенный на схеме сервис `ClusterIP` перенаправляет запросы, поступающие на его IP-адрес и TCP-порт `8080`, на контейнерный TCP-порт `80`. Если IP-адрес сервиса `ClusterIP` не указан в спецификации, то он назначается динамически. Разрешение имени `nginx-deployment-example` в фактический адрес `ClusterIP` выполняет внутренний DNS-сервис кластера Kubernetes в рамках работы механизма обнаружения сервисов.

Компонент kube-proxy управляет виртуальными IP-адресами на узлах и настраивает правила маршрутизации трафика. Сервисы Kubernetes, по сути, представляют собой логические, а не физические сущности. То есть фактическое проксирование и маршрутизацию выполняет не отдельный процесс, а компонент kube-proxy на основе конфигурации сервисов.

В отличие от сервисов NodePort сервисы ClusterIP не занимают порт рабочего узла, поэтому к ним невозможно обратиться извне кластера.

Учтите, что для одного и того же пула подов можно использовать оба типа сервисов. Если вы хотите сделать приложение общедоступным и организовать его внутреннее взаимодействие с другими подами, можете создать два сервиса — один типа NodePort и один типа ClusterIP.

В этом случае нужно лишь дать сервисам разные имена, чтобы избежать конфликтов при их создании через kube-apiserver. Никаких других ограничений не существует.

Просмотр списка сервисов ClusterIP

Для вывода списка сервисов типа ClusterIP используется та же команда, что и для перечисления сервисов NodePort:

```
$ kubectl get svc
```

В ее выводе будут перечислены все сервисы с указанием их типа.

Императивный способ создания сервисов ClusterIP

Сервис ClusterIP является одним из наиболее часто используемых объектов Kubernetes, поэтому для его создания предусмотрено несколько способов. В частности, его создать можно:

- с помощью параметра --expose или метода kubectl expose (императивный способ);
- с помощью YAML-манифеста (декларативный способ).

Императивный способ сводится к использованию параметра --expose с командой kubectl run. В следующем примере мы создадим под nginx-clusterip и сервис ClusterIP для его публикации. Чтобы сделать под доступным, при использовании параметра --expose необходимо также указать порт, который сервис ClusterIP будет прослушивать. Код для решения этой задачи выглядит следующим образом:

```
$ kubectl run nginx-clusterip --image nginx --expose=true --port=80
service/nginx-clusterip created
pod/nginx-clusterip created
```

Как видите, в результате создается и под, и публикующий его сервис. Теперь посмотрим описание этого сервиса.

Просмотр подробных сведений о сервисе ClusterIP

Как и в случае с любыми другими объектами Kubernetes, просмотреть сведения о сервисах типа `ClusterIP` можно с помощью команды `kubectl describe`. Для этого достаточно знать имя сервиса.

Применим эту команду к сервису `ClusterIP`, созданному ранее:

```
$ kubectl describe svc/nginx-clusterip
Name:              nginx-clusterip
Namespace:         default
Labels:            <none>
Annotations:       <none>
Selector:          run=nginx-clusterip
Type:              ClusterIP
IP Family Policy:  SingleStack
IP Families:       IPv4
IP:                10.101.229.225
IPs:               10.101.229.225
Port:              <unset>  80/TCP
TargetPort:        80/TCP
Endpoints:         10.244.0.10:80
Session Affinity:  None
Events:            <none>
```

Блок `Selector` показывает, что сервис `ClusterIP` был создан с помощью параметра `--expose` и содержит значение, совпадающее с меткой, назначенной поду `nginx-clusterip`, созданному одновременно с ним. Чтобы убедиться в этом, выведем метки пода:

```
$ kubectl get pods/nginx-clusterip --show-labels
NAME              READY   STATUS    RESTARTS   AGE   LABELS
nginx-clusterip   1/1     Running   0          76s   run=nginx-clusterip
```

Как видите, селектор сервиса совпадает с меткой пода, благодаря чему между этими объектами возможно взаимодействие. Теперь обратимся к сервису `ClusterIP` напрямую из другого пода в кластере.

Поскольку сервис называется `nginx-clusterip`, он доступен по адресу `nginx-clusterip.default.svc.cluster.local`.

Для проведения проверки снова воспользуемся контейнерным образом `k8sutils`:

```
$ kubectl exec k8sutils -- curl nginx-clusterip.default.svc.cluster.local
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
0   0    0     0    0     0      0      0 --:--:-- --:--:-- --:--:--     0
<!DOCTYPE ... <фрагмент опущен для краткости> ...
<body>
```

```
<h1>Welcome to nginx!</h1>
<p>If you see this page, the nginx web server is successfully installed and
working. Further configuration is required.</p>
 ... <фрагмент опущен для краткости> ...
```

Судя по всему, сервис `ClusterIP` работает. Он корректно перенаправил запрос к поду `nginx`, и мы получили стандартную домашнюю страницу `nginx`.

Хотя в данном случае мы не использовали образ `containous/whoami` в качестве веб-сервиса, стоит помнить о том, что сервис `ClusterIP` также предусматривает внутреннюю балансировку нагрузки, выполняемую по циклическому алгоритму. Например, если за сервисом `ClusterIP` развернуто 10 подов, а сервис получил 1000 запросов, то каждый под обработает около 100 запросов.

Далее мы обсудим способ создания сервиса `ClusterIP` с помощью YAML-манифеста.

Декларативный способ создания сервисов ClusterIP

Сервисы ClusterIP можно создавать и декларативно, применяя YAML-манифесты через `kube-apiserver`.

Ниже приведен YAML-манифест, с помощью которого можно создать тот же сервис `ClusterIP`, который мы создали ранее императивным способом:

```
# clusterip-service.yaml
apiVersion: v1
kind: Service
metadata:
  name: nginx-clusterip
spec:
  type: ClusterIP    # Указывает, что сервис относится к типу ClusterIP
  ports:
    - port: 80       # Порт, открываемый сервисом
      protocol: TCP
      targetPort: 80 # Целевой порт подов
  selector:
    run: nginx-clusterip
```

Обратите внимание на комментарии в YAML-манифесте, особенно относящиеся к полям `port` и `targetPort`.

У сервиса `ClusterIP` есть собственный порт, который не зависит от того, какой порт открыт на стороне пода. Вы обращаетесь к сервису `ClusterIP` по его DNS-имени и номеру порта, а затем трафик перенаправляется на порт пода, выбираемого на основе соответствия метки и селектора.

Помните, что порты рабочих узлов в данном случае не задействуются. Порты, используемые сервисами `ClusterIP`, не связаны с портами хостовой машины!

Прежде чем двигаться дальше, очистим окружение, удалив сервис `nginx-clusterip`:

```
$ kubectl delete svc nginx-clusterip
service "nginx-clusterip" deleted
```

Учтите, что удаление сервиса не приведет к удалению публикуемых им подов. Поды необходимо удалять отдельно. Далее вы познакомитесь с headless-сервисами — еще одним ресурсом, имеющим отношение к сервисам типа `ClusterIP`.

Headless-сервисы

Технически headless-сервис (или «безголовый» сервис) представляет собой не отдельный тип сервиса (вроде `NodePort`), а особый режим работы сервиса `ClusterIP`.

Его можно активизировать, задав значение `None` для параметра `.spec.clusterIP` в YAML-файле для сервиса `ClusterIP`. Вот пример, основанный на предыдущем манифесте:

```
# clusterip-headless.yaml
apiVersion: v1
kind: Service
metadata:
  name: nginx-clusterip-headless
spec:
  clusterIP: None
  type: ClusterIP    # Указывает, что сервис относится к типу ClusterIP
  ports:
    - port: 80       # Порт, открываемый сервисом
      protocol: TCP
      targetPort: 80 # Целевой порт подов
  selector:
    run: nginx-clusterip
```

По сути, headless-сервис — это сервис `ClusterIP`, который не выполняет балансировку нагрузки и не имеет заранее выделенного кластерного IP-адреса. В связи с этим логика балансировки нагрузки и взаимодействия с подами не определяется системой Kubernetes.

Поскольку headless-сервис не имеет IP-адреса, доступ к подам, развернутым за ним, осуществляется напрямую, без проксирования трафика и балансировки нагрузки. Такой сервис возвращает DNS-имена публикуемых им подов, благодаря чему вы можете обращаться к ним напрямую. Определенная логика балансировки нагрузки при этом все-таки используется, но она реализуется на уровне DNS, а не средствами Kubernetes.

При использовании обычного сервиса `ClusterIP` вы всегда обращаетесь к одному статичному IP-адресу, закрепленному за сервисом, который перенаправляет запросы к подам. В случае headless-сервиса система Kubernetes просто возвращает список DNS-имен подов, а клиент сам выбирает, к какому из них обращаться.

Как правило, headless-сервисы Kubernetes применяются в сценариях, требующих прямого взаимодействия с отдельными подами в обход единой точки входа и балансировщика нагрузки.

Они особенно полезны для организации взаимодействия с кластеризованными сервисами с сохранением состояния, например с *LDAP*-серверами (Lightweight Directory Access Protocol — легковесный протокол доступа к каталогам). В таких случаях может потребоваться, чтобы LDAP-клиент имел доступ к отдельным DNS-именам подов, на которых запущен LDAP-сервер. Обычный сервис типа `ClusterIP` для этого не подходит, так как он предоставляет статический IP-адрес и реализует балансировку нагрузки средствами Kubernetes. Далее вы познакомитесь еще с одним типом сервиса.

Сервис типа LoadBalancer

Работа сервисов типа `LoadBalancer` зависит от облачной платформы, на которой развернут кластер Kubernetes. То есть для их использования необходимо разместить кластер в облаке, поддерживающем сервисы данного типа.

На облачных платформах, предоставляющих внешние балансировщики нагрузки, указание значения `LoadBalancer` в поле `type` приводит к созданию такого балансировщика для вашего сервиса. Этот процесс выполняется асинхронно, а сведения о созданном балансировщике становятся доступны в поле `.status.loadBalancer` объекта Service.

Некоторые облачные провайдеры позволяют задать параметр `loadBalancerIP` для создания балансировщика с указанным IP-адресом. Если параметр `loadBalancerIP` не указан, балансировщику назначается эфемерный IP-адрес. Однако если задать `loadBalancerIP` на платформе, не поддерживающей эту возможность, данный параметр будет проигнорирован.

Для сервиса типа `LoadBalancer` также доступно поле `.spec.loadBalancerClass`, которое позволяет использовать реализацию балансировщика нагрузки, отличную от предоставляемой облачным провайдером по умолчанию. Если значение `loadBalancerClass` не задано, Kubernetes использует стандартный балансировщик провайдера (при условии, что кластер сконфигурирован с его указанием через флаг `--cloud-provider`). При заданном параме-

тре `.spec.loadBalancerClass` реализация балансировщика соответствующего класса активно отслеживает сервисы. В этом случае стандартная реализация балансировщика (например, предоставляемая облачным провайдером) будет игнорировать сервисы с заданным в этом поле значением. Важно отметить, что значение в поле `spec.loadBalancerClass` можно задать только для сервиса типа `LoadBalancer`, после чего его уже нельзя будет изменить. Значение `spec.loadBalancerClass` должно соответствовать формату идентификатора в стиле метки. Допускается использование префикса, например `internal-vip` или `example.com/internal-vip`. Имена без префиксов зарезервированы для конечных пользователей.

Принцип работы сервиса типа `LoadBalancer` в Kubernetes проиллюстрирован на рис. 8.8.

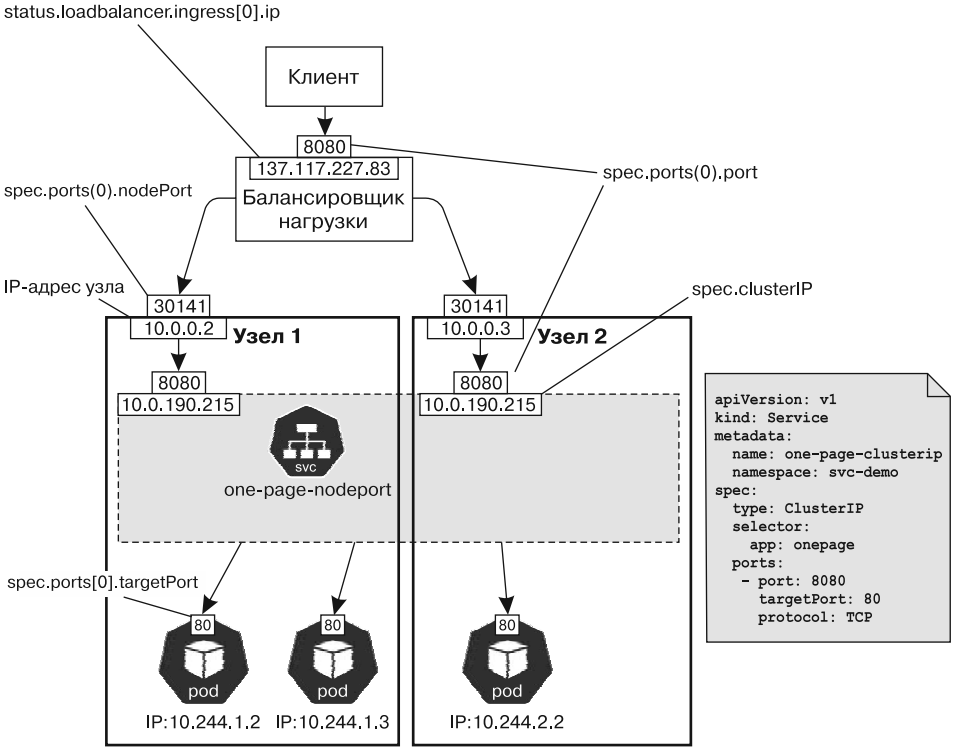

Рис. 8.8. Сервис типа LoadBalancer

В следующем разделе мы перечислим облачных провайдеров, поддерживающих сервисы типа `LoadBalancer`.

Облачные провайдеры, поддерживающие сервисы LoadBalancer

Не все облачные провайдеры поддерживают сервисы типа `LoadBalancer`. Среди поддерживающих можно выделить:

- AWS;
- GCP;
- Azure;
- OpenStack.

Этот список не является исчерпывающим, но, как видите, в трех крупнейших публичных облачных провайдерах такая поддержка есть.

Если ваш облачный провайдер входит в их число, учтите, что логика балансировки нагрузки реализуется самим провайдером. Kubernetes не позволит вам контролировать процесс распределения трафика между подами, и вам придется разбираться в принципе работы балансировщика нагрузки конкретного облачного провайдера. Фактически его стоит рассматривать как сторонний компонент, доступный в Kubernetes в виде ресурса.

Стоит ли использовать сервис LoadBalancer

На этот вопрос сложно дать однозначный ответ, но многие специалисты предпочитают не использовать сервисы этого типа.

Главная причина заключается в том, что сервисы `LoadBalancer` практически невозможно настроить средствами Kubernetes. Если вы работаете в облаке, то для настройки балансировщика нагрузки лучше использовать инструменты самого провайдера. Сервис `LoadBalancer` в Kubernetes является общим механизмом выделения балансировщика нагрузки, но он не предоставляет всех возможностей, которые может предложить облачный провайдер.

Кроме того, использование сторонних балансировщиков нагрузки, как правило, сопряжено с дополнительными затратами, которые могут значительно варьироваться в зависимости от провайдера и объема обрабатываемого трафика.

В следующем разделе мы обсудим сервис типа `ExternalName`.

Сервис типа ExternalName

Сервисы типа `ExternalName` связывают кластер Kubernetes с такими внешними ресурсами, как базы данных, API-интерфейсы и сервисы, размещенные за пределами кластера. Они сопоставляют сервис внутри кластера с DNS-именем, а не с подами, что позволяет приложениям, работающим в кластере, беспре-

пятственно обращаться к внешнему ресурсу, не зная его IP-адреса и деталей внутреннего устройства.

Ниже приведен пример YAML-манифеста для создания сервиса типа `ExternalName`:

```
# externalname-service.yaml
apiVersion: v1
kind: Service
metadata:
  name: mysql-db
  namespace: prod
spec:
  type: ExternalName
  externalName: app-db.database.example.com
```

В данном случае вместо связывания внешнего имени или IP-адреса с внутренними подами вы просто указываете DNS-имя, например `app-db.database.example.com`, в конфигурации сервиса. После этого при обращении к `mysql-db` DNS-сервис кластера будет перенаправлять запросы внутрикластерных приложений к внешней базе данных. С точки зрения приложений взаимодействие выглядит так же, как и с любым другим сервисом. При этом перенаправление запросов осуществляется на уровне DNS, что делает процесс максимально прозрачным (рис. 8.9).

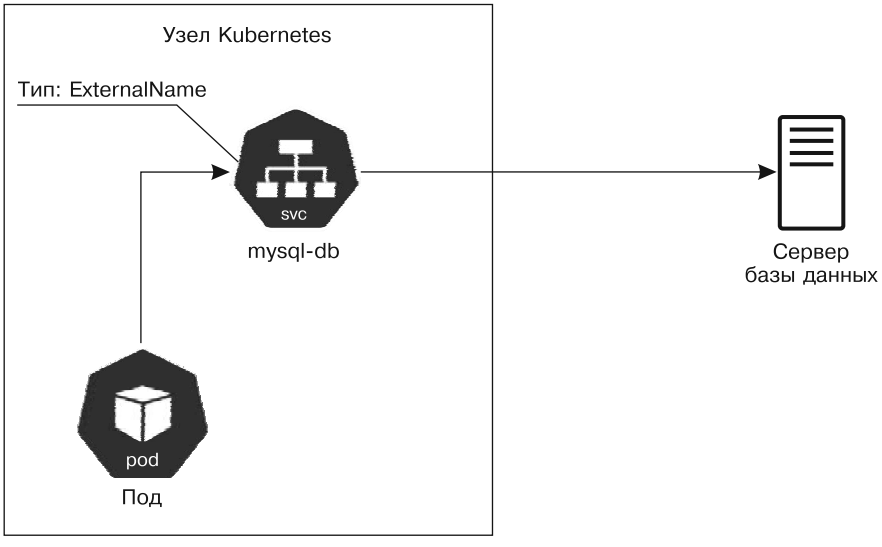

Рис. 8.9. Сервис типа ExternalName

Сервисом типа `ExternalName` может быть сервис, работающий в другом пространстве имен, сервис, расположенный за пределами кластера Kubernetes, сервис, развернутый в другом кластере, и т. д.

Такой подход имеет ряд преимуществ.

- *Упрощенная конфигурация*. Приложениям необходимо знать только имя сервиса, они не нуждаются в подробных сведениях о внешнем ресурсе, что значительно упрощает настройку.
- *Гибкое управление ресурсами*. Если позднее вы перенесете базу данных в свой кластер, вам достаточно будет обновить конфигурацию сервиса и управлять им внутри кластера, не затрагивая работу приложений.
- *Повышенная безопасность*. Конфиденциальные сведения, такие как IP-адреса, остаются скрытыми внутри кластера, что повышает общий уровень безопасности.

Помните, что сервисы `ExternalName` предназначены исключительно для подключения к внешним ресурсам. Для организации доступа к внутренним ресурсам кластера следует использовать обычные или headless-сервисы.

Теперь, когда вы познакомились с различными типами сервисов Kubernetes, перейдем к рассмотрению проб, которые используются для проверки их доступности.

Обеспечение работоспособности сервисов с помощью проверок

Когда вы создаете сервис для открытия доступа к приложению, работающему в подах, Kubernetes не проверяет его работоспособность автоматически. Поды могут находиться в состоянии `Running`, но само приложение при этом может работать некорректно, а сервис продолжит направлять к нему трафик. В результате пользователи или другие приложения столкнутся с ошибками либо вовсе останутся без ответа. Для предотвращения таких ситуаций в Kubernetes предусмотрены механизмы проверки состояния приложений, также называемые пробами. В данном разделе мы поговорим о трех типах таких проверок, и о том, как они помогают обеспечить работоспособность вашего приложения.

Что такое проверка готовности и зачем она нужна

Проверка готовности (`ReadinessProbe`) наряду с проверкой жизнеспособности (`LivenessProbe`) является важным инструментом, который вам следует освоить для обеспечения максимально качественного пользовательского опыта. Для начала посмотрим, как можно реализовать проверку `ReadinessProbe` и как она помогает убедиться в готовности контейнеров обслуживать трафик.

Технически проверки готовности не являются частью сервисов Kubernetes, однако их важно изучать именно в контексте работы этих сервисов.

Как и все в Kubernetes, проверка `ReadinessProbe` была реализована для решения конкретной проблемы. Она позволяет убедиться в готовности пода, прежде чем начать направлять на него трафик, в том числе поступающий от сервиса.

Принцип работы сервисов очень прост: они распределяют трафик между всеми подами, соответствующими их селектору меток. Новый под, чьи метки соответствуют селектору, немедленно начинает получать входящие запросы. Однако такой подход порождает проблему: если приложение в контейнере запускается медленно (например, из-за ожидания внешнего API или выполнения подготовительных операций), оно может оказаться не готово обрабатывать пользовательские запросы в момент начала получения трафика. Это приводит к ошибкам и ухудшает пользовательский опыт.

Чтобы этого не происходило, в описании пода можно настроить проверку `ReadinessProbe`.

Под, предусматривающий проверку готовности, может послать управляющему слою сигнал о том, что он еще не готов обслуживать трафик. В этом случае сервисы не будут перенаправлять на него запросы. Посмотрим, как можно реализовать такую проверку.

Реализация проверки готовности

Чтобы реализовать проверку готовности, достаточно добавить соответствующие конфигурационные данные в YAML-манифест пода. Следует подчеркнуть, что эта проверка никак не связана с самим объектом Service. Добавляя особые параметры в спецификацию контейнера, входящего в состав объекта Pod, мы приказываем Kubernetes не направлять трафик сервисов на этот под до успешного прохождения проверки готовности.

Проверка `ReadinessProbe` предусматривает три варианта.

- *Команда*. Выполнение команды внутри пода. Код выхода `0` указывает на готовность пода.
- *HTTP*. Отправка HTTP-запроса. О готовности пода свидетельствует код ответа `>=200` и `<400`.
- *TCP*. Попытка установления TCP-соединения. Если соединение успешно установлено, под считается готовым к обработке трафика.

Вот пример YAML-файла для пода `nginx` с HTTP-проверкой готовности:

```
# nginx-pod-with-readiness-http.yaml
apiVersion: v1
kind: Pod
metadata:
  name: nginx-pod-with-readiness-http
```

```
spec:
  containers:
    - name: nginx-pod-with-readiness-http
      image: nginx
      readinessProbe:
        initialDelaySeconds: 5
        periodSeconds: 5
        httpGet:
          path: /ready
          port: 80
```

Обратите внимание на два важных параметра в разделе `readinessProbe`:

- `initialDelaySeconds` — время ожидания перед запуском первой проверки (в секундах);
- `periodSeconds` — интервал между двумя последовательными проверками (в секундах).

Проверка готовности будет выполняться регулярно с периодичностью, определяемой параметром `periodSeconds`.

В нашем примере проверка готовности предполагает выполнение HTTP-запроса к пути `/ready`. Если ответ содержит HTTP-код `>= 200` и `< 400`, проверка считается успешно пройденной, а под — работоспособным.

Помните, что проверка `ReadinessProbe` предназначена для того, чтобы удостовериться в готовности приложения принимать трафик. Поэтому стоит вызывать такую конечную точку, которая отражает фактическое состояние приложения. Например, чтобы убедиться в готовности приложения взаимодействовать с базой данных, можно вызывать страницу, открывающую внутреннее соединение с MySQL. Разработчикам имеет смысл создавать отдельную конечную точку для проверки соединений с различными бэкендами.

После успешного прохождения проверки под включается в пул, обслуживаемый сервисом, и начинает принимать трафик. Проверка `ReadinessProbe` также может быть проведена путем установления TCP-соединения или выполнения команды, но эти примеры мы рассмотрим в следующем подразделе в контексте обсуждения проверки жизнеспособности.

Что такое проверка жизнеспособности и зачем она нужна

Проверка жизнеспособности (`LivenessProbe`) во многом похожа на проверку готовности (`ReadinessProbe`). Если вы когда-либо работали с облачными провайдерами, то, вероятно, уже знакомы с механизмом проверки работоспособности (health check). По сути, `LivenessProbe` представляет собой именно его.

Проверки `LivenessProbe` позволяют оценить работоспособность пода и особенно актуальны в случае долгоживущих процессов вроде веб-сервисов. Представьте ситуацию: у вас есть сервис, который направляет трафик на три пода, и один из них выходит из строя. Сам сервис не сможет это обнаружить и продолжит равномерно распределять трафик между всеми тремя подами, включая неисправный. В результате 33 % запросов будут завершаться ошибкой, что ухудшит пользовательский опыт (рис. 8.10).

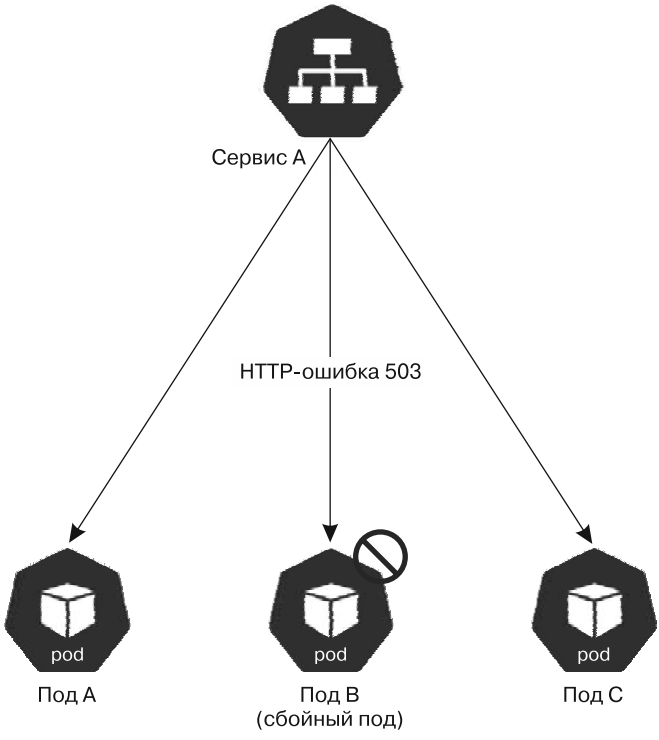

Рис. 8.10. Один из подов вышел из строя, но сервис продолжает пересылать ему трафик

Чтобы избежать подобных ситуаций, необходим механизм, позволяющий выявлять сбойные поды и исключать соответствующие контейнеры из пула, обслуживаемого сервисом.

Проверка `LivenessProbe` решает именно эту задачу и настраивается на уровне пода. Однако учтите, что `LivenessProbe` не может «починить» под. Она лишь выявляет его неработоспособность и инициирует его завершение. Далее мы обсудим способы реализации такой проверки.

Реализация проверки жизнеспособности

Проверка жизнеспособности проводится по расписанию для отслеживания состояния приложения в долгосрочной перспективе. Эта проверка выполняется компонентом `kubelet` и предусматривает несколько вариантов.

- *Команда*. Выполнение команды в контейнере. Код выхода, равный `0`, означает, что под работает корректно.
- *HTTP*. Выполнение HTTP-запроса к поду. Код ответа `>=200` и `<400` говорит о работоспособности пода.
- *TCP*. Попытка установить TCP-соединение. Если соединение успешно установлено, значит, под работает нормально.
- *gRPC*. Если приложение поддерживает и реализует протокол проверки работоспособности gRPC (gRPC Health Checking Protocol).

Для любой проверки жизнеспособности необходимо задать параметр `periodSeconds` (в виде целого числа), указывающий интервал между последовательными проверками (в секундах). Дополнительно можно использовать параметр `initialDelaySeconds`, определяющий задержку перед выполнением первой проверки. Это важно, так как в некоторых случаях из-за слишком ранней проверки «неработоспособным» может быть признано приложение, которое просто еще не успело полностью запуститься.

Конфигурация `LivenessProbe` задается в YAML-манифесте пода (а не сервиса). При этом каждый контейнер в поде может предусматривать собственную проверку.

Проверка жизнеспособности с помощью HTTP-запроса

HTTP-проверки в Kubernetes предусматривают дополнительные настраиваемые поля: `host`, `scheme`, `path`, `httpHeaders`, `port`, позволяющие регулировать способ выполнения запросов для проверки состояния приложения. Ниже приведен пример манифеста, который проверяет работоспособность пода, выполняя HTTP-запрос к конечной точке `/healthcheck` в контейнере `nginx`:

```
# nginx-pod-with-liveness-http.yaml
apiVersion: v1
kind: Pod
metadata:
  name: nginx-pod-with-liveness-http
spec:
  containers:
    - name: nginx-pod-with-liveness-http
      image: nginx
      livenessProbe:
```

```
      initialDelaySeconds: 5
      periodSeconds: 5
      httpGet:
        path: /healthcheck
        port: 80
        httpHeaders:
          - name: My-Custom-Header
            value: My-Custom-Header-Value
```

Обратите внимание на ключевые параметры в блоке `livenessProbe`. Система будет ждать 5 секунд перед выполнением первой проверки состояния, а затем каждые 5 секунд отправлять один HTTP-запрос к пути `/healthcheck` на порте `80`. В запрос добавляется пользовательский HTTP-заголовок, помогающий идентифицировать проверки состояния в журналах доступа. Учтите, что путь `/healthcheck`, скорее всего, отсутствует в контейнере `nginx`, поэтому этот контейнер никогда не будет признан работоспособным, так как проверка `LivenessProbe` будет возвращать HTTP-ответ `404`. Для успешного прохождения HTTP-проверки состояния необходимо, чтобы ответ содержал код `>= 200` и `< 400`. Поскольку код `404` не попадает в этот диапазон, под будет считаться неработоспособным.

Проверка жизнеспособности с помощью команды

Проверить состояние пода можно и с помощью команды. Сделаем это на примере того же YAML-манифеста:

```
# nginx-pod-with-liveness-command.yaml
apiVersion: v1
kind: Pod
metadata:
  name: nginx-pod-with-liveness-command
spec:
  containers:
    - name: nginx-pod-with-liveness-command
      image: nginx
      livenessProbe:
        initialDelaySeconds: 5
        periodSeconds: 5
        exec:
          command:
            - cat
            - /hello/world
```

Как видите, этот вариант гораздо проще по сравнению с HTTP-проверкой. В данном случае каждые 5 секунд выполняется команда `cat /hello/world`. Если файл существует и команда завершается с кодом выхода `0`, проверка считается успешно пройденной. Если файла нет, проверка завершится ошибкой, под будет признан неработоспособным и отключен.

Проверка жизнеспособности путем установки TCP-соединения

В данной ситуации выполняется попытка установить соединение с TCP-сокетом на порте `80`. В случае успеха проверка жизнеспособности считается пройденной, а контейнер — готовым к работе. В противном случае проверка завершается с ошибкой и под отключается. Вот пример соответствующей конфигурации:

```
# nginx-pod-with-liveness-tcp.yaml
apiVersion: v1
kind: Pod
metadata:
  name: nginx-pod-with-liveness-tcp
spec:
  containers:
    - name: nginx-pod-with-liveness-tcp
      image: nginx
      livenessProbe:
        initialDelaySeconds: 5
        periodSeconds: 5
        tcpSocket:
          port: 80
```

TCP-проверка во многом похожа на HTTP-проверку, так как протокол HTTP работает поверх TCP. Она особенно полезна для оценки работоспособности приложения, которое не использует HTTP в качестве протокола, а также в тех случаях, когда применение HTTP-запросов не имеет смысла, например при проверке состояния соединения с LDAP.

Использование именованного порта в рамках TCP- и HTTP-проверки

Для настройки проверки жизнеспособности типа TCP и HTTP (но не gRPC) можно использовать именованный порт, как показано ниже:

```
# nginx-pod-with-liveness-http-named-port.yaml
apiVersion: v1
kind: Pod
metadata:
  name: nginx-pod-with-liveness-http-named-port
spec:
  containers:
    - name: nginx-pod-with-liveness-http-named-port
      image: nginx
      ports:
        - name: liveness-port
          containerPort: 8080
          hostPort: 8080
      livenessProbe:
        initialDelaySeconds: 5
        periodSeconds: 5
        httpGet:
          path: /healthcheck
          port: liveness-port
```

В этом примере порт `liveness-port` определяется в разделе `ports`, а затем задействуется в разделе `httpGet` для проведения проверки `livenessProbe`.

Теперь обсудим проверку успешности запуска.

Проверка успешности запуска

Первичный запуск устаревших приложений может занимать больше времени. Это создает определенную проблему при настройке проверок жизнеспособности, так как для выявления взаимных блокировок критически важна быстрая реакция.

Решением является использование параметра `initialDelaySeconds` или проведение отдельной проверки успешности запуска (`startupProbe`). Параметр `initialDelaySeconds` позволяет отложить выполнение первой проверки готовности и дать приложению время на инициализацию.

Однако для более точного контроля рекомендуется использовать проверку `startupProbe`. Она работает так же, как и проверка жизнеспособности (выполняемая с помощью команды, HTTP- или TCP-запроса), но предусматривает более длинный интервал ожидания (`failureThreshold * periodSeconds`), гарантирующий то, что у приложения будет достаточно времени для инициализации, прежде чем оно будет признано готовым к приему трафика. После успешного запуска проверка `livenessProbe` выполняется в обычном режиме, обеспечивая быстрое обнаружение сбоев (см. следующий YAML-файл):

```
# nginx-pod-with-startupprobe.yaml
apiVersion: v1
kind: Pod
metadata:
  name: nginx-pod-with-startupprobe
spec:
  containers:
    - name: nginx-pod-with-startupprobe
      image: nginx
      ports:
        - name: liveness-port
          containerPort: 8080
          hostPort: 8080
      livenessProbe:
        initialDelaySeconds: 5
        periodSeconds: 5
        httpGet:
          path: /healthcheck
          port: liveness-port
      startupProbe:
        httpGet:
          path: /healthz
          port: liveness-port
        failureThreshold: 30
        periodSeconds: 10
```

Как видите, комбинирование проверок позволяет убедиться в том, что приложение действительно готово обрабатывать трафик. В следующем подразделе мы посмотрим, как можно сочетать проверки `ReadinessProbe` и `LivenessProbe`.

Совместное использование проверок готовности и жизнеспособности

Проверки готовности (`ReadinessProbe`) и жизнеспособности (`LivenessProbe`) можно выполнять в отношении одного и того же пода.

Эти проверки выполняют разные функции, но настраиваются почти одинаково, поэтому их совместное использование вполне оправданно. Они имеют следующие общие параметры:

- `initialDelaySeconds` — период ожидания перед выполнением первой проверки (в секундах);
- `periodSeconds` — интервал между последовательными проверками (в секундах);
- `timeoutSeconds` — время ожидания перед тайм-аутом (в секундах);
- `successThreshold` — количество успешных проверок, необходимых для признания пода готовым (в случае `ReadinessProbe`) или работоспособным (в случае `LivenessProbe`);
- `failureThreshold` — количество неудачных проверок, необходимых для признания пода неготовым (в случае `ReadinessProbe`) или подлежащим отключению (в случае `LivenessProbe`);
- `terminationGracePeriodSeconds` — время, отведенное контейнеру на корректное завершение работы до принудительной остановки (значение по умолчанию наследуется от параметра, заданного на уровне пода).

На этом мы завершаем главу, посвященную сервисам Kubernetes, методам их реализации и способам проверки готовности и жизнеспособности.

Резюме

Глава получилась весьма насыщенной и предоставила большой объем информации о сетевых взаимодействиях в Kubernetes. Как и поды, сервисы являются краеугольным камнем Kubernetes, и их освоение критически важно для успешной работы с этим оркестратором.

В этой главе мы выяснили, что в момент создания поду динамически назначается уникальный IP-адрес. Для обеспечения надежного подключения к подам в Kubernetes используется прокси под названием Service. Мы также рассмотрели разные типы сервисов Kubernetes и их назначение. Кроме того, мы обсудили

проверки `ReadinessProbe` и `LivenessProbe`, которые позволяют контролировать работоспособность и гарантировать направление трафика только к тем подам, которые готовы его обрабатывать.

В следующей главе вы продолжите изучение основ Kubernetes и познакомитесь с концепциями PersistentVolume и PersistentVolumeClaim, которые применяются для работы с постоянными данными. Эта глава будет особенно интересна тем, кто собирается создавать и развертывать приложения с сохранением состояния в кластере Kubernetes, например базы данных или файловые хранилища.

Дополнительные источники информации

- Сервисы, балансировка нагрузки и сетевое взаимодействие: https://kubernetes.io/docs/concepts/services-networking/.
- Headless-сервисы Kubernetes: https://kubernetes.io/docs/concepts/services-networking/service/#headless-services.
- Настройка проверок жизнеспособности, готовности и успешности запуска: https://kubernetes.io/docs/tasks/configure-pod-container/configure-liveness-readiness-startup-probes/.
- Сетевые политики: https://kubernetes.io/docs/concepts/services-networking/network-policies/.
- Объявление сетевой политики: https://kubernetes.io/docs/tasks/administer-cluster/declare-network-policy/.
- Сетевая политика по умолчанию: https://kubernetes.io/docs/concepts/services-networking/network-policies/#default-policies.
- Механизм EndpointSlice: https://kubernetes.io/docs/concepts/services-networking/endpoint-slices/.
- Сетевые плагины: https://kubernetes.io/docs/concepts/extend-kubernetes/compute-storage-net/network-plugins/.
- Сетевое взаимодействие в кластере: https://kubernetes.io/docs/concepts/cluster-administration/networking/.

9 Постоянное хранилище в Kubernetes

В предыдущих главах вы познакомились с некоторыми ключевыми концепциями Kubernetes. Вы также узнали, что эта система позволяет задавать желаемое состояние для всех традиционных уровней ИТ-инфраструктуры путем создания объекта в хранилище `etcd`, который затем преобразуется в реальные вычислительные ресурсы внутри кластера.

В этой главе мы поговорим о постоянном хранилище данных для приложений с сохранением состояния. Как и в случае с другими абстракциями ресурсов, для организации постоянного хранилища в кластере используется набор объектов. В Kubernetes оно реализуется с помощью ресурса типа PersistentVolume, обладающего особой механикой. Не переживайте, если она покажется вам сложной, далее мы подробно разберем все ее особенности.

Технические требования

Для работы с главой вам потребуется:

- локальный или облачный кластер Kubernetes;
- рабочий интерфейс командной строки `kubectl`, настроенный для взаимодействия с кластером.

Инструкции по созданию кластера Kubernetes и настройке клиента `kubectl` приведены в главах 2 и 3.

Примеры кода для этой главы можно загрузить из официального репозитория GitHub по адресу https://github.com/PacktPublishing/The-Kubernetes-Bible-Second-Edition/tree/main/Chapter09.

Назначение постоянного хранилища данных

Хранилище данных является важным ресурсом в мире ИТ, поскольку обеспечивает логический способ выполнения *CRUD*-операций, то есть *создания*, *чтения*, *обновления* и *удаления* всевозможной информации — от зарплатных ведомостей

до медицинских записей. Хотя хранилище играет ключевую роль в предоставлении пользователям актуальной информации, контейнеры и микросервисы не должны хранить состояние. Поэтому данные работающего контейнера становятся недоступными после его повторного запуска или переноса в другой кластер. То же самое относится и к микросервисам: компонент, отвечающий за хранение данных, должен быть отделен, чтобы микросервис не зависел от состояния данных и их доступности при повторном запуске.

Так где же хранить данные приложения? В специализированном хранилище. С точки зрения обеспечения непрерывности бизнеса, если такое хранилище работает в том же кластере Kubernetes, что и микросервисы, оно должно иметь механизм репликации, учитывающий специфику приложения. При этом важно помнить, что Kubernetes является оркестратором ресурсов, который управляет приложениями в соответствии с заданным желаемым состоянием. При настройке подов вы можете указать конкретное хранилище, предоставив контейнерам возможность создавать, читать, обновлять и удалять данные. Теперь посмотрим, какие способы постоянного хранения данных предлагает Kubernetes.

Знакомство с томами

Самый простой способ организации хранилища в Kubernetes предполагает монтирование объектов в контейнеры в виде томов. Это можно реализовать для:

- объекта ConfigMap;
- объекта Secret;
- токена ServiceAccount (разновидности секрета).

Такой подход позволяет команде разработчиков отделить конфигурацию микросервиса от контейнера или манифеста развертывания. На протяжении жизненного цикла приложения учетные данные, сертификаты или токены для внешних сервисов, а также конфигурационные параметры могут нуждаться в обновлении. По соображениям безопасности их не стоит прописывать в манифестах развертывания или в образах контейнеров.

Рассмотрим пример объекта ConfigMap, создаваемого с помощью манифеста `nginx-configmap.yaml`:

```
# nginx-configmap.yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: nginx-hello
  labels:
    app: test
immutable: false
```

```
data:
  hello1.html: |
    <html>
    hello world 1
    </html>
  hello2.html: |
    <html>
    hello world 2
    </html>
```

Здесь определены два файла, которые будут смонтированы в под `nginx` с помощью манифеста `nginx-pod.yaml`:

```
# nginx-pod.yaml
apiVersion: v1
kind: Pod
metadata:
  name: nginx-hello
  labels:
    app: test
spec:
  containers:
    - name: nginx
      image: nginx:1.14.2
      ports:
        - containerPort: 80
      volumeMounts:
        - name: nginx-hello
          mountPath: "/usr/share/nginx/html/hello"
  volumes:
    - name: nginx-hello
      configMap:
        name: nginx-hello
```

Применим оба манифеста:

```
$ kubectl apply -f nginx-configmap.yaml
configmap/nginx-hello created
$ kubectl apply -f nginx-pod.yaml
pod/nginx-hello created
```

А затем проверим состояние созданных объектов:

```
$ kubectl get pod,cm
NAME              READY   STATUS    RESTARTS   AGE
pod/nginx-hello   1/1     Running   0          7m26s

NAME                          DATA   AGE
configmap/kube-root-ca.crt    1      7d17h
configmap/nginx-hello         2      7m31s
```

Убедимся, что файлы доступны в каталоге `/usr/share/nginx/hello`, указанном в качестве пути монтирования:

```
$ kubectl exec -t pod/nginx-hello -- ls -al /usr/share/nginx/html/hello/
total 12
... <фрагмент опущен для краткости> ...
lrwxrwxrwx 1 root root 18 Sep 7 21:19 hello1.html -> ..data/hello1.html
lrwxrwxrwx 1 root root 18 Sep 7 21:19 hello2.html -> ..data/hello2.html
```

А теперь убедимся, что данные действительно предоставляются сервером Nginx, выполнив перенаправление портов, чтобы не настраивать сервис:

```
$ kubectl port-forward nginx-hello 8080:80
Forwarding from 127.0.0.1:8080 -> 80
Forwarding from [::1]:8080 -> 80
```

Затем во втором терминале можно выполнить curl-запросы к обоим URL-адресам:

```
$ curl 127.0.0.1:8080/hello/hello1.html
<html>
  hello world 1
</html>

$ curl 127.0.0.1:8080/hello/hello2.html
<html>
  hello world 2
</html>
```

Это неплохое начало, однако у таких объектов есть ограничение по объему хранимых данных. Поскольку они зависят от хранилища `etcd`, во избежание проблем с производительностью этот лимит составляет 1,5 Мбайт (https://etcd.io/docs/v3.5/dev-guide/limit). Следующая группа объектов позволяет приложению хранить данные, объем которых ограничен емкостью хранилища хостовой системы.

Представим кластер Kubernetes с двумя рабочими узлами, на которых могут размещаться поды, и рассмотрим пять типов томов:

- `emptyDir`;
- `hostPath`;
- локальный том;
- блочный диск, подключаемый по протоколу *Fiber Channel (FC)*;
- том, экспортируемый по протоколу *Network File System (NFS)*.

Первые три типа: `emptyDir`, `hostPath` и локальные тома — имеют два существенных ограничения:

- они ограничены объемом дискового пространства рабочего узла, для которого были выделены;
- они привязаны к узлу, на котором развернут под. Если под создан на рабочем узле 1, то и данные будут храниться только на нем.

Использование этих типов томов может привести к деградации сервиса или даже к более серьезным последствиям, например к возникновению проблемы разделенного мозга (split-brain). Если рабочий узел 1 выйдет из строя и под будет переназначен на узел 2, приложение запустится без своих данных, что может вызвать серьезный сбой.

Обратите внимание, что некоторые приложения имеют встроенный механизм репликации. Типичная конфигурация развертывания такого приложения предполагает запуск двух реплик и создание тома `hostPath` на каждом узле. В таком случае отказ одного рабочего узла приведет к снижению доступности и производительности приложения, но не к его полной неработоспособности.

Будучи внешними по отношению к вычислительным ресурсам кластера Kubernetes, два последних типа, блочный диск FC и том NFS, устраняют упомянутые ранее ограничения, но привносят дополнительную сложность. Первые три типа томов не требуют взаимодействия с администраторами хранилища, чего нельзя сказать о последних двух. Если вкратце, то вашим администраторам хранилища необходимо будет выделить:

- *номер логической единицы хранения* (Logical Unit Number, LUN, — блочный диск FC) в *сети хранения данных* (Storage Area Network, SAN), подключенной к рабочим узлам Kubernetes через сеть FC-коммутаторов, и открыть к нему доступ путем настройки зонирования;
- пространство данных в *сетевом хранилище* (Network Attached Storage, NAS), подключенном к корпоративной сети и доступном для рабочих узлов Kubernetes, и открыть к нему доступ через политику экспорта.

Учтите, что использование этих двух типов томов требует специализированного оборудования и нетривиальной настройки, хотя NAS все чаще используется и в домашних лабораториях. Однако с точки зрения Kubernetes, настройка таких томов мало чем отличается от примера с ConfigMap. Ниже приведены измененные версии определения пода `nginx`.

- Для тома FC (`nginx-pod-fiberchannel.yaml`):

```
...
    - name: nginx
      image: nginx:1.14.2
```

```
      ports:
        - containerPort: 80
      volumeMounts:
        - name: fc-vol
          mountPath: "/usr/share/nginx/html/hello"

  volumes:
    - name: fc-vol
      fc:
        targetWWNs:
          - 500a0982991b8dc5
          - 500a0982891b8dc5
        lun: 2
        fsType: ext4
        readOnly: true
```

Параметры SAN и LUN должны быть заданы в разделе `fc`.

- Для тома NFS (`nginx-pod-nfs-volume.yaml`):

```
...
  containers:
    - name: nginx
      image: nginx:1.14.2
      ports:
        - containerPort: 80
      volumeMounts:
        - name: nfs-volume
          mountPath: "/usr/share/nginx/html/hello"

  volumes:
    - name: nfs-volume
      nfs:
        server: nfs.corp.mycompany.org
        path: /k8s-nginx-hello
        readOnly: true
```

Параметры NAS и экспортируемого тома должны быть заданы в разделе `nfs`.

Обратите внимание на следующее.

- Оба типа томов, блочный диск FC и NFS, будут подключаться к узлам по мере необходимости, обусловленной наличием пода.
- Несмотря на то что эти два типа томов решают ряд задач, их использование считается антипаттерном с точки зрения принципа отделения конфигураций от ресурсов.
- В то время как ConfigMap монтируется в виде тома с двумя HTML-файлами напрямую в контейнер, другие типы томов требуют применения иного подхода к предоставлению данных.
- Существует множество других типов томов, о которых можно узнать здесь: https://kubernetes.io/docs/concepts/storage/volumes/.

Концепция томов в Kubernetes является отличной отправной точкой для развертывания приложений с сохранением состояния. Однако из-за ограничений одних томов, сложности других и потребности в специальных знаниях в области хранения данных масштабирование сотен или тысяч микросервисов с помощью таких объектов является весьма затруднительным. Благодаря дополнительному уровню абстракции Kubernetes предлагает независимый от технологий способ организации масштабного использования хранилищ с помощью объекта PersistentVolume, который мы рассмотрим в следующем подразделе.

Объект PersistentVolume

Подобно поду и объекту ConfigMap, объект PersistentVolume (PV, постоянный том) представляет собой ресурс, доступный через компонент `kube-apiserver`. Вы можете создавать, обновлять и удалять *постоянные тома* с помощью YAML-манифестов и утилиты `kubectl`, как и любые другие объекты Kubernetes.

Список объектов PersistentVolume, уже созданных в кластере Kubernetes, можно вывести с помощью следующей команды:

```
$ kubectl get persistentvolume
No resources found
```

Для перечисления объектов данного типа также можно использовать множественную форму `persistentvolumes` и псевдоним `pv`. Следующие три команды, по сути, эквивалентны:

```
$ kubectl get persistentvolume
No resources found
$ kubectl get persistentvolumes
No resources found
$ kubectl get pv
No resources found
```

Стоит отметить, что псевдоним `pv` очень широко распространен в мире Kubernetes, и, говоря о постоянных томах, многие люди используют именно это сокращение. В настоящее время в нашем кластере Kubernetes объекты PersistentVolume отсутствуют, поэтому в выводе предыдущей команды эти ресурсы не отображаются.

Объект PersistentVolume представляет собой участок хранилища, подключаемый к поду. Этот участок называется *постоянным*, потому что он не привязан к жизненному циклу пода.

В главе 5 вы узнали о том, что Kubernetes поддерживает концепцию томов, а также познакомились с томами типа `emptyDir`, которые создают пустой каталог, доступный для совместного использования несколькими подами, и определяют

путь в файловой системе рабочего узла, предоставляемый этим подам. Оба типа томов привязаны к жизненному циклу пода. Это означает, что при удалении пода данные, хранящиеся в таких томах, также будут уничтожены.

Однако иногда бывает необходимо сохранить том и содержащиеся в нем данные даже в случае сбоя пода. Именно для этого используются объекты PersistentVolume, представляющие собой тома, не привязанные к жизненному циклу пода. Поскольку постоянный том является ресурсом, как и сами поды, он существует независимо от них. По сути, объект PersistentVolume гарантирует то, что хранилище останется доступным и после завершения работы пода, что критически важно для обеспечения целостности данных в приложениях с сохранением состояния. Объекты PersistentVolume включают в себя два ключевых элемента: базовую технологию хранения (тип PersistentVolume) и режим доступа (например, *ReadWriteOnce (RWO)*). Понимание этих концепций необходимо для эффективного использования постоянных томов в среде Kubernetes.

Помните, что объекты PersistentVolume — это записи в хранилище etcd, а не сами диски.

PersistentVolume представляет собой своеобразный указатель на участок хранилища, например NFS, диск, том Amazon Elastic Block Store (EBS) и т. д. Это позволяет задействовать соответствующие технологии в системе Kubernetes стандартными для нее методами.

В следующем подразделе вы познакомитесь с типами постоянных томов.

Типы объектов PersistentVolume

Вы уже знаете, что простейшая конфигурация Kubernetes может представлять собой базовую установку `minikube`, в то время как наиболее сложная может включать десятки серверов в рамках масштабируемой инфраструктуры. Разумеется, такие разнообразные варианты развертывания требуют разных способов управления постоянным хранилищем. Три крупнейших публичных облачных провайдера предлагают для этого собственные решения:

- тома Amazon EBS;
- файловые системы Amazon *Elastic File System (EFS)*;
- диски Google GCE *Persistent Disk (PD)*;
- диски Microsoft Azure.

Каждое из этих решений предусматривает особые принципы, архитектуру и механизмы работы. Система Kubernetes изначально была призвана абстрагировать все подобные варианты с помощью единого объекта, скрывающего различия базовых технологий. Именно таким объектом и является ресурс PersistentVolume,

который подключается к работающему поду. Будучи объектом Kubernetes, под не знает, что такое EBS или PD; он взаимодействует только с объектом PersistentVolume, который также является ресурсом Kubernetes.

Где бы ни работал ваш кластер Kubernetes — на платформе Google GKE, Amazon EKS или на локальной машине, для управления постоянным хранилищем вы будете создавать и развертывать объекты PersistentVolume, а затем связывать их с подами.

По умолчанию система Kubernetes поддерживает целый ряд технологий хранения, включая следующие:

- `csi` — CSI (Container Storage Interface — интерфейс хранилища контейнеров);
- `fc` — FC-хранилище;
- `iscsi` — SCSI поверх IP;
- `local` — использование локального хранилища;
- `hostPath` — тома `hostPath`;
- `nfs` — сетевое файловое хранилище.

Приведенный выше список не является исчерпывающим: система Kubernetes универсальна и поддерживает множество решений для хранения данных, которые можно абстрагировать в виде объектов PersistentVolume в кластере.

Однако учтите, что в последних версиях Kubernetes несколько типов PersistentVolume были признаны устаревшими или полностью удалены. Это отражает изменения в подходах к управлению хранилищем в системе Kubernetes и является признаком ее эволюции, направленной на упрощение API и улучшение совместимости с современными решениями для хранения данных.

В частности, с момента выхода версии Kubernetes 1.29 были удалены или объявлены устаревшими следующие типы PersistentVolume:

- `awsElasticBlockStore` — том Amazon EBS;
- `azureDisk` — том Azure Disk;
- `azureFile` — том Azure File;
- `portworxVolume` — том Portworx;
- `flexVolume` — том FlexVolume;
- `vsphereVolume` — том vSphere VMDK;
- `cephfs` — том CephFS;
- `cinder` — том Cinder.

Эти изменения отражают более широкую тенденцию к использованию стандартизированных интерфейсов хранилища и переносимых, не зависящих от облачного провайдера решений. Подробную информацию об объекте PersistentVolume

и его типах, поддерживаемых системой Kubernetes, можно найти в официальной документации: https://kubernetes.io/docs/concepts/storage/persistent-volumes/#types-of-persistent-volumes.

Преимущества объектов PersistentVolume

Объекты PersistentVolume (PV) являются ключевым инструментом Kubernetes для управления приложениями с сохранением состояния. В отличие от эфемерного хранилища PV гарантируют сохранность данных за пределами жизненного цикла отдельных подов, что делает их оптимальным выбором для долговременного хранения данных приложений и обеспечения их согласованности. Эти ресурсы привносят гибкость и надежность в экосистему Kubernetes, повышая производительность и устойчивость приложений.

Использование объекта PersistentVolume дает три основных преимущества.

- Постоянный том продолжает существовать в системе Kubernetes независимо от пода, который его использует. Это значит, что в случае удаления или пересоздания пода, подключенного к объекту PersistentVolume, данные, содержащиеся в этом томе, не исчезают. Сохранность данных зависит от политики освобождения ресурса (`reclaimPolicy`), заданной для объекта PersistentVolume. При выборе политики `Retain` данные остаются доступными для повторного использования, а выбор политики `Delete` приводит к удалению тома и содержащихся в нем данных при удалении пода. Таким образом, вы можете управлять подами, не опасаясь потери данных, хранящихся в постоянных томах.
- При сбое в работе пода объект PersistentVolume не удаляется из кластера.
- Объект PersistentVolume доступен на уровне кластера, то есть может быть подключен к любому поду, запущенному на любом узле (соответствующие методы и ограничения мы рассмотрим далее в этой главе).

Однако учтите, что эти три утверждения не всегда оказываются на 100 % верными, так как объект PersistentVolume может зависеть от используемой технологии хранения.

В качестве примера рассмотрим объект PersistentVolume, представляющий собой указатель на хранилище `hostPath`, находящееся на конкретном вычислительном узле. В такой конфигурации постоянный том не будет доступен другим узлам.

При использовании NFS ситуация иная, так как к NFS можно одновременно обращаться с нескольких машин. Поэтому объект PersistentVolume, основанный на NFS, может без проблем подключаться к нескольким подам, работающим на разных узлах. Чтобы сделать постоянный том доступным для разных узлов, нужно понимать концепцию режимов доступа, которую мы обсудим в следующем разделе.

Режимы доступа, поддерживаемые объектом PersistentVolume

Режим доступа — это параметр, который задается при создании объекта PersistentVolume и определяет способ монтирования тома.

Объект PersistentVolume поддерживает четыре режима доступа.

- *ReadWriteOnce (RWO)*. В любой момент времени только один узел может получить доступ к тому для чтения и записи.
- *ReadOnlyMany (ROX)*. Том доступен только для чтения нескольким узлам одновременно.
- *ReadWriteMany (RWX)*. Том доступен для чтения и записи нескольким узлам одновременно.
- *ReadWriteOncePod*. Новый режим, стабилизированный в версии Kubernetes 1.29. В этом режиме том доступен для чтения/записи только одному поду во всем кластере. Используйте этот режим в тех случаях, когда необходимо, чтобы только один под в кластере имел возможность читать данные из объекта *PersistentVolumeClaim (PVC)* или записывать их в него.

Для каждого объекта PersistentVolume необходимо указать как минимум один режим доступа, даже если сам том поддерживает несколько. Режимы, поддерживаемые разными типами PersistentVolume, показаны в табл. 9.1.

Таблица 9.1. Режимы доступа, поддерживаемые различными типами PersistentVolume (Источник: kubernetes.io/docs/concepts/storage/persistent-volumes)

Плагин	ReadWriteOnce	ReadOnlyMany	ReadWriteMany	ReadWrite-OncePod
AzureFile	✓	✓	✓	—
CephFS	✓	✓	✓	—
CSI	Зависит от драйвера	Зависит от драйвера	Зависит от драйвера	Зависит от драйвера
FC	✓	✓	—	—
FlexVolume	✓	✓	Зависит от драйвера	—
HostPath	✓	–	—	—
iSCSI	✓	✓	—	—
NFS	✓	✓	✓	—
RBD	✓	✓	—	—

Плагин	ReadWriteOnce	ReadOnlyMany	ReadWriteMany	ReadWrite-OncePod
VsphereVolume	✓	—	— (Работает при совместном размещении подов)	—
PortworxVolume	✓	—	✓	—

В Kubernetes режимы доступа PersistentVolume тесно связаны с используемой технологией хранения и тем, как она управляет данными. Вот почему разные типы PV поддерживают разные режимы.

Файловое и блочное хранилище

- Файловое хранилище (например, *Network File System (NFS)* или *Common Internet File System (CIFS)*) позволяет нескольким клиентам одновременно обращаться к одним и тем же файлам. Поэтому файловые системы могут поддерживать разнообразные режимы доступа, например RWO, ROX и RWX. Они изначально предназначались для организации многопользовательского доступа по сети и обеспечения одновременного чтения и записи данных несколькими узлами без повреждения информации.
- Блочное хранилище (например, локальное хранилище или `hostPath`) имеет принципиальное отличие. Оно рассчитано на использование одним клиентом, так как работает с секторами диска, а не с файлами. Одновременный доступ нескольких клиентов приводит к несогласованности или повреждению данных. По этой причине блочное хранилище поддерживает только режим RWO, позволяющий одному узлу читать и записывать данные в том.

Внутреннее и внешнее хранилище

- Тома `hostPath`, которые используют хранилище на том же узле, где выполняется рабочая нагрузка, изначально ограничены этим узлом. Поскольку такое хранилище привязано к физическому узлу, к нему нельзя получить доступ с других узлов кластера. По этой причине том `hostPath` совместим исключительно с режимом доступа ReadWriteOnce (RWO).
- В отличие от них такие решения, как NFS или иные системы внешнего хранения данных, изначально спроектированы для организации сетевого доступа, позволяющего нескольким узлам одновременно использовать одно и то же хранилище. Благодаря такой гибкости они поддерживают дополнительные режимы доступа, в частности ReadWriteMany (RWX).

Понимание этого различия проясняет, почему одни типы PersistentVolume поддерживают более гибкие режимы доступа, а другие остаются ограниченными.

Создание первого объекта PersistentVolume

Создадим объект PersistentVolume в кластере Kubernetes, используя декларативный подход. Поскольку постоянные тома относятся к более сложным ресурсам, применять императивный метод при их создании крайне не рекомендуется. Декларативный подход позволяет определять и управлять ресурсами с помощью YAML-файлов, что упрощает отслеживание изменений, контроль версий и обеспечивает воспроизводимость конфигураций в разных средах. Кроме того, данный подход облегчает управление такими крупными и сложными ресурсами, как PersistentVolume, требующее точной настройки и тщательного планирования.

Вот пример YAML-манифеста для создания объекта PersistentVolume:

```
---
apiVersion: v1
kind: PersistentVolume
metadata:
  name: pv-hostpath
  labels:
    type: local
spec:
  storageClassName: manual
  capacity:
    storage: 1Gi
  accessModes:
    - ReadWriteOnce
  hostPath:
    path: "/mnt/data"
```

Это самый простой вариант постоянного тома. По сути, данный YAML-файл создает запись PersistentVolume в кластере Kubernetes. В этом примере PersistentVolume относится к типу `hostPath`.

Объект PersistentVolume типа hostPath не рекомендуется использовать в рабочих средах, а также при запуске критически важных рабочих нагрузок. В данном примере он используется исключительно в демонстрационных целях.

Применить конфигурацию PV можно следующим образом:

```
$ kubectl apply -f pv-hostpath.yaml
persistentvolume/pv-hostpath created
```

Разумеется, объект PersistentVolume может представлять собой нечто более сложное, например облачный диск или NFS, но его простейшей формой является том типа `hostPath` на узле, где выполняется ваш под.

Управление хранением данных с помощью объектов PersistentVolume

Как мы уже выяснили, ресурс PersistentVolume является указателем на место хранения, которым может быть, например, диск, NFS или дисковый том, управляемый сторонним оператором. Все эти технологии работают по-разному. К счастью, в Kubernetes все они представлены через объект PersistentVolume.

Таким образом, YAML-манифесты для создания объектов PersistentVolume будут немного различаться в зависимости от технологии, лежащей в основе хранилища. Например, если вы хотите, чтобы PersistentVolume указывал на общий ресурс NFS, вам необходимо соблюсти два условия:

- общий ресурс NFS должен быть предварительно настроен и доступен для узлов Kubernetes;
- YAML-манифест для PersistentVolume должен содержать сведения об NFS-сервере и соответствующем общем ресурсе.

Ниже приведен пример YAML-манифеста для создания объекта PersistentVolume, использующего NFS в качестве технологии хранения данных:

```
# pv-nfs.yaml
apiVersion: v1
kind: PersistentVolume
metadata:
  name: pv-nfs
spec:
  capacity:
    storage: 5Gi
  volumeMode: Filesystem
  accessModes:
    - ReadWriteOnce
  persistentVolumeReclaimPolicy: Recycle
  storageClassName: slow
  mountOptions:
    - hard
    - nfsvers=4.1
  nfs:
    path: /appshare
    server: nfs.example.com
```

Чтобы объект PersistentVolume работал корректно, он должен обеспечить связь между системой Kubernetes и реальным хранилищем. Для этого необходимо создать или выделить участок хранилища за пределами Kubernetes, а затем создать объект PersistentVolume, указав уникальный идентификатор диска или тома, предоставляемого внешней по отношению к Kubernetes технологией хранения данных. В следующем разделе мы рассмотрим пример YAML-манифеста для создания объекта PersistentVolume на основе блочного тома.

Создание объекта PersistentVolume на основе блочного тома

Следующий YAML-манифест описывает объект PersistentVolume, указывающий на блочный том:

```
# pv-block.yaml
apiVersion: v1
kind: PersistentVolume
metadata:
  name: pv-block
spec:
  capacity:
    storage: 100Gi
  accessModes:
    - ReadWriteOnce
  volumeMode: Block
  persistentVolumeReclaimPolicy: Retain
  fc:
    targetWWNs: ["50060e801049cfd1"]
    lun: 0
    readOnly: false
```

Как видите, раздел `fc` этого YAML-манифеста содержит сведения о FC-томе, на который ссылается данный объект PersistentVolume. Конкретный блочный том определяется ключом `targetWWNs`. Такого YAML-манифеста системе Kubernetes достаточно для нахождения соответствующего всемирного имени *WWN* (World Wide Name) и поддержания ссылки на него.

Теперь поговорим о выделении ресурсов хранилища.

Может ли Kubernetes выделять или создавать ресурсы хранилища самостоятельно

Необходимость по отдельности создавать ресурсы хранилища и объекты PersistentVolume может показаться утомительной.

К счастью, благодаря способности взаимодействовать с *API* облачного провайдера и другими технологиями хранения Kubernetes может создать тома или диски на лету. Для этого используется механизм *динамического выделения постоянных томов*, который существенно упрощает управление объектами PersistentVolume. Однако он работает только с поддерживаемыми технологиями хранения и облачными провайдерами.

Эту более продвинутую тему мы подробно обсудим далее в главе.

Теперь, когда мы разобрались со способом создания объектов PersistentVolume внутри кластера, давайте поговорим об их монтировании. Перед использова-

нием постоянного тома его необходимо подключить к поду. Для этого используется промежуточный объект PersistentVolumeClaim, который мы обсудим в следующем разделе.

Монтирование объекта PersistentVolume в под

Чтобы смонтировать объект PersistentVolume в под, нам понадобится еще один важный ресурс — PersistentVolumeClaim.

Объект PersistentVolumeClaim

Подобно PersistentVolume и ConfigMap, PersistentVolumeClaim (PVC, заявка на выделение постоянного тома) является самостоятельным ресурсом в кластере Kubernetes.

Несмотря на схожесть названий, PersistentVolume и PersistentVolumeClaim — это два разных ресурса, представляющих разные сущности.

Вывести список объектов PersistentVolumeClaim, созданных в кластере, можно с помощью следующей команды `kubectl`:

```
$ kubectl get persistentvolumeclaims
No resources found in default namespace.
```

Вывод говорит о том, что в нашем кластере пока нет ресурсов PersistentVolumeClaim. Обратите внимание, что эта команда также работает с псевдонимом `pvc`:

```
$ kubectl get pvc
No resources found in default namespace.
```

Многие специалисты, работающие с Kubernetes, обозначают объекты PersistentVolumeClaim как `pvc`, поэтому не удивляйтесь, если встретите это сокращение в документации или в ходе практической работы. Теперь разберемся, что же представляют собой объекты PersistentVolumeClaim в Kubernetes.

Разделение этапов создания и потребления ресурсов хранилища

Ключом к пониманию различий между PersistentVolume и PersistentVolumeClaim является то, что первый объект представляет само хранилище, а второй — запрос пода на использование этого хранилища.

Причина данного разделения заключается в том, что Kubernetes обычно используется двумя категориями специалистов.

- *Администратор Kubernetes* поддерживает и эксплуатирует кластер, а также добавляет вычислительные ресурсы и ресурсы постоянного хранилища.
- *Разработчик Kubernetes-приложений* разрабатывает и развертывает приложения, то есть использует вычислительные ресурсы и хранилище, предоставленные администратором.

Разумеется, в организации один и тот же человек может совмещать обе роли, однако это разграничение важно для понимания процесса подключения объектов PersistentVolume к подам.

Система Kubernetes изначально проектировалась с учетом того, что объект PersistentVolume будет относиться к сфере ответственности администратора кластера, а объект PersistentVolumeClaim — к сфере ответственности разработчика приложений. Добавление PersistentVolume (или использование операторов динамического выделения томов) является задачей администратора, поскольку эти тома могут представлять собой аппаратные ресурсы. Разработчики же лучше понимают, какой объем и какой тип хранилища необходим приложению, поэтому для них был создан объект PersistentVolumeClaim.

По сути, под не может смонтировать объект PersistentVolume напрямую. Для этого он должен явно запросить его. Такой запрос осуществляется через создание объекта PersistentVolumeClaim и его привязку к поду, которому требуется PersistentVolume.

Именно для этого и существует дополнительный уровень абстракции. В следующем разделе мы рассмотрим процесс использования объекта PersistentVolume.

Процесс использования объекта PersistentVolume

После создания приложения разработчик при необходимости может запросить объект PersistentVolume. Для этого создаются два YAML-манифеста:

- один — для пода или объекта Deployment;
- другой — для объекта PersistentVolumeClaim.

Манифест пода должен быть написан так, чтобы объект PersistentVolumeClaim подключался к нему через ключ конфигурации `volumeMount` в YAML-файле. Учтите, что для корректной работы объект PersistentVolumeClaim должен находиться в том же пространстве имен, что и под приложения, в который он монтируется. После применения обоих YAML-манифестов и создания соответствующих ресурсов в кластере объект PersistentVolumeClaim начнет поиск объекта PersistentVolume, соответствующего требованиям, указанным в заявке. Если в кластере Kubernetes присутствует готовый к использованию объект

PersistentVolume, способный удовлетворить запрос, он будет автоматически связан с объектом PersistentVolumeClaim.

Если все прошло успешно, запрос считается выполненным, а том — корректно подключенным к поду. Понимание этого процесса, по сути, означает понимание всей логики работы с PersistentVolume в Kubernetes.

Следующая схема иллюстрирует рабочий процесс статического выделения ресурсов хранилища в Kubernetes (рис. 9.1).

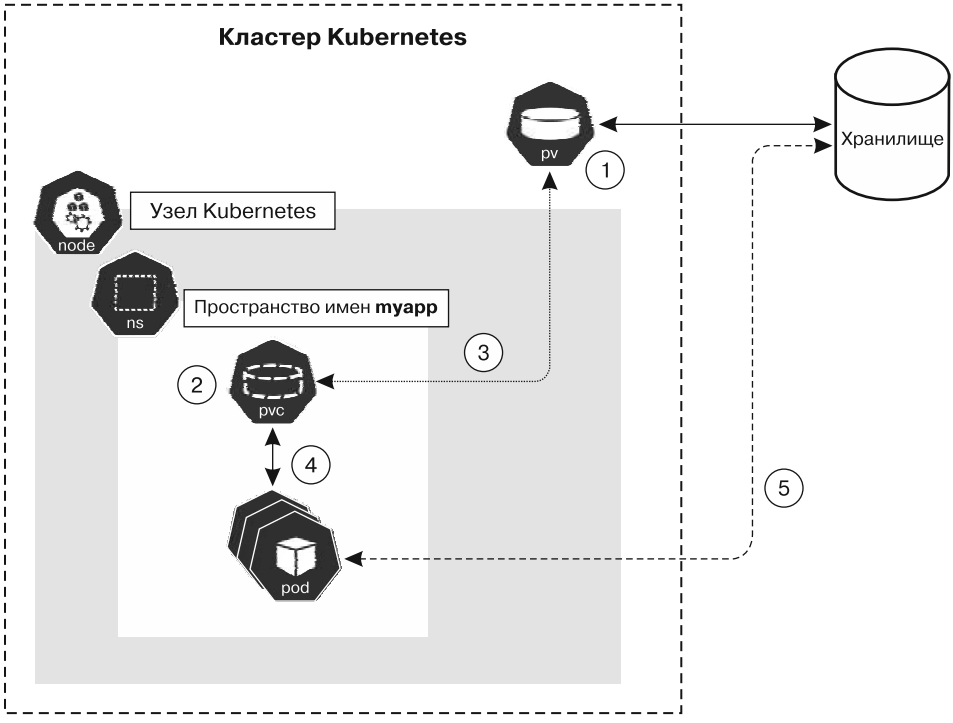

Рис. 9.1. Статическое выделение ресурсов хранилища в Kubernetes

О механизме динамического выделения ресурсов хранилища мы поговорим далее в этой главе, в подразделе «Динамическое выделение ресурсов».

Если разработчику требуется постоянное хранилище для приложения, работающего в Kubernetes, последовательность его действий будет такой.

1. *Подготовка объекта PersistentVolume*. Администратор Kubernetes подготавливает базовое хранилище и создает объект PersistentVolume (PV). Этот PV выступает в качестве декларации, задающей такие параметры хранилища,

как емкость, режим доступа (чтение/запись, только чтение) и используемая технология хранения (например, `hostPath`, NFS).

2. *Создание запроса с помощью PersistentVolumeClaim*. Разработчик создает объект PersistentVolumeClaim (PVC), который представляет собой запрос на выделение хранилища с указанием таких его параметров, как размер, режим доступа и предпочтительный класс хранилища. Кроме того, в YAML-манифесте пода разработчик указывает способ монтирования тома, определяющий то, как под будет получать доступ к постоянному хранилищу.
3. *Обработка запроса*. После создания пода и PVC система Kubernetes ищет среди имеющихся ресурсов PV, соответствующий требованиям, указанным в PVC.
4. *Использование хранилища через* volumeMount. Обнаружив подходящий PV, система Kubernetes связывает его с PVC, после чего хранилище становится доступным для пода.
5. *Обработка потока данных (операции чтения/записи)*. Теперь под может взаимодействовать с постоянным хранилищем в соответствии с режимом доступа, заданным в PV. Он может выполнять операции чтения или записи данных в том, обеспечивая их сохранность даже при перезапуске пода.

Обратите внимание, что объект PersistentVolume является кластерным ресурсом, тогда как PersistentVolumeClaim, под и volumeMount относятся к уровню пространства имен.

Такое взаимодействие между PV, PVC и Kubernetes гарантирует разработчикам доступ к постоянному хранилищу для приложений, позволяя им сохранять и извлекать данные на протяжении жизненного цикла подов.

На первый взгляд эта схема может показаться сложной, но вы быстро к ней привыкнете.

В следующем разделе мы рассмотрим процесс использования хранилища в подах с помощью объектов PersistentVolume и PersistentVolumeClaim.

Создание пода с объектом PersistentVolumeClaim

В этом разделе мы создадим под, который монтирует том типа `hostPath` в кластере `minikube`. Этот том будет представлять собой разновидность объекта PersistentVolume, имеющую собственный жизненный цикл, независимый от пода.

Первым делом нужно создать объект PersistentVolume типа `hostPath`. Чтобы впоследствии объект PersistentVolumeClaim мог найти этот том, мы добавим произвольные метки в раздел `metadata` его YAML-манифеста:

```
# pv.yaml
apiVersion: v1
kind: PersistentVolume
metadata:
  name: my-hostpath-pv
  labels:
    type: hostpath
    env: prod
spec:
  capacity:
    storage: 1Gi
  accessModes:
    - ReadWriteOnce
  hostPath:
    path: "/tmp/test"
  storageClassName: slow
```

При сопоставлении PV с PVC будут использоваться следующие элементы YAML-файла:

- `labels`;
- `capacity`;
- `accessModes`;
- `storageClassName`.

Теперь мы можем создать объект PersistentVolume и вывести список доступных в кластере ресурсов этого типа. В этом списке должен присутствовать созданный нами объект. Обратите внимание, что данная команда работает и с псевдонимом `pv`:

```
$ kubectl apply -f pv.yaml
persistentvolume/my-hostpath-pv created

$ kubectl get pv
NAME             CAPACITY   ACCESS MODES   RECLAIM POLICY   STATUS      CLAIM
STORAGECLASS   VOLUMEATTRIBUTESCLASS   REASON   AGE
my-hostpath-pv   1Gi        RWO            Retain           Available
slow           <unset>                          3s
```

Как видите, объект PersistentVolume был успешно создан и имеет статус `Available`.

Далее нам нужно создать два ресурса для монтирования объекта PersistentVolume:

- объект PersistentVolumeClaim, нацеленный на этот конкретный PersistentVolume;
- под, который будет использовать созданный объект PersistentVolumeClaim.

Чтобы продемонстрировать разницу между ресурсами уровня пространства имен и ресурсами уровня кластера, создадим отдельное пространство имен для PVC и пода (см. файл `pv-ns.yaml`):

```
$ kubectl apply -f pv-ns.yaml
namespace/pv-ns created
```

Теперь можно переходить к созданию объекта `PersistentVolumeClaim`:

```
# pvc.yaml
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: my-hostpath-pvc
  namespace: pv-ns
spec:
  resources:
    requests:
      storage: 1Gi
  accessModes:
    - ReadWriteOnce
  selector:
    matchLabels:
      type: hostpath
      env: prod
  storageClassName: slow
```

Создадим объект PVC и убедимся в том, что он появился в кластере. Учтите, что эта команда также работает с псевдонимом `pvc`:

```
$ kubectl apply -f pvc.yaml
persistentvolumeclaim/my-hostpath-pvc created


$ kubectl get pvc -n pv-ns
NAME              STATUS   VOLUME           CAPACITY   ACCESS MODES
STORAGECLASS   VOLUMEATTRIBUTESCLASS   AGE
my-hostpath-pvc   Bound    my-hostpath-pv   1Gi        RWO            slow
<unset>                 2m29s
```

Статус `Bound` говорит о том, что PVC уже сопоставлен с PV и готов к использованию ресурсов хранилища.

При наличии объектов PersistentVolume и PersistentVolumeClaim мы можем создать под, который будет монтировать PV через PVC.

Для решения этой задачи создадим под `nginx`:

```
# pod.yaml
apiVersion: v1
kind: Pod
```

```
metadata:
  name: nginx
  namespace: pv-ns
spec:
  containers:
    - image: nginx
      name: nginx
      volumeMounts:
        - mountPath: "/var/www/html"
          name: mypersistentvolume
  volumes:
    - name: mypersistentvolume
      persistentVolumeClaim:
        claimName: my-hostpath-pvc
```

Как видно в разделе `volumeMounts`, объект PersistentVolumeClaim указан как том, при этом мы ссылаемся на PVC по его имени. Учтите, что объект PVC должен находиться в том же пространстве имен, что и под, который его монтирует. Это связано с тем, что PVC является *ресурсом уровня пространства имен*, в то время как PV относится к кластерным ресурсам. В данном случае мы не используем метки или селекторы: чтобы привязать объект PVC к поду, достаточно указать его имя.

Таким образом, под будет привязан к объекту PersistentVolumeClaim, который найдет соответствующий объект PersistentVolume. В результате том `hostPath` будет смонтирован в под `nginx` и доступен для него.

Создадим под и проверим его статус:

```
$ kubectl apply -f pod.yaml
pod/nginx created

$ kubectl get pvc,pod -n pv-ns
NAME                                  STATUS   VOLUME           CAPACITY   ACCESS
MODES   STORAGECLASS   VOLUMEATTRIBUTESCLASS   AGE
persistentvolumeclaim/my-hostpath-pvc   Bound    my-hostpath-pv   1Gi     RWO
slow           <unset>                 4m32s

NAME        READY   STATUS    RESTARTS   AGE
pod/nginx   1/1     Running   0          13s
```

Как видите, под успешно запущен и том `/tmp/test` смонтирован в него через PV и PVC. Теперь вы знаете, что представляют собой объекты PersistentVolume и PersistentVolumeClaim и как использовать их для подключения постоянного хранилища к подам.

Поскольку объекты PersistentVolume и PersistentVolumeClaim существуют независимо от подов, их жизненные циклы имеют особенности, которые мы рассмотрим далее.

Жизненный цикл объекта PersistentVolume в Kubernetes

Объекты PersistentVolume позволяют сохранять состояние приложения, не привязывая его к жизненному циклу подов или контейнеров, в которых оно выполняется.

Однако у таких объектов есть собственный жизненный цикл, особенности которого важно учитывать при их использовании.

Почему объекты PersistentVolume не связаны с пространствами имен

Прежде всего, следует помнить о том, что PersistentVolume не является объектом уровня пространства имен, в отличие от PersistentVolumeClaim.

```
$ kubectl api-resources --namespaced=false | grep -i volume
persistentvolumes     pv   v1                     false   PersistentVolume
volumeattachments          storage.k8s.io/v1      false   VolumeAttachment
```

Поэтому для того, чтобы под мог использовать PersistentVolume, в его пространстве имен должен быть создан объект PersistentVolumeClaim.

Жизненный цикл типичного объекта PersistentVolume (PV) включает следующие этапы.

- *Выделение*. На этом этапе администратор создает объект PV, задавая емкость, режимы доступа и при необходимости дополнительные параметры, такие как класс хранилища и политика освобождения.
- *Состояние непривязанности*. Изначально PV доступен, но не привязан ни к одному поду.
- *Запрос на использование*. Разработчик создает заявку PVC, указывая размер, режим доступа и предпочтительный класс хранилища.
- *Сопоставление и привязка*. Система Kubernetes находит непривязанный PV, соответствующий требованиям, указанным в заявке PVC, и связывает их.
- *Использование*. Под получает доступ к привязанному PV через ключ `volumeMount`, определенный в YAML-манифесте.
- *Освобождение*. При удалении пода, использующего PVC, данная заявка отвязывается (состояние PV зависит от политики освобождения).
- *Удаление*. Объект PV может быть удален администратором в соответствии с заданной политикой освобождения ресурса хранилища.

Далее мы рассмотрим еще один важный аспект работы с PersistentVolume — политику освобождения, которая играет ключевую роль в отвязывании PVC от работающего пода.

Освобождение объекта PersistentVolume

Важным параметром объекта PersistentVolume является политика освобождения (reclaim policy).

Этот параметр указывает системе Kubernetes, что делать с объектом PersistentVolume при удалении объекта PersistentVolumeClaim, связывающего его с подами.

По сути, удаление объекта PersistentVolumeClaim означает разрыв связи между подами и объектом PersistentVolume. Этот процесс можно рассматривать как размонтирование тома, после которого он снова становится доступным для использования другим приложением.

Однако в некоторых случаях необходимо, чтобы объект PersistentVolume автоматически удалялся при удалении связанного с ним объекта PersistentVolumeClaim. Именно для этого и предусмотрена политика освобождения.

Рассмотрим три ее варианта.

- `Delete`. Самый простой вариант. При использовании этой политики освобождения объект PersistentVolume будет удален, а его запись исчезнет из кластера Kubernetes сразу после удаления соответствующего объекта PersistentVolumeClaim. Это удобно, если вы хотите уничтожить данные, чтобы другие приложения не могли их использовать. Учтите, что эта операция необратима, поэтому рекомендуется заранее настроить стратегию резервного копирования у вашего поставщика услуг хранения данных на случай необходимости их восстановления.

В нашем примере объект PV был создан вручную на основе hostPath по пути /tmp/. В данном случае операция удаления сработает без проблем. Однако при ручном создании PV некоторых типов операция удаления может выполняться некорректно. Поэтому настоятельно рекомендуется использовать механизм динамического выделения PV, о котором мы поговорим далее в этой главе.

- `Retain`. Вторая политика, противоположная `Delete`. При ее использовании объект PersistentVolume не удаляется в момент удаления соответствующего объекта PVC. Вместо этого PV переходит в состояние `Released`, то есть

остается в кластере, администратор которого может вручную извлечь все его данные.

- `Recycle`. Представляет собой комбинацию двух предыдущих. Сначала том очищается от всех данных (как при выполнении простой операции `rm -rf volume/*`) и остается доступным в кластере для очередного подключения к приложению.

Политика Recycle была объявлена устаревшей, поэтому вместо нее рекомендуется использовать механизм динамического выделения ресурсов.

Политику освобождения можно задать непосредственно в YAML-манифесте на уровне объекта PersistentVolume.

Обновление политики освобождения

Политика освобождения является настраиваемым параметром, который можно изменить и после создания объекта PersistentVolume.

Чтобы продемонстрировать различия между политиками, воспользуемся ранее созданными объектами:

```
$ kubectl get pod,pvc -n pv-ns
NAME        READY   STATUS    RESTARTS   AGE
pod/nginx   1/1     Running   0          30m

NAME                                    STATUS   VOLUME           CAPACITY
ACCESS MODES   STORAGECLASS   VOLUMEATTRIBUTESCLASS   AGE
persistentvolumeclaim/my-hostpath-pvc Bound    my-hostpath-pv   1Gi        RWO
slow           <unset>                 34m
```

Сначала удалим под, так как он использует объект PVC:

```
$ kubectl delete pod nginx -n pv-ns
pod "nginx" deleted
$ kubectl delete pvc my-hostpath-pvc -n pv-ns
persistentvolumeclaim "my-hostpath-pvc" deleted
```

Теперь проверим статус объекта PV:

```
$ kubectl get pv
NAME             CAPACITY   ACCESS MODES   RECLAIM POLICY   STATUS     CLAIM
STORAGECLASS   VOLUMEATTRIBUTESCLASS   REASON   AGE
my-hostpath-pv   1Gi        RWO            Retain           Released   pv-ns/
my-hostpath-pvc   slow           <unset>                          129m
```

Из вывода следует, что PV находится в состоянии `Released`, но еще не переведен в состояние `Available` для использования другим объектом PVC.

Изменим политику освобождения на `Delete`, используя команду `kubectl patch`:

```
$ kubectl patch pv/my-hostpath-pv -p '{"spec":{"persistentVolumeReclaimPolicy":
"Delete"}}'
persistentvolume/my-hostpath-pv patched
```

Поскольку PV не привязан ни к одному объекту PVC, он будет сразу удален при политике `Reclaim=Delete`:

```
$ kubectl get pv
No resources found
```

Как видите, мы изменили политику освобождения объекта PV, после чего этот объект был удален из кластера Kubernetes.

Теперь обсудим возможные статусы объектов PV и PVC.

Статусы объектов PersistentVolume и PersistentVolumeClaim

Подобно подам, которые могут находиться в разных состояниях вроде `Pending`, `ContainerCreating` и `Running`, объекты PersistentVolume и PersistentVolumeClaim тоже предусматривают различные статусы. Определить их можно с помощью команд `kubectl get pv` и `kubectl get pvc`.

Объект PersistentVolume может находиться в следующих состояниях.

- `Available`. Исходное состояние объекта PV, означающее его готовность к привязке к PVC.
- `Bound`. Этот статус указывает на то, что данный PV в настоящее время связан с определенным объектом PVC и используется подом. По сути, это означает, что том находится в работе. Если этот статус имеет объект PersistentVolumeClaim, это значит, что PVC используется подом, который получает через него доступ к PV.
- `Terminating`. В это состояние переходит объект PersistentVolumeClaim после выполнения команды `kubectl delete pvc`.
- `Released`. Если объект PVC, использующий PV, был удален (и при этом для PV задана политика освобождения `Retain`), PV переходит в это состояние. Оно означает, что том отвязан, но остается доступным для будущих PVC.
- `Failed`. Статус указывает на проблему с объектом PV, не позволяющую его использовать. Возможными причинами являются ошибки на стороне поставщика ресурсов хранилища, проблемы с доступом или сбои в работе механизма выделения (если применимо).
- `Unknown`. В редких случаях статус PV может быть неизвестным из-за сбоев связи с базовой системой хранения.

Итак, у вас есть все необходимые знания об объектах PersistentVolume и PersistentVolumeClaim для работы с постоянным хранилищем в Kubernetes. Однако мы еще не затронули такую важную тему, как динамическое выделение ресурсов. Это одна из самых впечатляющих возможностей Kubernetes, благодаря которой данная система может взаимодействовать с API облачного провайдера для создания постоянного хранилища в облаке, а затем делать его доступным в кластере путем динамического создания объектов PV. В следующем разделе мы сравним статическое и динамическое выделение ресурсов хранилища.

Статическое и динамическое выделение объектов PersistentVolume

До этого момента мы говорили только о статическом выделении объектов PersistentVolume. Теперь обсудим динамическое выделение, позволяющее создавать объекты PersistentVolume непосредственно из кластера Kubernetes.

Сравнение статического и динамического выделения

Как вы помните, при статическом выделении объектов PersistentVolume необходимо выполнить следующие действия.

1. Создать участок хранилища у облачного провайдера или в используемой системе хранения данных.
2. Создать объект PersistentVolume, который будет выступать в Kubernetes в роли указателя на это хранилище.
3. Создать под и объект PersistentVolumeClaim, позволяющий связать PV с подом.

Этот процесс называется статическим выделением, потому что участок хранилища должен быть создан до создания объектов PV и PVC в Kubernetes. Такой подход работает, однако при масштабировании управление существенно усложняется, особенно если речь идет о сотнях PV и PVC. Например, для создания тома Amazon EBS и его подключения в качестве PersistentVolume путем статического выделения вам придется сделать следующее.

1. Аутентифицироваться в консоли AWS.
2. Создать том EBS.
3. Скопировать его уникальный идентификатор в YAML-манифест PersistentVolume.
4. Создать PV с помощью этого YAML-файла.
5. Создать PVC для получения доступа к этому PV.
6. Смонтировать объект PVC в под.

Все эти действия вполне можно выполнить вручную или автоматизировать, однако при большом количестве ресурсов это становится чрезмерно сложным и отнимает слишком много времени.

Именно поэтому разработчики Kubernetes решили, что будет лучше, если сама система сможет выделять участок хранилища вместе с указывающим на него объектом PersistentVolume. Такой подход называется динамическим выделением.

Динамическое выделение ресурсов

При использовании механизма динамического выделения кластер Kubernetes первым делом аутентифицируется во внешней системе хранения (например, AWS, Azure или в системе другого поставщика). Затем вы выполняете команду для выделения диска или тома, в результате чего автоматически создается объект PersistentVolume, который может быть использован через PVC. Такой подход позволяет экономить много времени благодаря автоматизации. Полезность динамического выделения ресурсов обусловлена тем, что Kubernetes поддерживает широкий спектр технологий хранения. Некоторые из них, в частности NFS, мы уже упоминали в этой главе.

Но как же системе Kubernetes удается достичь такой универсальности? Ответ заключается в использовании такого ресурса, как объект StorageClass, о котором мы поговорим далее.

Знакомство с CSI

Прежде чем перейти к обсуждению объекта StorageClass, вы познакомитесь с CSI (Container Storage Interface — интерфейс хранилища контейнеров), который служит мостом между Kubernetes и различными системами хранения и определяет стандартный интерфейс для предоставления ресурсов хранилища контейнерным рабочим нагрузкам. CSI создает слой абстракции для взаимодействия с такими примитивами Kubernetes, как PersistentVolume, что позволяет интегрировать разные решения для хранения данных в Kubernetes, не привязываясь к конкретному поставщику.

Типичный процесс динамического выделения ресурсов хранилища в Kubernetes включает следующие этапы.

1. Установка и настройка объекта StorageClass и провиженера. Администратор устанавливает CSI-драйвер (или встроенный провиженер) и настраивает объект StorageClass, определяющий тип хранилища, его параметры и политику освобождения.
2. Создание PVC с информацией о StorageClass. Разработчик создает объект PersistentVolumeClaim, указывая требуемый размер тома, режим доступа

и ссылку на нужный объект StorageClass, чтобы запросить динамическое выделение ресурсов хранилища.

3. Отправка запроса провиженеру. Обнаружив объект PVC, система Kubernetes автоматически активизирует CSI-драйвер (или провиженер) и отправляет запрос на выделение ресурса во внешней системе хранения.
4. Взаимодействие провиженера с системой хранения для создания тома. Провиженер связывается с внешней системой хранения, создает том и генерирует объект PersistentVolume в Kubernetes, который привязывается к PVC.
5. Монтирование PVC в под для предоставления доступа к хранилищу. Объект PVC подключается к запрашивающему поду, предоставляя ему доступ к хранилищу согласно настройке `volumeMount` в конфигурации пода.

Следующая диаграмма иллюстрирует процесс динамического выделения объекта PV (рис. 9.2).

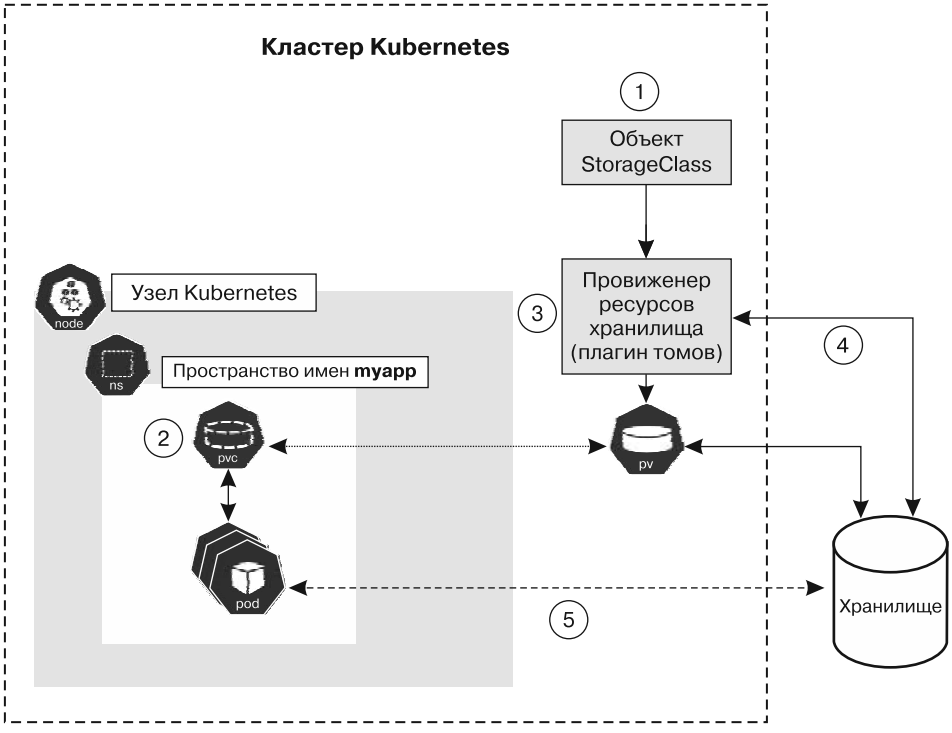

Рис. 9.2. Динамическое выделение PV в Kubernetes

CSI-драйверы — это предоставляемые поставщиками услуг хранения данных контейнеризованные реализации, которые соответствуют спецификации CSI и обеспечивают функциональность, связанную с выделением, подключением, отключением и управлением томами.

CSI Node и CSI Controller — это сервисы Kubernetes, которые выполняют логику CSI-драйвера соответственно на рабочих узлах и в управляющем слое, обеспечивая взаимодействие между подами и системой хранения.

После развертывания CSI-совместимого драйвера томов в кластере Kubernetes вы можете использовать тома типа `csi` (список CSI-драйверов, доступных в Kubernetes, можно найти в документации по адресу https://kubernetes-csi.github.io/docs/drivers.html). Это позволяет подключать или монтировать тома, предоставляемые CSI-драйвером. Существует три способа использования тома типа `csi` внутри пода.

- *Ссылка на PersistentVolumeClaim*. Этот подход предполагает связывание пода с постоянным хранилищем, управляемым Kubernetes.
- *Использование универсального эфемерного тома*. Этот метод предполагает предоставление временного хранилища, не сохраняемого при перезапуске пода.
- *Использование CSI-специфичного эфемерного тома (если он поддерживается драйвером)*. Этот вариант предоставляет специфичные для драйвера возможности временного хранения в дополнение к общим.

Помните, что вам не нужно взаимодействовать с CSI напрямую. Объект StorageClass может ссылаться на CSI-драйвер по имени в поле `provisioner`, используя его для выделения томов.

Знакомство с объектом StorageClass

StorageClass — это еще один тип ресурса, предоставляемого через `kube-apiserver`. Вероятно, вы уже замечали соответствующее поле в выводе команды `kubectl get pv`. Именно этот ресурс обеспечивает прозрачное взаимодействие Kubernetes с различными технологиями хранения.

StorageClass выступает в роли интерфейса, позволяющего пользователю задать требования к хранилищу. *CSI-драйверы*, на которые ссылаются объекты StorageClass, обеспечивают фактическое выделение и управление хранилищем в соответствии с особенностями конкретной системы. По сути, StorageClass служит своеобразным мостом между вашими потребностями в хранении данных и возможностями, предоставляемыми CSI-драйверами.

Список объектов StorageClass, созданных в кластере, можно вывести с помощью следующей команды `kubectl`:

```
$ kubectl get sc
NAME                 PROVISIONER               RECLAIMPOLICY   VOLUMEBINDINGMODE
ALLOWVOLUMEEXPANSION   AGE
standard (default)   k8s.io/minikube-hostpath Delete          Immediate
false                  7d
```

Подробные сведения об объекте StorageClass можно посмотреть с помощью параметра `-o yaml`:

```
$ kubectl get storageclasses standard -o yaml
apiVersion: storage.k8s.io/v1
kind: StorageClass
... <фрагмент опущен для краткости> ...
metadata:
  name: standard
  resourceVersion: "290"
  uid: f41b765f-301f-4781-b9d0-46aec694336b
provisioner: k8s.io/minikube-hostpath
reclaimPolicy: Delete
volumeBindingMode: Immediate
```

Кроме того, допустимо использовать форму во множественном числе `storageclasses`, а также псевдоним `sc`. Следующие три варианта команды эквивалентны:

```
$ kubectl get storageclass
$ kubectl get storageclasses
$ kubectl get sc
```

Во всех трех случаях вывод одинаков, поэтому мы опустили его для краткости. Наибольшую важность для нас представляют следующие два поля этого вывода:

- `NAME` — имя и уникальный идентификатор объекта StorageClass;
- `PROVISIONER` — название базовой технологии хранения; по сути, это фрагмент кода, который кластер Kubernetes использует для взаимодействия с выбранной системой хранения.

Вы можете создать несколько объектов StorageClass, использующих один и тот же провиженер.

Поскольку мы работаем с кластером `minikube` в лабораторной среде, у нас есть объект StorageClass `standard`, использующий провиженер `k8s.io/minikube-hostpath`.

Этот провиженер взаимодействует с файловой системой нашего хоста и автоматически создает тома `hostPath` для подов. Но аналогичным образом можно было бы использовать тома Amazon EBS или Google PD.

В системе GKE предусмотрен класс хранилища с провиженером, способным взаимодействовать с API Google PD. Эта встроенная возможность платформы Google Cloud реализуется с помощью StorageClass следующим образом:

```
# gce-pd-sc.yaml
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: slow
provisioner: kubernetes.io/gce-pd
parameters:
  type: pd-standard
```

В свою очередь, в AWS существует объект StorageClass с провиженером, работающим с API томов EBS. Эти провиженеры представляют собой библиотеки, способные взаимодействовать с API соответствующих облачных провайдеров.

Таким образом, именно объекты StorageClass позволяют Kubernetes работать со множеством различных технологий хранения. С точки зрения пода не имеет значения, используется ли том EBS, NFS или Google PD: под «видит» только объект PersistentVolume. Вся логика взаимодействия с конкретной системой хранения реализована в провиженере, на который ссылается объект StorageClass.

Хорошая новость заключается в том, что вы можете добавить в кластер произвольное количество объектов StorageClass с их провиженерами.

Ничто не мешает расширять возможности кластера, добавляя новые объекты StorageClass, подобно плагинам. При этом вы просто увеличиваете количество технологий хранения, с которыми умеет работать ваш кластер. Например, можно добавить в кластер `minikube` объект StorageClass для Amazon EBS. Однако пользы от этого не будет: если кластер `minikube` работает не на экземпляре EC2, а на локальной машине, подключить том EBS все равно не удастся.

С практической точки зрения лучше использовать CSI-драйверы поставщиков, поддерживающих локальное развертывание, например OpenEBS, TopoLVM или Portworx. Они позволяют работать с постоянным хранилищем даже в локальной среде, в том числе в `minikube`. Кроме того, большинство облачных провайдеров предлагает бесплатные тарифы для небольших Kubernetes-развертываний, что позволяет тестировать созданные решения в облаке без существенных затрат.

В следующем подразделе мы поговорим о динамическом выделении ресурсов хранилища с использованием PVC.

Роль PersistentVolumeClaim в динамическом выделении ресурсов хранилища

В ходе динамического выделения ресурсов хранилища объект PersistentVolumeClaim играет иную роль. Так как объект PersistentVolume в этом сценарии будет формироваться с помощью объекта StorageClass, вам придется управлять только заявкой PersistentVolumeClaim.

В качестве примера создадим под `nginx`, который будет динамически монтировать том типа `hostPath`. В данном случае администратору вообще не нужно вручную выделять объект PersistentVolume, об этом позаботятся объекты PersistentVolumeClaim и StorageClass.

Для начала создадим новое пространство имен `dynamicstorage`, в котором будут выполняться наши примеры:

```
$ kubectl create ns dynamicstorage
namespace/dynamicstorage created
```

Теперь выполним команду `kubectl get sc`, чтобы убедиться в наличии объекта StorageClass, способного работать с томом `hostPath`, доступным в кластере.

В данной конкретной среде Kubernetes (`minikube`) объект StorageClass создается по умолчанию во время установки кластера. Однако в другом дистрибутиве Kubernetes дела могут обстоять иначе.

Учтите, что кластеры, развернутые в GKE, по умолчанию могут иметь объект StorageClass для работы с хранилищем Google, а кластеры в AWS — предусматривать StorageClass для взаимодействия с решениями Amazon и т. д. В случае с `minikube` у нас есть как минимум один объект StorageClass, способный работать с объектами PersistentVolume на основе `hostPath`. Это означает, что вывод команды `kubectl get sc` будет зависеть от того, где развернут ваш кластер:

```
$ kubectl get sc
NAME                 PROVISIONER               RECLAIMPOLICY   VOLUMEBINDINGMODE
ALLOWVOLUMEEXPANSION   AGE
standard (default)   k8s.io/minikube-hostpath Delete          Immediate
false                  21d
```

Как видите, в нашем кластере присутствует объект StorageClass `standard`, способный взаимодействовать с `hostPath`.

В некоторых сложных кластерах, развернутых одновременно в нескольких облаках или в гибридной среде, может присутствовать множество объектов StorageClass для работы с различными технологиями хранения. Учтите, что система Kubernetes не привязана к какому-либо облачному провайдеру, поэтому не ограничивает вас в выборе хранилищ.

Теперь создадим объект PersistentVolumeClaim, который динамически создаст том типа `hostPath`. Ниже приведен YAML-манифест для объекта PVC. Обратите внимание на значение `standard` в поле `storageClassName`:

```
# pvc-dynamic.yaml
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: my-dynamic-hostpath-pvc
spec:
  accessModes:
    - ReadWriteOnce
  storageClassName: standard   # ВАЖНО
  resources:
    requests:
      storage: 1Gi
  selector:
    matchLabels:
      type: hostpath
      env: prod
```

А сейчас создадим объект PVC в нужном пространстве имен:

```
$ kubectl apply -f pvc-dynamic.yaml -n dynamicstorage
persistentvolumeclaim/my-dynamic-hostpath-pvc created
```

И проверим статус объектов PV и PVC:

```
$ kubectl get pod,pvc,pv -n dynamicstorage
NAME                                          STATUS   VOLUME
CAPACITY   ACCESS MODES   STORAGECLASS   VOLUMEATTRIBUTESCLASS   AGE
persistentvolumeclaim/my-dynamic-hostpath-pvc   Bound
pvc-4597ab27-c894-40de-a7ac-1b6ca961bcdc   1Gi        RWO            standard
<unset>                 7s

NAME                                                        CAPACITY
ACCESS MODES   RECLAIM POLICY   STATUS   CLAIM
STORAGECLASS   VOLUMEATTRIBUTESCLASS   REASON   AGE
persistentvolume/pvc-4597ab27-c894-40de-a7ac-1b6ca961bcdc   1Gi        RWO
Delete           Bound    dynamicstorage/my-dynamic-hostpath-pvc      standard
<unset>                          7s
```

Как видите, PV был создан с помощью StorageClass и привязан к PVC в соответствии с запросом.

После создания объекта PersistentVolumeClaim можно добавить новый под, в который он будет смонтирован. Вот YAML-манифест для этого пода:

```
# pod-with-dynamic-pvc.yaml
apiVersion: v1
kind: Pod
metadata:
  name: nginx-dynamic-storage
```

```
spec:
  containers:
    - image: nginx
      name: nginx
      volumeMounts:
        - mountPath: "/var/www/html"
          name: mypersistentvolume
  volumes:
    - name: mypersistentvolume
      persistentVolumeClaim:
        claimName: my-dynamic-hostpath-pvc
```

Теперь создадим его в подходящем пространстве имен:

```
$ kubectl apply -f pod-with-dynamic-pvc.yaml -n dynamicstorage
pod/nginx-dynamic-storage created

$ kubectl get pod,pvc,pv -n dynamicstorage
NAME                        READY   STATUS    RESTARTS   AGE
pod/nginx-dynamic-storage 1/1     Running   0          45s

NAME                                        STATUS   VOLUME
CAPACITY   ACCESS MODES   STORAGECLASS   VOLUMEATTRIBUTESCLASS   AGE
persistentvolumeclaim/my-dynamic-hostpath-pvc   Bound
pvc-4597ab27-c894-40de-a7ac-1b6ca961bcdc   1Gi        RWO            standard
<unset>                 7m39s

NAME                                                        CAPACITY
ACCESS MODES   RECLAIM POLICY   STATUS   CLAIM
STORAGECLASS   VOLUMEATTRIBUTESCLASS   REASON   AGE
persistentvolume/pvc-4597ab27-c894-40de-a7ac-1b6ca961bcdc   1Gi        RWO
Delete           Bound    dynamicstorage/my-dynamic-hostpath-pvc      standard
<unset>                          7m39s
```

Динамическое выделение ресурсов хранилища настроено. Обратите внимание на то, что по умолчанию используется политика освобождения `Delete`, поэтому объект PV удаляется при удалении создавшего его PVC. Если вы хотите сохранять критически важные данные, измените политику освобождения.

Чтобы проверить эту настройку, удалите под и PVC. PV будет автоматически удален объектом StorageClass:

```
$ kubectl delete po nginx-dynamic-storage -n dynamicstorage
pod "nginx-dynamic-storage" deleted

$ kubectl delete pvc my-dynamic-hostpath-pvc -n dynamicstorage
persistentvolumeclaim "my-dynamic-hostpath-pvc" deleted

$ kubectl get pod,pvc,pv -n dynamicstorage
No resources found
```

Из вывода видно, что PV также удаляется автоматически вместе с PVC.

Итак, мы рассмотрели основы работы с PV, PVC и StorageClass, а также различия между статическим и динамическим выделением ресурсов. В следующем разделе мы обсудим продвинутые темы, связанные с хранением данных в Kubernetes, и посмотрим, как можно оптимизировать и расширять стратегии работы с хранилищем.

Продвинутые темы, связанные с хранением данных

Помимо получения базовых знаний о PV, PVC и StorageClass, вам также стоит ознакомиться с продвинутыми темами, связанными с хранением данных в Kubernetes. Хотя они не являются обязательными, понимание этих концепций позволит существенно расширить ваши способности как специалиста по использованию системы Kubernetes. В следующих подразделах мы рассмотрим такие темы, как временное хранение данных с помощью эфемерных томов, клонирование томов CSI для более гибкого управления ими и увеличение объема хранилища путем изменения объекта PersistentVolumeClaim. Это позволит вам получить более широкое представление о практической работе с хранилищем в Kubernetes.

Эфемерные тома для временного хранения данных в Kubernetes

Эфемерные тома предоставляют удобный способ организации временного хранилища для подов в Kubernetes. Они идеально подходят для приложений, которым требуется рабочее пространство для кэширования или доступ к данным только для чтения (например, к конфигурационным файлам или объектам Secret). В отличие от PV эфемерные тома автоматически удаляются при завершении работы пода, что упрощает процессы развертывания и управления.

Вот основные преимущества эфемерных томов:

- возможность организации временного хранилища для подов;
- автоматическое удаление при завершении работы пода;
- упрощенное развертывание и управление.

В Kubernetes доступно несколько видов эфемерного хранилища:

- `emptyDir` — создает пустой каталог в локальном хранилище узла;
- ConfigMap, downwardAPI, Secret — внедряют в под данные из объектов Kubernetes;

- эфемерные CSI-тома — предоставляются внешними CSI-драйверами (для их использования необходима поддержка со стороны конкретного драйвера);
- универсальные эфемерные тома — предоставляются драйверами хранения, поддерживающими работу с PV.

Теперь, когда мы разобрались с особенностями эфемерных томов, поговорим о клонировании томов CSI и о снимках состояния томов.

Клонирование томов CSI и снимки состояния томов

CSI предусматривает такую мощную функцию, как клонирование томов, позволяющее создать точную копию существующего объекта PersistentVolumeClaim в виде нового PVC.

Вот пример YAML-манифеста для клонирования PVC:

```
# pv-cloning.yaml
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: cloned-pvc
  namespace: mynamespace
spec:
  accessModes:
    - ReadWriteOnce
  storageClassName: custom-storage-class
  resources:
    requests:
      storage: 5Gi
  dataSource:
    kind: PersistentVolumeClaim
    name: original-pvc
```

Рассмотрим основные преимущества клонирования CSI-томов.

- *Упрощение рабочих процессов.* Клонирование CSI-томов автоматизирует репликацию данных, устраняя необходимость в ручном копировании и упрощая управление хранилищем.
- *Повышение эффективности.* Мы можем легко создавать реплики существующих томов, оптимизируя процессы развертывания и использование ресурсов.
- *Отладка с использованием копий данных.* Вместо того чтобы работать с производственными данными, можно создать копию и использовать ее для тестирования, контроля качества или устранения проблем.

Подробнее о клонировании CSI-томов можно узнать по адресу https://kubernetes.io/docs/concepts/storage/volume-pvc-datasource.

Помимо клонирования томов, Kubernetes поддерживает еще один механизм создания резервных копий данных через CSI-драйверы — снимки томов. Объ-

ект VolumeSnapshot предоставляет стандартизированный способ получения точной копии данных тома на конкретный момент времени. Для управления снимками состояния Kubernetes использует три ресурса: VolumeSnapshot, VolumeSnapshotContent и VolumeSnapshotClass. Объект VolumeSnapshot представляет собой пользовательский запрос на создание снимка, а VolumeSnapshotContent — сам снимок. Эти ресурсы позволяют фиксировать состояние томов без выделения нового тома, что особенно полезно, например, при создании резервных копий баз данных перед выполнением критически важных обновлений или удалений. В отличие от обычных объектов PV ресурсы для работы со снимками состояния томов реализованы в виде *CRD* (Custom Resource Definition — определение пользовательского ресурса) и требуют наличия CSI-драйвера, поддерживающего создание таких снимков. Для выполнения операций `CreateSnapshot` и `DeleteSnapshot` CSI-драйвер использует контейнер-прицеп под названием `csi-snapshotter`.

Когда пользователь создает снимок, он может быть либо заранее выделен администратором, либо динамически создан из существующего PVC. В обоих случаях контроллер снимков связывает ресурсы VolumeSnapshot и VolumeSnapshotContent, гарантируя соответствие содержимого снимка запросу пользователя. Снимки легко удаляются или сохраняются в зависимости от заданной политики удаления (`DeletionPolicy`), что обеспечивает гибкость управления данными. Кроме того, Kubernetes предоставляет возможность при восстановлении данных из снимка задать иной режим тома (например, `Filesystem` вместо `Block`) и создать на его основе новый объект PVC. Такая функциональность делает VolumeSnapshot мощным инструментом защиты данных. В сочетании с возможностью клонирования томов CSI она позволяет эффективно создавать резервные копии или тестовые среды, повышая гибкость управления хранилищем в Kubernetes.

Клонирование томов в Kubernetes идеально подходит для создания идентичных копий PersistentVolume, что особенно востребовано в средах разработки и тестирования. Снимки состояния, в свою очередь, фиксируют состояние тома в конкретный момент времени, что делает их полезными для резервного копирования и восстановления данных.

Подробнее о снимках состояния томов можно узнать по адресу https://kubernetes.io/docs/concepts/storage/volume-snapshots/.

В следующем разделе мы рассмотрим способ увеличить размер тома через PVC.

Увеличение размера тома через PersistentVolumeClaim

Система Kubernetes поддерживает встроенный механизм расширения PVC, что позволяет с легкостью увеличивать объем хранилища для приложений. Эта возможность доступна только для томов, выделяемых через CSI-драйверы (начиная с версии 1.29, другие типы томов признаны устаревшими).

Чтобы включить данный механизм для конкретного объекта StorageClass, необходимо задать параметр `allowVolumeExpansion: true` в его определении. Этот флаг управляет возможностью объектов PVC, ссылающихся на данный StorageClass, запрашивать увеличение объема хранилища.

Пример конфигурации StorageClass:

```
# storageclass-expandable.yaml
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: expandable-sc
  provisioner: vendor-name.example/magicstorage
... <фрагмент опущен для краткости> ...
allowVolumeExpansion: true
```

Когда приложению требуется дополнительное место, достаточно отредактировать объект PVC и указать большее значение в поле `resources.requests.storage`. Kubernetes инициирует процесс расширения, изменив размер тома, управляемого CSI-драйвером. Это устраняет необходимость в создании нового тома и переносе данных в него, что значительно упрощает управление хранилищем.

Подробнее об этой возможности можно узнать по адресу https://kubernetes.io/docs/concepts/storage/persistent-volumes/#expanding-persistent-volumes-claims.

Резюме

Итак, мы подошли к концу главы, посвященной управлению постоянным хранилищем в Kubernetes. В ней вы узнали о том, что PersistentVolume — это объект, который выступает в роли указателя на базовую технологию хранения, будь то том `hostPath`, NFS или облачные решения вроде Amazon EBS и Google PD.

Кроме того, вы узнали о взаимосвязи между объектами PersistentVolume, PersistentVolumeClaim и StorageClass. Мы обсудили политики освобождения PersistentVolume, которые позволяют удалять, сохранять или повторно использовать эти объекты при удалении связанного с ними PVC.

Наконец, мы изучили механизм динамического выделения ресурсов, который упрощает работу с хранилищем. Следует помнить, что чрезмерное количество созданных и сохраненных томов может увеличить затраты на облачную инфраструктуру, даже несмотря на то, что использование ресурсов хранилища можно ограничивать с помощью квот для пространств имен.

Таким образом, мы завершили изучение основ Kubernetes, и эта глава стала заключительной в данной части. Далее в книге вы познакомитесь с контроллерами Kubernetes, которые предназначены для автоматизации задач вроде поддержания заданного количества реплик подов с использованием ресурсов типа Deployment или StatefulSet. Так что впереди еще много нового!

Дополнительные источники информации

- Объекты Persistent Volume: https://kubernetes.io/docs/concepts/storage/persistent-volumes/.
- Типы объектов PersistentVolume: https://kubernetes.io/docs/concepts/storage/persistent-volumes/#types-of-persistent-volumes.
- Динамическое выделение томов: https://kubernetes.io/docs/concepts/storage/dynamic-provisioning/.
- Увеличение размера тома через PersistentVolumeClaim: https://kubernetes.io/docs/concepts/storage/persistent-volumes/#expanding-persistent-volumes-claims.
- Клонирование CSI-томов: https://kubernetes.io/docs/concepts/storage/volume-pvc-datasource/.

10 Запуск рабочих нагрузок производственного уровня в Kubernetes

В предыдущих главах мы обсуждали концепции контейнеризации и такие базовые строительные блоки Kubernetes, как поды, задания и объекты ConfigMap. До сих пор мы в основном рассматривали сценарии с одной машиной, в которых приложению требуется только один контейнерный хост или узел Kubernetes. Когда речь заходит о Kubernetes-приложениях *производственного уровня*, необходимо учитывать такие аспекты, как *масштабируемость*, *обеспечение высокой доступности* и *балансировка нагрузки*. А это всегда предполагает *оркестрацию* контейнеров, работающих на нескольких узлах.

Если вкратце, то *оркестрация контейнеров* — это способ управления жизненным циклом множества контейнеров в крупных и динамичных средах. Он охватывает развертывание и поддержание желаемого состояния сетей контейнеров, обеспечение избыточности и высокой доступности (с помощью внешних компонентов), масштабирование кластера и изменение числа реплик контейнеров, проведение автоматизированных проверок работоспособности и сбор телеметрии (логов и метрик). Организовать эффективную оркестрацию контейнеров в облаке непросто. Именно для решения этой задачи и была создана система Kubernetes.

Технические требования

Для работы с этой главой вам потребуется:

- развернутый кластер Kubernetes. Вы можете использовать как локальный, так и облачный кластер, однако для полноценного усвоения материала рекомендуется отдать предпочтение многоузловому облачному кластеру;
- рабочий интерфейс командной строки `kubectl`, настроенный для взаимодействия с кластером.

Инструкции по созданию кластера Kubernetes и настройке клиента `kubectl` приведены в главах 2 и 3.

Примеры кода для этой главы можно загрузить из официального репозитория GitHub по адресу https://github.com/PacktPublishing/The-Kubernetes-Bible-Second-Edition/tree/main/Chapter10.

Обеспечение высокой доступности и отказоустойчивости в Kubernetes

Сначала вспомним, что подразумевается под высокой доступностью (High Availability, HA) и отказоустойчивостью (Fault Tolerance, FT) и чем эти понятия различаются. Эти ключевые аспекты облачных приложений описывают способность системы или решения сохранять работоспособность в течение необходимого периода времени. С точки зрения конечного пользователя, важнейшим требованием обычно является именно высокая доступность наряду с согласованностью данных.

Высокая доступность

В области системной инженерии под *доступностью* понимается процент времени, в течение которого система полноценно функционирует и доступна для конечного пользователя. Иными словами, это отношение времени безотказной работы к общему времени, охватывающему периоды работы и периоды простоя. Например, если за последние 30 дней (720 часов) ваше облачное приложение находилось в состоянии незапланированного простоя в течение 1 часа и было недоступно для пользователя, то его доступность составляет: 719 ч / 720 ч = 99,861 %. Обычно при проектировании систем в целях упрощения доступность выражается в «девятках». Например, если мы говорим, что система обладает доступностью «пять девяток», это означает, что она доступна как минимум 99,999 % общего времени. Такое значение допускает не более 26 секунд простоя в месяц. Эти показатели часто лежат в основе *соглашений об уровне обслуживания* (Service-Level Agreement, SLA) платных облачных сервисов.

Исходя из этого, мы можем дать относительно простое, хотя и не абсолютно строгое определение высокой доступности: система считается высокодоступной, если она функционирует без перерывов на протяжении длительных периодов времени. Обычно эталоном высокой доступности служит показатель «пять девяток».

Добиться высокой доступности системы можно с помощью одного из следующих подходов или их комбинации.

- *Устранение единой точки отказа* (Single Point of Failure, SPOF) за счет дублирования компонентов.

- *Настройка механизма автоматического аварийного переключения* (failover), который переносит нагрузку с текущего (неисправного) компонента на резервный.
- *Балансировка нагрузки*, то есть распределение входящего трафика между резервными компонентами, способными его обслуживать. В большинстве случаев это требует корректной настройки механизма автоматического аварийного переключения, мониторинга работы компонентов и сбора телеметрии.

Теперь обсудим понятие отказоустойчивости, которое также играет важную роль в таких распределенных системах, как приложения, работающие в Kubernetes.

Отказоустойчивость

Понятие отказоустойчивости дополняет концепцию высокой доступности: система считается отказоустойчивой, если она продолжает функционировать и оставаться работоспособной даже при сбое одного или нескольких компонентов. К типичным механизмам обеспечения отказоустойчивости относятся RAID-массивы для хранения данных, которые распределяют информацию по нескольким дискам, и балансировщики нагрузки, перенаправляющие трафик на исправные узлы. Эти решения обеспечивают устойчивость системы и сводят к минимуму перебои в ее работе. Полная отказоустойчивость означает достижение 100%-ной доступности, что во многих случаях требует использования сложных механизмов, активно выявляющих сбои и устраняющих проблемы, возникающие в компонентах, без перерывов в их работе. В зависимости от реализации сбой может привести к постепенному снижению производительности, пропорциональному степени его серьезности. Иными словами, незначительный сбой будет оказывать минимальное влияние на общую производительность системы при обслуживании запросов конечных пользователей.

Обеспечение высокой доступности и отказоустойчивости приложений в Kubernetes

В предыдущих главах мы говорили о подах и о том, как сервисы делают их доступными для внешнего трафика (см. главу 8). Сервисы — это объекты Kubernetes, предоставляющие стабильный сетевой адрес для набора исправных подов. Внутри кластера Kubernetes сервис делает поды доступными по виртуальным IP-адресам, которыми управляет компонент `kube-proxy` на каждом узле. Снаружи, в облачных средах, обычно используется облачный балансировщик нагрузки, который интегрируется с кластером Kubernetes через облачный плагин в составе компонента `cloud-controller-manager`. При наличии внешнего балансировщика нагрузки микросервисы или рабочие нагрузки, выполняемые в Kubernetes, получают возможность распределять трафик между исправными

подами — как на одном, так и на нескольких узлах, что является ключевым условием обеспечения высокой доступности.

Сервисы необходимы для распределения запросов между подами, однако мы еще не обсудили способ поддержания нескольких реплик одного и того же пода, размещенных на разных узлах. Для этого в Kubernetes предусмотрены следующие инструменты.

- *Объект ReplicationController* — исходный механизм репликации подов в Kubernetes.
- *Объект ReplicaSet* — механизм, пришедший на смену ReplicationController. Главное отличие заключается в том, что ReplicaSet поддерживает выборку подов с помощью селекторов, основанных на множествах.

Управлять объектами ReplicaSet рекомендуется с помощью объекта Deployment, который упрощает процесс обновлений и откатов.

- *Объект Deployment* — это еще один уровень абстракции поверх ReplicaSet. Он обеспечивает *декларативные* обновления подов и объектов ReplicaSet, включая выкатки и откаты. Объект Deployment используется для управления микросервисами и рабочими нагрузками *без сохранения состояния*.
- *Объект StatefulSet* — напоминает Deployment, но применяется для управления микросервисами и рабочими нагрузками *с сохранением состояния* внутри кластера. Управление состоянием в кластере обычно представляет наибольшую сложность при проектировании распределенных систем.
- *Объект DaemonSet* — предназначен для запуска единственного экземпляра пода на каждом или некоторых узлах кластера. Как правило, такие объекты используются для управления внутренними сервисами, например для агрегации логов или мониторинга узлов.

В следующих разделах мы поговорим об объектах ReplicationController и ReplicaSet. Более сложные объекты Deployment, StatefulSet и DaemonSet будут подробно описаны в следующих главах.

В данной главе основное внимание уделяется вопросам обеспечения высокой доступности и отказоустойчивости рабочих нагрузок и приложений, выполняющихся в Kubernetes. Если вас интересует обеспечение высокой доступности и отказоустойчивости самой системы Kubernetes, обратитесь к официальной документации: https://kubernetes.io/docs/setup/production-environment/tools/kubeadm/high-availability/. Учтите, что такие управляемые облачные Kubernetes-сервисы, как Azure Kubernetes Service (AKS), Amazon Elastic Kubernetes Service (EKS) или Google Kubernetes Engine (GKE), предоставляют высокодоступные кластеры, что избавляет вас от необходимости самостоятельно управлять мастер-узлами.

Объект ReplicationController

Обеспечение высокой доступности и отказоустойчивости требует дублирования компонентов и корректного распределения входящего трафика между их репликами. Первым объектом Kubernetes, позволявшим создавать и поддерживать несколько реплик подов в кластере, был ReplicationController (контроллер репликации). Важно отметить, что мы рассматриваем его здесь в основном по историческим причинам, так как именно объект ReplicationController являлся исходным способом создания нескольких реплик подов в Kubernetes. В настоящее время рекомендуется использовать объект ReplicaSet, который фактически представляет собой контроллер репликации следующего поколения, обладающий расширенными возможностями API.

Контроллеры Kubernetes призваны отслеживать текущее и желаемое состояние кластера, объявленное через API-сервер, и инициировать изменения, необходимые для приведения текущего состояния к желаемому. Контроллеры представляют собой непрерывные циклы обратной связи, делающие все возможное для обеспечения соответствия фактического состояния кластера тому, которое описано в шаблонах объектов.

Задача объекта ReplicationController довольно проста: он должен гарантировать, что в кластере исправно работает указанное количество реплик подов (определяемое шаблоном). Это означает, что если контроллеру репликации дано задание поддерживать три реплики заданного пода, то он будет поддерживать именно это количество, создавая и удаляя поды по необходимости. Например, сразу после создания объекта ReplicationController будут запущены три новых пода на основе его шаблона. Если по какой-то причине в кластере окажется четыре таких пода, контроллер репликации удалит один из них. Если же какой-то под будет удален или окажется неисправным, он будет заменен новым подом.

Поскольку в настоящее время для управления репликацией рекомендуется использовать объект Deployment, который настраивает ReplicaSet, мы не будем подробно останавливаться на обсуждении объекта ReplicationController. В следующем разделе мы обсудим объект ReplicaSet и попрактикуемся в работе с ним. Подробный разбор объекта Deployment вы найдете в главе 11.

Объект ReplicaSet

Объект ReplicaSet (набор реплик) тесно связан с ReplicationController, о котором мы говорили выше. Более того, ReplicaSet является его *преемником* и обладает очень похожей спецификацией API и функциональностью. Объект

ReplicaSet тоже призван поддерживать фиксированное количество исправных, идентичных подов (реплик), удовлетворяющих заданным условиям. Для этого вы определяете шаблон пода, задаете соответствующие селекторы меток и требуемое количество реплик, а контроллер Kubernetes ReplicaSetController (так официально называется контроллер, отвечающий за работу объектов ReplicaSet) выполняет все необходимые действия для поддержания подов в рабочем состоянии.

Прежде чем обсуждать принцип работы ReplicaSet, рассмотрим его основные отличия от ReplicationController.

Отличия ReplicaSet от ReplicationController

Различия между объектами ReplicaSet и ReplicationController перечислены в табл. 10.1.

Таблица 10.1. Сравнение объектов ReplicaSet и ReplicationController

Характеристика	ReplicaSet	ReplicationController
Селекторы меток	Поддерживает селекторы на основе множеств для включения или исключения меток. Это позволяет задавать более сложную логику, например добавлять метку `environment=test` или `environment=dev`, исключая при этом `environment=prod`	Поддерживает только селекторы на основе равенства типа *`ключ=значение`*. Не предусматривает расширенных механизмов сопоставления меток
Интеграция с другими объектами Kubernetes	Служит основой для таких более продвинутых объектов, как Deployment и HorizontalPodAutoscaler (HPA)	В основном управляет репликацией подов напрямую без подобных интеграций
Выкатка обновлений подов	Управляется декларативно через объекты Deployment, что позволяет выполнять поэтапные выкатки и откаты	Управление осуществлялось вручную с помощью устаревшей императивной команды `kubectl rolling-update`
Дальнейшая поддержка	Более современный и гибкий ресурс с перспективными возможностями	Ожидается, что этот объект будет объявлен устаревшим

Таким образом, вам всегда следует выбирать объект ReplicaSet вместо ReplicationController. Однако учтите, что использование «голых» объектов ReplicaSet в рабочих кластерах, как правило, нецелесообразно; для управления ими лучше

применять более высокоуровневые абстракции, такие как объекты Deployment. Этот подход будет подробно рассмотрен в следующей главе.

А сейчас обсудим способ создания объектов ReplicaSet и управления ими.

Создание объекта ReplicaSet

В следующем примере мы будем использовать многоузловой кластер, созданный с помощью инструмента `kind`, описанного в главе 3:

```
$ kind create cluster --config Chapter03/kind_cluster --image kindest/node:v1.31.0

$ kubectl get nodes
NAME                 STATUS   ROLES           AGE   VERSION
kind-control-plane   Ready    control-plane   60s   v1.31.0
kind-worker          Ready    <none>          47s   v1.31.0
kind-worker2         Ready    <none>          47s   v1.31.0
kind-worker3         Ready    <none>          47s   v1.31.0
```

Первым делом создадим пространство имен, в котором будут размещаться наши ресурсы ReplicaSet.

```
$ kubectl create -f ns-rs.yaml
namespace/rs-ns created
```

Теперь рассмотрим структуру YAML-манифеста `nginx-replicaset.yaml`, поддерживающего четыре реплики пода `nginx`:

```
# nginx-replicaset-example.yaml
apiVersion: apps/v1
kind: ReplicaSet
metadata:
  name: nginx-replicaset-example
  namespace: rs-ns
spec:
  replicas: 4
  selector:
    matchLabels:
      app: nginx
      environment: test
  template:
    metadata:
      labels:
        app: nginx
        environment: test
    spec:
      containers:
        - name: nginx
          image: nginx:1.17
          ports:
            - containerPort: 80
```

В спецификации ReplicaSet можно выделить три основных компонента.

- `replicas` — определяет количество реплик пода, которые запускаются на основе заданного шаблона (`template`) и соответствуют селектору меток (`selector`). Для поддержания требуемого числа реплик поды могут создаваться или удаляться.
- `selector` — селектор меток, задающий правила идентификации подов, находящихся под управлением ReplicaSet. Учтите, что, как и в случае с ReplicationController, существующие «голые» поды (поды без контроллера) могут попасть под управление ReplicaSet при совпадении их меток с селектором.
- `template` — шаблон для создания пода. Метки, указанные в разделе `metadata`, должны соответствовать условиям, определенным в разделе `selector`.

Эти концепции проиллюстрированы на рис. 10.1.

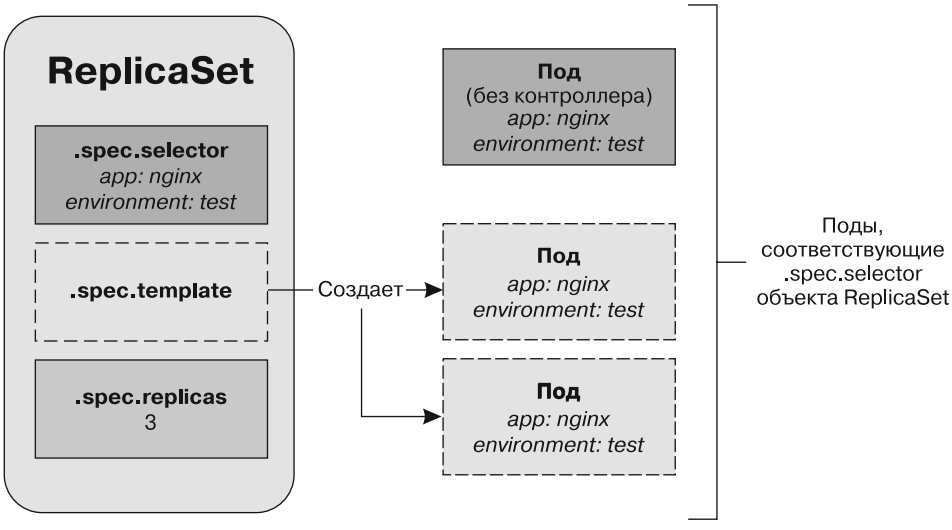

Рис. 10.1. Объект ReplicaSet в Kubernetes

Как видите, для создания подов объект ReplicaSet использует `.spec.template`. Созданные поды должны соответствовать селектору меток, заданному в разделе `.spec.selector`. Обратите внимание, что мы также можем захватить существующие «голые» поды, метки которых совпадают с условиями, заданными в ReplicaSet. В показанном на рис. 10.1 случае объект ReplicaSet создает только два новых пода, а третий является захваченным «голым» подом.

В предыдущем примере мы использовали простой селектор *на основе равенства*, определенный через `.spec.selector.matchLabels`. Более сложный селектор

на основе множеств можно задать с помощью .spec.selector.matchExpressions. Например:

```
# nginx-replicaset-expressions.yaml
...<фрагмент опущен для краткости>...
spec:
  replicas: 4
  selector:
    matchLabels:
      app: nginx
    matchExpressions:
      - key: environment
        operator: In
        values:
          - test
          - dev
  template:
...<фрагмент опущен для краткости>...
```

Эта конфигурация предписывает ReplicaSet выбирать только те поды, у которых есть метка app=nginx, а также метка environment=test или environment=dev.

При определении объекта ReplicaSet значения в разделе .spec.template.metadata.labels должны совпадать с содержимым .spec.selector, иначе объект будет отклонен API.

Теперь применим манифест ReplicaSet к кластеру с помощью команды kubectl apply:

```
$ kubectl apply -f nginx-replicaset-example.yaml
replicaset.apps/nginx-replicaset-example created
```

После этого можно проверить статус нового объекта nginx-replicaset-example:

```
$ kubectl describe replicaset/nginx-replicaset-example -n rs-ns
...
Replicas:    4 current / 4 desired
Pods Status: 4 Running / 0 Waiting / 0 Succeeded / 0 Failed
...
```

Для просмотра подов, которыми управляет ReplicaSet, можно выполнить команду kubectl get pods -n rs-ns. Подробно изучить метки пода и убедиться, что в его описании присутствует свойство Controlled By: ReplicaSet/nginx-replicaset-example, можно с помощью команды kubectl describe pod <Id_пода>.

При выполнении команд kubectl можно использовать сокращение rs вместо полной формы replicaset.

В следующем подразделе мы поговорим о тестировании поведения объекта ReplicaSet.

Тестирование поведения объекта ReplicaSet

Чтобы продемонстрировать гибкость объекта ReplicaSet, удалим один из подов, которыми управляет объект `nginx-replicaset-example`, с помощью команды `kubectl delete`:

```
$ kubectl delete po nginx-replicaset-example-6qc9p -n rs-ns
```

Если действовать быстро[1], то с помощью команды `kubectl get pods` можно заметить, как один из подов завершает работу, а ReplicaSet немедленно создает новый для восстановления целевого количества реплик.

Чтобы получить больше информации о событиях, произошедших с объектом ReplicaSet из нашего примера, используйте команду `kubectl describe`:

```
$ kubectl describe rs/nginx-replicaset-example -n rs-ns
...
Events:
  Type    Reason            Age   From                   Message
  ----    ------            ----  ----                   -------
...<фрагмент опущен для краткости>...
  Normal  SuccessfulCreate  9m9s  replicaset-controller  Created pod: nginx-
replicaset-example-r2qfn
  Normal  SuccessfulCreate  3m1s  replicaset-controller  Created pod: nginx-
replicaset-example-krdrs
```

В данном случае `nginx-replicaset-example-krdrs` представляет собой новый под, созданный объектом ReplicaSet.

Теперь создадим «голый» под, метки которого совпадают с селектором нашего объекта ReplicaSet. В результате количество подов, удовлетворяющих условиям ReplicaSet, станет равным четырем, и этот объект удалит один под, чтобы вернуть число реплик к целевому значению, равному трем.

Будьте осторожны с метками «голых» подов (подов, не находящихся под управлением объекта ReplicaSet). Объект ReplicaSet может взять под контроль любой под с соответствующими метками, что может привести к нежелательным результатам. Чтобы избежать подобных конфликтов, снабжайте «голые» поды уникальными метками.

[1] Для наглядности в соседнем терминале лучше выполнить watch -n 1 kubectl get pods -n rs-ns.

Сначала создадим простой манифест пода `nginx-pod-bare.yaml`:

```
# nginx-pod-bare.yaml
apiVersion: v1
kind: Pod
metadata:
  name: nginx-pod-bare-example
  namespace: rs-ns
  labels:
    app: nginx
    environment: test
spec:
  containers:
    - name: nginx
      image: nginx:1.17
      ports:
        - containerPort: 80
```

Метаданные этого пода должны содержать метки, совпадающие с селектором ReplicaSet. Примените этот манифест к кластеру:

```
$ kubectl apply -f nginx-pod-bare.yaml
pod/nginx-pod-bare-example created
$ kubectl get pods
NAME                            READY   STATUS        RESTARTS   AGE
nginx-pod-bare-example          0/1     Terminating   0          1s
nginx-replicaset-example-74kq9  1/1     Running       0          23h
nginx-replicaset-example-qfvx6  1/1     Running       0          23h
nginx-replicaset-example-s5cwc  1/1     Running       0          23h
```

Сразу после этого проверьте события, связанные с объектом ReplicaSet, выполнив команду `kubectl describe`:

```
$ kubectl describe rs/nginx-replicaset-example -n rs-ns
...
Events:
  Type    Reason            Age   From                   Message
  ----    ------            ----  ----                   -------
...
  Normal  SuccessfulDelete  29s   replicaset-controller  Deleted pod:
                                                         nginx-pod-bare-example
```

Как видите, объект ReplicaSet незамедлительно обнаружил новый под с совпадающими метками и удалил его.

Аналогичным образом можно вывести поды из-под управления ReplicaSet, изменив их метки так, чтобы они больше не соответствовали его селектору. Это бывает актуально при отладке и расследовании инцидентов.

В следующем подразделе вы узнаете, как объект ReplicaSet помогает обеспечить высокую доступность и отказоустойчивость приложения.

Тестирование доступности и отказоустойчивости с помощью ReplicaSet

В качестве демонстрации мы используем ранее развернутый объект `nginx-replicaset-example` и создадим сервис, чтобы предоставить доступ к этому приложению:

```
$ kubectl apply -f nginx-service.yaml
service/nginx-service created
```

Выполним проброс портов для тестирования этого сервиса:

```
$ kubectl port-forward svc/nginx-service 8080:80 -n rs-ns
Forwarding from 127.0.0.1:8080 -> 80
Forwarding from [::1]:8080 -> 80
```

Проверим доступ к приложению в другой консоли:

```
$ curl localhost:8080
```

В ответ вы должны получить стандартную страницу приветствия Nginx.

Ранее мы уже тестировали команду удаления пода и убедились в том, что объект ReplicaSet воссоздает поды для восстановления заданного числа их реплик. Теперь проверим поведение при удалении узла Kubernetes. Сначала выясним текущее распределение подов:

```
$ kubectl get po -n rs-ns -o wide
NAME                              READY   STATUS    RESTARTS   AGE   IP
NODE           NOMINATED NODE   READINESS GATES
nginx-replicaset-example-cfcfs   1/1     Running   0          24m   10.244.1.3
kind-worker2   <none>           <none>
nginx-replicaset-example-krdrs   1/1     Running   0          17m   10.244.3.5
kind-worker    <none>           <none>
nginx-replicaset-example-kw7cl   1/1     Running   0          24m   10.244.3.4
kind-worker    <none>           <none>
nginx-replicaset-example-r2qfn   1/1     Running   0          24m   10.244.2.4
kind-worker3   <none>           <none>
```

Как видите, два пода работают на узле `kind-worker`. Удалим этот узел из кластера:

```
$ kubectl cordon kind-worker
node/kind-worker cordoned

$ kubectl drain kind-worker --ignore-daemonsets
...<фрагмент опущен для краткости>...
```

```
pod/nginx-replicaset-example-kw7cl evicted
node/kind-worker drained

$ kubectl delete node kind-worker
node "kind-worker" deleted
```

Теперь снова проверим распределение подов и количество реплик:

```
$ kubectl get po -n rs-ns -o wide
NAME                             READY   STATUS    RESTARTS   AGE     IP
NODE          NOMINATED NODE   READINESS GATES
nginx-replicaset-example-8lz9x   1/1     Running   0          2m33s   10.244.1.4
kind-worker2  <none>           <none>
nginx-replicaset-example-cfcfs   1/1     Running   0          30m     10.244.1.3
kind-worker2  <none>           <none>
nginx-replicaset-example-d8rz5   1/1     Running   0          2m33s   10.244.2.5
kind-worker3  <none>           <none>
nginx-replicaset-example-r2qfn   1/1     Running   0          30m     10.244.2.4
kind-worker3  <none>           <none>
```

Согласно выводу, объект ReplicaSet воссоздал требуемое количество подов на доступных узлах.

Если ваш процесс `kubectl port-forward` все еще работает, можно повторно выполнить команду `curl localhost:8080` и убедиться в том, что приложение по-прежнему доступно. В рабочие среды рекомендуется интегрировать такие инструменты мониторинга, как *Prometheus* или *Grafana*, для визуализации данных о состоянии ресурсов в реальном времени, а также использовать *Fluentd* для сбора логов подов и диагностики сбоев.

Теперь, когда мы ознакомились с поведением объекта ReplicaSet, поговорим о его масштабировании.

Масштабирование объекта ReplicaSet

Объект ReplicaSet масштабируется так же, как и ReplicationController. Однако в реальных ситуациях масштабирование ReplicaSet обычно не выполняется вручную. Вместо этого его размером управляет другой, более высокоуровневый объект, например Deployment.

Сначала увеличим количество реплик в описании нашего объекта ReplicaSet. Для этого откройте файл `nginx-replicaset-example.yaml` и задайте для параметра `replicas` значение `5`:

```
...
spec:
  replicas: 5
...
```

Теперь нам необходимо изменить состояние кластера с помощью декларативной команды `kubectl apply`:

```
$ kubectl apply -f ./nginx-replicaset-example.yaml
```

Чтобы убедиться в том, что количество подов, которыми управляет ReplicaSet, изменилось, используйте команду `kubectl get pods -n rs-ns` или `kubectl describe rs/nginx-replicaset-example -n rs-ns`.

Аналогичного результата можно достичь и императивным способом, выполнив команду kubectl scale rs/nginx-replicaset-example --replicas=5. Однако такие императивные команды рекомендуется использовать только в учебных или отладочных сценариях.

А теперь уменьшим количество реплик. Откройте файл `nginx-replicaset.yaml` и задайте для параметра `replicas` значение `2`:

```
...
spec:
  replicas: 2
...
```

Затем снова примените изменения декларативным способом с помощью команды `kubectl apply`:

```
$ kubectl apply -f ./nginx-replicaset-example.yaml
```

На этом этапе вы можете удостовериться в том, что число подов уменьшилось до двух, с помощью команды `kubectl get pods -n rs-ns` или `kubectl describe rs/nginx-replicaset-example -n rs-ns`.

Использование проверок жизнеспособности подов с объектом ReplicaSet

Иногда бывает необходимо признать под неисправным и инициировать перезапуск контейнера, даже если основной процесс внутри этого контейнера не завершился аварийно. В главе 8 вы уже познакомились с механизмом проверок. В этом разделе вы узнаете, как можно использовать *проверки жизнеспособности* совместно с объектом ReplicaSet для повышения отказоустойчивости контейнеризованных компонентов.

В примере мы создадим объект ReplicaSet, запускающий поды `nginx` с дополнительной проверкой жизнеспособности основного контейнера. Эта проверка

будет выполнять HTTP-запрос типа `GET` к пути `/` и проверять, возвращает ли он код *успешного* ответа. Процесс `nginx` в контейнере может оставаться исправным (пока не произойдет сбой), но это не гарантирует работоспособность пода в целом. Если веб-сервер не может корректно предоставлять контент, значит, процесс работает, но возникла другая ошибка и такой под больше не должен использоваться. Мы смоделируем эту ситуацию, удалив файл `/index.html` из контейнера, что приведет к сбою проверки жизнеспособности.

Сначала создадим YAML-манифест `nginx-replicaset-livenessprobe.yaml` для нового объекта `nginx-replicaset-livenessprobe-example`:

```
# nginx-replicaset-livenessprobe.yaml
apiVersion: apps/v1
kind: ReplicaSet
metadata:
  name: nginx-replicaset-livenessprobe-example
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx
      environment: test
  template:
    metadata:
      labels:
        app: nginx
        environment: test
    spec:
      containers:
        - name: nginx
          image: nginx:1.17
          ports:
            - containerPort: 80
          livenessProbe:
            httpGet:
              path: /
              port: 80
            initialDelaySeconds: 2
            periodSeconds: 2
```

Единственным отличием данного объекта ReplicaSet от предыдущих является выделенный блок с определением проверки жизнеспособности. Эта проверка предполагает выполнение HTTP-запроса типа `GET` к пути `/` на порте `80` контейнера с интервалом 2 секунды (`periodSeconds`). Первая проверка будет выполнена через 2 секунды после запуска контейнера (`initialDelaySeconds`).

При модификации существующего объекта ReplicaSet необходимо сначала удалить его и создать заново, чтобы применить изменения к шаблону пода.

Примените манифест к кластеру с помощью следующей команды:

```
$ kubectl apply -f ./nginx-replicaset-livenessprobe.yaml
```

Удостоверьтесь в том, что поды успешно запустились:

```
$ kubectl get pods
```

Теперь выберите один из подов ReplicaSet, чтобы смоделировать сбой внутри контейнера, который приведет к провалу проверки жизнеспособности. В нашем примере мы выберем первый под из списка и удалим файл `index.html`. Выполните приведенную ниже команду. Она удалит файл `index.html`, который предоставляет сервер `nginx`, и спровоцирует ошибку при обработке HTTP-запроса типа `GET`:

```
$ kubectl exec -it nginx-replicaset-livenessprobe-example-lgvqv -- rm /usr/
share/nginx/html/index.html
```

Чтобы просмотреть события, связанные с этим подом, выполните команду `kubectl describe`:

```
$ kubectl describe pod/nginx-replicaset-livenessprobe-example-lgvqv
Name:           nginx-replicaset-livenessprobe-example-lgvqv
...<фрагмент опущен для краткости>...
Events:
  Type     Reason     Age                  From               Message
  ----     ------     ----                 ----               -------
  Normal   Scheduled  2m9s                 default-scheduler  Successfully
assigned default/nginx-replicaset-livenessprobe-example-lgvqv to kind-worker2
  Normal   Pulled     60s (x2 over 2m8s)   kubelet            Container image
"nginx:1.17" already present on machine
  Normal   Created    60s (x2 over 2m8s)   kubelet            Created container
nginx
  Normal   Started    60s (x2 over 2m8s)   kubelet            Started container
nginx
  Warning  Unhealthy  60s (x3 over 64s)    kubelet            Liveness probe
failed: HTTP probe failed with statuscode: 403
  Normal   Killing    60s                  kubelet            Container nginx
failed liveness probe, will be restarted
```

Как видите, проверка жизнеспособности правильно зафиксировала неисправность веб-сервера и инициировала перезапуск контейнера внутри пода.

Важно отметить, что сам объект ReplicaSet не участвовал в перезапуске — это действие было выполнено на уровне пода. Пример демонстрирует то, как отдельные объекты Kubernetes, обладающие разными функциями, могут взаимодействовать друг с другом для повышения отказоустойчивости системы.

Без проверки жизнеспособности конечный пользователь мог бы получить доступ к реплике, не способной корректно обслуживать запросы, и этот сбой остался бы незамеченным.

Удаление объекта ReplicaSet

Наконец, поговорим об удалении объекта ReplicaSet. Это можно сделать двумя способами.

1. Удалить объект ReplicaSet вместе с подами, которыми он управляет, — при этом сначала произойдет автоматическое уменьшение числа реплик.
2. Удалить только объект ReplicaSet, оставив поды нетронутыми.

Чтобы удалить ReplicaSet вместе с подами, используйте стандартную команду `kubectl delete`:

```
$ kubectl delete rs/nginx-replicaset-livenessprobe-example
```

В данном случае перед удалением объекта ReplicaSet будет завершена работа подов.

Если же вы хотите удалить только ReplicaSet, оставив поды, используйте параметр `--cascade=orphan`:

```
$ kubectl delete rs/nginx-replicaset-livenessprobe-example --cascade=orphan
```

Если вы проверите список подов в кластере после выполнения этой команды, то увидите все те поды, которыми ранее управлял объект `nginx-replicaset-livenessprobe-example`. Эти поды могут попасть под управление другого объекта ReplicaSet, если их метки совпадают с его селектором.

Резюме

В этой главе вы познакомились с основными строительными блоками, обеспечивающими высокую доступность и отказоустойчивость приложений, работающих в кластерах Kubernetes. Сначала мы обсудили важность высокой доступности и отказоустойчивости. Затем мы рассмотрели способы репликации компонентов и переключения на резервные экземпляры с помощью объекта ReplicaSet, который используется в Kubernetes для поддержки нескольких копий (реплик) идентичных подов. Мы обсудили различия между объектами ReplicationController и ReplicaSet и выяснили, почему именно ReplicaSet в настоящее время является рекомендуемым инструментом создания нескольких реплик подов.

Далее в книге мы обсудим способы оркестрации контейнерных приложений и рабочих нагрузок с помощью Kubernetes и такие ключевые объекты этой системы, как Deployment, StatefulSet и DaemonSet. В частности, следующая глава будет посвящена объектам Deployment, которые позволяют выполнять развертывание приложений и легко управлять выкатками и откатами новых версий.

Дополнительные источники информации

- Объект ReplicaSet: https://kubernetes.io/docs/concepts/workloads/controllers/replicaset.
- Объект ReplicationController: https://kubernetes.io/docs/concepts/workloads/controllers/replicationcontroller.

11 Использование объекта Deployment для управления рабочими нагрузками без сохранения состояния

В предыдущей главе вы познакомились с двумя важными объектами Kubernetes: *ReplicationController* и *ReplicaSet*. Теперь вы знаете, что они оба предназначены для поддержания заданного количества идентичных и исправных реплик (копий) подов. Фактически объект ReplicaSet является преемником ReplicationController, и в последних версиях Kubernetes рекомендуется использовать именно его вместо устаревшего контроллера репликации.

Пришло время познакомиться с объектом *Deployment* (развертывание), который обеспечивает удобное масштабирование, постепенное обновление и откат к определенной версии ваших Kubernetes-приложений и сервисов без сохранения состояния. Объект Deployment строится поверх ReplicaSet и предоставляет декларативный способ управления этими объектами: вы описываете желаемое состояние в манифесте Deployment, а Kubernetes осуществляет оркестрацию соответствующих объектов ReplicaSet контролируемым и предсказуемым образом. Наряду со StatefulSet (который будет рассмотрен в следующей главе) Deployment является одним из ключевых объектов для управления рабочими нагрузками в Kubernetes. Он представляет собой основополагающее средство разработки и эксплуатации Kubernetes-приложений. В этой главе вы получите все знания и инструменты, необходимые для развертывания компонентов приложений без сохранения состояния с помощью объектов Deployment, а также для безопасного выпуска новых версий этих компонентов.

Технические требования

Для работы с этой главой вам потребуется:

- развернутый кластер Kubernetes. Вы можете использовать как локальный, так и облачный кластер, однако для полноценного усвоения материала рекомендуется отдать предпочтение многоузловому облачному кластеру;
- рабочий интерфейс командной строки `kubectl`, настроенный для взаимодействия с кластером.

Инструкции по созданию кластера Kubernetes и настройке клиента `kubectl` приведены в главах 2 и 3.

Примеры кода для этой главы можно загрузить из официального репозитория GitHub по адресу https://github.com/PacktPublishing/The-Kubernetes-Bible-Second-Edition/tree/main/Chapter11.

Знакомство с объектом Deployment

Kubernetes позволяет запускать различные типы рабочих нагрузок в зависимости от конкретных сценариев использования. Понимание того, какой тип нагрузки лучше соответствует потребностям вашего приложения, позволяет принимать более взвешенные решения, оптимизировать использование ресурсов и обеспечивать высокую производительность и надежность облачных приложений. Эти базовые знания помогают раскрыть весь потенциал Kubernetes и с уверенностью развертывать и масштабировать приложения. Рассмотрим типы рабочих нагрузок, поддерживаемые Kubernetes, чтобы понять, какое место в этой системе занимает объект Deployment и какова его роль.

Следующая схема демонстрирует различные типы рабочих нагрузок в Kubernetes, которые будут подробно рассмотрены в этом разделе (рис. 11.1).

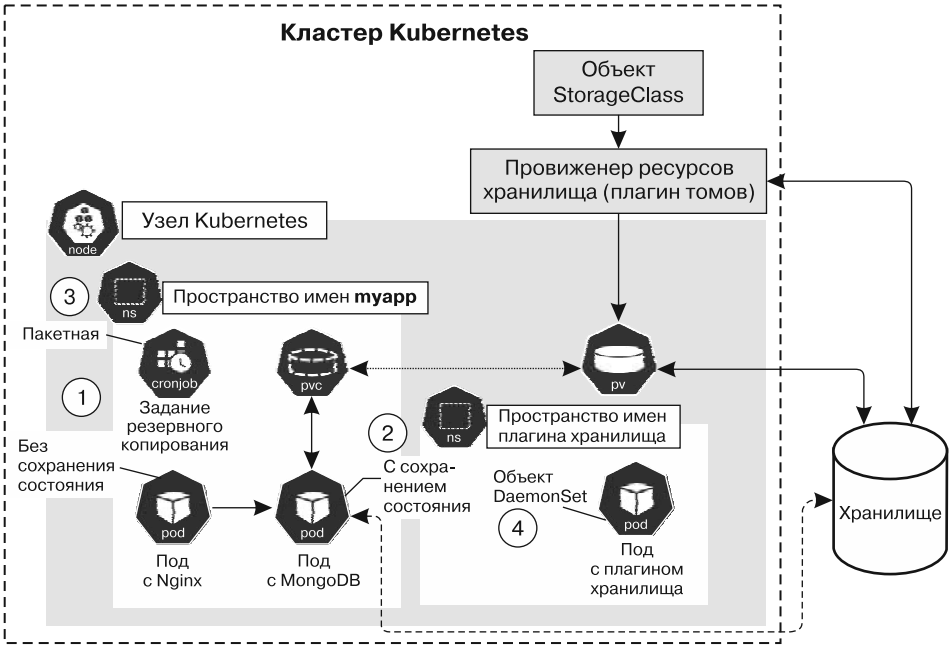

Рис. 11.1. Типы рабочих нагрузок в Kubernetes

При реализации облачных приложений обычно требуются следующие типы рабочих нагрузок.

- *Без сохранения состояния (stateless)*. В мире контейнеров приложениями без сохранения состояния считаются те, которые не сохраняют данные (состояние) внутри самого контейнера. Представьте два контейнера с сервером Nginx, выполняющих одинаковую функцию: один хранит пользовательские данные в файле внутри контейнера, а другой использует отдельный контейнер с MongoDB для постоянного хранения данных. Хотя оба контейнера решают одну и ту же задачу, первый сохраняет состояние, так как полагается на внутреннее хранилище. Второй, полагающийся на внешнюю базу данных, не сохраняет состояние. Такой подход упрощает управление и масштабирование приложений в Kubernetes, поскольку позволяет легко перезапускать или заменять их без риска потери данных. Обычно для управления рабочими нагрузками без сохранения состояния в Kubernetes используются объекты Deployment.
- *С сохранением состояния (stateful)*. Поды и контейнеры считаются сохраняющими состояние, если они хранят изменяемые данные внутри себя. Хорошим примером является под с MySQL или MongoDB, который читает и записывает данные в объект PersistentVolume. Управление такими рабочими нагрузками усложняется необходимостью контролировать «липкие» сессии и разделение данных при обновлениях, откатах и масштабировании. По возможности рабочие нагрузки с сохранением состояния следует выносить за пределы кластера Kubernetes, используя, например, облачные базы данных, предоставляемые по модели *SaaS* (Software-as-a-Service — программное обеспечение как услуга). В Kubernetes для управления рабочими нагрузками с сохранением состояния применяются объекты StatefulSet. Подробнее о них мы поговорим в главе 12.
- *Объекты Job и CronJob*. Эти рабочие нагрузки предназначены для выполнения заданий по расписанию или по требованию. В зависимости от приложения пакетные (batch) рабочие нагрузки могут использовать тысячи контейнеров и множество узлов. К ним могут относиться любые процессы, выполняемые *в фоновом режиме*. Контейнеры для пакетной обработки также не должны хранить состояние, чтобы упростить возобновление прерванных заданий. В Kubernetes для управления такими нагрузками используются объекты Job и CronJob (см. главу 4).
- *Объект DaemonSet*. Бывают случаи, когда необходимо запустить рабочую нагрузку на каждом узле для обеспечения работы кластера Kubernetes. Это может быть актуально для системы мониторинга, логирования, агента управления хранилищем (например, для PersistentVolume) и т. д. В таких случаях

используется специальный тип объекта — DaemonSet, который гарантирует запуск копии рабочей нагрузки на каждом узле кластера. Подробнее о нем вы узнаете в главе 13.

Теперь, когда вы знаете о различных типах рабочих нагрузок в Kubernetes, можем перейти к более детальному рассмотрению рабочих нагрузок без сохранения состояния, управляемых объектами Deployment. В сущности, они предоставляют декларативный и контролируемый механизм обновлений подов и объектов ReplicaSet.

С помощью объектов Deployment можно декларативно выполнять следующие операции.

- *Выкатывание нового объекта ReplicaSet*. Объект Deployment отлично подходит для управления контролируемыми выкатываниями. Внутри него можно определить новый объект ReplicaSet с желаемым шаблоном пода. После этого система Kubernetes организует постепенное выкатывание обновлений, увеличивая число реплик в новом ReplicaSet и одновременно уменьшая их в старом. Это минимизирует время простоя и обеспечивает плавный переход.
- *Контролируемое выкатывание обновлений при изменении шаблона пода*. Объект Deployment позволяет обновлять шаблон пода в его определении. При применении обновленного объекта Deployment Kubernetes выполняет постепенное обновление, увеличивая число реплик в новом ReplicaSet и одновременно уменьшая их в старом. Это позволяет внедрять изменения в поды приложения контролируемым образом.
- *Откат к предыдущей версии*. Объект Deployment сохраняет историю своих ревизий. Если новая версия вызывает проблемы, вы можете легко откатиться к предыдущей стабильной версии. Это позволяет быстро отменить изменения и минимизировать сбои.
- *Масштабирование объекта ReplicaSet*. Масштабирование объекта Deployment напрямую влияет на связанный с ним объект ReplicaSet. Достаточно указать желаемое количество реплик для Deployment, и Kubernetes автоматически скорректирует число реплик в ReplicaSet.
- *Приостановка и возобновление выкатывания*. Объект Deployment позволяет приостановить выкатывание нового объекта ReplicaSet при необходимости устранить проблемы или осуществить дополнительную настройку. После этого можно возобновить процесс. Подобная гибкость весьма полезна в ходе развертывания.

Таким образом, объекты Deployment обеспечивают сквозной конвейер управления компонентами без сохранения состояния, работающими в кластере

Kubernetes. Как правило, они используются совместно с объектами Service (описанными в главе 8) для достижения высокой отказоустойчивости приложения, мониторинга его состояния и интеллектуальной балансировки входящего трафика.

Теперь рассмотрим спецификацию Deployment и создадим такой объект в кластере Kubernetes.

Создание объекта Deployment

Для создания YAML-манифеста объекта Deployment можно использовать любые поддерживаемые инструменты. Например, команда `kubectl create deployment` выводит готовый YAML-шаблон с заполненными значениями:

```
$ kubectl create deployment my-deployment --replicas=1 --image=my-image:latest
--dry-run=client --port=80 -o yaml
```

Вы можете перенаправить этот вывод в файл и использовать его как основу для своих конфигураций Deployment:

```
$ kubectl create deployment my-deployment --replicas=1 --image=my-image:latest
--dry-run=client --port=80 -o yaml > my-deployment.yaml
```

Сначала рассмотрим структуру YAML-манифеста `nginx-deployment.yaml`, который поддерживает три реплики пода с Nginx:

```
# nginx-deployment.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment-example
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx
      environment: test
  minReadySeconds: 10
  strategy:
    type: RollingUpdate
    rollingUpdate:
      maxUnavailable: 1
      maxSurge: 1
  template:
    metadata:
      labels:
        app: nginx
        environment: test
```

```
spec:
  containers:
    - name: nginx
      image: nginx:1.17
      ports:
        - containerPort: 80
```

Как видите, структура спецификации Deployment почти идентична объекту ReplicaSet, но предусматривает дополнительные параметры для настройки стратегии выкатывания новых версий. Приведенный выше YAML-манифест содержит четыре основных компонента.

- `replicas` — определяет количество реплик пода, которые запускаются на основе заданного шаблона (`template`) и соответствуют селектору меток (`selector`). Для поддержания требуемого числа реплик поды могут создаваться или удаляться. Это свойство используется базовым объектом ReplicaSet.
- `selector` — селектор меток, задающий правила идентификации подов, находящихся под управлением базового объекта ReplicaSet. Он может быть основан как на равенстве, так и на множестве. В случае с Deployment базовый объект ReplicaSet также использует сгенерированную метку `pod-template-hash` для предотвращения конфликтов между дочерними объектами ReplicaSet при выкатывании новой версии. Это также предотвращает случайное попадание «голых» подов под управление контроллера, что нередко случается с простыми объектами ReplicaSet. Тем не менее Kubernetes допускает использование перекрывающихся селекторов для разных объектов Deployment и других контроллеров. Однако в таких случаях существует риск их конфликтов и непредсказуемого поведения.
- `template` — определяет шаблон для создания пода. Метки, указанные в разделе `metadata`, должны соответствовать условиям, определенным в разделе `selector`.
- `strategy` — определяет стратегию замены существующих подов новыми. Подробнее об этих стратегиях вы узнаете в следующих разделах. В предыдущем примере мы использовали стратегию по умолчанию — `RollingUpdate`, которая поочередно заменяет поды предыдущей версии подами новой версии. Это гарантирует отсутствие простоя. А в сочетании с объектами Service и проверками готовности такая стратегия обеспечивает балансировку нагрузки, благодаря которой трафик направляется только на те поды, которые готовы его обрабатывать.

Спецификация объекта Deployment предоставляет широкие возможности для его настройки под ваши нужды. Более подробно о них можно узнать в документации: https://kubernetes.io/docs/reference/generated/kubernetes-api/v1.31/#deployment-v1-apps (укажите версию своего кластера Kubernetes, например v1.31).

Чтобы лучше понять взаимосвязь между объектом Deployment, его дочерним объектом ReplicaSet и подами, рассмотрим следующую схему (рис. 11.2).

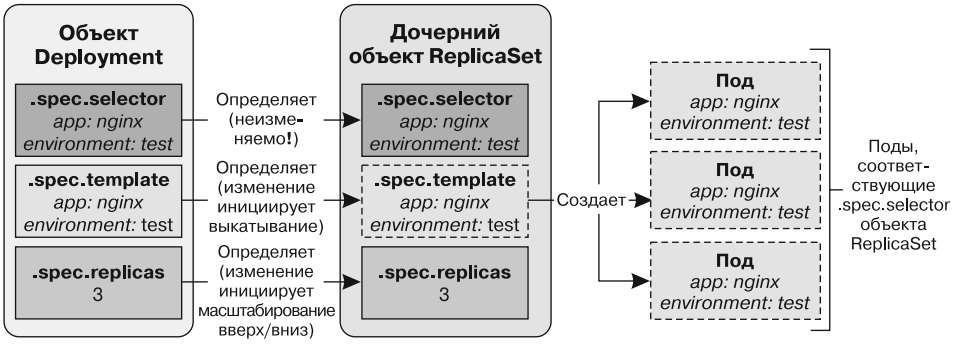

Рис. 11.2. Объект Deployment в Kubernetes

После определения и создания объекта Deployment изменить его селектор невозможно. В противном случае могли бы появиться «осиротевшие» объекты ReplicaSet. Над существующим объектом Deployment можно выполнить две важные операции.

- *Изменение шаблона.* Чаще всего бывает необходимо обновить определение пода, указав новую версию образа приложения. Это инициирует выкатывание обновлений в соответствии с выбранной стратегией.
- *Изменение числа реплик.* Инициирует масштабирование объекта ReplicaSet вверх или вниз.

Теперь применим наш YAML-манифест `nginx-deployment.yaml` к кластеру декларативным способом с помощью команды `kubectl apply`:

```
$ kubectl apply -f ./nginx-deployment.yaml
```

Использование флага --record позволяет отслеживать вносимые в объекты изменения и определять, какие именно команды их вызвали. Результатом его применения является автоматическое добавление аннотации kubernetes.io/change-cause, содержащей сведения о команде. Однако флаг --record был признан устаревшим и в будущем будет удален. Поэтому лучшей практикой является самостоятельное добавление этой аннотации при обновлении Deployment, если ваши процессы зависят от ее наличия.

При необходимости вы можете вручную добавить аннотацию к объекту Deployment с помощью команды `kubectl annotate`:

```
$ kubectl annotate deployment/nginx-deployment-example kubernetes.io/
change-cause='Updated image to 1.2.3'
```

Сразу после создания объекта Deployment можно использовать команду `kubectl rollout`, чтобы отслеживать статус развертывания в реальном времени:

```
$ kubectl rollout status deployment nginx-deployment-example
Waiting for deployment "nginx-deployment-example" rollout to finish: 0 of 3
updated replicas are available...
deployment "nginx-deployment-example" successfully rolled out
```

Эта команда позволяет получить подробные сведения о происходящем во время выкатывания объекта Deployment. Можно также использовать стандартные команды `kubectl get` или `kubectl describe`:

```
$ kubectl get deploy nginx-deployment-example
NAME                       READY   UP-TO-DATE   AVAILABLE   AGE
nginx-deployment-example   3/3     3            3           6m21s
```

Как видите, объект Deployment был успешно создан и все три пода находятся в состоянии готовности.

Вместо слова deployment в командах kubectl можно использовать сокращение deploy.

Просмотреть базовые объекты ReplicaSet можно следующим образом:

```
$ kubectl get rs
NAME                                  DESIRED   CURRENT   READY   AGE
nginx-deployment-example-5b8dc6b8cd   3         3         3       2m17s
```

Обратите внимание на сгенерированный хеш `5b8dc6b8cd` в имени объекта ReplicaSet — он совпадает со значением упомянутой ранее метки `pod-template-hash`.

Наконец, мы можем просмотреть поды в кластере, созданные объектом Deployment, выполнив следующую команду:

```
$ kubectl get pods
NAME                                        READY   STATUS    RESTARTS   AGE
nginx-deployment-example-5b8dc6b8cd-lj2bz   1/1     Running   0          3m30s
nginx-deployment-example-5b8dc6b8cd-nxkbj   1/1     Running   0          3m30s
nginx-deployment-example-5b8dc6b8cd-shzmd   1/1     Running   0          3m30s
```

Итак, вы создали и исследовали свой первый объект Deployment в Kubernetes! Далее мы поговорим об использовании сервисов для публикации объектов Deployment и обеспечения доступа внешнего трафика в кластер.

Публикация подов Deployment с помощью объектов Service

Объекты Service (сервис) подробно рассматривались в главе 8, поэтому в данном подразделе мы лишь кратко опишем их роль и то, как они обычно используются вместе с объектами Deployment.

Следующая схема служит простейшей иллюстрацией различных сетей внутри кластера Kubernetes (рис. 11.3).

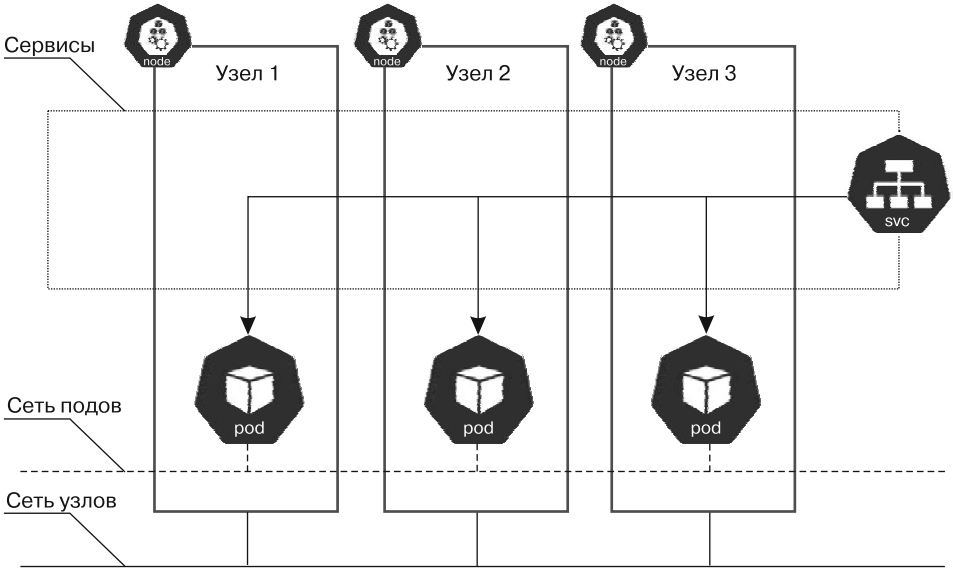

Рис. 11.3. Разные сети в Kubernetes

Объекты Service в Kubernetes позволяют сделать поды доступными как для других подов внутри кластера, так и для конечных пользователей. Они являются важнейшими строительными блоками для создания высокодоступных и отказоустойчивых Kubernetes-приложений, так как обеспечивают балансировку нагрузки, распределяя входящий трафик по готовым к работе и исправным подам.

Объекты Deployment, в свою очередь, обеспечивают репликацию подов, автоматический перезапуск в случае сбоев, удобное масштабирование, а также контролируемое выкатывание обновлений и откат к предыдущим версиям. Но здесь есть нюанс: поды, создаваемые объектами ReplicaSet или Deployment, имеют ограниченный жизненный цикл. Рано или поздно их работа будет завершена и их место займут новые реплики подов с другими IP-адресами. Представьте, что у вас есть объект Deployment с подами веб-сервера, которым необходимо взаимодействовать с подами другого объекта Deployment, например с бэкенд-

подами. Поды веб-сервера не могут полагаться на IP-адреса или DNS-имена бэкенд-подов, так как они подвержены изменениям. Эта проблема решается с помощью сервисов, обеспечивающих надежное сетевое взаимодействие между подами.

Если вкратце, то объект Service нацеливается на определенный набор подов, задаваемый селекторами меток. Эти селекторы работают по тому же принципу, что и в случае с ReplicaSet и Deployment. Чаще всего сервис создают для уже существующего объекта Deployment, используя тот же селектор меток.

Сервис предоставляет стабильное DNS-имя и IP-адрес, а также отслеживает результат работы селекторов, обновляя список IP-адресов соответствующих подов в связанном объекте Endpoint. Для организации внутреннего взаимодействия в кластере чаще всего используются сервисы типа ClusterIP. Для предоставления доступа к подам извне можно применять сервисы типа NodePort или LoadBalancer, что особенно актуально в облачных средах.

Следующая схема демонстрирует процесс взаимодействия сервисов с объектами Deployment в Kubernetes (рис. 11.4).

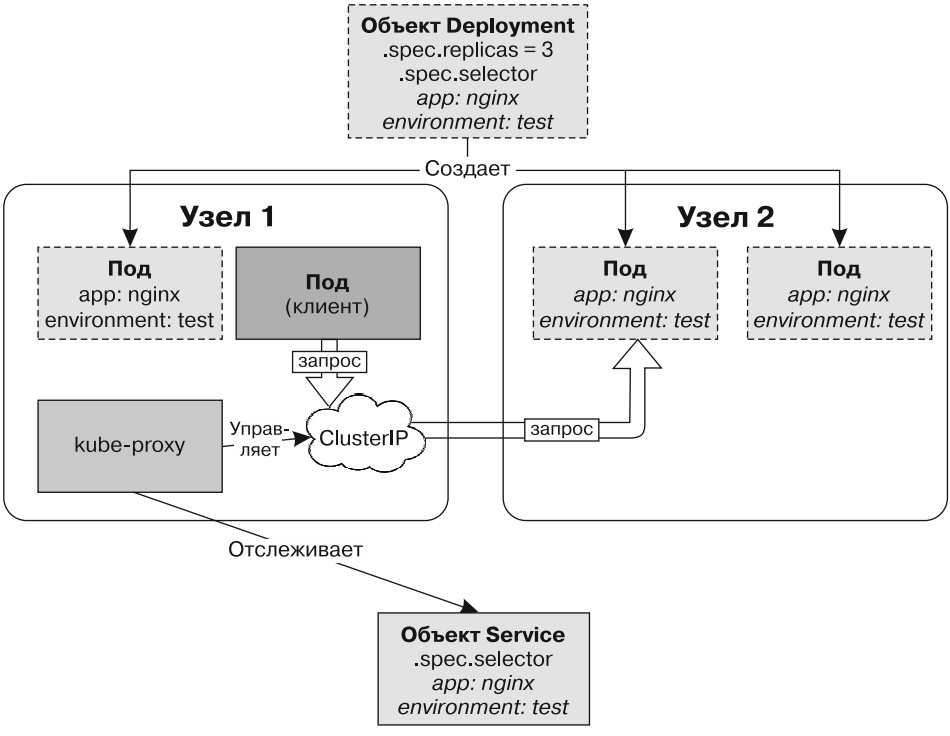

Рис. 11.4. Под-клиент выполняет запросы к объекту Deployment, опубликованному через сервис типа ClusterIP

Эта схема показывает, как любой клиентский под в кластере может взаимодействовать с подами `nginx`, созданными объектом Deployment и опубликованными через сервис типа ClusterIP. По сути, ClusterIP является виртуальным IP-адресом, которым управляет `kube-proxy`, запущенный на каждом узле. Компонент `kube-proxy` отвечает за сложную маршрутизацию трафика внутри кластера и гарантирует ее полную прозрачность для клиентских подов — им не нужно знать, взаимодействуют ли они с подом на том же узле, на другом узле или даже с внешним компонентом. `kube-proxy` отслеживает изменения в объекте Service и обеспечивает соблюдение всех необходимых правил маршрутизации на каждом узле для корректной доставки трафика. Обычно для управления маршрутизацией `kube-proxy` использует режим iptables или IPVS (IP Virtual Server).

Роль объекта Service заключается в определении набора готовых подов, которые должны быть *скрыты* за стабильным ClusterIP. Обычно внутренние клиенты не обращаются к подам напрямую через ClusterIP, а используют короткое DNS-имя, совпадающее с именем сервиса, например `nginx-service-example`. Это имя разрешается в ClusterIP с помощью внутреннего DNS-сервиса кластера. Альтернативным вариантом является использование *полного доменного имени* (FQDN) в формате `<`*`имя_сервиса`*`>.<`*`пространство_имен`*`>.svc.<`*`домен_кластера`*`>`, например: `nginx-service-example.default.svc.cluster.local`.

Сервисы типа LoadBalancer или NodePort, делающие поды доступными для внешнего трафика, имеют схожий принцип работы: они также предоставляют ClusterIP для внутреннего взаимодействия, но предусматривают настройку дополнительных компонентов для доставки внешнего трафика в кластер.

Теперь, когда у вас есть все необходимые знания о сервисах и их взаимодействии с объектами Deployment, применим их на практике.

Более подробную информацию о сервисах Kubernetes и их типах можно найти в главе 8.

Создание сервиса декларативным способом

Для публикации объекта Deployment `nginx-deployment-example` с помощью сервиса `nginx-service-example` типа LoadBalancer выполните такие шаги.

1. Создайте манифест `nginx-service.yaml` со следующим содержимым:

```
# nginx-service.yaml
apiVersion: v1
kind: Service
metadata:
  name: nginx-service-example
spec:
  selector:
```

```
    app: nginx
    environment: test
  type: LoadBalancer
  ports:
    - port: 80
      protocol: TCP
      targetPort: 80
```

Селектор меток этого сервиса совпадает с селектором объекта Deployment. Спецификация сервиса указывает на необходимость опубликовать наш объект Deployment на порте `80` облачного балансировщика нагрузки и перенаправить трафик с этого порта на поды.

В зависимости от того, как развернут ваш кластер Kubernetes, использование сервиса типа LoadBalancer может оказаться невозможным. В этом случае для выполнения данного упражнения вы можете использовать сервис типа NodePort или ограничиться простым ClusterIP, пропустив часть, связанную с организацией внешнего доступа. При использовании таких локальных развертываний, как minikube, для получения доступа к сервису следует использовать команду minikube service. Более подробную информацию можно найти по адресу https://minikube.sigs.k8s.io/docs/commands/service/.

2. Создайте объект `nginx-service-example` и используйте команды `kubectl get` или `kubectl describe`, чтобы просмотреть информацию о новом сервисе и связанном с ним балансировщике нагрузки:

```
$ kubectl apply -f nginx-service.yaml
service/nginx-service-example created

$ kubectl describe service nginx-service-example
Name:                     nginx-service-example
Namespace:                default
Labels:                   <none>
Annotations:              <none>
Selector:                 app=nginx,environment=test
...<фрагмент опущен для краткости>...
Endpoints:                10.244.1.2:80,10.244.2.2:80,10.244.2.3:80
Session Affinity:         None
External Traffic Policy:  Cluster
Events:                   <none>
```

3. Теперь попробуем получить доступ к этому сервису из другого пода, как мы это делали в главе 8. Создадим под `k8sutils` и протестируем доступ к сервису:

```
$ kubectl apply -f ../Chapter07/k8sutils.yaml
pod/k8sutils created
$ kubectl exec -it k8sutils -- curl nginx-service-example.default.svc.
cluster.local | grep Welcome -A2
<title>Welcome to nginx!</title>
<style>
```

```
    body {
--
  <h1>Welcome to nginx!</h1>
<p>If you see this page, the nginx web server is successfully installed and
working. Further configuration is required.</p>
```

Этот пример демонстрирует то, как сервисы позволяют сделать поды объектов Deployment доступными для внешнего трафика. Далее мы посмотрим, как можно добиться того же результата с помощью императивных команд.

Создание сервиса императивным способом

Аналогичного результата можно добиться с помощью императивной команды `kubectl expose`. Создадим сервис для нашего объекта Deployment `nginx-deployment-example`:

```
$ kubectl expose deployment --type=LoadBalancer nginx-deployment-example
service/nginx-deployment-example exposed
```

Вот подробный разбор этого фрагмента кода.

- Команда создаст сервис с именем объекта Deployment, то есть `nginx-deployment-example`. Если вы хотите использовать другое имя (как в предыдущем примере), укажите параметр `--name=nginx-service-example`.
- Этот сервис будет использовать порт `80` — тот же, что и поды. Чтобы его изменить, используйте параметры `--port=<`*номер*`>` и `--target-port=<`*номер*`>`.

Все доступные параметры, используемые для публикации объекта Deployment, можно просмотреть с помощью команды kubectl expose deployment --help.

Императивный способ рекомендуется использовать только в ходе разработки или отладки. В рабочих средах следует применять такие декларативные подходы, как «инфраструктура как код» (IaC) и «конфигурация как код» (CaC).

В следующем пункте вы узнаете, как использовать проверки готовности, жизнеспособности и успешности запуска с объектами Deployment.

Роль проверок готовности, жизнеспособности и успешности запуска

В главе 8 вы узнали о том, что для каждого контейнера, работающего в поде, можно настроить три типа проверок (или проб):

- проверку готовности (`readinessProbe`);
- проверку жизнеспособности (`livenessProbe`);
- проверку успешности запуска (`startupProbe`).

Все эти проверки чрезвычайно полезны при настройке объекта Deployment. Всегда старайтесь предвидеть возможные сценарии жизненного цикла процессов, работающих в контейнерах, и настраивать соответствующие проверки для развертываний.

Учтите, что по умолчанию контейнеры в подах запускаются без каких-либо проверок. Система Kubernetes будет направлять трафик к контейнерам с подами, работающим за сервисом, только если эти контейнеры успешно стартовали. В случае сбоя контейнеры будут перезапущены согласно политике `always-restart`, используемой по умолчанию. Это означает, что ответственность за выбор проверок и их параметров в каждом конкретном случае лежит на вас. Важно также понимать возможные последствия некорректной настройки таких проверок. Например, чрезмерно строгая проверка жизнеспособности со слишком короткими тайм-аутами может привести к ошибочному перезапуску контейнеров, снижающему доступность приложения.

Теперь посмотрим, как можно настроить *проверку готовности* объекта Deployment и как она работает в реальном времени.

Более подробно о настройке других типов проверок можно узнать в главе 8 данной книги, а также в официальной документации по адресу https://kubernetes.io/docs/tasks/configure-pod-container/configure-liveness-readiness-startup-probes/.

Используемый нами объект Deployment с сервером Nginx очень прост и не требует настройки отдельной проверки готовности. Вместо этого мы настроим контейнер так, чтобы иметь возможность вызывать успешное или неуспешное прохождение проверки готовности по своему усмотрению. Идея состоит в следующем: во время инициализации контейнера создается пустой файл `/usr/share/nginx/html/ready`, который будет доступен по пути `/ready` и предоставляться сервером Nginx в качестве обычного файла. Затем настраивается проверка готовности типа `httpGet`, которая периодически обращается к пути `/ready` и признает контейнер готовым при получении кода, означающего успешную обработку запроса. Таким образом, удаляя или воссоздавая файл `ready` с помощью команды `kubectl exec`, мы можем легко моделировать сбои в подах, приводящие к провалу или успешному прохождению проверки готовности.

Для настройки и тестирования проверки готовности выполните следующие шаги.

1. Удалите существующий объект Deployment с помощью команды:

```
$ kubectl delete deployment nginx-deployment-example
```

2. Создайте новый YAML-манифест объекта Deployment:

```
# nginx-deployment-readinessprobe.yaml
apiVersion: apps/v1
kind: Deployment
```

```
  metadata:
    name: nginx-deployment-readiness
  spec:
    replicas: 3
    selector:
      matchLabels:
        app: nginx
        environment: test
    minReadySeconds: 10
    strategy:
      type: RollingUpdate
      rollingUpdate:
        maxUnavailable: 1
        maxSurge: 1
    ...<продолжение далее>...
```

Теперь раздел `spec.template` выглядит так:

```
# nginx-deployment-readinessprobe.yaml
...<продолжение>...
  template:
    metadata:
      labels:
        app: nginx
        environment: test
    spec:
      containers:
        - name: nginx
          image: nginx:1.25.4
          ports:
            - containerPort: 80
          command:
            - /bin/sh
            - -c
            - |
              touch /usr/share/nginx/html/ready
              echo "You have been served by Pod with IP address:
                $(hostname -i)" > /usr/share/nginx/html/index.html
              nginx -g "daemon off;"
          readinessProbe:
            httpGet:
              path: /ready
              port: 80
            initialDelaySeconds: 5
            periodSeconds: 2
            timeoutSeconds: 10
            successThreshold: 1
            failureThreshold: 2
```

В манифесте Deployment было изменено несколько частей. Во-первых, мы переопределили команду, выполняемую при запуске контейнера, с помощью параметра `command` и передали дополнительные аргументы. Параметр `command`

установлен в значение `/bin/sh` для выполнения пользовательской команды оболочки. Дополнительные аргументы сформированы следующим образом.

- `-c` — аргумент для `/bin/sh`, указывающий на то, что далее следует команда, которую нужно выполнить в оболочке.
- `touch /usr/share/nginx/html/ready` — первая команда, выполняемая в оболочке контейнера. Она создает пустой файл `ready`, который сервер Nginx будет предоставлять через конечную точку `/ready`.
- `echo "You have been served by Pod with IP address: $(hostname -i)" > /usr/share/nginx/html/index.html` — вторая команда, которая записывает в `index.html` информацию об IP-адресе пода в кластере. Команда `hostname -i` используется для получения IP-адреса контейнера. Для каждого пода Deployment это значение будет своим.
- `nginx -g "daemon off;"` — наконец, выполняется стандартная команда запуска образа `nginx:1.25.4`. Это инициирует запуск веб-сервера Nginx в качестве основного процесса контейнера.

Обычно подобные настройки выполняются с помощью нового контейнерного образа, который наследуется от базового (например, образа nginx) и содержит специализированный сценарий приложения. Представленный здесь метод используется в демонстрационных целях и показывает, насколько гибкой является среда выполнения Kubernetes. Для создания пользовательских контейнерных образов обратитесь к примеру Chapter11/Containerfile в репозитории GitHub.

Вторая группа изменений, внесенных в YAML-манифест Deployment, касается определения проверки готовности (`readinessProbe`), которая настроена следующим образом.

- Данная проверка имеет тип `httpGet` и выполняет `GET`-запрос к конечной точке `/ready` на порте `80` контейнера.
- `initialDelaySeconds`: запросы начинают отправляться через 5 секунд после запуска контейнера.
- `periodSeconds`: запросы отправляются каждые 2 секунды.
- `timeoutSeconds`: время ожидания перед тайм-аутом составляет 10 секунд.
- `successThreshold`: для признания пода готовым достаточно одной успешной проверки.
- `failureThreshold`: для признания пода неготовым требуются две последовательные неудачные проверки. Значение больше `1` помогает избежать ложных срабатываний.

Для создания объекта Deployment выполните следующие шаги.

1. Примените новый YAML-манифест:

```
$ kubectl apply -f ./nginx-deployment-readinessprobe.yaml
```

2. Убедитесь в том, что сервис `nginx-service-example` отображает IP-адреса бэкенд-подов. Вы увидите три конечные точки, соответствующие подам Deployment и готовые обслуживать трафик:

```
$ kubectl describe svc nginx-service-example
Name:                     nginx-service-example
Namespace:                default
Labels:                   <none>
Annotations:              <none>
Selector:                 app=nginx,environment=test
Type:                     LoadBalancer
IP Family Policy:         SingleStack
IP Families:              IPv4
IP:                       10.96.231.126
IPs:                      10.96.231.126
Port:                     <unset>  80/TCP
TargetPort:               80/TCP
NodePort:                 <unset>  32563/TCP
Endpoints:                10.244.1.6:80,10.244.1.7:80,10.244.2.6:80
Session Affinity:         None
External Traffic Policy:  Cluster
Events:                   <none>
```

3. Протестируйте доступ к веб-серверу Nginx с помощью пода `k8sutils`, созданного ранее в главе. Вы заметите, что ответы приходят от подов с разными IP-адресами. Это объясняется тем, что объект Deployment содержит три реплики, поэтому запросы могут направляться разным подам:

```
$ kubectl exec -it k8sutils -- curl nginx-service-example.default.svc.
cluster.local
You have been served by Pod with IP address: 10.244.1.7
$ kubectl exec -it k8sutils -- curl nginx-service-example.default.svc.
cluster.local
You have been served by Pod with IP address: 10.244.2.6
```

4. А теперь смоделируем неудачное прохождение проверки готовности первого пода. В нашем случае это `nginx-deployment-readiness-69dd4cfdd9-4pkwr` с IP-адресом `10.244.1.7`. Для этого достаточно удалить файл `ready` внутри контейнера с помощью команды `kubectl exec`:

```
$ kubectl exec -it nginx-deployment-readiness-69dd4cfdd9-4pkwr -- rm /usr/
share/nginx/html/ready
```

5. Теперь проверка готовности начнет возвращать ошибки, но не сразу. Мы задали параметр `failureThreshold=2`, поэтому для провала проверки

необходимо как минимум две неудачные попытки подряд, выполняемые с интервалом 2 секунды. Позже вы заметите, что трафик распределяется между двумя оставшимися подами, находящимися в состоянии готовности.

6. Если вы выполните команду `kubectl describe svc nginx-service-example`, то увидите, что вам доступны только две конечные точки:

```
$ kubectl describe svc nginx-service-example | grep Endpoint
Endpoints:              10.244.1.6:80,10.244.2.6:80
```

7. В списке событий, связанных с подом, также будет указано, что он не находится в состоянии готовности:

```
$ kubectl describe pod nginx-deployment-readiness-69dd4cfdd9-4pkwr
Name:     nginx-deployment-readiness-69dd4cfdd9-4pkwr
...<фрагмент опущен для краткости>...
Normal   Started    21m                  kubelet   Started container nginx
Warning  Unhealthy  72s (x25 over 118s)  kubelet   Readiness probe failed
with status code: 404
```

Можно пойти еще дальше и удалить файл `ready` в двух других подах, чтобы сервис полностью перестал работать:

```
$ kubectl exec -it nginx-deployment-readiness-69dd4cfdd9-7n2kz -- rm /usr/
share/nginx/html/ready
$ kubectl exec -it nginx-deployment-readiness-69dd4cfdd9-t7rp2 -- rm /usr/
share/nginx/html/ready
```

Поды находятся в состоянии `Running`, однако ни один из них не может принимать трафик из-за непрохождения проверки готовности.

```
$ kubectl get po -w
NAME                                          READY   STATUS    RESTARTS  AGE
k8sutils                                      1/1     Running   0         166m
nginx-deployment-readiness-69dd4cfdd9-4pkwr   0/1     Running   0         25m
nginx-deployment-readiness-69dd4cfdd9-7n2kz   0/1     Running   0         25m
nginx-deployment-readiness-69dd4cfdd9-t7rp2   0/1     Running   0         25m
```

Теперь при проверке сервиса с помощью пода `k8sutils` вы увидите, что запросы остаются в состоянии ожидания и в конечном счете завершаются по тайм-ауту. Таким образом, мы смоделировали провал проверки готовности всех реплик пода в объекте Deployment.

```
$ kubectl exec -it k8sutils -- curl nginx-service-example.default.svc.
cluster.local
curl: (7) Failed to connect to nginx-service-example.default.svc.cluster.
local port 80 after 5 ms: Couldn't connect to server
command terminated with exit code 7
```

8. Восстановим работу одного из подов, воссоздав файл `ready`. Для этого обновите страницу, чтобы запрос перешел в состояние ожидания, и выполните команду для создания файла `ready`:

```
$ kubectl exec -it nginx-deployment-readiness-69dd4cfdd9-4pkwr -- touch /
usr/share/nginx/html/ready
```

   Примерно через 2 секунды (интервал между проверками) ожидающий запрос в браузере будет успешно выполнен и вы получите ответ от Nginx.

```
$ kubectl exec -it k8sutils -- curl nginx-service-example.default.svc.
cluster.local
You have been served by Pod with IP address: 10.244.1.7
```

Итак, вы успешно настроили и протестировали проверку готовности подов вашего объекта Deployment! Теперь вы знаете, как работают подобные проверки и как их можно использовать вместе с сервисами, предоставляющими доступ к объекту Deployment.

Далее обсудим способ масштабирования таких объектов.

Масштабирование объекта Deployment

Главное достоинство объекта Deployment заключается в возможности его практически мгновенного масштабирования. Если он опубликован через сервис, новые поды будут автоматически обнаружены и добавлены в список конечных точек при увеличении числа реплик или удалены из этого списка при его уменьшении. Порядок действий при этом следующий.

1. Для начала отмасштабируем объект Deployment декларативным способом. Откройте манифест `nginx-deployment-readinessprobe.yaml` и измените число реплик:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment-readiness
spec:
  replicas: 10
...
```

2. Примените изменения к кластеру с помощью команды `kubectl apply`:

```
$ kubectl apply -f ./nginx-deployment-readinessprobe.yaml
deployment.apps/nginx-deployment-readiness configured
```

   Теперь, если вы отобразите список подов с помощью команды `kubectl get pods`, то заметите, как создаются новые поды. Аналогичным образом, если вы

посмотрите вывод команды `kubectl describe` для этого объекта Deployment, в списке событий увидите следующее:

```
$ kubectl describe deployments.apps nginx-deployment-readiness
Name:   nginx-deployment-readiness
...<фрагмент опущен для краткости>...
Events:
  Type    Reason             Age   From                   Message
  ----    ------             ----  ----                   -------
  Normal  ScalingReplicaSet  32m   deployment-controller   Scaled up replica
set nginx-deployment-readiness-69dd4cfdd9 to 3
  Normal  ScalingReplicaSet  9s    deployment-controller   Scaled up replica
set nginx-deployment-readiness-69dd4cfdd9 to 10 from 3
```

Того же результата можно достичь и с помощью императивной команды, однако такой способ рекомендуется использовать только в процессе разработки:

```
$ kubectl scale deploy nginx-deployment-readiness --replicas=10
deployment.apps/nginx-deployment-readiness scaled
```

3. Чтобы уменьшить число реплик декларативным способом, измените параметр `replicas` в манифесте `nginx-deployment-readinessprobe.yaml`:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment-readiness
spec:
  replicas: 2
  ...
```

4. Примените изменения с помощью команды `kubectl apply`:

```
$ kubectl apply -f ./nginx-deployment-readinessprobe.yaml
```

5. Аналогичный результат можно получить и императивным способом, выполнив следующую команду:

```
$ kubectl scale deploy nginx-deployment-readiness --replicas=2
```

Если вы примените к объекту Deployment команду `kubectl describe`, то увидите запись об уменьшении числа реплик в списке событий:

```
$ kubectl describe deploy nginx-deployment-readiness
...<фрагмент опущен для краткости>...
Events:
  Type    Reason             Age   From                   Message
  ----    ------             ----  ----                   -------
  Normal  ScalingReplicaSet  27s   deployment-controller   Scaled down
replica set nginx-deployment-readiness-69dd4cfdd9 to 2 from 10
```

Список событий очень полезен для отслеживания хронологии масштабирования и других операций, выполняемых над объектом Deployment.

Объекты Deployment также возможно масштабировать автоматически с помощью объекта HorizontalPodAutoscaler, о котором вы узнаете в главе 20.

Теперь обсудим способ удаления объекта Deployment из кластера.

Удаление объекта Deployment

Объект Deployment можно удалить двумя способами.

- Удалить объект Deployment вместе с принадлежащими ему подами. Это можно сделать, предварительно выполнив автоматическое уменьшение числа реплик.
- Удалить только объект Deployment, оставив поды нетронутыми.

Чтобы удалить объект Deployment вместе с подами, используйте стандартную команду `kubectl delete`:

```
$ kubectl delete deploy nginx-deployment-readiness
```

После завершения работы подов будет удален и сам объект Deployment.

Если же вы хотите удалить только объект Deployment, оставив поды, используйте параметр `--cascade=orphan`:

```
$ kubectl delete deploy nginx-deployment-readiness --cascade=orphan
```

После выполнения этой команды при проверке списка подов в кластере вы по-прежнему увидите все поды, ранее принадлежавшие объекту `nginx-deployment-example`.

В следующем разделе вы узнаете, как объект Deployment позволяет управлять ревизиями и выкатыванием новых версий.

Управление ревизиями и версиями приложений в Kubernetes с помощью объекта Deployment

До сих пор мы обсуждали только масштабирование объекта Deployment путем изменения значения параметра `replicas` в его спецификации. Однако это далеко не все, что можно сделать! Объект Deployment допускает изменение шаблона пода (`.spec.template`) в спецификации, запускающее выкатывание обновлений. Этот процесс может быть инициирован простым изменением, например

модификацией меток подов. Но чаще всего он связан с обновлением образа контейнера в определении пода. Это наиболее распространенный сценарий, позволяющий оператору кластера Kubernetes выполнять контролируемое и предсказуемое выкатывание новой версии образа, сопровождающееся созданием новой ревизии объекта Deployment.

Стратегия выкатывания обновлений, используемая объектом Deployment, указывается в поле YAML-манифеста `.spec.strategy.type`. Система Kubernetes изначально поддерживает две такие стратегии.

- `RollingUpdate` — стратегия по умолчанию, которая позволяет постепенно выкатывать новую версию приложения. Для этого используются два объекта ReplicaSet. Когда в спецификацию Deployment вносится изменение, инициирующее выкатывание обновлений, Kubernetes создает новый объект ReplicaSet с обновленным шаблоном пода и нулевым количеством реплик. Старый объект ReplicaSet на этом этапе остается без изменений. Далее начинается постепенное сокращение количества подов в старом объекте ReplicaSet с одновременным пропорциональным увеличением их числа в новом. Число подов, которые могут быть недоступны в процессе обновления (из-за неудачных проверок готовности), задается параметром `.spec.strategy.rollingUpdate.maxUnavailable`.

  Максимальное число дополнительных подов, которые могут быть запланированы сверх требуемого количества, задается параметром `.spec.strategy.rollingUpdate.maxSurge`. Кроме того, данная стратегия автоматически сохраняет историю ревизий, что позволяет в случае необходимости быстро откатиться к предыдущей версии.
- `Recreate` — стратегия попроще, подходящая для этапа разработки, в ходе которой все старые поды удаляются, а на их месте создаются новые. При этом старый объект ReplicaSet сразу удаляется и заменяется новым. Эту стратегию не рекомендуется применять в рабочей среде, за исключением специфических случаев.

Стратегии развертывания можно рассматривать как базовые строительные блоки для более сложных сценариев использования объекта Deployment. Например, сине-зеленые развертывания (blue/green deployments) легко реализовать в Kubernetes с помощью комбинации объектов Deployment и Service при изменении селекторов меток. Подробнее об этом можно прочитать в официальном блоге Kubernetes: https://kubernetes.io/blog/2018/04/30/zero-downtime-deployment-kubernetes-jenkins/.

Далее мы выполним выкатывание обновлений с использованием стратегии `RollingUpdate`. Более простую стратегию `Recreate` можно реализовать аналогичным образом.

Обновление объекта Deployment

В этом подразделе мы применим стратегию `RollingUpdate` на практике. Для начала пересоздадим объект Deployment, который использовали ранее для демонстрации проверки готовности.

1. Сделайте копию предыдущего YAML-манифеста:

```
$ cp nginx-deployment-readinessprobe.yaml nginx-deployment-rollingupdate.yaml
```

2. Убедитесь, что в манифесте `nginx-deployment-readinessprobe.yaml` указана стратегия `RollingUpdate` и используется образ `nginx:1.17`. Если вы выполняли шаги, описанные в предыдущих подразделах, эти параметры уже должны быть настроены:

```
# nginx-deployment-rollingupdate.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment-rollingupdate
spec:
  replicas: 3
...
  minReadySeconds: 10
  strategy:
    type: RollingUpdate
    rollingUpdate:
      maxUnavailable: 1
      maxSurge: 1
  template:
...
    spec:
      containers:
        - name: nginx
          image: nginx:1.17
...
          readinessProbe:
            httpGet:
...
```

3. В нашем примере для параметра `maxUnavailable` задано значение `1`. Это значит, что из трех подов (целевое количество) только один может быть временно недоступен (не готов). Следовательно, в любой момент времени как минимум два пода должны быть готовы обслуживать трафик.
4. Параметр `maxSurge`, значение которого также равно `1`, разрешает создание одного дополнительного пода сверх целевого количества (равного трем) во время выкатывания обновлений. Таким образом, в ходе этого процесса в кластере может присутствовать до четырех (готовых или не готовых) подов.

Значения этих параметров можно задавать и в процентах (например, 25 %), что особенно полезно при автоматическом масштабировании.

5. Параметр `minReadySeconds` (значение которого равно `10`) задает дополнительный промежуток времени, в течение которого под должен оставаться в состоянии готовности, чтобы можно было *объявить* о его успешном обновлении.
6. Примените этот YAML-манифест к кластеру:

```
$ kubectl apply -f nginx-deployment-rollingupdate.yaml
deployment.apps/nginx-deployment-rollingupdate created

$ kubectl get pod
NAME                                           READY STATUS   RESTARTS AGE
nginx-deployment-rollingupdate-69d855cf4b-nshn2  1/1   Running  0        24s
nginx-deployment-rollingupdate-69d855cf4b-pqvjh  1/1   Running  0        24s
nginx-deployment-rollingupdate-69d855cf4b-tdxzl  1/1   Running  0        24s
```

После создания объекта Deployment можно приступать к выкатыванию новой версии приложения. Для этого мы изменим образ в шаблоне пода на более свежую версию и посмотрим, что произойдет в процессе выкатывания обновлений. Выполните следующие шаги:

7. Измените контейнерный образ в манифесте объекта Deployment на `nginx:1.18`:

```
...
  template:
    metadata:
      labels:
        app: nginx
        environment: test
    spec:
      containers:
        - name: nginx
          image: nginx:1.18
...
```

8. Примените изменения к кластеру:

```
$ kubectl apply -f nginx-deployment-rollingupdate.yaml
deployment.apps/nginx-deployment-rollingupdate configured
```

9. Сразу после этого используйте команду `kubectl rollout status`, чтобы отслеживать прогресс в реальном времени:

```
$ kubectl rollout status deployment.apps/nginx-deployment-rollingupdate
deployment "nginx-deployment-rollingupdate" successfully rolled out
```

10. Выкатывание обновлений займет некоторое время, так как для объекта Deployment был задан параметр `minReadySeconds`, а для проверки готовности контейнера — `initialDelaySeconds`.

11. Используя команду `kubectl describe`, можно просмотреть список событий, показывающий, как масштабировались объекты ReplicaSet:

```
$ kubectl describe deploy nginx-deployment-rollingupdate
Name: nginx-deployment-rollingupdate
<фрагмент опущен для краткости>
Events:
  Type    Reason             Age    From                   Message
  ----    ------             ----   ----                   -------
  Normal  ScalingReplicaSet  8m21s  deployment-controller   Scaled up
replica set nginx-deployment-rollingupdate-69d855cf4b to 3
  Normal  ScalingReplicaSet  2m51s  deployment-controller   Scaled up
replica set nginx-deployment-rollingupdate-5479f5d87f to 1
  Normal  ScalingReplicaSet  2m51s  deployment-controller   Scaled down
replica set nginx-deployment-rollingupdate-69d855cf4b to 2 from 3
  Normal  ScalingReplicaSet  2m51s  deployment-controller   Scaled up
replica set nginx-deployment-rollingupdate-5479f5d87f to 2 from 1
  Normal  ScalingReplicaSet  2m24s  deployment-controller   Scaled down
replica set nginx-deployment-rollingupdate-69d855cf4b to 1 from 2
  Normal  ScalingReplicaSet  2m24s  deployment-controller   Scaled up
replica set nginx-deployment-rollingupdate-5479f5d87f to 3 from 2
  Normal  ScalingReplicaSet  2m14s  deployment-controller   Scaled down
replica set nginx-deployment-rollingupdate-69d855cf4b to 0 from 1
```

12. Теперь выведем список объектов ReplicaSet в кластере:

```
$ kubectl get rs
NAME                                          DESIRED  CURRENT  READY  AGE
nginx-deployment-rollingupdate-5479f5d87f  3        3        3      4m22s
nginx-deployment-rollingupdate-69d855cf4b  0        0        0      9m52s
```

Вы заметите, что старый объект ReplicaSet все еще присутствует, но число его подов было уменьшено до нуля. Это связано с тем, что объект Deployment сохраняет историю ревизий. Каждая ревизия соответствует своему набору реплик, который может использоваться для отката обновлений. Количество сохраняемых ревизий задается параметром `.spec.revisionHistoryLimit` (по умолчанию его значение равно `10`). История ревизий особенно важна, если вы вносите изменения императивным способом. Если же вы используете декларативный подход и фиксируете изменения в системе контроля версий, сохранение этой истории может быть менее критическим.

13. Наконец, убедимся, что поды действительно обновились до новой версии образа. Для этого используйте следующую команду и проверьте один из подов в выводе:

```
$ kubectl get po
NAME                                           READY STATUS  RESTARTS AGE
nginx-deployment-rollingupdate-5479f5d87f-2k7d6 1/1   Running 0        5m41s
nginx-deployment-rollingupdate-5479f5d87f-6gn9m 1/1   Running 0        5m14s
```

```
nginx-deployment-rollingupdate-5479f5d87f-mft6b 1/1   Running 0        5m41s

$ kubectl describe pod nginx-deployment-rollingupdate-5479f5d87f-2k7d6 |
grep 'Image:'
Image:  nginx:1.18
```

Этот вывод подтверждает успешное завершение выкатывания новой версии контейнерного образа `nginx`.

Вы также можете изменить контейнерный образ Deployment императивным способом с помощью команды kubectl set image deployment nginx-deployment-rollingupdate nginx=nginx:1.18. Этот подход хорошо сочетается с императивными откатами, но его не рекомендуется использовать в рабочей среде.

Далее мы поговорим об откате изменений, примененных к объектам Deployment.

Откат обновлений объекта Deployment

Если вы используете декларативный подход для внесения изменений в кластер Kubernetes и фиксируете каждое изменение в репозитории исходного кода, то выполнение отката сводится к отмене соответствующего коммита и повторному применению конфигурации. Обычно применение изменений осуществляется в рамках работы CI/CD-конвейера репозитория исходного кода, а не вручную оператором (например, командой разработчиков или администраторами). Это самый простой способ управления объектами Deployment, который рекомендуется применять при использовании таких парадигм, как «инфраструктура как код» (IaC) и «конфигурация как код» (CaC).

Ярким примером практического использования декларативной модели является проект Flux (https://fluxcd.io/). Изначально он развивался под эгидой организации Cloud Native Computing Foundation (CNCF), но впоследствии вышел из стадии инкубации и стал полноценным проектом в экосистеме CNCF. Flux лежит в основе подхода GitOps — методологии, предназначенной для реализации непрерывного развертывания облачных приложений. Этот подход ориентирован на удобство разработчиков, опирается на привычные инструменты, такие как Git и конвейеры непрерывного развертывания, и тем самым упрощает управление инфраструктурой.

Тем не менее Kubernetes поддерживает и императивный способ отката с использованием истории ревизий объекта Deployment. Причем императивный откат можно выполнить и для тех объектов Deployment, которые обновлялись

декларативным способом. Ниже показано, как использовать команду `kubectl` для выполнения отката.

1. Для начала императивно выкатим новую версию объекта Deployment. На этот раз обновим образ `nginx` до версии `1.19`:

```
$ kubectl set image deployment nginx-deployment-rollingupdate
nginx=nginx:1.19
deployment.apps/nginx-deployment-rollingupdate image updated
```

2. Здесь `nginx=nginx:1.19` означает, что мы задаем образ `nginx:1.19` для контейнера `nginx` в объекте `nginx-deployment-rollingupdate`.
3. Дождитесь завершения развертывания, проверив его статус с помощью команды `kubectl rollout status`:

```
$ kubectl rollout status deployment.apps/nginx-deployment-rollingupdate
deployment "nginx-deployment-rollingupdate" successfully rolled out
```

4. Допустим, новая версия `1.19` вызывает проблемы и команда решила откатиться к предыдущей стабильной версии.
5. Чтобы просмотреть все доступные ревизии объекта Deployment, используйте команду `kubectl rollout history`:

```
$ kubectl rollout history deployment.apps/nginx-deployment-rollingupdate
deployment.apps/nginx-deployment-rollingupdate
REVISION  CHANGE-CAUSE
1         <none>
2         <none>
3         <none>
```

6. Как видите, у нас есть три ревизии. Первая соответствует созданию исходного объекта Deployment. Вторая ревизия — это декларативное обновление Deployment с использованием образа `nginx:1.18`. Наконец, третья ревизия отражает последнее императивное обновление, инициировавшее выкатывание версии `nginx:1.19`. Поле `CHANGE-CAUSE` является пустым, поскольку мы не использовали флаг `--record`. Этот флаг признан устаревшим и будет удален в будущих версиях Kubernetes. При необходимости указать причину изменения в поле `CHANGE-CAUSE` вам следует вручную обновить аннотацию объекта Deployment, как было показано ранее в этой главе.
7. Ревизии, созданные в результате декларативных изменений, обычно содержат немного информации в поле `CHANGE-CAUSE`. Чтобы узнать больше о второй ревизии, выполните следующую команду:

```
$ kubectl rollout history deploy nginx-deployment-rollingupdate --revision=2
deployment.apps/nginx-deployment-rollingupdate with revision #2
Pod Template:
...<фрагмент опущен для краткости>...
```

```
  Containers:
    nginx:
      Image:    nginx:1.18
...<фрагмент опущен для краткости>...
```

8. Теперь выполним откат к этой ревизии. Поскольку речь идет о предыдущей ревизии, для этого достаточно ввести следующую команду:

```
$ kubectl rollout undo deploy nginx-deployment-rollingupdate
deployment.apps/nginx-deployment-rollingupdate rolled back
```

9. Команда для отката к конкретному номеру ревизии выглядит так:

```
$ kubectl rollout undo deploy nginx-deployment-rollingupdate --to-revision=2
```

10. Как и в случае с выкаткой обновлений, вы можете следить за процессом отката в реальном времени с помощью команды:

```
$ kubectl rollout status deploy nginx-deployment-rollingupdate
$ kubectl rollout history deployment.apps/nginx-deployment-rollingupdate
deployment.apps/nginx-deployment-rollingupdate
REVISION  CHANGE-CAUSE
1         <none>
3         <none>
4         <none>
```

11. Вы заметите, что версия 2 отсутствует, и для объекта Deployment была создана новая версия.

Обратите внимание, что откат можно выполнять и во время выкатывания обновлений. Это можно сделать как декларативным, так и императивным способом.

Чтобы приостановить и возобновить процесс выкатывания объекта Deployment, используйте команды kubectl rollout pause deployment nginx-deployment-example и kubectl rollout resume deployment nginx-deployment-example.

Итак, вы успешно выполнили откат объекта Deployment! В следующем разделе вы найдете рекомендации по управлению объектами Deployment в Kubernetes.

Обратите внимание, что откаты можно выполнять и во время уже запущенного отката — как декларативно, так и императивно.

Если нужно приостановить или возобновить выполняющийся откат, используйте команды kubectl rollout pause deployment nginx-deployment-example и kubectl rollout resume deployment nginx-deployment-example.

Поздравляем — вы успешно выполнили откат Deployment! В следующем подразделе мы поговорим о канареечном развертывании.

Канареечное развертывание

Канареечное развертывание (canary deployment) представляет собой эффективную стратегию снижения рисков при обновлении приложений. Название этого подхода восходит к практике использования канареек в угольных шахтах для обнаружения опасных газов. По аналогии канареечные развертывания предполагают выпуск новой версии приложения для ограниченного числа пользователей до того, как она станет доступна всем остальным.

Такой контролируемый процесс выкатывания позволяет протестировать новую версию в реальных условиях с минимальными рисками возникновения сбоев. Представьте, что вы обновляете сайт компании электронной коммерции. При традиционном подходе на новую версию переключается сразу весь сайт. В случае канареечного развертывания создается два объекта Deployment: стабильный, который обслуживает большинство пользователей, используя текущую версию приложения, и канареечный с новой версией, обслуживающей небольшую долю пользователей. При этом механизмы маршрутизации трафика, такие как Ingress-контроллеры или аннотации сервисов, направляют определенную часть запросов (например, 10 %) на канареечную версию.

Внимательно отслеживая работу канареечного развертывания с помощью таких метрик, как частота ошибок, время отклика и отзывы пользователей, вы можете оценить стабильность новой версии. Если все работает корректно, долю трафика, направляемого на канареечную версию, можно постепенно увеличивать. В случае возникновения проблем можно быстро откатить обновления и сохранить стабильную версию, не допустив влияния сбоев на всю пользовательскую базу.

Рассмотрим пример YAML-манифеста для канареечного развертывания фронтенд-приложения. Ниже приведена конфигурация стабильного объекта Deployment с образом `frontend-app:1.0`:

```
# Стабильная версия приложения
  ...
  name: frontend-stable
  replicas: 3
  ...
  labels:
    app: myapp
    tier: frontend
    version: stable
  ...
  image: frontend-app:1.0
```

Конфигурация канареечного развертывания с образом `frontend-app:2.0` выглядит так:

```
# Канареечная версия
  ...
  name: frontend-canary
  replicas: 1
  ...
  labels:
    app: myapp
    tier: frontend
    version: canary
  ...
  image: frontend-app:2.0
```

Нам также нужен сервис, распределяющий трафик между двумя версиями объекта Deployment. Обратите внимание на то, что в качестве селекторов мы используем только метки `app: myapp` и `tier: frontend`, чтобы сервис направлял трафик на поды версий `stable` и `canary`:

```
# Сервис (направляет трафик на оба объекта Deployment)
apiVersion: v1
kind: Service
metadata:
  name: frontend-service
spec:
  selector:
    app: myapp
    tier: frontend
  ports:
    - protocol: TCP
      port: 80
      targetPort: 8080
```

Изменяя количество реплик в каждом объекте Deployment, можно корректировать процент пользователей, обслуживаемых той или иной версией.

Альтернативный подход предполагает создание отдельных сервисов для стабильной и канареечной версии и использовать механизмы вроде Ingress-контроллеров или аннотаций сервисов для направления определенной доли трафика (например, 10 %) на канареечную версию.

Ключевым моментом является мониторинг канареечного развертывания и проверка того, что новая версия работает ожидаемым образом. На основе полученных результатов вы можете либо закрепить канареечную версию в качестве стабильной, либо удалить ее и обновить образ приложения (например, `image: frontend-app:2.0`) в стабильном объекте Deployment.

Подробнее о канареечном развертывании можно узнать по адресу https://kubernetes.io/docs/concepts/workloads/management/#canary-deployments.

В следующем разделе мы обсудим лучшие практики работы с объектами Deployment.

Лучшие практики работы с объектами Deployment

В этом разделе описаны лучшие практики работы с объектами Deployment. Этот список не является исчерпывающим, но служит хорошей отправной точкой при освоении Kubernetes.

Управляйте объектами Deployment декларативным способом

В мире DevOps и контейнеризованных приложений хорошей практикой считается использование декларативного подхода при изменении инфраструктуры и ПО. Именно этот подход лежит в основе таких парадигм, как «инфраструктура как код» и «конфигурация как код». В Kubernetes декларативные обновления легко выполняются с помощью команды `kubectl apply`, которую можно применить как к отдельному YAML-файлу, так и к целой папке таких манифестов.

Для удаления объектов, напротив, предпочтительнее использовать императивные команды, так как это дает более предсказуемый результат и снижает вероятность возникновения ошибок. Декларативное удаление ресурсов кластера уместно в CI/CD- и GitOps-сценариях, в которых весь процесс полностью автоматизирован.

Тот же принцип применим и к объектам Deployment. Выкатывать и откатывать их обновления гораздо проще, если файлы YAML-манифестов версионируются и хранятся в репозитории системы контроля версий. Команды `kubectl rollout undo` и `kubectl set image deployment` обычно не рекомендуется использовать в рабочих средах, особенно если с кластером одновременно взаимодействует несколько операторов.

Не используйте стратегию Recreate в рабочей среде

Использование стратегии `Recreate` может показаться хорошей идеей, поскольку она обеспечивает мгновенное обновление объектов Deployment. Однако при этом в работе приложения неизбежно возникает простой. Это происходит потому, что все поды старой ревизии объекта Deployment одновременно завершают работу и заменяются новыми. Между удалением старых подов и приведением новых в состояние готовности может пройти некоторое время. Такого простоя легко избежать, применяя стратегию `RollingUpdate` в рабочих средах.

Стратегия Recreate лучше всего проявляет себя в ситуациях, допускающих кратковременное прекращение работы приложения. Например, при развертывании значительных изменений или внедрении совершенно новой версии стратегия Recreate позволяет начать с чистого листа и гарантирует то, что новая версия работает независимо от предыдущей.

Не создавайте поды с метками, совпадающими с селектором существующего объекта Deployment

Kubernetes позволяет вручную создавать поды с метками, совпадающими с селектором уже существующего объекта Deployment. Это можно сделать, создав «голый» под или используя другой объект Deployment или ReplicaSet. Такая ситуация приводит к конфликтам, предотвращение которых не входит в обязанности Kubernetes, и заставляет существующий объект Deployment «считать», что он создал эти поды сам. Результаты могут быть непредсказуемыми, поэтому вам следует внимательно относиться к назначению меток ресурсам в кластере. Мы рекомендуем использовать семантическую систему назначения меток, о которой можно узнать в официальной документации: https://kubernetes.io/docs/concepts/configuration/overview/#using-labels.

Внимательно настраивайте проверки контейнеров

Проверки жизнеспособности, готовности и успешности запуска контейнеров способны принести большую пользу, однако при неправильной настройке они могут вызвать перебои в работе, включая каскадные отказы. Всегда важно понимать последствия неудачного прохождения каждой проверки и то, как это может повлиять на другие ресурсы Kubernetes, например на сервисы.

Ниже перечислены рекомендации по использованию проверки готовности.

- Выполняйте эту проверку в тех случаях, когда контейнеры могут оказаться не готовы обслуживать трафик сразу после запуска.
- При оценке готовности следует контролировать такие процессы, как прогрев кэша и выполнение миграций базы данных. Иногда имеет смысл инициировать процесс прогрева кэша в рамках проверки готовности, если он еще не запущен, однако делать это нужно с осторожностью. Проверка готовности выполняется постоянно на протяжении всего жизненного цикла пода, и ресурсоемкие операции не должны выполняться при каждом ее вызове. Такие операции лучше выполнять в рамках проверки успешности запуска.
- Для микросервисных приложений, предоставляющих конечные точки HTTP, рекомендуется настраивать проверку готовности типа `httpGet`. Это позволяет учесть ситуацию, когда контейнер уже запущен, но HTTP-сервер еще не полностью инициализирован.

- Хорошей практикой считается использование специально выделенной конечной точки HTTP для проведения проверок готовности. Довольно часто с этой целью используется путь `/health`.

Если в рамках проверки готовности вы оцениваете состояние внешних зависимостей (базы данных, системы логирования и т. д.), вам следует задавать тайм-аут проверки, превышающий максимально допустимое время ожидания ответа внешней зависимости. В противном случае вместо кратковременного увеличения задержки вы можете столкнуться с каскадными сбоями и снижением доступности приложения.

Ниже перечислены рекомендации по использованию проверки жизнеспособности.

- Выполняйте проверку жизнеспособности с осторожностью, так как ошибки в ее конфигурации могут привести к каскадным сбоям в работе сервисов и возникновению бесконечных циклов перезапуска контейнеров.
- Не используйте проверки жизнеспособности без особой необходимости. Такая необходимость может возникнуть, например, в случае наличия в приложении неизвестного дефекта, приводящего к взаимной блокировке.
- Выполняйте простые и быстрые проверки, определяющие состояние самого процесса, а не его зависимостей. Иными словами, не проверяйте статус внешних зависимостей в рамках проверки жизнеспособности, так как это может вызвать лавинообразные перезапуски контейнеров и перегрузку небольшой группы подов, обслуживаемых сервисом.
- Если процесс в контейнере может завершиться из-за возникновения критической ошибки, скорее всего, вам вовсе не нужна проверка жизнеспособности.
- Задавайте консервативные значения параметра `initialDelaySeconds`, чтобы избежать преждевременных перезапусков контейнеров и возникновения их бесконечного цикла.

Это были основные моменты, касающиеся проверок подов. Далее приведены рекомендации по поводу тегирования (маркирования) контейнерных образов.

Используйте осмысленные и семантические теги образов

Управление процессом отката к предыдущей версии Deployment и анализ истории выкаток требуют правильного подхода к тегированию контейнерных образов. Если полагаться на тег `latest`, выполнить откат будет невозможно, поскольку в разные моменты времени этот тег может указывать на разные

версии образа. Хорошей практикой является использование семантического версионирования контейнерных образов. Кроме того, стоит рассмотреть вариант тегирования образов с помощью хешей исходного кода, например хеша коммита Git, чтобы всегда можно было точно определить, какая версия выполняется в кластере Kubernetes.

Переходите на новые версии Kubernetes

Если вы взаимодействуете с рабочими нагрузками, разработанными для старых версий Kubernetes, вы могли заметить, что, начиная с версии 1.16, применение объекта Deployment в кластере вызывает следующую ошибку:

```
error: unable to recognize "deployment": no matches for kind "Deployment" in
version "extensions/v1beta1"
```

Это происходит потому, что в Kubernetes 1.16 объект Deployment был удален из группы API `extensions/v1beta1` в соответствии с политикой версионирования API. Вместо нее следует использовать группу API `apps/v1`, частью которой объект Deployment является начиная с версии 1.9. Старайтесь всегда следовать политике версионирования и обновляйте ресурсы до актуальных групп API при переходе на новую версию Kubernetes. Это позволит избежать неприятных сюрпризов в будущем, когда ресурс будет признан устаревшим в прежних группах API.

Задействуйте механизмы управления ресурсами в объекте Deployment

Задавайте минимальное количество ресурсов, гарантированно выделяемое контейнеру внутри объекта Deployment (запрос), и максимальное количество ресурсов, которое контейнер может потребить (лимит). Это помогает планировщику Kubernetes эффективно распределять ресурсы и предотвращать ситуации ресурсного голода. При этом потребности в ресурсах не следует ни завышать, ни занижать. Анализируйте поведение приложения, чтобы определить оптимальные запросы и лимиты. Это позволит избежать как недоиспользования ресурсов, так и возникновения узких мест в производительности.

Масштабирование и управление репликами

Используйте объект *HorizontalPodAutoscaler (HPA)* для автоматического масштабирования объекта Deployment на основе заранее определенных метрик, таких как загрузка ЦП или объем потребляемой памяти.

Следующая схема иллюстрирует то, как HPA управляет количеством реплик (рис. 11.5).

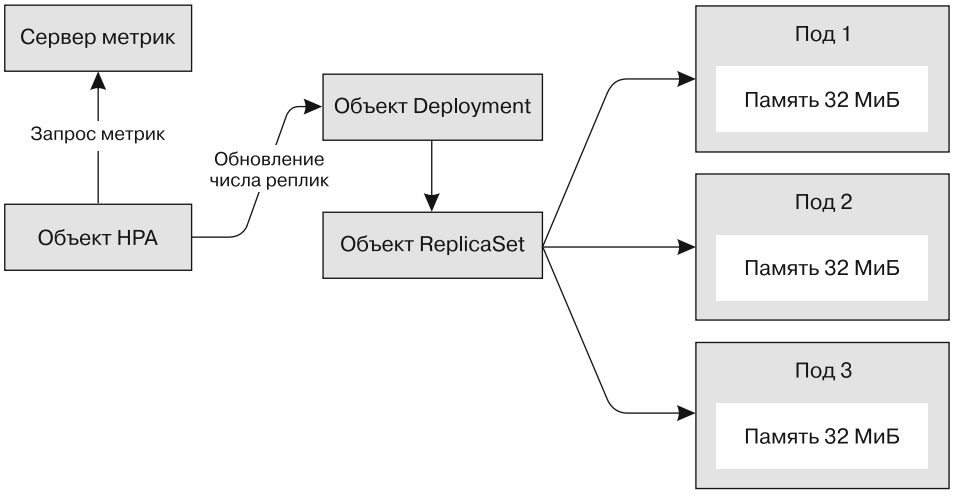

Рис. 11.5. Автоматическое масштабирование объекта Deployment на основе метрик

Хорошей практикой также является задание начального числа реплик (количества экземпляров подов) для объекта Deployment с учетом ожидаемой нагрузки и доступности ресурсов.

Вопросы безопасности

Реализуйте политики развертывания, запрещающие запуск контейнеров с избыточными привилегиями (например, от имени пользователя root). Это уменьшает поверхность атаки и минимизирует потенциальные уязвимости. Помните также о том, что чувствительная информация, такая как пароли или конфигурационные данные, должна храниться в объектах Secret и ConfigMap, а не встраиваться непосредственно в объект Deployment. Более подробно об обеспечении безопасности Kubernetes мы поговорим в главе 18.

Резюме

В этой главе вы узнали о том, как работать с приложениями и рабочими нагрузками *без сохранения состояния* в Kubernetes, используя объекты Deployment. Сначала вы создали экземпляр этого объекта и опубликовали его поды через сервис типа `LoadBalancer` для предоставления к ним доступа извне. Затем вы познакомились с методами масштабирования и управления объектами

Deployment в кластере. Мы рассмотрели такие операции, как выкатывание новой ревизии Deployment и откат к предыдущей версии в случае сбоя. В завершение вы ознакомились с лучшими практиками работы с объектами Deployment.

В следующей главе мы поговорим об управлении рабочими нагрузками и приложениями *с сохранением состояния* и вы познакомитесь с объектом StatefulSet.

Дополнительные источники информации

- Объект Deployment: https://kubernetes.io/docs/concepts/workloads/controllers/deployment.
- Объект StatefulSet: https://kubernetes.io/docs/concepts/workloads/controllers/statefulset/.
- Канареечное развертывание: https://kubernetes.io/docs/concepts/workloads/management/#canary-deployments.

12 Объект StatefulSet: развертывание приложений с сохранением состояния

В предыдущей главе мы говорили о запуске приложений и рабочих нагрузок *без сохранения состояния* в кластере Kubernetes с использованием объекта Deployment. Запуск таких нагрузок в облачной среде обычно не представляет сложности, поскольку любая реплика контейнера способна обработать запрос, не полагаясь на результаты предыдущих операций пользователя. Иными словами, все реплики контейнера обрабатывают запрос одинаковым образом, и нам необходимо позаботиться лишь о корректной балансировке нагрузки.

Основная сложность заключается в управлении *состоянием* приложений, под которым подразумеваются *данные*, которые приложение или компонент использует для обработки запросов и которые могут изменяться в результате выполнения этих запросов. Типичным компонентом, сохраняющим состояние, является база данных (БД), например *реляционная БД MySQL* или *нереляционная БД NoSQL MongoDB*. В Kubernetes для запуска рабочих нагрузок и приложений, сохраняющих состояние, предусмотрен специальный объект — StatefulSet (набор с сохранением состояния). При работе с объектами StatefulSet обычно используются постоянные тома (PV, PersistentVolume), подробно рассмотренные в главе 9. В этой главе вы узнаете о роли объекта StatefulSet в Kubernetes и научитесь создавать и управлять такими объектами при выпуске новых версий приложений с сохранением состояния.

Технические требования

Для работы с этой главой вам потребуется следующее.

- Развернутый кластер Kubernetes. Вы можете использовать как локальный, так и облачный кластер, однако для полноценного усвоения материала рекомендуется отдать предпочтение *многоузловому* облачному кластеру. Этот кластер должен поддерживать создание объектов PersistentVolumeClaim (PVC). Подойдет любой облачный кластер или локальная установка, например `minikube` с провиженером `k8s.io/minikube-hostpath`.

- Рабочий интерфейс командной строки `kubectl`, настроенный для взаимодействия с кластером

Инструкции по созданию кластера Kubernetes и настройке клиента `kubectl` приведены в главах 2 и 3.

Примеры кода для этой главы можно загрузить из официального репозитория GitHub по адресу https://github.com/PacktPublishing/The-Kubernetes-Bible-Second-Edition/tree/main/Chapter12.

Знакомство с объектом StatefulSet

Вероятно, вам интересно, почему выполнение рабочих нагрузок с сохранением состояния в распределенной облачной среде обычно считается *более сложной* задачей, чем запуск нагрузок без сохранения состояния. В классических трехуровневых приложениях все состояния хранятся в базе данных (на *уровне данных*, или в *слое персистентности*). SQL-серверы, как правило, дополняются механизмом автоматического аварийного переключения и репликации данных, а для повышения производительности выполняется *вертикальное* масштабирование, то есть приобретение более мощного оборудования для размещения БД. Позднее может возникнуть необходимость в кластерных SQL-решениях с *шардированием* (горизонтальным разбиением) *данных*. Тем не менее с точки зрения веб-сервера, на котором выполняется ваше приложение, база данных является всего лишь строкой подключения, используемой для операций чтения и записи. При этом за сохранение *изменяемого состояния* отвечает сама БД.

Помните, что любое приложение как целое в той или иной степени сохраняет состояние, если только его функциональность не ограничивается предоставлением статического контента или преобразованием пользовательского ввода. Однако это не означает, что все компоненты приложения сохраняют состояние. Например, веб-сервер, реализующий прикладную логику, может его не сохранять, в отличие от базы данных, хранящей пользовательский ввод и сеансы.

Далее мы поговорим об управлении состоянием в контейнерах и уточним, что именно подразумевается под состоянием приложения или системы.

Управление состоянием в контейнерах

Если вы развернете один экземпляр сервера SQL внутри контейнера, то заметите, что после перезапуска контейнера все данные в базе будут потеряны, то есть при каждом запуске вы будете получать «чистый» экземпляр SQL-сервера. Контейнеры по своей природе *эфемерны*, что, очевидно, не вполне нас устраивает. К счастью, они поддерживают возможность монтирования томов

данных. Таким томом может быть, например, *каталог хост-системы* или *внешний дисковый том*, который *монтируется* в файловую систему контейнера по определенному пути. Все, что вы там сохраняете, остается в томе даже после завершения работы или перезапуска контейнера. В качестве тома также можно использовать общий ресурс NFS или экземпляр внешнего диска. Если вы настроите сервер SQL так, чтобы файлы с его данными размещались в каталоге, соответствующем смонтированному тому, то обеспечите сохранность данных даже при перезапуске контейнера. Сам контейнер при этом по-прежнему остается эфемерным, а данные (то есть состояние) — *нет*.

Теперь вы в общих чертах представляете, как можно сохранять состояние в обычных контейнерах без использования Kubernetes. Прежде чем обсуждать возможности этой системы, нам следует уточнить, что именно понимается под *состоянием*.

Предположим, у вас есть веб-сервер, предоставляющий простой *статический* (неизменяемый) контент (например, HTML-страницу). В этом случае в системе действительно хранятся некоторые данные — те самые HTML-файлы. Но это *не* является состоянием, поскольку запросы пользователей не изменяют эти данные и *предыдущие* обращения не влияют на результат выполнения *текущего* запроса. Файлы конфигурации веб-сервера и журналы событий, записываемые на диск, тоже не относятся к состоянию (по крайней мере с точки зрения конечного пользователя).

Теперь рассмотрим веб-сервер, который хранит сеансы и статус авторизации пользователя. В данном случае речь уже идет о состоянии, поскольку сервер будет возвращать разные страницы (ответы) в зависимости от того, вошел ли пользователь в систему. Предположим, что этот веб-сервер работает в контейнере. Тогда возникает вопрос, можно ли считать его компонентом, сохраняющим состояние (stateful). Если процесс веб-сервера хранит пользовательские сеансы в виде файлов внутри контейнера (крайне неудачное решение), то контейнер веб-сервера является stateful-компонентом. Если же сеансы хранятся во внешней базе данных или в кеше Redis, работающих в отдельных контейнерах, то сам веб-сервер является компонентом *без сохранения состояния* (stateless), а база данных или контейнер Redis — stateful-компонентами.

Так выглядит управление состоянием с точки зрения одного контейнера. Теперь посмотрим, как управление состоянием реализуется в *подах Kubernetes*.

Управление состоянием в подах Kubernetes

В Kubernetes концепция томов контейнера расширяется за счет трех специализированных объектов, связанных с хранением данных: *PersistentVolume (PV)*, *PersistentVolumeClaim (PVC)* и *StorageClass (SC)*. Главная цель PVC (заявка на выделение постоянного тома) — разорвать связь между подом и конкретным хранилищем, предоставив абстракцию для создания запроса на хранение

данных определенного типа, класса или размера. Проще говоря, объект PVC позволяет создать запрос наподобие: *«Мне нужно 10 Гбайт SSD-хранилища с режимом доступа read/write-once»*. Для удовлетворения такого запроса требуется объект PV (постоянный том) — реальный участок хранилища, выделенный автоматизированным процессом кластера. Его можно представить в виде каталога в хост-системе или диска, управляемого драйвером хранилища. Типы PV реализуются в виде плагинов, подобно томам в Docker или Podman. Выделение PV может осуществляться *динамически*. Для этого создается объект StorageClass, который используется при определении PVC. При создании нового StorageClass вы указываете параметры провиженера (или плагина), после чего каждый объект PVC, использующий этот StorageClass, будет автоматически создавать PV с помощью выбранного провиженера. Такие провиженеры могут, например, создавать диски, управляемые облачным провайдером и выступающие в качестве базового хранилища. Кроме того, контейнеры, входящие в один и тот же под, могут совместно использовать данные, монтируя общий PV в свою файловую систему.

Это лишь краткий обзор механизмов хранения состояния, предоставляемых Kubernetes. Их подробное описание приведено в главе 9.

Помимо управления состоянием одного пода и его контейнеров, бывает необходимо управлять состоянием *нескольких реплик* пода. Представьте, что вы используете объект Deployment для запуска нескольких подов с сервером MySQL. Сначала вам нужно обеспечить сохранение состояния в томе контейнера. Для этого можно использовать объекты PV Kubernetes. Однако в результате вы получите несколько независимых экземпляров MySQL-сервера, что не обеспечит высокой доступности и отказоустойчивости. Публикация такого объекта Deployment через сервис тоже не принесет особой пользы, так как при каждом запросе вы будете обращаться к разным подам MySQL и получать разные данные.

Таким образом, возникает необходимость построения *многоузловой отказоустойчивой конфигурации* с синхронизацией данных между главным узлом и его репликами или создания сложного *кластера*, предусматривающего *шардирование данных*. В любом случае каждую реплику пода с сервером MySQL нужно наделить *уникальной идентичностью* и желательно *предсказуемыми сетевыми именами*, чтобы узлы и клиенты могли корректно взаимодействовать.

Разрабатывая облачное приложение для кластера Kubernetes, всегда оценивайте все за и против хранения его состояния в виде stateful-компонентов внутри кластера.

Именно для этого в Kubernetes предусмотрен объект StatefulSet, о котором мы поговорим далее.

Объект StatefulSet и его отличие от объекта Deployment

Объект StatefulSet в Kubernetes во многом похож на объект Deployment. Он также предназначен для управления набором подов и их масштабирования, но, помимо этого, гарантирует *упорядоченность и уникальность* (идентичности) этих подов. Подобно Deployment, StatefulSet использует шаблон пода для описания того, как должна выглядеть каждая реплика. Он тоже позволяет изменять число реплик, а также выкатывать новые версии. Однако отдельные реплики подов в StatefulSet *не являются взаимозаменяемыми*. Каждая из них сохраняет уникальную идентичность, включающую *имя пода* и его *DNS-имя в кластере*, при переназначении или обновлении. С помощью этого уникального идентификатора можно определить, какие PV были назначены поду, даже если исходный под был заменен по причине сбоя. Для этого в спецификации StatefulSet предусмотрен отдельный тип шаблона — `volumeClaimTemplates`. Он позволяет динамически создавать объекты PersistentVolumeClaim (PVC) на основе заданного StorageClass (SC). Таким образом, весь процесс выделения ресурсов хранилища становится полностью динамическим — достаточно создать объект StatefulSet, а все сопутствующие объекты хранилища будут управляться его контроллером.

Кластерные DNS-имена подов в StatefulSet остаются неизменными, чего нельзя сказать об их кластерных IP-адресах. Поэтому при подключении к отдельным подам StatefulSet следует использовать именно их DNS-имена.

Объект StatefulSet предназначен для приложений, которым требуются:

- постоянное хранилище, управляемое кластером Kubernetes (это основной, но не единственный сценарий использования этих объектов);
- стабильные и уникальные сетевые идентификаторы (обычно DNS-имена) для каждой реплики пода;
- упорядоченное развертывание и масштабирование;
- упорядоченные постепенные обновления.

Следующая схема показывает, что StatefulSet можно рассматривать как более предсказуемый вариант объекта Deployment, дополненный возможностью использования постоянного хранилища через PVC (рис. 12.1).

Рассмотрим основные различия между объектами StatefulSet и Deployment.

- StatefulSet гарантирует присвоение подам *детерминированных* (постоянных) имен, формируемых по шаблону `<имя_StatefulSet>-<порядковый_номер>`. В случае объекта Deployment имена подов генерируются *случайным образом* и имеют вид `<имя_Deployment>-<хеш_шаблона_пода>-<случайный_хеш>`.

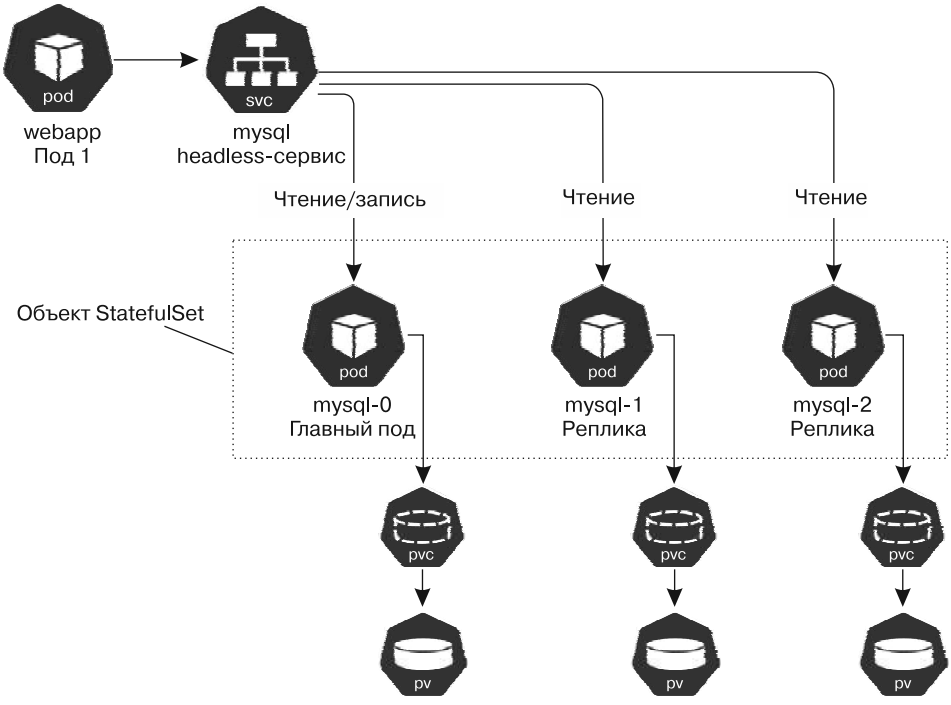

Рис. 12.1. Общая схема объекта StatefulSet

- Поды StatefulSet запускаются и завершают работу в *строго определенном* и *предсказуемом* порядке, что обеспечивает согласованность, стабильность и координацию при изменении числа реплик. Например, в случае с объектом StatefulSet MySQL (см. предыдущую схему) поды будут создаваться последовательно: `mysql-0`, `mysql-1`, `mysql-2`. При уменьшении числа реплик в StatefulSet завершение работы подов будет происходить в обратном порядке: `mysql-2`, `mysql-1`, `mysql-0`.
- Что касается хранилища, то Kubernetes создает отдельные объекты PVC на основе параметра `volumeClaimTemplates`, определенного в спецификации StatefulSet, для каждого пода и всегда привязывает их к поду *с тем же именем*. В случае с Deployment, если указать `persistentVolumeClaim` в шаблоне пода, Kubernetes создаст один объект PVC, который будет использоваться всеми подами этого объекта Deployment. Хотя такой подход может быть полезен в некоторых сценариях, он применяется нечасто.

Для управления *детерминированной сетевой идентичностью* подов (их DNS-именами в кластере) необходимо создать так называемый headless-сервис («безголовый» сервис). В отличие от обычного сервиса типа ClusterIP, который возвращает одну DNS-запись типа A, headless-сервис возвращает A-записи

с IP-адресами *всех* подов, работающих за ним. Headless-сервис необходим только в том случае, если вы не используете обычный сервис. При этом в поле `.spec.serviceName` спецификации StatefulSet должно быть указано его имя. О headless-сервисах речь шла в подразделе «Headless-сервисы» в главе 8.

Прежде чем перейти к подробному изучению объекта StatefulSet, нам стоит обсудить его ограничения.

Ограничения объекта StatefulSet

Далее перечислено, что необходимо учитывать при работе с объектами StatefulSet.

- *Настройка хранилища*. Объект StatefulSet не создает хранилище для подов автоматически. Вам необходимо либо использовать встроенный инструмент (например, PersistentVolume Provisioner), либо подготовить хранилище вручную заранее.
- *Неосвобожденные ресурсы хранилища*. При сокращении количества реплик или удалении объекта StatefulSet используемое подами хранилище не освобождается автоматически. Это сделано специально для защиты ценных данных от случайной потери. Хранилище необходимо освобождать вручную. Если этого не делать, увеличение объема неиспользуемого хранилища приведет к нерациональному расходованию ресурсов и росту затрат на хранение данных.
- *Адреса подов*. Для назначения подам уникальных и стабильных сетевых имен необходимо создать отдельный headless-сервис.
- *Завершение работы StatefulSet*. Kubernetes не гарантирует, что при удалении StatefulSet поды будут завершать работу в определенном порядке. Чтобы обеспечить корректное завершение работы StatefulSet, рекомендуется сначала уменьшить количество реплик до нуля, а затем удалить сам объект.
- *Обновление StatefulSet*. Использование стандартного метода обновления в отношении объектов StatefulSet может вызывать проблемы, требующие ручного вмешательства. Поэтому следует учитывать возможные риски и при необходимости применять альтернативные стратегии обновления.

Перед выполнением практических упражнений ознакомьтесь с важной информацией об объекте StatefulSet, представленной в следующем подразделе.

Управление данными в объекте StatefulSet

Объект StatefulSet представляет собой мощный инструмент для управления приложениями с сохранением состояния. Однако успешное развертывание и сопровождение таких приложений требует участия пользователя, которое

не ограничивается определением объекта StatefulSet. Рассмотрим, на что следует обратить внимание.

- *Клонирование и синхронизация данных*. В отличие от приложений без сохранения состояния, приложения с сохранением состояния опираются на постоянно хранящиеся в них данные. Объект StatefulSet управляет порядком создания подов и гарантирует их идентичность, но не осуществляет репликацию данных между ними. Эту функциональность необходимо реализовать самостоятельно. Для этого обычно применяются следующие подходы: использование контейнеров инициализации для копирования данных из заранее определенного источника при создании пода, применение встроенных механизмов репликации в самих приложениях (например, репликация MySQL); использование внешних сценариев для управления процессом синхронизации данных.
- *Доступность удаленного хранилища*. Объект StatefulSet гарантирует возможность переназначения подов любым доступным узлам кластера. Чтобы данные сохранялись в ходе этого процесса, хранилище, предоставляемое объектом PersistentVolume, должно быть доступно со всех рабочих узлов. Это требует выбора класса хранилища, обеспечивающего репликацию данных между узлами, либо использования сетевых систем хранения данных, доступных всем машинам в кластере.
- *Внешнее резервное копирование*. Объект StatefulSet предназначен для управления жизненным циклом подов и обеспечения сохранности данных внутри кластера, но он не выполняет внешнее резервное копирование. Для защиты данных от последствий катастрофических сбоев необходимо реализовать отдельное внешнее решение для резервного копирования. В данном случае речь может идти о создании резервных копий постоянных томов в облачном хранилище или на выделенном сервере резервного копирования, расположенном вне кластера Kubernetes.

Существуют лучшие практики и проверенные подходы к управлению данными в объектах StatefulSet. В следующем подразделе мы обсудим техники управления репликацией.

Управление репликацией

Как следует из названия, приложения с сохранением состояния, работающие с данными, часто требуют их инициализации или синхронизации. Для их реализации можно использовать следующее.

- *Контейнеры инициализации* — копируют данные из источника (например, из объекта ConfigMap) в постоянное хранилище перед запуском основного приложения.

- *Репликацию на уровне приложения* — предполагает использование встроенных механизмов репликации, предусмотренных самим приложением, для синхронизации данных между подами.
- *Внешние сценарии или инструменты* — позволяют управлять миграцией данных во время обновления приложения.

Теперь рассмотрим конкретный пример с объектом StatefulSet, который развертывает поды MySQL с поддержкой постоянного хранилища.

Управление объектом StatefulSet

Чтобы продемонстрировать принцип работы StatefulSet, мы модифицируем наш объект Deployment с MySQL, так как значительная часть спецификации этих объектов совпадает. Поскольку мы хотим показать, как работает автоматическое создание PVC в объектах StatefulSet, в спецификации будет использован параметр `volumeClaimTemplates`. С его помощью создаются объекты PVC и PV, которые затем подключаются к подам. Каждый под будет монтировать свой PV во внутренний каталог контейнера по пути `/var/lib/mysql`, где MySQL по умолчанию хранит файлы данных. Таким образом, мы наглядно покажем, как сохраняется *состояние* даже при принудительном перезапуске подов.

Пример, приведенный в этой главе, предназначен исключительно для демонстрационных целей и максимально упрощен. Если вы хотите ознакомиться с более сложными сценариями, например с развертыванием и управлением распределенными базами данных в StatefulSet, обратитесь к официальной публикации в блоге Kubernetes, посвященной развертыванию базы данных Cassandra: https://kubernetes.io/docs/tutorials/stateful-application/cassandra/. Основная сложность подобных сценариев обычно связана с добавлением и удалением реплик подов при масштабировании объекта StatefulSet.

Далее мы подробно рассмотрим YAML-манифесты, необходимые для создания StatefulSet, и применим их к кластеру.

Создание объекта StatefulSet

Мы уже обсудили концепцию объекта StatefulSet, и теперь пришло время научиться им управлять. Начнем с рассмотрения манифеста `mysql-statefulset.yaml`:

```
# mysql-statefulset.yaml
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: mysql-stateful
```

```
  labels:
    app: mysql
  namespace: mysql
spec:
  serviceName: mysql-headless
  replicas: 3
  selector:
    matchLabels:
      app: mysql
      environment: test
# (продолжение далее)
```

Первая часть файла похожа на спецификацию объекта Deployment — здесь также необходимо указать число реплик (`replicas`) и селектор подов (`selector`). Однако появился и новый параметр — `serviceName`, который мы обсудим чуть позже.

Следующий фрагмент файла описывает шаблон пода, используемый объектом StatefulSet:

```
# (продолжение)
  template:
    metadata:
      labels:
        app: mysql
        environment: test
    spec:
      containers:
        - name: mysql
          image: mysql:8.2.0
          ports:
            - containerPort: 3306
          volumeMounts:
            - name: mysql-data
              mountPath: /var/lib/mysql
          env:
            - name: MYSQL_ROOT_PASSWORD
              valueFrom:
                secretKeyRef:
                  name: mysql-secret
                  key: MYSQL_ROOT_PASSWORD
            - name: MYSQL_USER
              valueFrom:
                secretKeyRef:
                  name: mysql-secret
                  key: MYSQL_USER
            - name: MYSQL_PASSWORD
              valueFrom:
                secretKeyRef:
                  name: mysql-secret
                  key: MYSQL_PASSWORD
# (продолжение далее)
```

Как видите, структура остается той же, что и в случае с Deployment. Обратите внимание на переменные окружения, которые передаются через объект Secret — его необходимо создать до создания объекта StatefulSet. В конце файла находится блок `volumeClaimTemplates`, предназначенный для определения шаблонов PVC, используемых подами:

```
# (продолжение)
  volumeClaimTemplates:
    - metadata:
        name: mysql-data
      spec:
        accessModes:
          - "ReadWriteOnce"
        resources:
          requests:
            storage: 1Gi
```

Таким образом, в целом структура спецификации StatefulSet схожа со спецификацией Deployment, но содержит дополнительные параметры, позволяющие настраивать PVC и связанные объекты Service. Спецификация StatefulSet состоит из пяти основных компонентов.

- `replicas` — определяет количество реплик пода, которые запускаются на основе заданного шаблона (`template`) и соответствуют селектору меток (`selector`). Для поддержания требуемого числа реплик поды могут создаваться или удаляться.
- `serviceName` — имя сервиса, управляющего объектом StatefulSet и обеспечивающего сетевую идентичность подов. Этот сервис должен быть создан до создания объекта StatefulSet. В нашем примере таким сервисом будет `mysql-headless`, который мы создадим на следующем шаге.
- `selector` — *селектор меток*, задающий правила идентификации подов, находящихся под управлением объекта StatefulSet. Он может быть основан как *на множестве*, так и *на равенстве*.
- `template` — определяет шаблон для создания пода. Метки, указанные в разделе `metadata`, должны соответствовать условиям, определенным в разделе `selector`. Имена подов являются детерминированными и формируются по шаблону `<имяStatefulSet>-<порядковыйНомер>`. При необходимости можно задать начальный порядковый номер с помощью параметра `.spec.ordinals`.
- `volumeClaimTemplates` — определяет шаблон объекта PVC, который будет автоматически создан для каждого пода в составе StatefulSet и жестко привязан к его имени. В нашем примере это будет том объемом 1 ГБ с режимом доступа `ReadWriteOnce`, который допускает единовременное выполнение операций чтения и записи только *одним* узлом. Поскольку параметр `storageClassName` не указан, PVC будут создаваться с использованием класса хранилища по умолчанию. Имена PVC также не являются случайными и формируются по шаблону `<имя_VolumeClaimTemplate>-<имя_StatefulSet>-<порядковый_номер>`.

Класс хранилища по умолчанию в кластере помечается аннотацией storageclass.kubernetes.io/is-default-class. Наличие класса по умолчанию и его конфигурация зависят от конкретного способа развертывания кластера. Например, в кластере Azure Kubernetes Service (AKS) класс по умолчанию имеет имя default и использует провиженер kubernetes.io/azure-disk; в minikube класс по умолчанию называется standard и использует провиженер k8s.io/minikube-hostpath.

В спецификации StatefulSet также присутствуют поля, связанные с выкатыванием новых версий этого объекта. Мы подробно рассмотрим их в следующем разделе.

Теперь создадим «безголовый» сервис с именем `mysql-headless`. Для этого создайте файл `mysql-headless-service.yaml` со следующим содержимым:

```
# mysql-headless-service.yaml
apiVersion: v1
kind: Service
metadata:
  name: mysql-headless
  namespace: mysql
spec:
  selector:
    app: mysql
    environment: test
  clusterIP: None
  ports:
    - port: 3306
```

Структура этого манифеста почти идентична структуре манифеста обычного сервиса, который мы создавали ранее для объекта Deployment. Единственное отличие в том, что поле `clusterIP` имеет значение `None`. Это приводит к созданию сервиса `mysql-headless`, который возвращает IP-адреса *всех* работающих за ним подов в виде DNS-записей типа A вместо одной A-записи, как в случае с сервисом ClusterIP. Далее мы продемонстрируем это на практике.

После подготовки всех необходимых YAML-манифестов можно приступать к развертыванию объекта StatefulSet. Для этого выполните следующие шаги.

1. Создайте пространство имен `mysql`, используя файл `mysql-ns.yaml`:

```
$ kubectl apply -f mysql-ns.yaml
namespace/mysql created
```

2. Создайте объект Secret для хранения переменных окружения MySQL:

```
$ kubectl create secret generic mysql-secret \
  --from-literal=MYSQL_ROOT_PASSWORD='mysqlroot' \
  --from-literal=MYSQL_USER='mysqluser' \
  --from-literal=MYSQL_PASSWORD='mysqlpassword' \
  -n mysql
secret/mysql-secret created
```

Тот же объект Secret можно создать и декларативно с помощью YAML-манифеста, указав значения в кодировке base64 (например: echo -n 'mysqlroot' | base64). Пример такого манифеста можно найти в репозитории в файле mysql-secret.yaml. В данном случае мы используем императивный способ для создания секрета с конкретными значениями.

3. Создайте сервис `mysql-headless` с помощью команды:

```
$ kubectl apply -f mysql-headless-service.yaml
service/mysql-headless created
```

4. Создайте объект StatefulSet `mysql-stateful`:

```
$ kubectl apply -f mysql-statefulset.yaml
statefulset.apps/mysql-stateful created
```

5. Проверьте результат с помощью команды `kubectl describe` (вместо `stateful-set` можно использовать сокращенную форму `sts`):

```
$ kubectl describe statefulset mysql-stateful -n mysql
<вывод опущен для краткости>
$ kubectl get sts -n mysql
NAME             READY  AGE
mysql-stateful   3/3    2m3s
```

6. Выполните команду `kubectl get pods`, чтобы убедиться в успешном создании трех реплик пода. Этот процесс может занять некоторое время, поскольку для каждого пода должен быть выделен постоянный том (PV) в соответствии с его заявкой PVC:

```
$ kubectl get pod -n mysql
NAME               READY   STATUS    RESTARTS   AGE
mysql-stateful-0   1/1     Running   0          2m32s
mysql-stateful-1   1/1     Running   0          2m29s
mysql-stateful-2   1/1     Running   0          2m25s
```

Обратите внимание на упорядоченные и предсказуемые имена подов — именно это обеспечивает уникальную идентичность подов в объекте StatefulSet.

7. Если вы примените к поду команду `describe`, то увидите подробные сведения о связанных с ним объектах PV и PVC:

```
$ kubectl -n mysql describe pod mysql-stateful-0
Name:             mysql-stateful-0
```

```
Namespace:        mysql
Priority:         0
Service Account:  default
<фрагмент опущен для краткости>
Volumes:
  mysql-data:
    Type:       PersistentVolumeClaim (a reference to a
PersistentVolumeClaim in the same namespace)
    ClaimName:  mysql-data-mysql-stateful-0
    ReadOnly:   false
... <фрагмент опущен для краткости> ...
```

Для второго пода (`mysql-stateful-1`) вывод будет аналогичным, если не считать PVC:

```
$ kubectl -n mysql describe pod mysql-stateful-1
Name:             mysql-stateful-1
Namespace:        mysql
Priority:         0
<фрагмент опущен для краткости>
Volumes:
  mysql-data:
    Type:       PersistentVolumeClaim (a reference to a
PersistentVolumeClaim in the same namespace)
    ClaimName:  mysql-data-mysql-stateful-1
    ReadOnly:   false
... <фрагмент опущен для краткости> ...
```

Итак, под `mysql-stateful-0` использует PVC `mysql-data-mysql-stateful-0`, а под `mysql-stateful-1` — PVC `mysql-data-mysql-stateful-1`. После назначения пода узлу объекты PV выделяются через соответствующий StorageClass и привязываются к объектам PVC. Затем создается контейнер, в который монтируется этот PV.

8. Просмотрите подробные сведения об объекте PVC с помощью команды `kubectl get`:

```
$ kubectl get pvc -n mysql
NAME                          STATUS   VOLUME
CAPACITY   ACCESS MODES   STORAGECLASS   VOLUMEATTRIBUTESCLASS   AGE
mysql-data-mysql-stateful-0   Bound    pvc-453dbfee-6076-48b9-8878-
e7ac6f79d271   1Gi        RWO            standard       <unset>
8m38s
mysql-data-mysql-stateful-1   Bound    pvc-36494153-3829-42aa-be6d-
4dc63163ea38   1Gi        RWO            standard       <unset>
8m35s
mysql-data-mysql-stateful-2   Bound    pvc-6730af33-f0b6-445d-841b-
4fbad5732cde   1Gi        RWO            standard       <unset>
8m31s
```

9. А теперь отобразите сведения о выделенных PV:

```
$ kubectl get pv
NAME                                       CAPACITY   ACCESS MODES   RECLAIM
POLICY   STATUS   CLAIM                                    STORAGECLASS
VOLUMEATTRIBUTESCLASS   REASON   AGE
pvc-36494153-3829-42aa-be6d-4dc63163ea38   1Gi        RWO            Delete
Bound    mysql/mysql-data-mysql-stateful-1       standard       <unset>
<none>   11m
pvc-453dbfee-6076-48b9-8878-e7ac6f79d271   1Gi        RWO            Delete
Bound    mysql/mysql-data-mysql-stateful-0       standard       <unset>
<none>   11m
pvc-6730af33-f0b6-445d-841b-4fbad5732cde   1Gi        RWO            Delete
Bound    mysql/mysql-data-mysql-stateful-2       standard       <unset>
<none>   11m
```

В нашем примере используется тип хранилища hostPath, предоставляемый minikube. Если ваш кластер Kubernetes использует другую технологию хранения, вывод будет другим.

Итак, мы успешно создали объект StatefulSet. Пришло время проверить его работоспособность. Для этого воспользуемся обновленным контейнерным образом `k8sutils`, в который был добавлен стандартный клиент MySQL. (Для получения подробных сведений об образе `k8sutils` обратитесь к файлу `Chapter12/Containerfile`.) Создайте файл `k8sutils.yaml` со следующим содержимым:

```
# k8sutils.yaml
apiVersion: v1
kind: Pod
metadata:
  name: k8sutils
  # namespace: default
spec:
  containers:
    - name: k8sutils
      image: quay.io/iamgini/k8sutils:debian12-1.1
      command:
        - sleep
        - "infinity"
      # imagePullPolicy: IfNotPresent
  restartPolicy: Always
```

Создайте под `k8sutils` следующим образом:

```
$ kubectl apply -f k8sutils.yaml -n mysql
pod/k8sutils created
```

Обратите внимание на то, что при применении манифеста мы указали параметр -n mysql для создания ресурса в пространстве имен mysql.

Выполните следующие шаги, чтобы проверить содержимое подов объекта StatefulSet.

1. Подключитесь к поду `k8sutils` для выполнения тестовых команд:

```
$ kubectl exec -it -n mysql k8sutils -- /bin/bash
root@k8sutils:/#
```

2. Подключитесь к приложению MySQL, работающему в StatefulSet, используя созданный ранее headless-сервис и пароль, указанный в объекте Secret:

```
root@k8sutils:/# mysql -u root -p -h mysql-headless.mysql.svc.cluster.local
Enter password: <mysqlroot>
Welcome to the MariaDB monitor.  Commands end with ; or \g.
Your MySQL connection id is 8
Server version: 8.2.0 MySQL Community Server - GPL

Copyright (c) 2000, 2018, Oracle, MariaDB Corporation Ab and others.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

MySQL [(none)]> show databases;
+--------------------+
| Database           |
+--------------------+
| information_schema |
| mysql              |
| performance_schema |
| sys                |
+--------------------+
4 rows in set (0.003 sec)

MySQL [(none)]>
```

Итак, мы установили подключение к MySQL и получили доступ к серверу MySQL, работающему в виде приложения StatefulSet. Теперь посмотрим, как именно ведет себя headless-сервис.

Использование headless-сервиса и стабильных сетевых идентификаторов

Ранее вы уже узнали, что такое headless-сервис в Kubernetes и как его можно использовать для получения доступа к приложениям, развернутым с помощью объекта StatefulSet (см. подраздел «Headless-сервисы» в главе 8). Теперь углубимся в детали и посмотрим, как headless-сервис работает в контексте StatefulSet.

Проведем небольшой эксперимент, демонстрирующий то, каким образом headless-сервис обеспечивает стабильную и предсказуемую сетевую идентичность подов.

1. Подключитесь к поду `k8sutils`, который использовался в предыдущем тесте.
2. Отобразите DNS-записи для сервиса `mysql-headless`:

```
root@k8sutils:/# nslookup mysql-headless
Server:         10.96.0.10
Address:        10.96.0.10#53

Name:   mysql-headless.mysql.svc.cluster.local
Address: 10.244.0.14
Name:   mysql-headless.mysql.svc.cluster.local
Address: 10.244.0.15
Name:   mysql-headless.mysql.svc.cluster.local
Address: 10.244.0.16
```

   Вы получите три A-записи с указанием IP-адресов подов. Помимо этого, в выводе будут присутствовать CNAME-записи вида `<имя_пода>-<порядковый_номер>.<имя_Headless-сервиса>.<пространство_имен>.svc.cluster.local`. Разница между обычным и headless-сервисом заключается в том, что обычный сервис (типа ClusterIP) осуществляет балансировку нагрузки на уровне *виртуального IP* (в ОС Linux это обычно реализуется на уровне ядра с помощью правил iptables, конфигурируемых компонентом `kube-proxy`). В случае с headless-сервисом ответственность за балансировку нагрузки или выбор целевого пода лежит на *клиенте*, отправляющем запрос.
3. Благодаря *предсказуемым* полным доменным именам (FQDN) подов в StatefulSet мы можем напрямую обращаться к конкретным подам, не зная их IP-адресов или имен. Попробуем подключиться к серверу MySQL, работающему в поде `mysql-stateful-0`, используя его короткое DNS-имя, предоставляемое headless-сервисом:

```
root@k8sutils:/# mysql -u root -p -h mysql-stateful-0.mysql-headless
Enter password: <mysqlroot>
Welcome to the MariaDB monitor.  Commands end with ; or \g.
Your MySQL connection id is 8
Server version: 8.2.0 MySQL Community Server - GPL

Copyright (c) 2000, 2018, Oracle, MariaDB Corporation Ab and others.

Type 'help;' or '\h' for help. Type '\c' to clear the current input
statement.

MySQL [(none)]>
```

   Как видите, соединение установлено непосредственно с нужным подом.

4. Создадим новую базу данных внутри сервера MySQL:

```
MySQL [(none)]> create database ststest;
Query OK, 1 row affected (0.002 sec)

MySQL [(none)]> exit;
Bye
```

5. Теперь убедимся, что DNS-имя остается неизменным, даже при перезапуске пода. IP-адрес пода изменится, но его DNS-имя останется прежним. Более того, смонтированный PV тоже не изменится, но мы убедимся в этом чуть позже. А сейчас в другом окне терминала (вне контейнера) выполните следующую команду для принудительного перезапуска пода `mysql-stateful-0`:

```
$ kubectl delete po -n mysql mysql-stateful-0
```

Отобразив список подов, вы увидите, что под `mysql-stateful-0` был пересоздан и к нему был подключен тот же PVC `mysql-data-mysql-stateful-0`:

```
$ kubectl get po -n mysql
NAME               READY   STATUS    RESTARTS   AGE
k8sutils           1/1     Running   0          35m
mysql-stateful-0   1/1     Running   0          6s
mysql-stateful-1   1/1     Running   0          52m
mysql-stateful-2   1/1     Running   0          51m
```

6. Вернувшись в оболочку `k8sutils`, выполните команду клиента MySQL, чтобы проверить содержимое базы данных:

```
root@k8sutils:/# mysql -u root -p -h mysql-stateful-0.mysql-headless
Enter password: <mysqlroot>
Welcome to the MariaDB monitor.  Commands end with ; or \g.
Your MySQL connection id is 8
Server version: 8.2.0 MySQL Community Server - GPL

Copyright (c) 2000, 2018, Oracle, MariaDB Corporation Ab and others.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

MySQL [(none)]> show databases;
+--------------------+
| Database           |
+--------------------+
| information_schema |
| mysql              |
| performance_schema |
| ststest            |
| sys                |
+--------------------+
5 rows in set (0.003 sec)
```

Вы увидите, что база данных `ststest`, созданная до удаления пода, все еще на месте. Это подтверждает то, что данные не исчезают, а приложение действительно сохраняет состояние. Обратите также внимание, что IP-адрес пода изменился, а DNS-имя осталось прежним.

Таким образом, headless-сервисы позволяют обеспечить стабильную и предсказуемую сетевую идентичность подов, которая не меняется при их перезапуске или пересоздании. Если вы задаетесь вопросом о практической ценности этого подхода и его важности для объектов StatefulSet, то вот несколько возможных сценариев его применения.

- Развертывание кластерных баз данных, например `etcd` или MongoDB, требует указания сетевых адресов других узлов кластера. Это особенно важно, если СУБД не поддерживает *автоматическое обнаружение* узлов. В таких случаях стабильные DNS-имена, предоставляемые headless-сервисами, позволяют запускать подобные кластеры в Kubernetes в виде объектов StatefulSet. Однако при масштабировании этих объектов (добавлении или удалении подов) может возникнуть необходимость в динамическом обновлении конфигурации базы данных. В некоторых случаях для этого используется *контейнер-прицеп*, который отслеживает API Kubernetes и изменяет конфигурацию базы данных на лету.
- Если вы реализуете собственное решение для хранения данных в виде StatefulSet, предусматривающее сложное шардирование, у вас, скорее всего, возникнет необходимость в сопоставлении логических шардов (фрагментов) с физическими подами в кластере. В этом могут помочь стабильные DNS-имена, гарантирующие то, что запросы к каждому логическому шарду всегда направляются в правильный под, независимо от того, был ли он перезапущен или переназначен другому узлу.

Теперь посмотрим, как обеспечивается сохранение состояния подов, работающих в составе StatefulSet.

Сохранение состояния

Как было показано ранее, данные, размещенные в PersistentVolume (PV), сохраняются и автоматически привязываются к новому поду с тем же порядковым номером после его пересоздания.

В следующем примере мы удалим все поды, входящие в состав StatefulSet:

```
$ kubectl delete po -n mysql mysql-stateful-0 mysql-stateful-1 mysql-stateful-2
pod "mysql-stateful-0" deleted
pod "mysql-stateful-1" deleted
pod "mysql-stateful-2" deleted
```

Система Kubernetes автоматически пересоздаст все поды в том же порядке:

```
$ kubectl get pod -n mysql
NAME               READY   STATUS    RESTARTS   AGE
k8sutils           1/1     Running   0          47m
mysql-stateful-0   1/1     Running   0          44s
mysql-stateful-1   1/1     Running   0          43s
mysql-stateful-2   1/1     Running   0          41s
```

Чтобы убедиться в корректном восстановлении связи между подами и заявками PVC, можно отобразить список смонтированных постоянных томов (PV):

```
$ kubectl describe pod -n mysql -l app=mysql | egrep 'ClaimName|Name:'
Name:           mysql-stateful-0
ClaimName:      mysql-data-mysql-stateful-0
ConfigMapName:  kube-root-ca.crt
Name:           mysql-stateful-1
ClaimName:      mysql-data-mysql-stateful-1
ConfigMapName:  kube-root-ca.crt
Name:           mysql-stateful-2
ClaimName:      mysql-data-mysql-stateful-2
ConfigMapName:  kube-root-ca.crt
```

Контроллер StatefulSet не удаляет PV при удалении подов или самого объекта StatefulSet. Очистка данных и удаление PV при полном удалении StatefulSet выполняются вручную.

Теперь рассмотрим способ масштабирования объекта StatefulSet.

Масштабирование объекта StatefulSet

Объекты StatefulSet можно *масштабировать* так же, как и объекты Deployment, — путем изменения числа реплик (`replicas`) в спецификации или с помощью императивной команды `kubectl scale`. При увеличении числа реплик новые поды автоматически добавляются в список конечных точек сервиса, а при уменьшении — удаляются из него.

Однако между объектами StatefulSet и Deployment существуют некоторые различия.

- При создании объекта StatefulSet с количеством реплик N поды создаются последовательно, в порядке от 0 до $N - 1$. В нашем примере при создании StatefulSet с тремя репликами сначала создается под `mysql-stateful-0`, затем `mysql-stateful-1` и `mysql-stateful-2`.
- При *увеличении* числа реплик в StatefulSet поды также создаются последовательно и по порядку.

- При *уменьшении* числа реплик поды завершают работу *в обратном порядке* — от $N - 1$ к 0. В нашем случае при масштабировании StatefulSet до нуля реплик сначала будет удален под `mysql-stateful-2`, затем `mysql-stateful-1` и наконец `mysql-stateful-0`.
- При *увеличении* числа реплик в StatefulSet очередной под создается только при условии, что все предыдущие поды находятся в состоянии `Running` и `Ready`.
- При *уменьшении* числа реплик в StatefulSet поды удаляются в порядке, обратном порядку создания, и только после полного завершения работы и удаления всех предыдущих подов.
- В целом перед выполнением *любой* операции масштабирования над подом объекта StatefulSet все предыдущие поды должны находиться в состоянии `Running` и `Ready`. Таким образом, если при уменьшении числа реплик с четырех до одной под `mysql-stateful-0` внезапно выйдет из строя, операции масштабирования над подами `mysql-stateful-1`, `mysql-stateful-2` и `mysql-stateful-3` не будут выполнены. Процесс масштабирования возобновится только после перехода пода `mysql-stateful-0` в состояние готовности.

Такое последовательное поведение при масштабировании можно изменить, задав другое значение в поле spec.podManagementPolicy спецификации StatefulSet. По умолчанию используется значение OrderedReady. Если заменить его на Parallel, операции масштабирования будут выполняться параллельно, как в случае с объектом Deployment. Однако важно помнить, что этот параметр влияет только на операции масштабирования. Способ обновления StatefulSet при использовании стратегии RollingUpdate остается прежним.

Теперь *отмасштабируем* объект StatefulSet императивным способом.

1. Увеличьте число реплик:

```
$ kubectl scale statefulset -n mysql mysql-stateful --replicas=4
statefulset.apps/mysql-stateful scaled
```

2. Если вы выведете список подов с помощью команды `kubectl get pods`, то заметите, как создаются новые поды:

```
$ kubectl get pod -n mysql
NAME               READY   STATUS    RESTARTS   AGE
k8sutils           1/1     Running   0          56m
mysql-stateful-0   1/1     Running   0          9m13s
mysql-stateful-1   1/1     Running   0          9m12s
mysql-stateful-2   1/1     Running   0          9m10s
mysql-stateful-3   1/1     Running   0          4s
```

Аналогичным образом, если вы посмотрите вывод команды `kubectl describe` для этого объекта `StatefulSet`, то в списке событий увидите следующее:

```
$ kubectl describe sts -n mysql mysql-stateful
Name:         mysql-stateful
Namespace:    mysql
...<фрагмент опущен для краткости>...
Events:
  Type     Reason                   Age                      From
Message
  ----     ------                   ----                     ---
-                    -------
  Normal   SuccessfulCreate         23m (x2 over 75m)
statefulsetcontroller   create Pod mysql-stateful-0 in StatefulSet mysql-
stateful successful
  Normal   RecreatingTerminatedPod  11m (x13 over 23m)
statefulsetcontroller   StatefulSet mysql/mysql-stateful is recreating
terminated Pod mysql-stateful-0
  Normal   SuccessfulDelete         11m (x13 over 23m)
statefulsetcontroller   delete Pod mysql-stateful-0 in StatefulSet mysql-
stateful successful
  Normal   SuccessfulCreate         2m28s
statefulsetcontroller   create Claim mysql-data-mysql-stateful-3 Pod mysql-
stateful-3 in StatefulSet mysql-stateful success
  Normal   SuccessfulCreate         2m28s
statefulsetcontroller   create Pod mysql-stateful-3 in StatefulSet mysql-
stateful successful
```

А теперь уменьшим количество реплик с помощью императивной команды:

```
$ kubectl scale statefulset -n mysql mysql-stateful --replicas=2
statefulset.apps/mysql-stateful scaled
```

Как видите, поды `mysql-stateful-3` и `mysql-stateful-2` были удалены по порядку. Выведем список подов в StatefulSet:

```
$ kubectl get pod -n mysql
NAME               READY   STATUS    RESTARTS   AGE
k8sutils           1/1     Running   0          61m
mysql-stateful-0   1/1     Running   0          15m
mysql-stateful-1   1/1     Running   0          15m
```

3. Проверив список PVC, вы увидите, что все эти объекты остались на месте, что вполне ожидаемо для объектов StatefulSet:

```
$ kubectl get pvc -n mysql
NAME                           STATUS   VOLUME
CAPACITY   ACCESS MODES   STORAGECLASS   VOLUMEATTRIBUTESCLASS   AGE
mysql-data-mysql-stateful-0    Bound    pvc-453dbfee-6076-48b9-8878-
e7ac6f79d271   1Gi        RWO            standard       <unset>         79m
mysql-data-mysql-stateful-1    Bound    pvc-36494153-3829-42aa-be6d-
4dc63163ea38   1Gi        RWO            standard       <unset>         79m
mysql-data-mysql-stateful-2    Bound    pvc-6730af33-f0b6-445d-841b-
4fbad5732cde   1Gi        RWO            standard       <unset>         79m
mysql-data-mysql-stateful-3    Bound    pvc-6ec1ee2a-5be3-4bf9-84e5-
4f5aee566c11   1Gi        RWO            standard       <unset>        7m4s
```

При управлении объектами StatefulSet с помощью Horizontal Pod Autoscaler (HPA) или других инструментов горизонтального масштабирования лучше не задавать значение параметра .spec.replicas в манифесте. В этом случае управляющий слой Kubernetes будет автоматически регулировать количество реплик в зависимости от текущей потребности в ресурсах, обеспечивая эффективное масштабирование приложения без ручного вмешательства.

Итак, теперь вы знаете, как разворачивать и масштабировать объекты StatefulSet. Далее мы поговорим об их удалении.

Удаление объекта StatefulSet

Удалить объект StatefulSet можно двумя способами:

- удалить StatefulSet вместе с принадлежащими ему подами;
- удалить только StatefulSet, оставив поды нетронутыми.

В обоих случаях объекты PVC и PV, созданные для подов на основе `volumeClaimTemplates`, по умолчанию *не* удаляются.

Однако в версии Kubernetes 1.27 появилось поле `.spec.persistentVolumeClaimRetentionPolicy`, позволяющее управлять удалением PVC в рамках жизненного цикла StatefulSet:

```
apiVersion: apps/v1
kind: StatefulSet
...
spec:
  persistentVolumeClaimRetentionPolicy:
    whenDeleted: Retain
    whenScaled: Delete
...
```

Подробнее об этом механизме можно узнать в официальной документации по адресу https://kubernetes.io/docs/concepts/workloads/controllers/statefulset/#persistentvolumeclaimretention.

Чтобы удалить объект StatefulSet вместе с подами, используйте стандартную команду `kubectl delete`:

```
$ kubectl delete sts -n mysql mysql-stateful
statefulset.apps "mysql-stateful" deleted
```

В этом случае сначала будет завершена работа подов, а затем удален сам объект StatefulSet. Обратите внимание, что эта операция отличается от *масштабирования* StatefulSet до нуля реплик и его последующего удаления. При удалении объекта StatefulSet с существующими подами нет никакой гарантии, что они

будут завершать работу в определенном порядке. Как правило, все они останавливаются одновременно.

Чтобы удалить только объект StatefulSet, сохранив поды, используйте параметр `--cascade=orphan`:

```
$ kubectl delete sts -n mysql mysql-stateful --cascade=orphan
```

В списке подов после выполнения этой команды вы увидите, что поды, ранее принадлежавшие объекту `mysql-stateful`, по-прежнему присутствуют в кластере.

Наконец, если вы хотите очистить хранилище после удаления StatefulSet, вам необходимо вручную удалить объекты PVC и PV. Для этого выполните команду:

```
$ kubectl delete -n mysql pvc mysql-data-mysql-stateful-0 mysql-data-mysql-
stateful-1 mysql-data-mysql-stateful-2 mysql-data-mysql-stateful-3
persistentvolumeclaim "mysql-data-mysql-stateful-0" deleted
persistentvolumeclaim "mysql-data-mysql-stateful-1" deleted
persistentvolumeclaim "mysql-data-mysql-stateful-2" deleted
persistentvolumeclaim "mysql-data-mysql-stateful-3" deleted
```

Команда удалит все заявки PVC и связанные с ними объекты PV.

Если вы планируете проверять сохранность данных после выкатки новой версии приложения (см. следующий раздел), не удаляйте объекты PVC, чтобы не потерять файлы MySQL, хранящиеся в постоянных томах.

Итак, мы рассмотрели основные операции с объектами StatefulSet в Kubernetes. Далее мы поговорим о выпуске новых версий приложений, развернутых в виде StatefulSet, и об управлении их ревизиями.

Выпуск новой версии приложения, развернутого в виде StatefulSet

В предыдущем разделе мы говорили о *масштабировании* объектов StatefulSet с помощью команды `kubectl scale` (или путем изменения значения `.spec.replicas` в спецификации). Все, что вы узнали о порядке изменения подов, имеет ключевое значение в ходе выкатывания новой ревизии StatefulSet с использованием стратегии обновления `RollingUpdate`. Объекты StatefulSet и Deployment во многом похожи. Механизмы обновления Deployment были подробно рассмотрены в главе 11. Изменение *шаблона* пода (`.spec.template`) в спецификации StatefulSet также инициирует выкатывание новой ревизии.

Объект StatefulSet поддерживает две *стратегии обновления*, задаваемые в поле спецификации `.spec.updateStrategy.type`.

- `RollingUpdate` — стратегия по умолчанию, позволяющая постепенно выкатывать новую версию приложения. Она немного отличается от аналогичной стратегии объекта Deployment. При обновлении StatefulSet поды удаляются и пересоздаются поочередно и в определенном порядке. Контроллер StatefulSet гарантирует, что каждый под будет успешно пересоздан и приведен в состояние готовности, прежде чем начнется обновление следующего.
- `OnDelete` — стратегия, реализующая устаревший механизм обновления StatefulSet, существовавший до версии Kubernetes 1.7, который по-прежнему может оказаться полезным в определенных сценариях. При использовании этой стратегии объект StatefulSet *не* обновляет поды автоматически — для применения нового шаблона необходимо удалить под вручную. Это удобно в тех случаях, когда перед обновлением очередной реплики необходимо выполнить дополнительные проверки или другие действия. Например, если в StatefulSet развернут кластер Cassandra или `etcd`, вы можете захотеть убедиться в корректном присоединении нового пода к кластеру после удаления предыдущей версии. Разумеется, аналогичные проверки можно реализовать, используя хуки жизненного цикла пода (`postStart` и `preStop`) в рамках стратегии `RollingUpdate`, однако это требует более сложной обработки ошибок.

Теперь подробно рассмотрим стратегию `RollingUpdate`, которая чаще всего применяется при обновлении объектов StatefulSet. Ее главная особенность заключается в соблюдении всех гарантий, описанных ранее в контексте обсуждения операций масштабирования. Процесс выкатывания обновлений выполняется в обратном порядке: например, сначала пересоздается под `mysql-stateful-2`, затем `mysql-stateful-1` и, наконец, `mysql-stateful-0`.

Если процесс обновления завершается сбоем (не обязательно на поде, пересоздаваемом в данный момент), контроллер StatefulSet откатывает все сбойные поды к их *текущей версии*. Это означает, что поды, успешно обновленные до новой версии, сохраняют ее, а необновленные остаются на предыдущей. Таким образом, объект StatefulSet поддерживает работоспособность и согласованность приложения. Однако это может привести к *прерыванию* процесса обновления: если хотя бы одна реплика пода не перейдет в состояние готовности, процесс обновления остановится в ожидании *ручного* вмешательства. Повторное применение шаблона к предыдущей ревизии StatefulSet не решит проблему — контроллер не возобновит обновление до перехода сбойного пода в состояние готовности. Единственное решение — вручную удалить сбойный под, после чего StatefulSet применит предыдущую ревизию его шаблона.

Стратегия `RollingUpdate` также поддерживает *поэтапное* обновление с помощью поля `.spec.updateStrategy.rollingUpdate.partition`, в котором за-

дается числовое значение. Реплики подов с *порядковым номером меньше* этого значения не будут обновляться и даже в случае удаления будут пересозданы на предыдущей версии. Например, если установить значение `partition` равным `1`, то в процессе выкатывания изменений будут обновлены только поды `mysql-stateful-1` и `mysql-stateful-2`, а под `mysql-stateful-0` останется необновленным и продолжит работать на прежней версии. Управляя значением `partition`, можно выполнять *канареечные* развертывания, начиная с одной реплики и *поэтапно* распространяя обновление на остальные поды. По умолчанию значение `partition` равно `0`, что обеспечивает обновление всех реплик подов.

Теперь выкатим новую версию нашего приложения `mysqlserver`, используя стратегию `RollingUpdate`.

Обновление объекта StatefulSet

В этом разделе мы рассмотрим способ выкатывания новой версии контейнерного образа с использованием созданного ранее YAML-манифеста StatefulSet.

1. Сделайте копию предыдущего YAML-манифеста:

```
$ cp mysql-statefulset.yaml mysql-statefulset-rolling-update.yaml
```

2. Убедитесь, что в спецификации указаны стратегия обновления `RollingUpdate` и параметр `partition`, равный `0`. Обратите внимание: если вы создавали StatefulSet с другой стратегией обновления, то не сможете изменить ее без удаления объекта:

```
# mysql-statefulset-rolling-update.yaml
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: mysql-stateful
  labels:
    app: mysql
  namespace: mysql
spec:
  serviceName: mysql-headless
  podManagementPolicy: OrderedReady
  updateStrategy:
    type: RollingUpdate
    rollingUpdate:
      partition: 0
  replicas: 3
... <фрагмент опущен для краткости> ...
```

   Полный YAML-манифест можно найти в репозитории GitHub.

   Эти значения используются по умолчанию, однако имеет смысл задать их явно, чтобы лучше разобраться в происходящем.

3. Примените манифест к кластеру, чтобы создать объект `mysql-stateful` с новыми параметрами:

```
$ kubectl apply -f mysql-statefulset-rolling-update.yaml
statefulset.apps/mysql-stateful created
```

   Дождитесь перехода подов в состояние `Running`, прежде чем приступать к обновлению. Отобразите список подов, созданных объектом StatefulSet, с помощью команды `kubectl get pods`:

```
$ kubectl get pods -n mysql
NAME              READY   STATUS    RESTARTS        AGE
k8sutils          1/1     Running   1 (21h ago)     23h
mysql-stateful-0  1/1     Running   0               65s
mysql-stateful-1  1/1     Running   0               62s
mysql-stateful-2  1/1     Running   0               58s
```

4. Когда объект StatefulSet будет готов, создайте внутри него новую базу данных через под `k8sutils`:

```
$ kubectl exec -it -n mysql k8sutils -- /bin/bash
root@k8sutils:/# mysql -u root -p -h mysql-stateful-0.mysql-headless

Enter password: <mysqlroot>

Welcome to the MariaDB monitor.  Commands end with ; or \g.
... <фрагмент опущен для краткости> ...

MySQL [(none)]> create database stsrolling;
Query OK, 1 row affected (0.027 sec)

MySQL [(none)]> exit;
```

   Итак, у нас есть объект StatefulSet с заданной стратегией обновления и базой данных.

Теперь обновим образ контейнера MySQL в StatefulSet до новой версии. Для этого выполните следующие шаги.

1. Измените образ контейнера, указанный в шаблоне пода, на `mysql:8.3.0`:

```
# mysql-statefulset-rolling-update.yaml
... <фрагмент опущен для краткости> ...
    spec:
      containers:
        - name: mysql
          image: mysql:8.3.0
... <фрагмент опущен для краткости> ...
```

2. Примените изменения к кластеру с помощью следующей команды:

```
$ kubectl apply -f mysql-statefulset-rolling-update.yaml
statefulset.apps/mysql-stateful configured
```

3. Сразу после этого запустите команду `kubectl rollout status` для отслеживания процесса обновления в реальном времени. Этот процесс займет больше времени, чем обновление объекта Deployment, поскольку в данном случае поды обновляют последовательно и в определенном порядке:

```
$ kubectl rollout status statefulset -n mysql
Waiting for 1 pods to be ready...
Waiting for 1 pods to be ready...
Waiting for 1 pods to be ready...
Waiting for 1 pods to be ready...
Waiting for partitioned roll out to finish: 1 out of 3 new pods have been
updated...
Waiting for 1 pods to be ready...
Waiting for 1 pods to be ready...
Waiting for 1 pods to be ready...
Waiting for 1 pods to be ready...
Waiting for partitioned roll out to finish: 2 out of 3 new pods have been
updated...
Waiting for 1 pods to be ready...
Waiting for 1 pods to be ready...
Waiting for 1 pods to be ready...
Waiting for 1 pods to be ready...
partitioned roll out complete: 3 new pods have been updated...
```

4. Если вы просмотрите список событий с помощью команды `kubectl describe`, то увидите, в каком порядке пересоздавались реплики подов:

```
$ kubectl describe sts -n mysql mysql-stateful
Name:               mysql-stateful
Namespace:          mysql
... <фрагмент опущен для краткости> ...
Events:
  Type     Reason                   Age                 From
Message
  ----     ------                   ----                ----
-------
  <фрагмент опущен для краткости>
  Normal   SuccessfulDelete         72s (x7 over 73s)   statefulsetcontroller
delete Pod mysql-stateful-2 in StatefulSet mysql-stateful successful
  Normal   RecreatingTerminatedPod  72s (x7 over 72s)   statefulsetcontroller
StatefulSet mysql/mysql-stateful is recreating terminated Pod mysql-stateful-2
  Warning  FailedDelete             72s                 statefulsetcontroller
delete Pod mysql-stateful-2 in StatefulSet mysql-stateful failed error: pods
"mysql-stateful-2" not found
```

```
  Normal   SuccessfulDelete         70s (x2 over 71s)   statefulsetcontroller
delete Pod mysql-stateful-1 in StatefulSet mysql-stateful successful
  Normal   RecreatingTerminatedPod  70s                 statefulsetcontroller
StatefulSet mysql/mysql-stateful is recreating terminated Pod mysql-stateful-1
```

Как и ожидалось, этот процесс выполнялся в *обратном* порядке. Первым был пересоздан под `mysql-stateful-2`, затем `mysql-stateful-1` и, наконец, `mysql-stateful-0`. Поскольку мы использовали значение `partition` по умолчанию, то есть равное `0`, все поды были обновлены. Это произошло потому, что порядковые номера всех реплик превышают или равны `0`.

5. Убедитесь, что поды действительно были пересозданы с новым образом. Выполните следующую команду, чтобы проверить первую реплику пода в объекте StatefulSet:

```
$ kubectl describe pod -n mysql mysql-stateful-0 | grep Image
      Image:     mysql:8.3.0
      Image ID:  docker.io/library/mysql@sha256:9de9d54fecee6253130e65154b9
                 30978b1fcc336bcc86dfd06e89b72a2588ebe
```

6. Наконец, убедитесь, что *состояние* сохранилось благодаря тому, что новые поды используют те же объекты PVC, что и прежние. Учтите, что это работает лишь в том случае, если вы не удаляли PVC для этого объекта StatefulSet вручную в предыдущем разделе:

```
$ kubectl exec -it -n mysql k8sutils -- /bin/bash
root@k8sutils:/# mysql -u root -p -h mysql-stateful-0.mysql-headless
Enter password: <mysqlroot>
...<фрагмент опущен для краткости> ...
Server version: 8.3.0 MySQL Community Server - GPL

Copyright (c) 2000, 2018, Oracle, MariaDB Corporation Ab and others.

Type 'help;' or '\h' for help. Type '\c' to clear the current input
statement.

MySQL [(none)]> show databases;
+--------------------+
| Database           |
+--------------------+
| information_schema |
| mysql              |
| performance_schema |
| stsrolling         |
| sys                |
+--------------------+
5 rows in set (0.004 sec)

MySQL [(none)]>
```

Как видите, выкатывание новой версии MySQL оказалось успешным, а данные сохранились, несмотря на пересоздание подов, что подтверждается наличием базы данных `stsrolling`, созданной до начала обновления.

Вы можете изменить контейнерный образ StatefulSet императивно, используя команду kubectl -n mysql set image sts mysql-stateful mysql=mysql:8.3.0. Этот метод не рекомендуется использовать в рабочей среде. Он подходит только для целей тестирования. В целом объектами StatefulSet гораздо проще управлять декларативным, нежели императивным способом.

Теперь рассмотрим способ использования поля `partition` для *поэтапного* выкатывания обновлений. Допустим, вы хотите обновить образ MySQL до версии 8.4.0, но перед этим проверить работоспособность этой версии в вашей среде, выполнив канареечное развертывание, то есть обновив только одну или несколько реплик пода. Для этого выполните следующие шаги.

1. В манифесте `mysql-statefulset-rolling-update.yaml` установите значение `partition`, равное текущему числу реплик, — в нашем случае это `3`:

```
... <фрагмент опущен для краткости> ...
spec:
  serviceName: mysql-headless
  podManagementPolicy: OrderedReady
  updateStrategy:
    type: RollingUpdate
    rollingUpdate:
      partition: 3
... <фрагмент опущен для краткости> ...
```

   Обновите образ до версии 8.4.0:

```
...
    spec:
      containers:
        - name: mysql
          image: mysql:8.4.0
... <фрагмент опущен для краткости> ...
```

   Когда значение `partition` совпадает с числом реплик, применение манифеста к кластеру *не вызывает никаких изменений в подах*. Это состояние называется *подготовкой к выкатыванию*.

2. Примените манифест к кластеру:

```
$ kubectl apply -f mysql-statefulset-rolling-update.yaml
statefulset.apps/mysql-stateful configured
```

3. Теперь выполним *канареечное* развертывание новой версии. Уменьшите значение параметра `partition` в манифесте до `2`. Это означает, что все реплики

подов с порядковыми номерами меньше 2 обновлены не будут — в нашем случае обновится только под mysql-stateful-2, а все остальные останутся без изменений:

```
...
spec:
  serviceName: mysql-headless
  podManagementPolicy: OrderedReady
  updateStrategy:
    type: RollingUpdate
    rollingUpdate:
      partition: 2
  replicas: 3
...
```

4. Снова примените манифест к кластеру:

```
$ kubectl apply -f mysql-statefulset-rolling-update.yaml
statefulset.apps/mysql-stateful configured
```

5. Используйте команду kubectl rollout status для отслеживания процесса. Как и ожидалось, будет пересоздан только один под:

```
$ kubectl rollout status statefulset -n mysql
Waiting for partitioned roll out to finish: 0 out of 1 new pods have been
updated...
Waiting for 1 pods to be ready...
Waiting for 1 pods to be ready...
Waiting for 1 pods to be ready...
partitioned roll out complete: 1 new pods have been updated...
```

6. Если вы отобразите подробные сведения о подах mysql-stateful-0 и mysql-stateful-2, то увидите, что первый под использует старую версию образа, а второй — новую:

```
$ kubectl describe pod -n mysql mysql-stateful-0 | grep Image
    Image:          mysql:8.3.0
    Image ID:       docker-pullable://mysql@sha256:9de9d54fecee6253130e6515
                    4b930978b1fcc336bcc86dfd06e89b72a2588ebe

$ kubectl describe pod -n mysql mysql-stateful-2 | grep Image
    Image:          mysql:8.4.0
    Image ID:       docker-pullable://mysql@sha256:4a4e5e2a19aab7a678705889
                    52e8f401e17a330466ecfc55c9acf51196da5bd0
```

7. На этом этапе можно провести проверку и базовое тестирование канареечной версии. Подключитесь к поду k8sutils и убедитесь в том, что обновленный под (mysql-stateful-2) корректно работает с новым образом (например, 8.4.0). Если все в порядке, можно продолжить развертывание новой версии:

```
$ kubectl exec -it -n mysql k8sutils -- /bin/bash
root@k8sutils:/# mysql -u root -p -h mysql-stateful-2.mysql-headless
Enter password: <mysqlroot>
Welcome to the MariaDB monitor.  Commands end with ; or \g.
Your MySQL connection id is 11
Server version: 8.4.0 MySQL Community Server - GPL
...
```

8. Для поэтапного выкатывания обновлений достаточно *уменьшить* значение параметра `partition` в манифесте. Вы можете выкатить обновления в несколько этапов или все сразу. Реализуем второй вариант, установив значение `partition` равным `0`:

```
...
  updateStrategy:
    type: RollingUpdate
    rollingUpdate:
      partition: 0
...
```

9. Снова примените манифест к кластеру:

```
$ kubectl apply -f mysql-statefulset-rolling-update.yaml
statefulset.apps/mysql-stateful configured
```

10. Наблюдайте за процессом с помощью команды `kubectl rollout status`:

```
$ kubectl rollout status statefulset -n mysql
Waiting for partitioned roll out to finish: 1 out of 3 new pods have been
updated...
Waiting for 1 pods to be ready...
Waiting for 1 pods to be ready...
Waiting for partitioned roll out to finish: 2 out of 3 new pods have been
updated...
Waiting for 1 pods to be ready...
Waiting for 1 pods to be ready...
Waiting for 1 pods to be ready...
Waiting for 1 pods to be ready...
partitioned roll out complete: 3 new pods have been updated...
```

Как видите, процесс поэтапного выкатывания версии образа `mysql:8.4.0` завершился успешно.

Поэтапное выкатывание обновлений можно выполнять и императивно, управляя параметром partition с помощью команды kubectl patch, например: kubectl patch sts mysql-stateful -p '{"spec":{"updateStrategy":{"type":"RollingUpdate", "rollingUpdate":{"partition":3}}}}' -n mysql. Однако этот подход является менее наглядным и более подверженным ошибкам по сравнению с декларативным.

Далее мы рассмотрим способы отката изменений, примененных к объектам StatefulSet.

Откат обновлений объекта StatefulSet

В главе 11 мы уже рассматривали *императивный* способ отката изменений в контексте обсуждения объектов Deployment. Над объектами StatefulSet можно выполнять те же операции, используя команду `kubectl rollout undo`. Однако в случае с объектами StatefulSet изменения в кластер Kubernetes рекомендуется вносить *декларативным* способом. В рамках такого подхода все изменения фиксируются в репозитории исходного кода, а откат выполняется путем отмены нужного коммита и повторного применения конфигурации. Как правило, процесс внесения изменений (как при развертывании, так и при обновлении) реализуется в рамках работы CI/CD-конвейера, что избавляет оператора от необходимости вручную применять манифесты. Это максимально упрощает управление объектами StatefulSet и полностью соответствует принципам парадигм «инфраструктура как код» и «конфигурация как код».

При выполнении отката изменений, примененных к объекту StatefulSet, необходимо четко понимать возможные последствия, особенно если речь идет о возврате к более ранней версии контейнерного образа при сохранении состояния. Например, если выкатывание новой версии привело к изменению схемы данных, то безопасно откатиться к предыдущей версии можно лишь при условии реализации процедуры обратной миграции данных о состоянии.

В нашем примере для возврата к образу `mysql:8.3.0` необходимо либо вручную изменить YAML-манифест, либо отменить коммит в репозитории, если он используется для хранения конфигураций. После этого останется лишь применить конфигурацию заново с помощью команды `kubectl apply`.

В заключительном разделе этой главы мы рассмотрим лучшие практики работы с объектами StatefulSet в Kubernetes.

Лучшие практики работы с объектами StatefulSet

В этом разделе описаны лучшие практики работы с объектами StatefulSet. Этот список не является исчерпывающим, но служит хорошей отправной точкой при освоении Kubernetes.

Управляйте объектами StatefulSet декларативным способом

В мире DevOps и контейнеризованных приложений хорошей практикой считается использование декларативного подхода при изменении инфраструктуры и ПО. Именно этот подход лежит в основе таких парадигм, как «инфраструктура

как код» (IaC) и «конфигурация как код» (CaC). В Kubernetes декларативные обновления легко выполняются с помощью команды `kubectl apply`, которую можно применить как к отдельному YAML-файлу, так и к целой директории таких манифестов.

Для удаления объектов, напротив, предпочтительнее использовать императивные команды, так как это дает более предсказуемый результат и снижает вероятность возникновения ошибок. Декларативное удаление ресурсов кластера уместно в CI/CD- и GitOps-сценариях, в которых весь процесс полностью автоматизирован.

Тот же принцип применим и к объектам StatefulSet. Выкатывать и откатывать их обновления гораздо проще, если файлы YAML-манифестов версионируются и хранятся в репозитории системы контроля версий. Команды `kubectl rollout undo` и `kubectl set image statefulset` обычно не рекомендуется использовать в рабочих средах, особенно если с кластером одновременно взаимодействует несколько операторов.

Не задавайте нулевое значение параметра terminationGracePeriodSeconds для подов StatefulSet

В спецификации пода можно задать параметр `terminationGracePeriodSeconds`, который сообщает утилите `kubelet`, сколько времени следует выделить поду для корректного завершения работы. Если указать для него нулевое значение (`terminationGracePeriodSeconds: 0`), это приведет к *мгновенной* принудительной остановке пода. Такой подход не рекомендуется применять в отношении объектов StatefulSet, поскольку они часто требуют корректного завершения работы контейнера или выполнения хуков жизненного цикла `preStop` перед его удалением. В противном случае существует риск нарушения согласованности состояния StatefulSet. Подробнее о хуках жизненного цикла контейнера можно узнать в документации: https://kubernetes.io/docs/concepts/containers/container-lifecycle-hooks/#container-hooks.

Перед удалением StatefulSet уменьшайте количество реплик до нуля

Если при удалении StatefulSet вы планируете повторно использовать заявки PVC, обеспечьте корректное завершение работы этого объекта. Это позволит избежать ошибок из-за несогласованного состояния данных в объектах PVC при последующем повторном развертывании. Если выполнить команду `kubectl`

`delete` без предварительного масштабирования StatefulSet, все поды будут остановлены *одновременно*, что не является желательным. Рекомендуется сначала уменьшить количество реплик до нуля, а затем удалить сам объект StatefulSet.

Обеспечьте совместимость состояния при откате изменений

Если вы планируете выполнять откат изменений, примененных к объекту StatefulSet, необходимо четко понимать возможные последствия, особенно если речь идет о возврате к более ранней версии контейнерного образа при сохранении состояния. Например, если выкатывание новой версии привело к изменению схемы данных, то безопасно откатиться к предыдущей версии можно лишь при условии реализации процедуры обратной миграции данных о состоянии. В противном случае откат приведет к пересозданию подов со старым образом контейнера и они не смогут корректно запуститься из-за несовместимости данных о состоянии.

Не создавайте поды с метками, совпадающими с селектором существующего объекта StatefulSet

Kubernetes позволяет вручную создавать поды с метками, совпадающими с селектором уже существующего объекта StatefulSet. Это можно сделать, создав «голый» под или используя другой объект Deployment или ReplicaSet. Такая ситуация приводит к конфликтам, предотвращение которых не входит в обязанности Kubernetes, и заставляет существующий объект StatefulSet «считать», что он создал эти поды сам. Результаты могут быть непредсказуемыми, поэтому вам следует внимательно относиться к назначению меток ресурсам в кластере. Мы рекомендуем использовать семантическую систему назначения меток, о которой можно узнать в официальной документации: https://kubernetes.io/docs/concepts/configuration/overview/#using-labels.

Используйте удаленное хранилище для объектов PV

При работе с объектами StatefulSet важно использовать удаленное хранилище, то есть хранить данные приложения в отдельной системе. Обычно в качестве такой системы выступает сетевое хранилище (Network-Attached Storage, NAS), сеть хранения данных (Storage Area Network, SAN) или облачное хранилище. Размещение данных в удаленном хранилище обеспечивает их доступность для любого экземпляра приложения (или любого узла в кластере) даже в случае перезапуска или перемещения пода. Такой подход гарантирует сохранность и устойчивость данных, снижая риск их потери при сбоях или обновлениях объекта StatefulSet.

Используйте проверки жизнеспособности и готовности

В случае приложений с сохранением состояния работоспособность пода определяется не только его переходом в состояние `Running`, но и возможностью получать доступ к своему постоянному состоянию и корректно его обрабатывать. Убедиться в соблюдении этих условий помогают проверки жизнеспособности. Если такая проверка стабильно завершается неудачей, это указывает на более глубокую проблему, не позволяющую поду корректно обрабатывать свое состояние. В этом случае перезапуск пода может активизировать механизмы восстановления или позволить контроллеру StatefulSet организовать аварийное переключение на другой исправный под с тем же состоянием.

Объекты StatefulSet часто используются для управления сервисами, которые требуют доступности определенных данных или конфигураций перед началом обслуживания трафика. Проверки готовности можно настроить так, чтобы они определяли готовность пода к работе и доступность его состояния. Блокируя направление трафика на неготовые к работе поды, вы обеспечиваете стабильную работу приложения и предотвращаете рассогласование данных.

Следите за состоянием объектов StatefulSet

Постоянный контроль состояния и производительности объектов StatefulSet имеет решающее значение. Используйте инструменты мониторинга для отслеживания таких ключевых метрик, как количество перезапусков подов, уровень потребления ресурсов и ошибки приложения. Это позволит быстрее выявлять и устранять потенциальные проблемы, предотвращая их влияние на стабильность и работоспособность вашего ПО.

Резюме

В этой главе мы говорили о развертывании приложений и рабочих нагрузок *с сохранением состояния* в Kubernetes с помощью объекта StatefulSet. Сначала мы рассмотрели подходы к сохранению состояния в контейнерах и подах Kubernetes, а затем на их основе описали то, как объект StatefulSet обеспечивает сохранение состояния. Затем мы создали экземпляр такого объекта вместе с headless-сервисом и выяснили, как объекты PVC и PV используются в StatefulSet для сохранения состояния между перезапусками подов. Далее вы узнали, как масштабировать StatefulSet и выполнять обновления с помощью *канареечных* развертываний и поэтапного выкатывания изменений. В завершение была представлена подборка лучших практик работы с объектами StatefulSet.

В следующей главе вы узнаете, как управлять особыми типами рабочих нагрузок, требующих поддержания ровно одного пода на каждом узле кластера Kubernetes, и познакомитесь с объектом DaemonSet.

Дополнительные источники информации

- Объекты StatefulSet: https://kubernetes.io/docs/concepts/workloads/controllers/statefulset/.
- Headless-сервисы: https://kubernetes.io/docs/concepts/services-networking/service/#headless-services.
- Хуки жизненного цикла контейнеров: https://kubernetes.io/docs/concepts/containers/container-lifecycle-hooks/#container-hooks.

13 Объект DaemonSet: запуск одного пода на каждом узле

В предыдущих главах мы рассмотрели самые распространенные контроллеры Kubernetes, предназначенные для управления подами, — ReplicaSet, Deployment и StatefulSet. Как правило, при запуске компонентов облачных приложений, содержащих *бизнес-логику*, для управления подами используются объекты Deployment или StatefulSet. Для выполнения пакетных рабочих нагрузок применяются объекты Job и CronJob.

Однако бывают ситуации, требующие запуска вспомогательных компонентов, например, для выполнения служебных задач, сбора логов или агрегирования метрик. В таких случаях используется объект *DaemonSet* (набор демонов), обеспечивающий запуск *одной* реплики пода на *каждом* узле кластера (если не указано иное). При добавлении нового узла в кластер ему автоматически назначается под, а при удалении узла работа соответствующего пода завершается. Все эти действия выполняются контроллером DaemonSet.

Технические требования

Для работы с этой главой вам потребуется:

- развернутый кластер Kubernetes. Вы можете использовать как локальный, так и облачный кластер, однако для полноценного усвоения материала рекомендуется отдать предпочтение *многоузловому* кластеру;
- интерфейс командной строки `kubectl`, настроенный для взаимодействия с кластером.

Инструкции по созданию кластера Kubernetes и настройке клиента `kubectl` приведены в главах 2 и 3.

Примеры кода для этой главы можно загрузить из официального репозитория GitHub по адресу https://github.com/PacktPublishing/The-Kubernetes-Bible-Second-Edition/tree/main/Chapter13.

Знакомство с объектом DaemonSet

Термин «демон» (daemon) имеет давнюю историю в мире операционных систем. В общем смысле он обозначает программу, работающую в фоновом режиме без непосредственного взаимодействия с пользователем. Чаще всего демоны отвечают за выполнение служебных задач, обработку сетевых запросов или мониторинг состояния оборудования. Это те процессы, которые должны выполняться постоянно и надежно — с момента загрузки системы до завершения ее работы.

Демоны традиционно ассоциируются с операционными системами семейства Unix. В среде Windows аналогичные фоновые процессы называются службами.

Представьте, что вам нужно запустить программу на каждом компьютере в офисе для отслеживания его состояния. Для решения этой задачи в Kubernetes предусмотрен объект DaemonSet. Он выступает в роли специального менеджера подов, гарантируя запуск ровно одной копии пода на каждом узле кластера. Это особенно важно в сценариях с использованием протокола gRPC, требующих создания выделенного сокета в файловой системе узла. Такую задачу проще решить при запуске только одного пода на каждом узле.

Поды, управляемые объектом DaemonSet, выполняют важные функции для всего кластера, включая:

- *мониторинг* — контроль состояния каждого узла;
- *сбор логов* — агрегирование информации о работе узлов и подов, запущенных на них;
- *управление хранилищем* — обработка запросов приложений на выделение дискового пространства;
- *управление сетью* — обеспечение сетевого взаимодействия с помощью таких компонентов кластера, как `kube-proxy` и CNI-плагины (например, Calico).

По мере добавления в кластер новых узлов объект DaemonSet автоматически создает дополнительные поды для их обслуживания. При удалении узлов соответствующие поды также автоматически завершают работу. Таким образом, DaemonSet можно рассматривать как самоорганизующуюся систему, которая непрерывно следит за тем, чтобы каждый узел получал необходимую поддержку.

На рис. 13.1 показана общая структура объекта DaemonSet.

В простых конфигурациях один объект DaemonSet может обеспечивать выполнение всей необходимой работы (например, мониторинга) на всех узлах кластера. В более сложных сценариях для решения одной и той же задачи может использоваться несколько объектов DaemonSet с разными настройками или требованиями к ресурсам в зависимости от типа узла (например, мощные серверы или базовые машины).

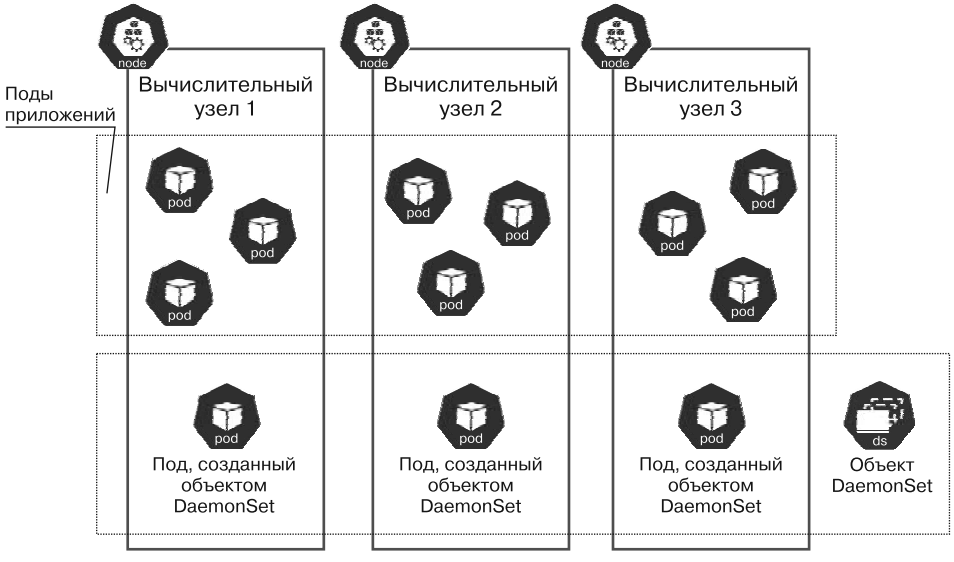

Рис. 13.1. Объект DaemonSet в Kubernetes

Использование DaemonSet гарантирует, что на каждом узле кластера Kubernetes будут запущены необходимые служебные компоненты, поддерживающие его стабильную и эффективную работу.

Все, что вы узнали в предыдущих главах об объектах ReplicaSet, Deployment и StatefulSet, в целом применимо и к DaemonSet. В его спецификации также нужно указать шаблон пода, селекторы меток и при необходимости селекторы узлов, позволяющие запускать поды только на определенных узлах.

В зависимости от конкретной задачи доступ к DaemonSet из других подов или из внешней сети может и не потребоваться. Например, если задача DaemonSet заключается в периодической очистке файловой системы узла, то устанавливать сетевое соединение с такими подами нет смысла. Если же ваш сценарий требует входящего или исходящего сетевого подключения к подам DaemonSet, то вы можете применить следующие стандартные подходы.

- *Сопоставление портов контейнера с портами узла.* Поскольку поды DaemonSet гарантированно существуют в единственном экземпляре на каждом узле кластера, можно использовать сопоставление портов контейнера с портами узла. В этом случае клиентам необходимо знать IP-адреса узлов.
- *Отправка данных другому сервису.* В некоторых случаях достаточно того, чтобы поды DaemonSet самостоятельно отправляли обновления другим сервисам без необходимости в приеме входящих соединений.
- *Использование headless-сервиса, соответствующего селекторам меток подов DaemonSet.* Этот подход работает так же, как в случае с объектом

StatefulSet: с помощью кластерной службы DNS можно получить несколько A-записей для подов, используя DNS-имя headless-сервиса.

- *Использование обычного сервиса, соответствующего селекторам меток подов DaemonSet.* В редких случаях вам может потребоваться доступ к *любому* поду в составе DaemonSet. В этом случае можно создать стандартный объект Service, например типа `ClusterIP`, позволяющий устанавливать соединение с произвольным подом DaemonSet.

Как уже было сказано, объект DaemonSet гарантирует то, что поды, обеспечивающие работу критически важных служб, будут запущены на всех узлах (или только на выбранных). В следующем подразделе мы поговорим о механизме планирования таких подов.

Планирование подов DaemonSet

Объекты DaemonSet гарантируют запуск единственного пода на каждом подходящем узле кластера Kubernetes. Контроллер DaemonSet использует правила предпочтения узлов (node affinity), обеспечивающие запуск подов DaemonSet только на тех узлах, которые соответствуют заданным условиям, что особенно полезно при работе со сложными кластерами. Затем стандартный планировщик привязывает под к выбранному узлу, вытесняя существующие поды при нехватке ресурсов. При желании можно указать пользовательский планировщик, однако контроллер DaemonSet в любом случае гарантирует размещение подов в соответствии с заданными правилами предпочтения узлов.

В следующем разделе мы поговорим о проверке ресурсов DaemonSet в кластере Kubernetes.

Проверка объектов DaemonSet

После развертывания кластера Kubernetes вы можете обнаружить, что некоторые объекты DaemonSet уже установлены по умолчанию в виде системных или вспомогательных компонентов вроде службы DNS или CNI-плагина.

В качестве примера создадим многоузловой кластер `minikube` с помощью следующего фрагмента кода:

```
$ minikube start \
  --driver=virtualbox \
  --nodes=3 \
  --cni=calico \
  --cpus=2 \
  --memory=2g \
  --kubernetes-version=v1.31.0 \
  --container-runtime=containerd
```

После создания кластера мы можем отобразить список узлов кластера, выполнив команду:

```
$ kubectl get nodes
NAME           STATUS   ROLES           AGE     VERSION
minikube       Ready    control-plane   3m28s   v1.31.0
minikube-m02   Ready    <none>          2m29s   v1.31.0
minikube-m03   Ready    <none>          91s     v1.31.0
```

Теперь проверим наличие объектов DaemonSet в только что созданном кластере:

```
$ kubectl get daemonsets -A
NAMESPACE     NAME          DESIRED   CURRENT   READY   UP-TO-DATE   AVAILABLE
NODE SELECTOR            AGE
kube-system   calico-node   3         3         3       3            3
kubernetes.io/os=linux   63m
kube-system   kube-proxy    3         3         3       3            3
kubernetes.io/os=linux   63m
```

Параметры `-A` или `--all-namespaces` использованной выше команды позволяют отобразить запрошенные объекты, принадлежащие всем пространствам имен.

Если вы создавали кластер `minikube` тем же способом, то в выводе будут присутствовать такие объекты, как `calico-node` и `kube-proxy`, развернутые в виде DaemonSet. (Вы также можете установить Calico в других кластерах Kubernetes и выполнить оставшиеся шаги этой инструкции.) Ранее мы уже включили Calico в качестве CNI-плагина в наш кластер `minikube`. При организации сетевого взаимодействия с помощью Kubernetes Calico обычно развертывается именно в виде объекта DaemonSet.

Пока можете не обращать внимания на DaemonSet kube-proxy, поскольку в minikube он по умолчанию запускается в виде данного объекта. Это обеспечивает постоянную доступность kube-proxy (компонента, управляющего сетевым трафиком в кластере) на каждом узле в среде minikube.

Теперь выведем список подов, созданных объектом DaemonSet `calico-node`:

```
$ kubectl get po -n kube-system -o wide | grep calico
calico-kube-controllers-ddf655445-jx26x   1/1     Running   0   82m
10.244.120.65   minikube       <none>   <none>
calico-node-fkjxb                         1/1     Running   0   82m
192.168.59.126  minikube       <none>   <none>
calico-node-nrzpb                         1/1     Running   0   81m
192.168.59.128  minikube-m03   <none>   <none>
calico-node-sg66x                         1/1     Running   0   82m
192.168.59.127  minikube-m02   <none>   <none>
```

Полученный вывод говорит о следующем.

- Поды Calico развернуты на всех узлах `minikube`. Каждый под работает на отдельном узле, который соответствует виртуальной машине `minikube` (`minikube`, `minikube-m02` и `minikube-m03`). Это указывает на то, что Calico использует DaemonSet для запуска пода на каждом узле.
- Под `calico-kube-controllers-ddf655445-jx26x` является контроллером CNI-плагина Calico.

Поскольку DaemonSet Calico в данном случае устанавливается автоматически вместе с `minikube`, мы не будем подробно рассматривать его конфигурацию. Вместо этого в следующем разделе мы создадим объект DaemonSet с нуля и подробно изучим принцип его работы.

Создание объектов DaemonSet и управление ими

Для демонстрации принципа работы объектов DaemonSet мы воспользуемся подами *Fluentd*. Fluentd — это популярный проект с открытым исходным кодом, предназначенный для агрегации логов из разных источников. Он эффективно собирает, фильтрует и преобразует сообщения журналов перед их передачей в другие системы для последующего анализа и хранения.

Для получения доступа к конечным точкам DaemonSet создадим headless-сервис, как делали это в главе 12. В реальных сценариях применения объекты DaemonSet, как правило, имеют более сложную конфигурацию и требуют монтирования различных системных ресурсов в поды. В нашем примере мы сохраним конфигурацию максимально простой, чтобы сосредоточиться на ключевых принципах работы DaemonSet.

Для читателей, желающих поработать над другим примером, мы подготовили рабочую версию экспортера Prometheus node-exporter, развернутого в виде DaemonSet и доступного через headless-сервис (см. файл node-exporter.yaml). Для этого при выполнении описанных далее шагов используйте имя сервиса node-exporter, порт 9100, а для запросов, отправляемых с помощью wget, добавьте путь /metrics. Данный объект DaemonSet публикует метрики узлов в формате Prometheus data model (модель данных Prometheus) на порте 9100 по пути /metrics.

Далее мы рассмотрим все YAML-манифесты, необходимые для создания объекта DaemonSet, и применим их к кластеру.

Создание объекта DaemonSet

Следуя лучшим практикам, мы будем использовать декларативный подход для создания объекта DaemonSet в этом примере. Сначала рассмотрим YAML-манифест `fluentd-daemonset.yaml`.

Первая часть манифеста создает отдельное пространство имен, предназначенное для логирования:

```
---
apiVersion: v1
kind: Namespace
metadata:
  name: logging
```

Далее следует описание самого объекта DaemonSet:

```
apiVersion: apps/v1
kind: DaemonSet
metadata:
  name: fluentd-elasticsearch
  namespace: kube-system
  labels:
    k8s-app: fluentd-logging
spec:
  selector:
    matchLabels:
      name: fluentd-elasticsearch
# (продолжение далее)
```

Первая часть показанного выше файла содержит метаданные и селектор меток подов, аналогичные тем, которые вы уже видели в манифестах Deployment и StatefulSet. Во второй части файла приводится шаблон пода, который будет использоваться объектом DaemonSet:

```
# (продолжение)
  template:
    metadata:
      labels:
        name: fluentd-elasticsearch
    spec:
      containers:
        - name: fluentd-elasticsearch
          image: quay.io/fluentd_elasticsearch/fluentd:v4.7
          resources:
            limits:
              memory: 200Mi
            requests:
              cpu: 100m
              memory: 200Mi
```

```
        volumeMounts:
          - name: varlog
            mountPath: /var/log
      terminationGracePeriodSeconds: 30
      volumes:
        - name: varlog
          hostPath:
            path: /var/log
```

Как видите, структура спецификации DaemonSet во многом совпадает с аналогичным разделом манифестов Deployment и StatefulSet. В данном случае также необходимо определить шаблон пода и задать селектор меток. Отличие состоит в том, что здесь отсутствует поле `replicas`, поскольку количество подов в кластере определяется числом его узлов. Спецификация DaemonSet включает два основных компонента.

- `spec.selector` — селектор меток, задающий правила идентификации подов, находящихся под управлением объекта DaemonSet. Он может быть основан как *на множестве*, так и *на равенстве*.
- `spec.template` — шаблон, определяющий способ создания пода. Метки, указанные в разделе `metadata`, должны соответствовать условиям, определенным в разделе `selector`.

Распространенной практикой также является задание параметров `.spec.template.spec.nodeSelector` или `.spec.template.spec.tolerations`, позволяющих контролировать выбор узлов, на которых будут запускаться поды DaemonSet. О планировании подов мы подробно поговорим в главе 19. Дополнительно можно задать параметры `.spec.updateStrategy`, `.spec.revisionHistoryLimit` и `.spec.minReadySeconds`, аналогичные тем, которые задаются для объектов Deployment.

В гибридных кластерах Kubernetes, состоящих из узлов Linux и Windows, селекторы узлов или правила их предпочтения, задаваемые для объекта DaemonSet, часто используются для гарантированного назначения подов узлам с определенной операционной системой. Это имеет смысл, поскольку контейнерные среды выполнения и ОС таких узлов фундаментально различаются.

Обратите внимание на строки, в которых выполняется монтирование томов: поды Fluentd будут получать доступ к каталогу `/var/log` хоста (на котором они запущены), чтобы обрабатывать данные журналов и передавать их агрегатору.

Следует отметить, что в реальной среде необходимо указать адреса целевых серверов Elasticsearch, куда Fluentd будет отправлять логи. Однако в рамках нашего демонстрационного примера мы не будем обсуждать настройку Elasticsearch.

Такие параметры могут передаваться контейнерам через переменные окружения, как показано в YAML-файле `fluentd-daemonset.yaml`.

```
        env:
          - name: FLUENT_ELASTICSEARCH_HOST
            value: "elasticsearch-logging"
          - name: FLUENT_ELASTICSEARCH_PORT
            value: "9200"
```

Теперь, имея все необходимые YAML-манифесты, мы можем применить их к кластеру. Для этого выполните следующие шаги.

1. Создайте объект DaemonSet `fluentd-elasticsearch`:

```
$ kubectl apply -f fluentd-daemonset.yaml
namespace/logging created
daemonset.apps/fluentd-elasticsearch created
```

2. Для наблюдения за процессом создания DaemonSet используйте команду `kubectl describe`:

```
$ kubectl describe daemonset fluentd-elasticsearch -n logging
```

   При работе с `kubectl` вместо `daemonset` можно использовать сокращение `ds`.

3. Выполнив команду `kubectl get pods` с параметром `-w`, вы увидите, как для каждого узла в кластере будет запланирован отдельный под:

```
$ kubectl get po -n logging -o wide
NAME                          READY   STATUS    RESTARTS   AGE     IP
NODE          NOMINATED NODE   READINESS GATES
fluentd-elasticsearch-cs4hm 1/1     Running   0          3m48s
10.244.120.68  minikube      <none>           <none>
fluentd-elasticsearch-stfqs 1/1     Running   0          3m48s
10.244.205.194 minikube-m02  <none>           <none>
fluentd-elasticsearch-zk6pt 1/1     Running   0          3m48s
10.244.151.2   minikube-m03  <none>           <none>
```

В нашем случае кластер состоит из трех узлов, поэтому было создано три пода.

Итак, мы успешно развернули объект DaemonSet и теперь можем убедиться в том, что он работает ожидаемым образом. Чтобы проверить наличие у подов Fluentd доступа к логам узлов Kubernetes, подключитесь к одному из них и просмотрите содержимое каталога `/var/log`:

```
$ kubectl exec -n logging -it fluentd-elasticsearch-cs4hm -- /bin/bash
root@fluentd-elasticsearch-cs4hm:/# ls -l /var/log/
total 20
drwxr-xr-x   3 root root 4096 May 29 10:56 calico
drwxr-xr-x   2 root root 4096 May 29 12:40 containers
drwx------   3 root root 4096 May 29 10:55 crio
drwxr-xr-x   2 root root 4096 May 29 11:53 journal
drwxr-x---  12 root root 4096 May 29 12:40 pods
root@fluentd-elasticsearch-cs4hm:/#
```

Этот пример наглядно демонстрирует ключевые принципы работы механизма планирования подов DaemonSet в Kubernetes.

Хорошей практикой считается задание ограничений (taints) и допусков (tolerations) для узлов, на которых запускаются объекты DaemonSet. Эти механизмы будут подробно рассмотрены в главе 19.

Теперь обсудим продвинутые параметры конфигурации DaemonSet.

Приоритизация критических объектов DaemonSet в Kubernetes

При управлении критически важными системными компонентами с помощью объектов DaemonSet необходимо обеспечить их бесперебойную работу. Для этого используется механизм назначения приоритетов подов и классы приоритета, гарантирующие то, что критически важные поды не будут вытеснены менее приоритетными задачами.

В Kubernetes каждому поду назначается уровень приоритета, определяющий его относительную важность в кластере. Поды с более высоким приоритетом считаются более значимыми по сравнению с низкоприоритетными подами.

Присвоение объекту DaemonSet более высокого класса приоритета (PriorityClass) повышает значимость его подов. Таким образом, при нехватке ресурсов планировщик не сможет вытеснить эти поды ради выполнения менее приоритетных задач.

Объект PriorityClass определяет конкретный уровень приоритета подов. Его значения могут варьироваться от отрицательных целых чисел до 1 миллиарда, при этом большие значения соответствуют более высокому приоритету.

Ниже приведен пример YAML-манифеста для объекта PriorityClass:

```
# priorityclass.yaml
apiVersion: scheduling.k8s.io/v1
kind: PriorityClass
metadata:
  name: fluentd-priority
value: 100000 # Высокое значение для критически важных подов
globalDefault: false # Не является классом по умолчанию для всех подов
description: "Fluentd Daemonset priority class"
```

После создания объекта PriorityClass его можно использовать в конфигурации DaemonSet, добавив ссылку на него:

```
spec:
  template:
    spec:
      priorityClassName: fluentd-priority
```

Для справки: системные компоненты, такие как `kube-proxy` и CNI-плагин кластера (например, Calico), часто используют встроенный объект PriorityClass `system-node-critical`. Этот класс обладает наивысшим приоритетом и гарантирует, что соответствующие поды не будут выселены (вытеснены) ни при каких обстоятельствах.

Далее мы посмотрим, как можно изменить объект DaemonSet, чтобы выкатить новую версию контейнерного образа для его подов.

Изменение объекта DaemonSet

Обновление объекта DaemonSet практически не отличается от обновления объекта Deployment. Изменение *шаблона пода* инициирует *выкатывание* новой ревизии DaemonSet в соответствии с выбранной стратегией обновления (`updateStrategy`). В Kubernetes доступны две такие стратегии.

- `RollingUpdate` — стратегия по умолчанию, позволяющая постепенно выкатывать новую версию демона. Она схожа со стратегией постепенного обновления объектов Deployment, так как позволяет задать параметр `.spec.updateStrategy.rollingUpdate.maxUnavailable`, определяющий максимальное число недоступных во время обновления подов (по умолчанию — 1), и `.spec.minReadySeconds` (по умолчанию — 0). При этом гарантируется, что в процессе выкатывания обновления на каждом узле кластера будет присутствовать *не более одного* пода DaemonSet, находящегося в состоянии `Running`.
- `OnDelete` — стратегия, реализующая устаревший механизм обновления DaemonSet, существовавший до версии Kubernetes 1.6. При использовании этой стратегии объект DaemonSet *не* обновляет поды автоматически — для применения нового шаблона необходимо удалить под вручную. Это удобно в тех случаях, когда перед переходом к следующему узлу необходимо выполнить дополнительные проверки или другие действия.

Выкатывание новой ревизии DaemonSet можно контролировать так же, как и в случае с объектами Deployment. Команда `kubectl rollout status` позволяет отслеживать прогресс обновления, а `kubectl rollout undo` — откатывать изменения *императивным* способом. Чтобы обновить версию контейнерного образа в подах DaemonSet *декларативным* способом, выполните следующие шаги.

1. Измените YAML-манифест `fluentd-daemonset.yaml`, указав в шаблоне пода контейнерный образ `quay.io/fluentd_elasticsearch/fluentd:v4.7.5`:

```
...
        containers:
          - name: fluentd-elasticsearch
            image: quay.io/fluentd_elasticsearch/fluentd:v4.7.5
...
```

2. Примените обновленный манифест к кластеру:

```
$ kubectl apply -f fluentd-daemonset.yaml
namespace/logging unchanged
daemonset.apps/fluentd-elasticsearch configured
```

3. Сразу после этого выполните команду `kubectl rollout status` для отслеживания прогресса в реальном времени:

```
$ kubectl rollout status ds -n logging
Waiting for daemon set "fluentd-elasticsearch" rollout to finish: 2 out
of 3 new pods have been updated...
Waiting for daemon set "fluentd-elasticsearch" rollout to finish: 2 out
of 3 new pods have been updated...
Waiting for daemon set "fluentd-elasticsearch" rollout to finish: 2 of 3
updated pods are available...
daemon set "fluentd-elasticsearch" successfully rolled out
```

4. Вы также можете воспользоваться командой `kubectl describe`, чтобы просмотреть события, связанные с объектом DaemonSet, и увидеть, в каком порядке пересоздавались поды.

```
$ kubectl describe ds -n logging
Name:           fluentd-elasticsearch
Selector:       name=fluentd-elasticsearch
... <фрагмент опущен для краткости> ...
Events:
  Type    Reason            Age    From                  Message
  ----    ------            ----   ----                  -------
... <фрагмент опущен для краткости> ...
  Normal  SuccessfulDelete  5m52s  daemonset-controller  Deleted pod:
fluentd-elasticsearch-zk6pt
  Normal  SuccessfulCreate  5m51s  daemonset-controller  Created pod:
fluentd-elasticsearch-fxffp
```

Как видите, поды заменялись один за другим.

Контейнерный образ DaemonSet также можно изменить императивным способом с помощью команды kubectl set image ds fluentd-elasticsearch fluentd-elasticsearch=quay.io/fluentd_ elasticsearch/fluentd:v4.7.5 -n logging. Однако этот способ не рекомендуется использовать в рабочих средах.

Кроме того, объект DaemonSet автоматически создает поды при добавлении нового узла в кластер, если этот узел соответствует селектору и правилам предпочтения узлов. Когда узел удаляется из кластера, соответствующий под тоже завершает работу. То же самое произойдет при изменении меток или ограничений на узле: если новые значения соответствуют условиям DaemonSet, на узле будет создан новый под; если же они перестанут соответствовать заданным условиям, существующий под будет удален.

В следующем подразделе мы рассмотрим способ отката обновлений объекта DaemonSet.

Откат обновлений объекта DaemonSet

Откатить изменения, внесенные в объект DaemonSet, можно с помощью команды `kubectl rollback`:

```
$ kubectl rollout undo daemonset fluentd-elasticsearch -n logging
```

Однако в рабочих средах настоятельно рекомендуется использовать декларативный подход, то есть вносить изменения в YAML-манифест и повторно применять конфигурацию.

Теперь посмотрим, как можно удалить объект DaemonSet.

Удаление объекта DaemonSet

Удалить объект DaemonSet можно двумя способами:

- удалить DaemonSet вместе с подами, которыми он управляет;
- удалить только сам объект DaemonSet, оставив поды нетронутыми.

Чтобы удалить DaemonSet вместе с подами, используйте команду `kubectl delete`:

```
$ kubectl delete ds fluentd-elasticsearch -n logging
```

При ее выполнении сначала будет завершена работа подов, а затем удален сам объект DaemonSet.

Чтобы удалить только объект DaemonSet, не затрагивая поды, используйте параметр `--cascade=orphan`:

```
$ kubectl delete ds fluentd-elasticsearch -n logging --cascade=orphan
```

Если после выполнения этой команды вы проверите список подов, присутствующих в кластере, то увидите, что поды, ранее принадлежавшие объекту DaemonSet `fluentd-elasticsearch`, никуда не исчезли.

Если вы освобождаете узел с помощью команды kubectl drain и на этом узле запущены поды, управляемые DaemonSet, то для успешного завершения операции необходимо добавить флаг --ignore-daemonsets.

Теперь рассмотрим наиболее распространенные сценарии использования объектов DaemonSet в Kubernetes.

Типичные сценарии использования объектов DaemonSet

На практике объект DaemonSet чаще всего используется либо для реализации фундаментальных функций кластера, без которых он не может работать, либо для запуска вспомогательных рабочих нагрузок, связанных с обслуживанием или сбором данных. Ниже перечислены наиболее распространенные и интересные сценарии использования объектов DaemonSet.

- В зависимости от способа развертывания кластера служба `kube-proxy`, являющаяся одним из ключевых компонентов Kubernetes, может быть установлена в виде объекта DaemonSet, а не в качестве обычной системной службы. Например, в сервисе *Azure Kubernetes Service (AKS)* определение этого объекта DaemonSet можно просмотреть с помощью команды `kubectl describe ds -n kube-system kube-proxy`. Это классический пример базовой службы, которая должна выполняться в единственном экземпляре на каждом узле кластера. Пример YAML-манифеста для `kube-proxy` доступен по ссылке: https://github.com/kubernetes/kubernetes/blob/release-1.30/cluster/addons/kube-proxy/kube-proxy-ds.yaml.
- К фундаментальным службам, запускаемым в виде DaemonSet, также относятся *CNI*-плагины и агенты, обеспечивающие сетевое взаимодействие в кластере Kubernetes. Мы уже рассматривали соответствующий пример в начале главы при использовании CNI-плагина Calico в кластере `minikube`. Еще одним распространенным примером является агент Flannel (https://github.com/flannel-io/flannel/blob/master/Documentation/kube-flannel.yml), который запускается на каждом узле и отвечает за выделение подсети из заранее настроенного адресного пространства. Конкретный механизм работы при этом зависит от типа сетевой технологии, используемой в кластере.
- Кластерные службы хранения данных также часто развертываются в виде объектов DaemonSet. Хорошим примером является *Object Storage Daemon* (OSD) системы *Ceph* — распределенной платформы, предоставляющей единое хранилище для объектных, блочных и файловых данных. Компонент OSD отвечает за хранение объектов в локальной файловой системе каждого узла и предоставляет к ним сетевой доступ. Пример манифеста можно найти в составе шаблона Helm-чарта по адресу https://github.com/ceph/ceph-container/blob/stable-7.0/examples/helm/ceph/templates/osd/daemonset.yaml.
- Ingress-контроллеры в Kubernetes (см. главу 21) также могут быть развернуты в виде DaemonSet. Например, при установке сервера Nginx в качестве Ingress-контроллера вы можете развернуть его в виде этого объекта: https://github.com/nginxinc/kubernetes-ingress/blob/master/deployments/daemon-set/nginx-ingress.yaml. Такой способ особенно распространен при развертывании кластеров Kubernetes на физических серверах.

- Агенты для сбора и агрегации логов, вроде `fluentd`, тоже часто развертываются в виде DaemonSet. Несколько примеров YAML-манифестов можно найти в официальном репозитории проекта: https://github.com/fluent/fluentd-kubernetes-daemonset.
- Агенты для сбора метрик узлов — это еще один классический случай использования DaemonSet. Хорошо известным примером является экспортер Prometheus `node-exporter`, манифест которого доступен по адресу https://github.com/prometheus-operator/kube-prometheus/blob/main/manifests/nodeExporter-daemonset.yaml.

Этот список можно продолжить. Как видите, объект DaemonSet является одним из ключевых строительных элементов, используемых инженерами при проектировании рабочих нагрузок в Kubernetes. Зачастую именно эти объекты образуют скрытую основу кластера, обеспечивающую его полноценное функционирование.

Далее мы рассмотрим лучшие практики реализации объектов DaemonSet.

Лучшие практики реализации объектов DaemonSet

Объекты DaemonSet — это мощный инструмент Kubernetes, предназначенный для управления подами, запускаемыми на каждом узле кластера. Однако, чтобы обеспечить их надежную и предсказуемую работу, стоит придерживаться следующих рекомендаций.

- *Запросы и лимиты ресурсов*. Задание адекватных значений параметров `requests` и `limits` для подов DaemonSet помогает эффективно распределять ресурсы и предотвращает ситуацию, в которой другие поды в кластере испытывают нехватку вычислительных мощностей.
- *Четкая структура*. Размещайте каждый объект DaemonSet в отдельном пространстве имен. Такой подход упрощает администрирование и делает структуру кластера более упорядоченной.
- *Гибкое планирование*. При создании DaemonSet рекомендуется использовать правило `preferredDuringSchedulingIgnoredDuringExecution` вместо `requiredDuringSchedulingIgnoredDuringExecution`. Первый вариант обеспечивает бóльшую гибкость при недостаточном количестве доступных узлов в кластере.
- *Ожидание готовности* (не обязательно). Задание параметра `minReadySeconds` в шаблоне пода позволят указать, сколько времени система Kubernetes должна ждать перед созданием новых подов во время обновления. Это гарантирует переход всех существующих подов в состояние готовности до запуска новых.

- *Мониторинг и логирование*. Наблюдение за состоянием и производительностью подов позволяет своевременно обнаруживать неполадки и оценивать работоспособность объектов DaemonSet.
- *Непрерывная работа*. Убедитесь в том, что для подов DaemonSet задана политика перезапуска `RestartPolicy: Always` (используется по умолчанию), чтобы обеспечить автоматический перезапуск подов в случае сбоя.
- *Высокий приоритет*. Присвойте подам DaemonSet повышенный приоритет (например, `10000`), чтобы они всегда получали необходимые ресурсы и не вытеснялись другими подами.
- *Совпадение меток*. Определите селектор подов, соответствующий меткам, указанным в шаблоне DaemonSet, чтобы этот объект управлял именно теми подами, которые вам нужны.

Следование этим рекомендациям позволяет обеспечить стабильную и эффективную работу объектов DaemonSet, а также оптимальное функционирование кластера Kubernetes.

Теперь рассмотрим альтернативы объекту DaemonSet.

Альтернативы объекту DaemonSet

Как правило, объект DaemonSet используется для запуска на каждом узле ровно одного пода, выполняющего определенную функцию. Однако в некоторых случаях стоит рассмотреть альтернативные подходы, которые могут лучше соответствовать вашим запросам.

- При проектировании архитектуры конвейера сбора логов вместо объекта DaemonSet можно использовать контейнер-прицеп. У каждого из этих вариантов есть свои преимущества и недостатки, но в целом контейнеры-прицепы, как правило, проще в реализации и более надежны, хотя могут потреблять больше системных ресурсов.
- Если вам требуется выполнять задания с определенной периодичностью, но не обязательно на каждом узле кластера, лучше использовать объект *CronJob*. Здесь важно проанализировать конкретный сценарий и определить, является ли запуск отдельного пода на каждом узле обязательным условием.
- Демоны операционной системы (например, службы, управляемые `systemd` в Ubuntu) могут выполнять те же функции, что и объект DaemonSet. Недостаток такого подхода заключается в невозможности управлять этими демонами средствами Kubernetes, например с помощью `kubectl`. Однако отсутствие зависимости от сервисов Kubernetes в некоторых случаях может являться преимуществом.

- Для достижения схожего результата можно использовать статические поды (https://kubernetes.io/docs/tasks/configure-pod-container/static-pod/). Такие поды создаются на основе манифестов, размещенных в определенном каталоге, который отслеживает компонент `kubelet`. Статическими подами нельзя управлять с помощью `kubectl`, и чаще всего они применяются для выполнения функций, связанных с инициализацией кластера.

Теперь подведем итоги и обобщим основные сведения об объектах DaemonSet.

Резюме

В этой главе вы научились работать с объектами DaemonSet в Kubernetes и использовать их для управления особыми типами рабочих нагрузок или процессов, которые должны выполняться в единственном экземпляре на каждом узле кластера. Сначала вы создали экземпляр DaemonSet и ознакомились с ключевыми элементами его спецификации. Затем вы научились выкатывать новые ревизии DaemonSet в кластере и отслеживать процесс обновления. В завершение мы обсудили наиболее распространенные сценарии использования этого типа объектов Kubernetes и его возможные альтернативы.

Это был последний тип контроллера для управления подами, обсуждаемый в этой книге. В следующей главе мы перейдем к более продвинутым возможностям Kubernetes. Мы начнем с изучения Helm-чартов, а затем вы познакомитесь с операторами Kubernetes.

Дополнительные источники информации

- Объекты DaemonSet: https://kubernetes.io/docs/concepts/workloads/controllers/daemonset/.
- Сетевая политика: https://minikube.sigs.k8s.io/docs/handbook/network_policy/.
- Развертывание Fluentd в Kubernetes: https://docs.fluentd.org/container-deployment/kubernetes.
- Постепенное обновление объекта DaemonSet: https://kubernetes.io/docs/tasks/manage-daemon/update-daemon-set/.

14 Работа с Helm-чартами и операторами

В экосистеме Kubernetes критически важной задачей является управление распространением приложений и их зависимостями. Это особенно необходимо в тех случаях, когда требуется обеспечить клиентам простую загрузку и установку приложений или организовать их совместное использование несколькими командами. В отличие от менеджеров пакетов Linux (таких как APT или YUM) инструменты Kubernetes, в частности Helm, являются универсальными и не привязаны к конкретному дистрибутиву Kubernetes. Helm позволяет распространять приложения, объединяя множество ресурсов в так называемые чарты. Такой подход упрощает повторное использование и настройку приложений в различных средах, избавляя разработчиков и операторов от необходимости вручную управлять многочисленными YAML-манифестами.

Операторы Kubernetes дополняют инструмент Helm, обеспечивая автоматизированное управление жизненным циклом приложений, в том числе обновлениями, аварийным переключением, резервным копированием и поддержанием согласованности конфигураций. В совокупности Helm и операторы помогают решать задачи масштабирования и управления Kubernetes-приложениями. В текущей главе мы поговорим о разработке Helm-чартов, установке популярных компонентов и эффективном распространении Kubernetes-приложений.

Технические требования

Для работы с этой главой вам потребуется следующее.

- Развернутый кластер Kubernetes. По возможности рекомендуется использовать *многоузловой* или облачный кластер. Вам также следует убедиться в том, что кластер располагает достаточным объемом ресурсов ЦП, памяти и хранилища для запуска нескольких подов. (Вы можете создать более крупный кластер `minikube`, выполнив команду `minikube start --kubernetes-version=1.31.0 --cpus=4 --memory=8g`.)
- Интерфейс командной строки `kubectl`, установленный на локальной машине и настроенный для взаимодействия с кластером.

Инструкции по созданию кластера Kubernetes и настройке клиента `kubectl` приведены в главах 2 и 3. В главах 15, 16 и 17 описан процесс развертывания полнофункционального кластера Kubernetes на различных облачных платформах. Примеры кода для этой главы можно загрузить из официального репозитория GitHub по адресу https://github.com/PacktPublishing/The-Kubernetes-Bible-Second-Edition/tree/main/Chapter14.

Знакомство с инструментом Helm

Распространенный способ демонстрации запуска приложений в контейнере на платформе Kubernetes — публикация сырых YAML-манифестов. Однако у такого подхода есть ряд недостатков.

- Все значения в YAML-шаблонах *жестко заданы*. Это значит, что, если вы решите изменить, например, количество реплик, на которые ссылается объект Service, или значение, хранящееся в объекте ConfigMap, вам придется вручную просмотреть и отредактировать соответствующие файлы манифестов. Аналогичным образом, если вам потребуется развернуть приложение в пространстве имен, отличном от предусмотренного разработчиками, вам необходимо будет вручную внести изменения во все YAML-файлы. Кроме того, без документации бывает трудно понять, какие параметры в YAML-шаблонах являются настраиваемыми.
- Разные приложения могут предусматривать разные процессы развертывания. В экосистеме Kubernetes отсутствует единый стандарт, определяющий, какие манифесты должен предоставлять автор приложения, а какие компоненты вам необходимо развертывать вручную.
- Отсутствие *управления зависимостями*. Например, если ваше приложение зависит от сервера *MySQL*, запущенного в кластере в виде объекта StatefulSet, вам придется либо развернуть его самостоятельно, либо надеяться на то, что автор приложения включил соответствующие YAML-манифесты в свой пакет.

Примерно так же обстоят дела при установке ПО без использования магазина приложений или пакетных менеджеров вроде *Chocolatey*, *APT*, *YUM* или *DNF*. Некоторые программы распространяются в виде установочных сценариев (`setup.sh`), а другие — в виде исполняемых файлов `.exe` или `.msi`, а иногда и простых архивов `.zip`, требующих распаковки и настройки.

В экосистеме Kubernetes для решения вышеперечисленных проблем используется *Helm* (https://helm.sh) — один из популярнейших менеджеров пакетов для приложений и сервисов Kubernetes. Если вы знакомы с такими инструментами, как APT, YUM, npm или Chocolatey, то многие концепции Helm покажутся вам знакомыми и интуитивно понятными. Далее рассмотрим три ключевых понятия Helm.

- *Чарт (chart)* — это *пакет*, который *устанавливается* с помощью *интерфейса командной строки* Helm. Чарт содержит все YAML-манифесты, необходимые для развертывания конкретного приложения в кластере Kubernetes. При этом YAML-файлы могут быть *параметризованы*, что позволяет пользователю передавать собственные конфигурационные значения при установке приложения.
- *Репозиторий (repository)* — это хранилище Helm-чартов, обеспечивающее их сбор и совместное использование. Репозитории бывают как публичными, так и частными. Содержимое публичных репозиториев можно просматривать, например, на Artifact Hub (https://artifacthub.io/). Частные репозитории, как правило, используются для обмена компонентами между командами, работающими над одним продуктом.
- *Релиз (release)* — это *экземпляр* Helm-чарта, установленный и запущенный в кластере Kubernetes. Именно с ним вы взаимодействуете через интерфейс командной строки Helm, выполняя, например, обновление или удаление. Один и тот же чарт можно установить несколько раз в рамках одного кластера, создав несколько релизов с уникальными именами.

Helm-чарты содержат параметризованные YAML-манифесты, которые хранятся в репозиториях Helm и распространяются через них. После установки чарта в кластере создается релиз, которым можно управлять.

Ниже перечислены наиболее распространенные сценарии использования инструмента Helm.

- Развертывание популярных приложений в кластере Kubernetes. Возможность устанавливать сторонние компоненты за считаные секунды значительно упрощает процесс *разработки*. Тот же подход можно применять и в кластерах *производственного уровня*. При этом создавать собственные YAML-манифесты для сторонних компонентов нет необходимости.
- *Управление зависимостями* между чартами. На случай, если чарт А требует предварительной установки чарта Б с определенными параметрами, приложение Helm предусматривает особый синтаксис для описания соответствующих зависимостей.
- Публикация собственных приложений в виде Helm-чартов. Это может включать упаковку продукта для конечных пользователей или применение Helm в качестве внутреннего менеджера пакетов и зависимостей, существующих между микросервисами вашего продукта.
- Обеспечение корректного обновления приложений. В Helm этот процесс реализован через специальный механизм обновления релизов.

- Настройка развертывания ПО под конкретные нужды. По сути, Helm-чарты представляют собой универсальные YAML-шаблоны для манифестов объектов Kubernetes, которые можно *параметризовать*. Для этого в Helm используются шаблоны *Go* (https://godoc.org/text/template).

В настоящее время приложение Helm распространяется в виде двоичного клиента (библиотеки) с интерфейсом командной строки, напоминающим `kubectl`. Операции, выполняемые с помощью Helm, не требуют установки дополнительных компонентов в кластер Kubernetes.

Следует отметить, что архитектура Helm изменилась после выхода версии 3.0.0. Ранее это приложение требовало наличия специального сервиса, запускаемого в кластере Kubernetes, под названием Tiller. Это вызывало ряд проблем, в частности связанных с безопасностью, механизмом управления доступом на основе ролей (RBAC) и необходимостью запуска подов с повышенными привилегиями внутри кластера. Подробнее о различиях между нынешней и предыдущими версиями Helm можно узнать в официальной документации: https://helm.sh/docs/faq/#changes-since-helm-2.

Обнаружив в онлайн-руководстве упоминание Tiller, имейте в виду, что оно, скорее всего, ориентировано на более старые версии Helm.

Теперь, когда мы разобрались с основными понятиями Helm, установим это приложение и проверим его работоспособность, развернув простой чарт с Artifact Hub.

Развертывание приложений в Kubernetes с помощью Helm

В этом разделе вам предстоит установить Helm и проверить корректность его работы путем развертывания Helm-чарта. Инструмент Helm распространяется в виде двоичных файлов, доступных для различных платформ (https://github.com/helm/helm/releases). Вы можете использовать эти готовые сборки или установить Helm с помощью менеджера пакетов, следуя приведенным далее инструкциям.

Установка Helm в Linux

Чтобы установить Helm в системе Fedora, убедитесь в корректной настройке и функционировании ее стандартного репозитория:

```
$ sudo dnf repolist | grep fedora
fedora                                   Fedora 39 – x86_64
```

Затем установите Helm:

```
$ sudo dnf install helm
```

После установки можно проверить установленную версию пакета Helm:

```
$ helm version
version.BuildInfo{Version:"v3.11", GitCommit:"", GitTreeState:"",
GoVersion:"go1.21rc3"}
```

Вы также можете установить Helm с помощью специального установочного сценария (https://helm.sh/docs/intro/install/#from-script). Этот сценарий автоматически определит вашу платформу, загрузит последнюю версию Helm и установит ее на вашу машину.

После успешной установки вы можете перейти к подразделу «Развертывание чарта WordPress».

Установка Helm в Windows

Самый простой способ установить Helm в среде Windows — воспользоваться менеджером пакетов Chocolatey. Подробные сведения и инструкцию по его установке можно найти в официальной документации: https://chocolatey.org/install.

После установки Chocolatey выполните следующую команду в оболочке PowerShell или командной строке, чтобы установить Helm:

```
PS C:\Windows\system32> choco install kubernetes-helm
PS C:\Windows\system32> helm version
version.BuildInfo{Version:"v3.15.0-rc.2",
GitCommit:"c4e37b39dbb341cb3f716220df9f9d306d123a58",
GitTreeState:"clean", GoVersion:"go1.22.3"}
```

После успешной установки вы можете перейти к подразделу «Развертывание чарта WordPress».

Установка Helm в macOS

Для установки Helm в системе macOS можно использовать стандартный менеджер пакетов *Homebrew*:

```
$ brew install helm
```

Проверьте корректность установки, запросив версию Helm в командной строке:

```
$ helm version
version.BuildInfo{Version:"v3.16.2", GitCommit:"13654a52f7c70a143b1dd51416d633
e1071faffb", GitTreeState:"dirty", GoVersion:"go1.23.2"}
```

После установки приложения Helm можно развернуть тестовый чарт, чтобы убедиться в его работоспособности.

Установка из двоичных файлов

Установить Helm также можно непосредственно из двоичного файла. Для этого нужно выбрать нужную версию на странице релизов (https://github.com/helm/helm/releases) и загрузить файл, предназначенный для вашей операционной системы. В следующем примере показан процесс установки последней версии Helm из двоичного файла на рабочей станции Fedora.

Выполните следующие команды для загрузки и установки Helm:

```
$ wget https://get.helm.sh/helm-v3.15.1-linux-amd64.tar.gz
$ tar -zxvf helm-v3.15.1-linux-amd64.tar.gz
linux-amd64/
linux-amd64/README.md
linux-amd64/LICENSE
linux-amd64/helm
$ sudo mv linux-amd64/helm /usr/local/bin/helm
$ helm version
version.BuildInfo{Version:"v3.15.1", GitCommit:"e211f2aa62992bd72586b395de5097
9e31231829", GitTreeState:"clean", GoVersion:"go1.22.3"}
```

Теперь мы можем протестировать установленный пакет Helm, развернув чарт WordPress.

Развертывание чарта WordPress

По умолчанию Helm не предусматривает настроенных репозиториев. Один из возможных (правда, уже не рекомендуемых) вариантов — добавить репозиторий `stable`, чтобы получить доступ к наиболее популярным Helm-чартам:

```
$ helm repo add stable https://charts.helm.sh/stable
"stable" has been added to your repositories
```

Добавление случайных репозиториев Helm для дальнейшей установки чартов может представлять серьезную угрозу. Перед установкой необходимо проводить аудит безопасности, чтобы гарантировать попадание в среду Kubernetes только доверенных пакетов.

Учтите, что большинство старых чартов постепенно признаются устаревшими и переносятся в другие репозитории Helm, где поддерживаются изначальными разработчиками. Соответствующие сведения можно получить, если выполнить поиск доступных чартов с помощью команды `helm search repo`:

```
$ helm search repo stable | grep -i deprecated | head
stable/acs-engine-autoscaler  2.2.2  2.1.1       DEPRECATED Scales worker nodes
                                                 within agent pools
stable/aerospike              0.3.5  v4.5.0.5    DEPRECATED A Helm chart for
                                                 Aerospike in Kubern...
stable/airflow                7.13.3  1.10.12    DEPRECATED
... <фрагмент опущен для краткости> ...
```

В настоящее время для поиска чартов рекомендуется использовать команду `helm search hub`, позволяющую просматривать содержимое Artifact Hub непосредственно через интерфейс командной строки:

```
$ helm search hub | head
URL                                             CHART VERSION  APP VERSION
DESCRIPTION
https://artifacthub.io/packages/helm/mya/12factor  24.1.2                Easily
deploy any application that conforms to ...
https://artifacthub.io/packages/helm/gabibbo97/...  0.1.0
fedora-32 389 Directory Server
... <фрагмент опущен для краткости> ...
```

Попробуем найти один из популярнейших Helm-чартов для тестирования нашей установки. В этом примере мы развернем приложение *WordPress* в нашем кластере Kubernetes. Мы выбрали его потому, что оно представляет собой классическое трехуровневое приложение, включающее уровень публичного доступа (объект Service), веб-уровень (WordPress), уровень базы данных (MariaDB). Сначала проверим, какие чарты WordPress доступны в Artifact Hub:

```
$ helm search hub wordpress
URL                                                 CHART VERSION  APP
VERSION  DESCRIPTION
https://artifacthub.io/packages/helm/kube-wordp...      0.1.0          1.1
this is my wordpress package
https://artifacthub.io/packages/helm/wordpress-...      1.0.2          1.0.0
A Helm chart for deploying Wordpress+Mariadb st...
https://artifacthub.io/packages/helm/bitnami-ak...     15.2.13         6.1.0
WordPress is the world's most popular blogging ...
... <фрагмент опущен для краткости> ...
```

Аналогичный поиск можно выполнить и через веб-интерфейс Artifact Hub (рис. 14.1).

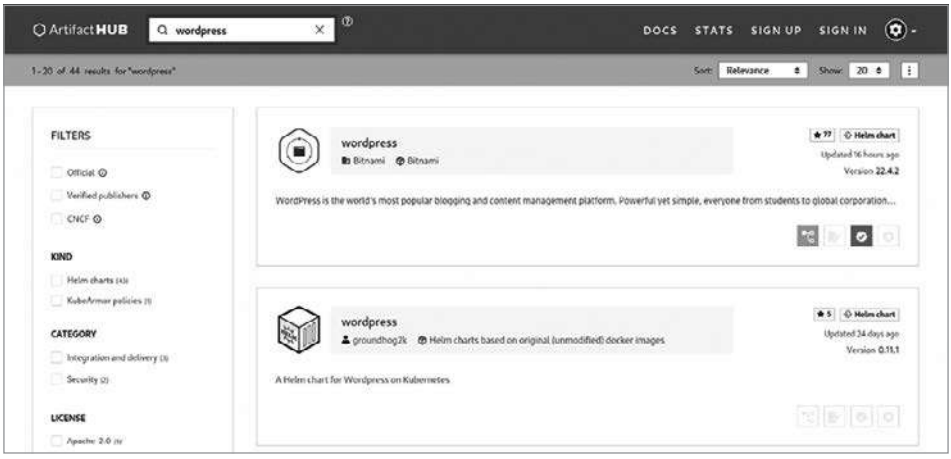

Рис. 14.1. Результаты поиска Helm-чартов WordPress на сайте Artifact Hub

Мы будем использовать Helm-чарт, предоставленный и поддерживаемый компанией *Bitnami*, специализирующейся на распространении ПО с открытым исходным кодом для различных платформ, включая Kubernetes. Если вы откроете страницу с результатами поиска чартов WordPress от Bitnami, то увидите следующее (рис. 14.2).

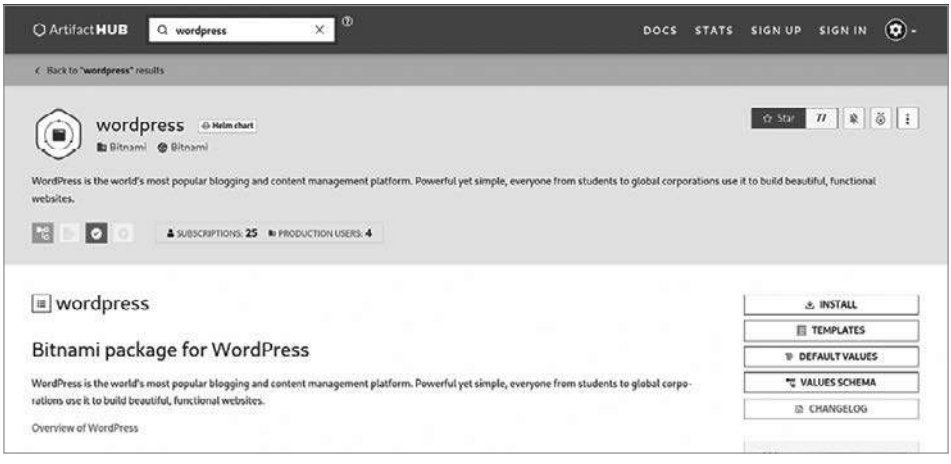

Рис. 14.2. Страница Helm-чарта WordPress от Bitnami на сайте Artifact Hub с инструкциями по установке

На этой странице приведена детальная информация о том, как добавить репозиторий `bitnami` и установить Helm-чарт WordPress. Здесь также можно найти

сведения о доступных параметрах конфигурации, известных ограничениях и способах устранения неполадок. Кроме того, мы можем перейти на домашнюю страницу конкретного чарта, чтобы просмотреть YAML-шаблоны, из которых он состоит (https://github.com/bitnami/charts/tree/master/bitnami/wordpress).

Выполним установку согласно инструкциям на этой странице. Сначала добавим репозиторий `bitnami` в локальную установку Helm:

```
$ helm repo add bitnami https://charts.bitnami.com/bitnami
"bitnami" has been added to your repositories
```

В соответствии с лучшими практиками создадим отдельное пространство имен `wordpress` для изоляции установки:

```
$ kubectl create ns wordpress
namespace/wordpress created
```

После добавления репозитория можно переходить к установке чарта `bitnami/wordpress`, однако перед этим нужно подготовить параметры развертывания. На странице Helm-чарта на сайте Artifact Hub (https://artifacthub.io/packages/helm/bitnami/wordpress) перечислены параметры, позволяющие настроить установку WordPress. Если их не указать, приложение Helm применит значения по умолчанию. Изменим некоторые из параметров, чтобы продемонстрировать процесс настройки.

Отдельные параметры можно передавать с помощью аргумента `--set`:

```
$ helm install wp-demo bitnami/wordpress -n wordpress --set
wordpressUsername=wpadmin
```

Для задания нескольких параметров можно использовать несколько аргументов `--set`, однако это предпочтительнее делать с помощью файлов с переменными.

Создадим файл `wp-values.yaml` для хранения переменных и их значений:

```
# wp-values.yaml
wordpressUsername: wpadmin
wordpressPassword: wppassword
wordpressEmail: admin@example.com
wordpressFirstName: WP
wordpressLastName: Admin
service:
  type: NodePort
volumePermissions:
  enabled: true
```

Как видите, в этот файл передаются некоторые параметры конфигурации WordPress для Helm-чарта. Обратите внимание на то, что мы изменяем тип

сервиса WordPress на `NodePort`, поскольку используем кластер `minikube`. Если вы работаете с другой средой Kubernetes (например, облачной), можете оставить значение по умолчанию — `LoadBalancer`.

После добавления репозитория Helm, создания отдельного пространства имен и определения параметров конфигурации в файле с переменными можно приступать к развертыванию приложения WordPress с помощью Helm. В качестве имени данного релиза мы зададим `wp-demo`:

```
$ helm install wp-demo bitnami/wordpress -n wordpress --values wp-values.yaml
NAME: wp-demo
LAST DEPLOYED: Tue Jun 4 21:27:49 2024
NAMESPACE: wordpress
STATUS: deployed
REVISION: 1
TEST SUITE: None
NOTES:
CHART NAME: wordpress
CHART VERSION: 22.4.2
APP VERSION: 6.5.3
... <фрагмент опущен для краткости> ...
```

После запуска приложение Helm отобразит основную информацию о релизе, как показано ниже. Через некоторое время вы также увидите подробные сведения о развертывании, включая имя сервиса, способ получения доступа к сайту WordPress и другую служебную информацию:

```
...
** Please be patient while the chart is being deployed **

Your WordPress site can be accessed through the following DNS name from within
your cluster:

    wp-demo-wordpress.wordpress.svc.cluster.local (port 80)

To access your WordPress site from outside the cluster follow the steps below:

Get the WordPress URL by running these commands:

    export NODE_PORT=$(kubectl get --namespace wordpress -o jsonpath="{.spec.
ports[0].nodePort}" services wp-demo-wordpress)
    export NODE_IP=$(kubectl get nodes --namespace wordpress -o jsonpath="{.
items[0].status.addresses[0].address}")
    echo "WordPress URL: http://$NODE_IP:$NODE_PORT/"
... <фрагмент опущен для краткости> ...
```

В этом состоит одно из ключевых преимуществ Helm: выполнив всего одну команду `helm install`, вы получаете подробную инструкцию о том, как использовать

развернутый компонент в *своем* кластере. Тем временем экземпляр WordPress устанавливается без какого-либо участия с вашей стороны.

Перед установкой рекомендуется проверить, какие именно YAML-манифесты объектов Kubernetes были сгенерированы Helm. Это можно сделать, используя команду helm install с дополнительными флагами: helm install wp-demo bitnami/wordpress --dry-run --debug. В результате будут отображены все YAML-манифесты без их применения к кластеру.

При необходимости можно указать конкретную версию Helm-чарта с помощью аргумента `--version`:

```
$ helm install my-wordpress bitnami/wordpress --version 22.4.2
```

Теперь выполним инструкцию, полученную при установке Helm-чарта.

1. Дождитесь завершения инициализации базы данных и развертывания подов, а затем проверьте их статус:

```
$ kubectl get po -n wordpress
NAME                                READY   STATUS    RESTARTS        AGE
wp-demo-mariadb-0                   1/1     Running   9 (5m57s ago)   31m
wp-demo-wordpress-5d98c44785-9xd6h  1/1     Running   0               31m
```

2. Обратите внимание, что база данных развернута в виде объекта StatefulSet:

```
$ kubectl get statefulsets.apps -n wordpress
NAME             READY   AGE
wp-demo-mariadb  1/1     99s
```

3. Дождитесь назначения порта сервису `wp-demo` (типа `NodePort`):

```
$ kubectl get svc -n wordpress
NAME               TYPE        CLUSTER-IP     EXTERNAL-IP   PORT(S)
AGE
wp-demo-mariadb    ClusterIP   10.100.118.79  <none>        3306/TCP
2m39s
wp-demo-wordpress  NodePort    10.100.149.20  <none>        80:30509/
TCP,443:32447/TCP  2m39s
```

   В нашем случае будет использоваться порт `80:31979/TCP`.

4. Поскольку мы работаем с кластером `minikube`, определим IP-адрес и номера портов (при использовании сервиса типа `LoadBalancer` можно сразу перейти по IP-адресу, чтобы открыть сайт WordPress):

```
$ minikube service --url wp-demo-wordpress -n wordpress
http://192.168.59.150:30509
http://192.168.59.150:32447
```

5. Откройте браузер и перейдите по адресу `http://192.168.59.150:30509` (второй порт используется для получения доступа по протоколу HTTPS) (рис. 14.3).

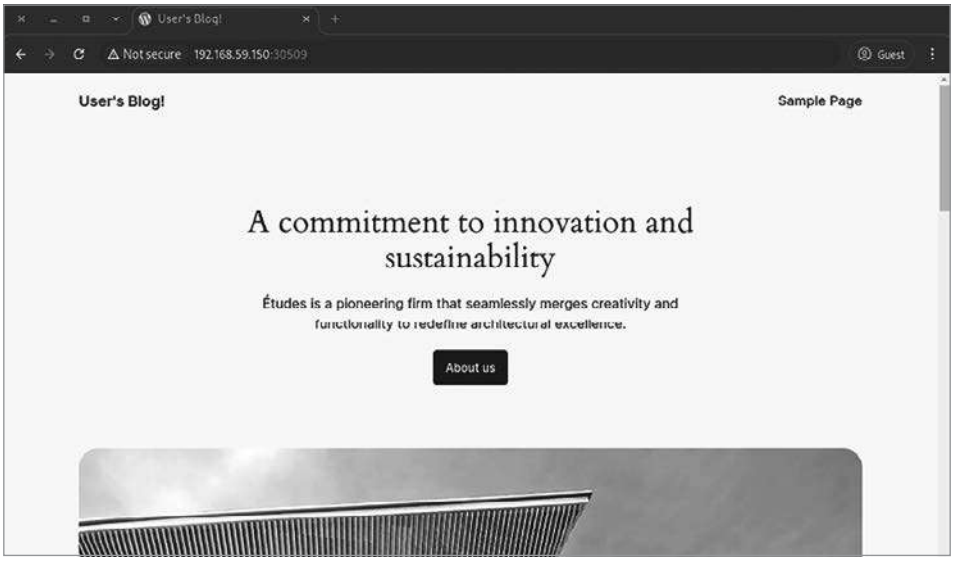

Рис. 14.3. Главная страница приложения WordPress, развернутого в Kubernetes с помощью Helm-чарта

6. Теперь можно войти на панель управления WordPress по адресу `http://192.168.59.150:30509/wp-admin`. Если вы не задали параметры WordPress, включая пароль, при установке, вам необходимо найти значения, которые инструмент Helm использовал по умолчанию. Например, чтобы получить пароль для входа в WordPress, можно проверить содержимое соответствующего секрета. Используйте следующие команды, чтобы извлечь учетные данные, хранящиеся в объекте Secret `wp-demo-wordpress`, созданном в рамках развертывания Helm-чарта:

```
$ kubectl get secret --namespace wordpress wp-demo-wordpress -o jsonpath="{.
data.wordpress-password}" | base64 --decode
wppassword
```

7. Используйте полученные учетные данные, чтобы войти в систему WordPress от имени администратора (рис. 14.4).

Итак, ваше приложение WordPress готово к работе! Теперь вы можете изучить развернутые поды, сервисы, а также объекты Deployment и StatefulSet, чтобы лучше разобраться в структуре чарта и взаимодействии его компонентов.

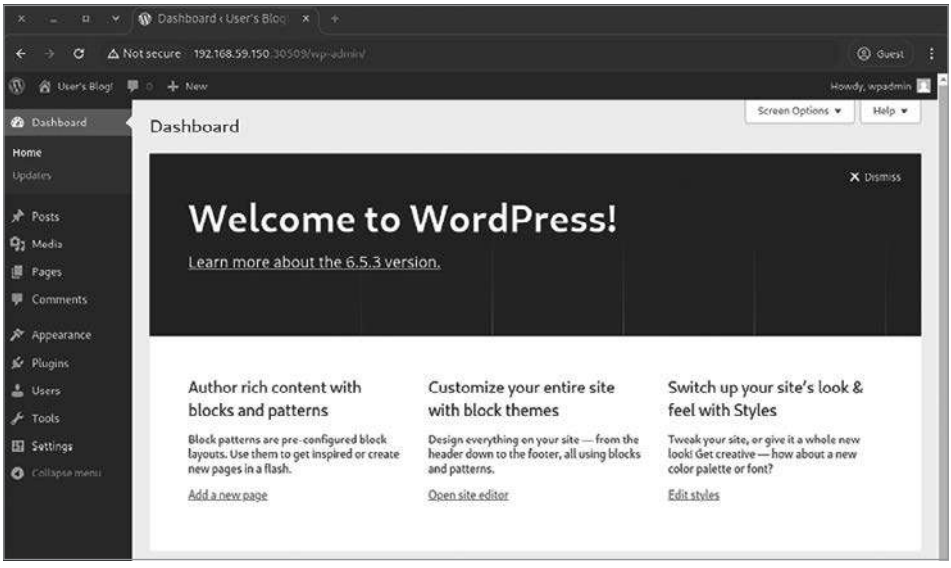

Рис. 14.4. Панель управления приложения WordPress, развернутого в Kubernetes с помощью Helm-чарта

Интерфейс командной строки Helm поддерживает функцию автодополнения для различных оболочек. Подробнее о ней можно узнать, выполнив команду helm completion.

Для просмотра списка релизов Helm, развернутых в вашем кластере Kubernetes, используйте такую команду:

```
$ helm list -n wordpress
NAME     NAMESPACE  REVISION  UPDATED                                  STATUS
CHART             APP VERSION
wp-demo  wordpress  1         2024-06-04 22:20:32.556683096 +0800 +08  deployed
wordpress-22.4.2   6.5.3
```

Далее вы узнаете, как удалить развернутый релиз Helm.

Удаление релиза Helm

Как уже было сказано, в процессе развертывания Helm-чарт создает несколько ресурсов, включая объекты Deployment, PVC, Service, Secret и др. Удалять их вручную по одному было бы крайне неудобно. К счастью, Helm предусматривает команду `helm uninstall`, позволяющую легко удалить развертывание. Когда

вы захотите очистить кластер, выполните следующую команду для удаления релиза Helm:

```
$ helm uninstall wp-demo -n wordpress
release "wp-demo" uninstalled
```

Она удалит все объекты Kubernetes, созданные данным релизом. Однако учтите, что постоянные тома (PersistentVolume) и заявки на их выделение (PersistentVolumeClaim), созданные Helm-чартом, не удаляются автоматически — их необходимо удалить вручную. Далее мы рассмотрим структуру Helm-чарта.

Структура Helm-чарта

В качестве примера мы рассмотрим Helm-чарт WordPress от Bitnami (https://github.com/bitnami/charts/tree/master/bitnami/wordpress), который использовали ранее для выполнения тестового развертывания в кластере. Helm-чарты представляют собой каталоги со строго определенной структурой, которые могут храниться как в локальной файловой системе, так и в репозиториях Git. Имя каталога совпадает с именем чарта — в нашем случае это `wordpress`. Типовая структура чарта включает следующие файлы и каталоги.

- `Chart.yaml` — YAML-файл с такими метаданными чарта, как версия, ключевые слова, а также ссылки на зависимые чарты, которые необходимо установить.
- `LICENSE` — необязательный текстовый файл с информацией о лицензии.
- `README.md` — README-файл для конечного пользователя, отображаемый на сайте Artifact Hub.
- `values.yaml` — файл со значениями по умолчанию, используемыми в качестве параметров YAML-шаблонов. Эти значения можно переопределять — либо по отдельности через интерфейс командной строки, либо указав отдельный YAML-файл (например, `--values wp-values.yaml`, как мы делали ранее).
- `values.schema.json` — необязательная JSON-схема, которой должен соответствовать файл `values.yaml`.
- `charts/` — необязательный каталог с зависимыми чартами.
- `crds/` — необязательный каталог с определениями пользовательских ресурсов Kubernetes.
- `templates/` — основной каталог, содержащий YAML-*шаблоны*, на основе которых генерируются итоговые манифесты Kubernetes. Эти шаблоны объединяются с переданными *значениями*, после чего получившиеся манифесты применяются к кластеру.
- `templates/NOTES.txt` — необязательный файл с краткими инструкциями по использованию чарта.

Если открыть файл `Chart.yaml`, содержащийся в чарте WordPress, можно увидеть, что он зависит от чарта *MariaDB* компании Bitnami — при условии, что для параметра `mariadb.enabled` задано значение `true`:

```
...

appVersion: 6.5.3
dependencies:
  - condition: memcached.enabled
    name: memcached
    repository: oci://registry-1.docker.io/bitnamicharts
    version: 7.x.x
  - condition: mariadb.enabled
    name: mariadb
    repository: oci://registry-1.docker.io/bitnamicharts
    version: 18.x.x
...
```

Если теперь обратиться к (довольно объемному) файлу `values.yaml` со значениями по умолчанию, можно заметить, что база данных MariaDB включена изначально:

```
...
## Настройки чарта MariaDB
## Справка: https://github.com/bitnami/charts/blob/main/bitnami/
## mariadb/values.yaml
##
mariadb:
  ## @param mariadb.enabled — развернуть сервер MariaDB,
  ## чтобы удовлетворить требования приложения к базе данных
  ## Если нужно использовать внешнюю базу данных, присвойте
  ## этому параметру значение false и настройте
  ## параметры `externalDatabase.*`
  ##
  enabled: true
...
```

Наконец, посмотрим, как выглядит один из YAML-шаблонов. Файл `deployment.yaml` (https://github.com/bitnami/charts/blob/master/bitnami/wordpress/templates/deployment.yaml) представляет собой шаблон объекта Deployment, предназначенный для подов с контейнерами WordPress. Обратите внимание, что число реплик (`replicas`) задается на основе переданных пользователем значений:

```
kind: Deployment
...
spec:
...
  replicas: {{ .Values.replicaCount }}
...
```

При установке этот параметр будет заменен на конкретное значение `replicaCount`, по умолчанию равное `1` и определенное в файле `values.yaml`. Подробную информацию об использовании шаблонов Go можно найти по адресу https://pkg.go.dev/text/template. Кроме того, вы можете изучить принципы работы шаблонов на примерах, проанализировав существующие Helm-чарты, так как большинство из них использует схожие паттерны обработки переданных значений.

Подробнее о структуре Helm-чартов можно узнать по адресу https://helm.sh/docs/topics/charts/.

В большинстве случаев при установке чарта вам потребуется переопределить некоторые значения по умолчанию из файла `values.yaml`, как было показано ранее.

Теперь, когда вы познакомились с внутренним устройством Helm-чартов, развернем с их помощью веб-интерфейс Kubernetes.

Установка веб-интерфейса Kubernetes с использованием Helm-чартов

Kubernetes Dashboard — это официальный веб-интерфейс, предназначенный для мониторинга состояния кластера. Helm-чарт этого компонента поддерживается сообществом Kubernetes (https://artifacthub.io/packages/helm/k8s-dashboard/kubernetes-dashboard). Мы установим этот веб-интерфейс с параметрами по умолчанию, поскольку на данном этапе в его дополнительной настройке нет необходимости.

В кластерах minikube веб-интерфейс Dashboard можно активизировать и открыть с помощью одной команды minikube dashboard. Однако в данном случае наша цель — научиться развертывать Dashboard в кластере любого типа.

Для начала добавьте репозиторий `kubernetes-dashboard` в Helm:

```
$ helm repo add kubernetes-dashboard https://kubernetes-retired.github.io/
dashboard/
"kubernetes-dashboard" has been added to your repositories
```

Затем установите Helm-чарт в кластер в виде релиза `kubernetes-dashboard`:

```
$ helm upgrade --install kubernetes-dashboard kubernetes-dashboard/
kubernetes-dashboard \
  --create-namespace --namespace kubernetes-dashboard
```

Дождитесь завершения установки и изучите вывод команд. Обратите внимание на показанное ниже сообщение, так как в нем содержится информация, которая понадобится нам позднее для получения доступа к веб-интерфейсу:

```
...
Congratulations! You have just installed Kubernetes Dashboard in your cluster.

To access the Dashboard, run the following:

$ kubectl -n kubernetes-dashboard port-forward svc/kubernetes-dashboard-kong-
proxy 8443:443
...
```

Убедитесь в том, что поды находятся в состоянии `Running`:

```
$ kubectl get pod -n kubernetes-dashboard
NAME                                                READY STATUS  RESTARTS AGE
kubernetes-dashboard-api-86c68c7896-7jxwz           1/1   Running 0        2m53s
kubernetes-dashboard-auth-59784dd8b-vsr99           1/1   Running 0        2m53s
kubernetes-dashboard-kong-7696bb8c88-6q7zs          1/1   Running 0        2m53s
kubernetes-dashboard-metrics-scraper-5485b64c47-9d69q 1/1   Running 0        2m53s
kubernetes-dashboard-web-84f8d6fff4-nxt5f           1/1   Running 0        2m53s
```

Остальные поды, развернутые чартом, можно проигнорировать. В следующем подразделе будет показано, как получить доступ к веб-интерфейсу Kubernetes.

Безопасный доступ к Kubernetes Dashboard

По умолчанию Kubernetes Dashboard отдает приоритет безопасности, используя минимально необходимую конфигурацию RBAC. Это помогает защитить данные кластера. В настоящее время для входа в интерфейс требуется токен на предъявителя (bearer-токен).

Следуя приведенной ниже инструкции, вы, скорее всего, создадите пользователя с административными привилегиями. Используйте ее только в учебных целях, а в рабочих средах обязательно реализуйте полноценные механизмы RBAC.

Выполните следующие шаги, чтобы создать токен для получения доступа к веб-интерфейсу Kubernetes Dashboard.

1. Подготовьте YAML-манифест с определением объекта ServiceAccount:

```
# dashboard-sa.yaml
apiVersion: v1
kind: ServiceAccount
metadata:
  name: admin-user
  namespace: kubernetes-dashboard
```

2. Создайте объект ServiceAccount, применив YAML-манифест:

```
$ kubectl apply -f dashboard-sa.yaml
serviceaccount/admin-user created
```

3. Подготовьте YAML-файл с описанием объекта ClusterRoleBinding, который позволит пользователю получить доступ к веб-интерфейсу:

```
# dashboard-rbac.yml
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: admin-user
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
  name: cluster-admin
subjects:
  - kind: ServiceAccount
    name: admin-user
    namespace: kubernetes-dashboard
```

4. Создайте объект ClusterRoleBinding, применив YAML-манифест:

```
$ kubectl apply -f dashboard-rbac.yml
clusterrolebinding.rbac.authorization.k8s.io/admin-user created
```

5. Создайте токен:

```
$ kubectl -n kubernetes-dashboard create token admin-user
```

Скопируйте сгенерированный токен, он понадобится вам в следующем подразделе для авторизации в веб-интерфейсе.

Получение доступа к веб-интерфейсу Dashboard

Получить доступ к веб-интерфейсу Kubernetes Dashboard можно несколькими способами. В этом разделе мы рассмотрим стандартный способ, предполагающий, что вы не изменяли конфигурацию чарта при установке. Если вы вносили правки, отдельные шаги могут отличаться.

Запустите показанную ниже команду (скопированную из вывода `helm install`), чтобы получить доступ к Dashboard. Эта команда активизирует режим проброса портов (`port-forward`) — не завершайте ее выполнение:

```
$ kubectl -n kubernetes-dashboard port-forward svc/kubernetes-dashboard-kong-
proxy 8443:443
Forwarding from 127.0.0.1:8443 -> 8443
Forwarding from [::1]:8443 -> 8443
...
```

Перейдите по адресу https://localhost:8443/ в браузере. Вы можете проигнорировать предупреждение о сертификате SSL, так как веб-интерфейс Dashboard использует самоподписанные SSL-сертификаты. Для входа в систему укажите токен, сгенерированный в предыдущем подразделе, как показано на рис. 14.5.

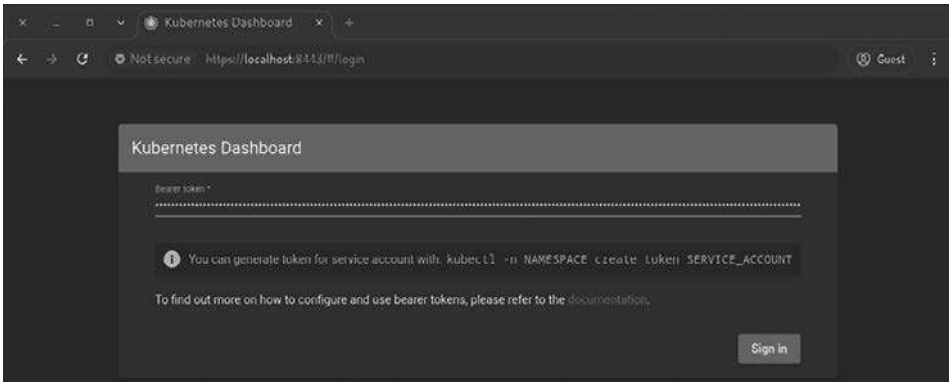

Рис. 14.5. Helm-чарт Kubernetes Dashboard — страница авторизации

Теперь у вас есть доступ к веб-интерфейсу и вы можете ознакомиться с его функционалом (рис. 14.6).

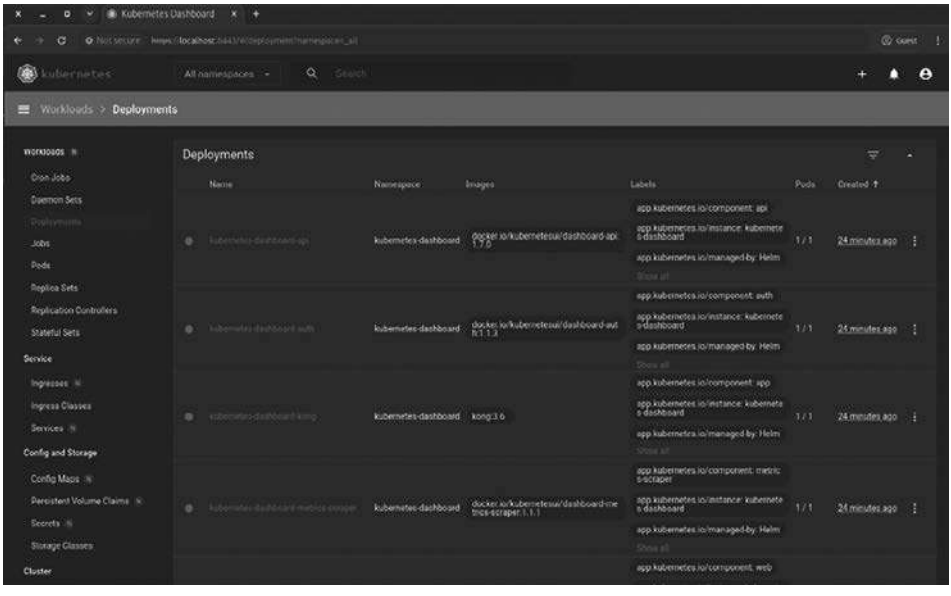

Рис. 14.6. Helm-чарт Kubernetes Dashboard — страница развертываний

Учтите, что используемый токен принадлежит пользователю с ролью `cluster-admin`, позволяющей выполнять любые операции, включая удаление ресурсов, поэтому будьте осторожны.

Итак, вы успешно развернули веб-интерфейс Kubernetes Dashboard с помощью Helm-чартов и получили к нему доступ с использованием токена. Теперь вы можете установить дополнительные компоненты.

Установка других популярных решений с помощью Helm-чартов

Чтобы закрепить навыки работы с Helm-чартами, вы можете установить ряд других программных решений для вашего кластера Kubernetes. Они могут оказаться полезными в процессе разработки или послужить готовыми строительными блоками для облачных приложений.

Elasticsearch и Kibana

Elasticsearch — это популярная система полнотекстового поиска, часто используемая для индексирования и анализа логов. Инструмент Kibana, входящий в экосистему Elasticsearch, представляет собой веб-интерфейс для визуализации данных, хранящихся в базе Elasticsearch. Для установки этого стека понадобятся два чарта, поддерживаемые разработчиками Elasticsearch:

- *Helm-чарт Elasticsearch* (https://artifacthub.io/packages/helm/elastic/elasticsearch);
- *Helm-чарт Kibana* (https://artifacthub.io/packages/helm/elastic/kibana).

Prometheus и Grafana

Prometheus — это популярная система мониторинга с базой данных временных рядов, а Grafana используется как интерфейс для их визуализации. Как и в случае со стеком Elastic, для установки Prometheus и Grafana понадобятся два Helm-чарта:

- *Prometheus* (https://artifacthub.io/packages/helm/prometheus-community/prometheus);
- *Grafana* (https://artifacthub.io/packages/helm/grafana/grafana).

Мы опустили пошаговые инструкции, поскольку процесс развертывания Helm-чартов в кластере Kubernetes был подробно рассмотрен ранее. Вы можете установить эти чарты и изучить их функциональные возможности самостоятельно.

В следующем разделе мы рассмотрим ключевые вопросы безопасности, связанные с Helm-чартами.

Вопросы безопасности, связанные с Helm-чартами

Helm-чарты значительно упрощают процесс развертывания приложений в Kubernetes, но могут создавать потенциальные уязвимости, которые необходимо контролировать. Ниже перечислены наиболее важные аспекты, на которые стоит обратить внимание.

- *Проверка источника*. Всегда проверяйте происхождение Helm-чарта перед установкой и никогда не используйте чарты из недоверенных или малоизвестных репозиториев. Внутри пакета могут содержаться вредоносные или небезопасные приложения. Используйте только официальные чарты или репозитории, поддерживаемые сообществом.
- *Регулярные аудиты*. Периодически проводите аудиты безопасности чартов и их зависимостей. Это позволяет выявлять известные уязвимости и обеспечивать соответствие развернутых приложений внутренним стандартам безопасности вашей организации. Для анализа уязвимостей можно использовать инструменты вроде Trivy или Anchore.
- *Конфигурация чарта*. Будьте осторожны с параметрами, заданными в Helm-чартах по умолчанию. Во многих случаях они могут не соответствовать требованиям вашей рабочей среды. Рекомендуется проверять и при необходимости изменять настройки в соответствии с политикой обеспечения безопасности и лучшими практиками, принятыми в вашей организации.
- *Управление доступом на основе ролей (RBAC)*. Реализуйте механизм RBAC для ограничения круга пользователей, способных развертывать и управлять Helm-чартами в вашем кластере. Это уменьшает риск несанкционированных изменений и гарантирует то, что только доверенные специалисты смогут устанавливать критически важные приложения.
- *Управление зависимостями*. Следите за списком зависимостей внутри Helm-чартов и регулярно обновляйте их, чтобы своевременно устранять уязвимости и исправлять проблемы безопасности.
- *Изоляция с помощью пространств имен*. Рассмотрите возможность установки каждого чарта в отдельное пространство имен. Это позволяет защитить приложения и их ресурсы, так как в случае инцидента его последствия будут локализованы в пределах одного пространства имен.

Придерживаясь этих рекомендаций, вы сможете значительно повысить уровень безопасности Kubernetes-развертываний и минимизировать влияние потенциальных уязвимостей на ваши приложения и данные.

На этом мы завершаем первую половину главы, посвященной работе с Helm-чартами. Теперь вы знаете, что инструмент Helm предоставляет удобный способ развертывания и управления сложными приложениями в Kubernetes.

Однако по мере роста масштабов использования Kubernetes возрастает и сложность рабочей среды, особенно в части управления приложениями. Хотя Helm и аналогичные инструменты упрощают процесс развертывания, они не решают задачи, связанные с эксплуатацией приложений с сохранением состояния, такие как масштабирование, управление конфигурацией и автоматическое восстановление после сбоев. Чтобы автоматизировать выполнение таких задач на всех этапах жизненного цикла приложения, обеспечив его стабильную работу и высокую производительность, необходимы решения, которые задействуют знания об архитектуре приложения и лучшие эксплуатационные практики. Такие решения минимизируют необходимость ручного вмешательства и связанные с ним ошибки, позволяя командам сосредоточиться на улучшении приложения, а не на обслуживании инфраструктуры. Для удовлетворения этих потребностей были разработаны операторы Kubernetes.

В следующем разделе вы узнаете, что представляют собой операторы Kubernetes и как с их помощью можно устанавливать сложные развертывания.

Знакомство с операторами Kubernetes

Ранее мы обсудили различия между объектами Deployment и StatefulSet (последний управляет приложениями с сохранением состояния, которым требуется постоянное хранилище данных). Мы также рассмотрели выполняемые вручную и автоматизированные операции, необходимые для корректной работы StatefulSet, такие как синхронизация данных между репликами подов и инициализационные процедуры.

Однако ручное вмешательство противоречит самой философии Kubernetes, основанной на принципах автоматизации и самовосстановления. Именно здесь на сцену выходят операторы Kubernetes.

От ручного управления к программному

Представьте, что функции людей, вручную управляющих жизненным циклом приложений, берут на себя программные компоненты. Операторы Kubernetes (Kubernetes operators) — это программные расширения, которые автоматизируют сложные задачи управления приложениями, в частности приложениями с сохранением состояния. Они используют встроенные программные механизмы и знания о работе конкретных приложений для их автоматического и эффективного обслуживания, не требуя вмешательства человека.

На рис. 14.7 показана общая схема взаимодействия компонентов в кластере Kubernetes, в котором применяются операторы.

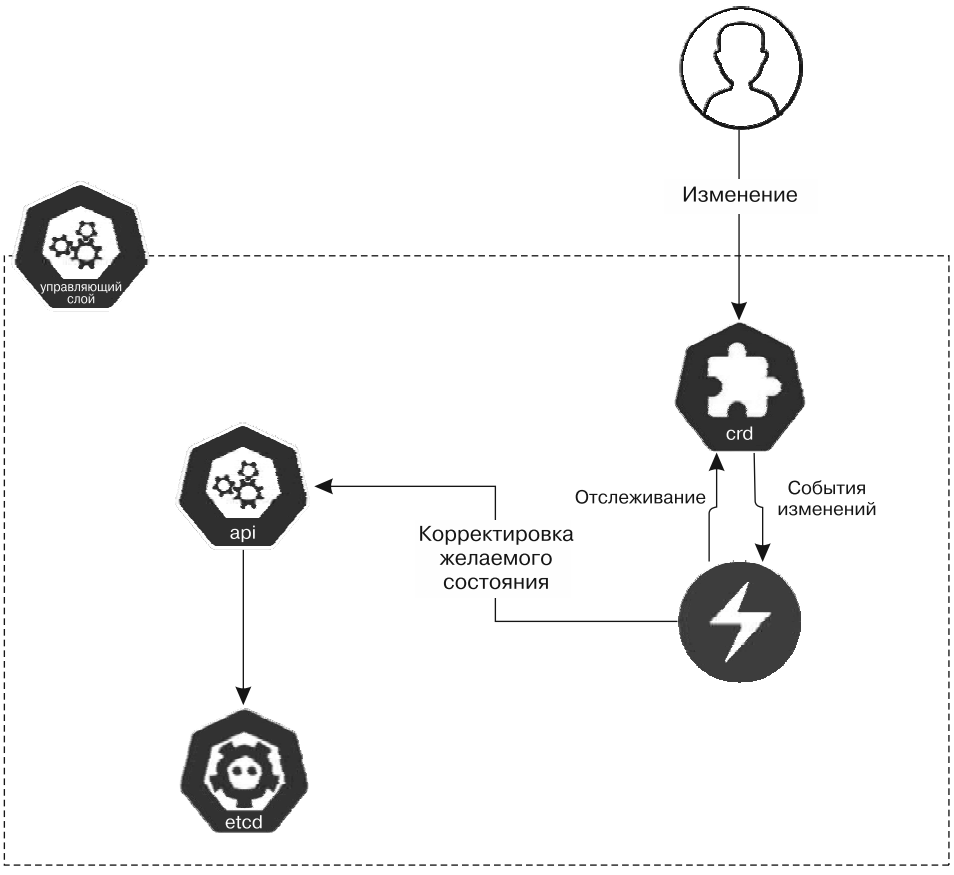

Рис. 14.7. Принцип управления ресурсами Kubernetes с помощью операторов

В следующих подразделах мы рассмотрим преимущества и ключевые особенности операторов Kubernetes.

Helm-чарты и операторы Kubernetes

Хотя Helm-чарты и операторы используются для управления приложениями в Kubernetes, между ними существуют принципиальные различия. Инструмент Helm позволяет устанавливать приложения гораздо быстрее, поскольку представляет собой эффективный менеджер пакетов, использующий готовые чарты, значительно упрощающие процесс. Этот инструмент лучше всего подходит для ситуаций, где важна скорость, что делает его идеальным инструментом для разовых установок или небольших приложений. Однако Helm в большей степени ориентирован на этап развертывания, а не на решение операционных задач на протяжении всего жизненного цикла приложения.

В свою очередь, операторы представляют собой программные компоненты, отвечающие за весь жизненный цикл сложных и stateful-приложений. С помощью определений пользовательских ресурсов (Custom Resource Definitions, CRD) они кодируют знания о работе приложения, что позволяет автоматизировать масштабирование, управление конфигурациями, обновления, а также реализовать механизмы самовосстановления. Операторы постоянно следят за состоянием приложения и предпринимают корректирующие действия, чтобы вернуть его к желаемой конфигурации. Это делает их особенно ценными для приложений, требующих интенсивного управления и наблюдения. Если вкратце, то Helm-чарты упрощают развертывание, а операторы расширяют возможности Kubernetes для промышленной эксплуатации ПО, позволяя командам автоматизировать и упростить управление приложениями в рабочей среде.

В следующем подразделе мы рассмотрим ключевые особенности операторов Kubernetes и сравним их с Helm-чартами.

Роль операторов в развертывании приложений

Операторы позволяют управлять и поддерживать приложение на всем протяжении его жизненного цикла, выполняя широкий спектр операций.

- *Управление жизненным циклом*. Операторы управляют всем жизненным циклом приложения или целого стека приложений, не ограничиваясь этапом развертывания. Они автоматизируют ключевые эксплуатационные задачи, включая обновление, масштабирование и восстановление после сбоев, обеспечивая стабильную работу и поддержание требуемого уровня производительности. В то время как Helm-чарты в основном ограничиваются процессом установки, операторы постоянно отслеживают текущее состояние приложения и при необходимости автоматически приводят его в соответствие с желаемой конфигурацией. Это позволяет выполнять обновления, изменять настройки и контролировать состояние приложения без ручного вмешательства. В случае сложных приложений с сохранением состояния, требующих интенсивного управления жизненным циклом, операторы обладают значительными преимуществами перед Helm-чартами и являются предпочтительным решением.
- *Оркестрация ресурсов*. Операторы создают такие ресурсы Kubernetes, как *ConfigMap*, *Secret*, *PVC*, необходимые для корректной работы приложения.
- *Автоматизация развертывания*. Операторы могут развертывать приложения на основе параметров, предоставленных пользователем, или значений по умолчанию, тем самым упрощая и ускоряя процесс установки.
- *Управление базами данных*. Например, при развертывании кластера PostgreSQL операторы могут использовать объект StatefulSet для синхронизации данных между репликами.

- *Самовосстановление*. Операторы способны обнаруживать сбои в работе приложения и автоматически запускать механизмы восстановления или аварийного переключения, обеспечивая непрерывность работы сервиса.

Повторное использование автоматизированных решений

Операторы способствуют повторному использованию автоматизированных решений. Один и тот же оператор может применяться в разных проектах или в нескольких кластерах Kubernetes, обеспечивая согласованное и эффективное управление приложениями по всей инфраструктуре.

Поддержание состояния приложения с помощью операторов

Операторы функционируют по тому же принципу, что и сама система Kubernetes, используя цикл управления для мониторинга и приведения приложения к нужной конфигурации. Этот цикл непрерывно отслеживает желаемое состояние, заданное в определении пользовательского ресурса (CRD), и сравнивает его с текущим состоянием приложения в кластере. При обнаружении расхождений оператор выполняет корректирующие действия, например масштабирует приложение, обновляет конфигурацию или перезапускает поды. Непрерывная работа управляющего цикла гарантирует соответствие приложения заданному состоянию, обеспечивая его самовосстановление и автоматизированное управление.

Определения пользовательских ресурсов — основа операторов

В основе операторов Kubernetes лежат так называемые определения пользовательских ресурсов (CRD, Custom Resource Definition). CRD представляют собой расширения API Kubernetes, позволяющие создавать собственные типы ресурсов, специфичные для конкретного приложения или адаптированные под его потребности. Их можно рассматривать как чертежи, описывающие желаемую конфигурацию приложения в экосистеме Kubernetes.

Ресурсы, созданные с использованием CRD, служат строительными блоками приложения в рамках кластера Kubernetes. С помощью CRD можно определить, например, следующие параметры приложения:

- желаемое количество реплик приложения (подов);
- запросы и лимиты на использование ресурсов памяти и ЦП;
- параметры хранения постоянных данных;
- переменные окружения и конфигурационные параметры.

Преимущества использования CRD

Применение определений пользовательских ресурсов дает несколько ключевых преимуществ при развертывании приложений.

- *Декларативное управление*. CRD позволяют декларативно описать желаемое состояние приложения, не прибегая к ручной настройке отдельных объектов, таких как Deployment или Service. Затем оператор автоматически преобразует это описание в набор ресурсов, работающих внутри кластера.
- *Конфигурация, специфичная для приложения*. CRD позволяют описать уникальные требования вашего приложения. Вы можете определять собственные поля, которые отражают его бизнес-логику или параметры настройки.
- *Упрощенное управление*. CRD обеспечивают единый центр управления конфигурацией приложения, что делает процесс настройки более прозрачным и удобным по сравнению с использованием множества отдельных конфигураций различных объектов Kubernetes.

Механизм распространения операторов

Как правило, операторы разрабатываются и распространяются поставщиками приложений, обладающими опытом в развертывании конкретных технологических стеков. Наряду с этим активно развивается сообщество, участники которого создают, публикуют и поддерживают широкий спектр операторов.

Централизованной площадкой для их распространения служит репозиторий OperatorHub (https://operatorhub.io/), позволяющий находить и устанавливать операторы для различных приложений и задач (рис. 14.8).

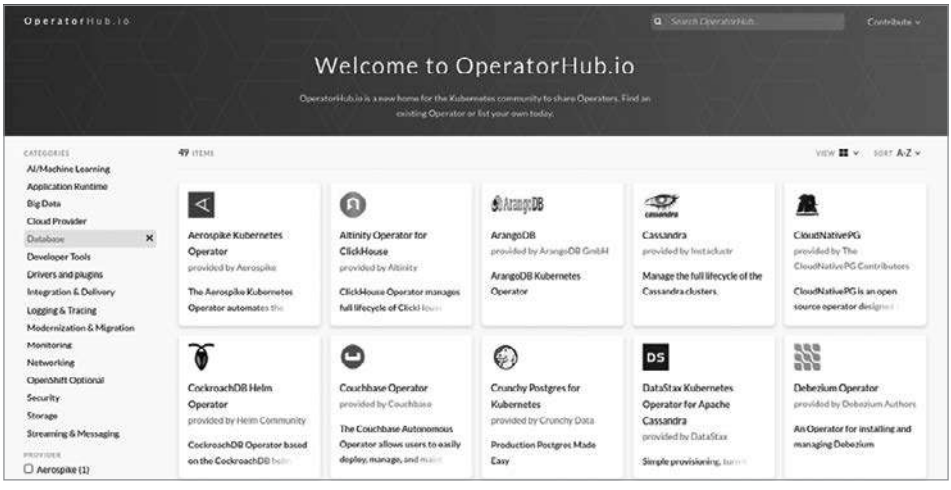

Рис. 14.8. Главная страница сайта OperatorHub

Создание собственного оператора

Operator Framework предоставляет мощный инструментарий с открытым исходным кодом под названием *Operator SDK*, позволяющий легко разрабатывать и создавать собственные операторы для получения полного контроля над процессом управления приложениями в Kubernetes. Operator SDK включает в себя готовые компоненты и функциональные модули, решающие такие типовые задачи, как организация взаимодействия с API Kubernetes, управление пользовательскими ресурсами и реализация цикла управления (control loop). Благодаря этому разработчик может сосредоточиться на уникальной логике и конфигурационных потребностях конкретного приложения.

Создание собственного оператора имеет ряд преимуществ. Во-первых, вы получаете полный контроль над всеми аспектами управления приложением. Оператор можно адаптировать под особенности ПО, обеспечив оптимальное управление его конфигурацией, стратегиями развертывания и масштабированием. Во-вторых, операторы автоматизируют повторяющиеся операции, связанные с жизненным циклом приложения, что существенно повышает эффективность работы. Наконец, пользовательские операторы можно использовать повторно в рамках разных развертываний, экономя время и снижая трудозатраты при сопровождении инфраструктуры Kubernetes.

Подробнее об инструментарии Operator SDK можно узнать в официальной документации: https://sdk.operatorframework.io/.

Прежде чем переходить к практической работе с операторами, рассмотрим компонент Operator Lifecycle Manager (OLM).

Компонент Operator Lifecycle Manager (OLM)

Operator Lifecycle Manager (OLM) — это компонент Kubernetes, который упрощает развертывание и управление приложениями, упакованными в виде операторов. Он использует декларативный подход, при котором все ресурсы описываются в YAML-файлах. Это устраняет необходимость в сложных развертываниях, предполагающих установку множества файлов в строго определенном порядке. Кроме того, OLM обеспечивает автоматическую установку и обновление операторов, поддерживая их в актуальном состоянии. Этот компонент также предоставляет механизм обнаружения пакетов под названием Catalog Sources, который позволяет задействовать операторы из таких источников, как OperatorHub, и других репозиториев. Использование OLM дает несколько преимуществ. Во-первых, это упрощает процесс развертывания за счет управления зависимостями и порядком установки. Во-вторых, OLM способен поддерживать тысячи операторов в масштабных кластерах. Кроме того, он обеспечивает со-

блюдение заданной конфигурации, упрощая процесс обновлений, а также способствует стандартизации, предлагая единый способ упаковки и развертывания приложений в виде операторов.

Выбор между OLM и Helm зависит от среды и личных предпочтений. OLM более тесно интегрирован с Kubernetes и является нативным инструментом для данной платформы, тогда как Helm предлагает более развитую экосистему пакетов, обеспечивающую быстрое развертывание.

При изучении операторов и OLM важно уделить внимание объекту *ClusterServiceVersion (CSV)*. CSV является центральным элементом Operator Lifecycle Manager, содержащим метаданные и сведения, необходимые для развертывания оператора. В частности, этот объект определяет имя оператора и его версию, а также предоставляет его краткое описание. В нем заданы разрешения и роли, необходимые для корректной работы, CRD, которыми управляет оператор, стратегия установки и логика обновления. Более подробную информацию об этом объекте можно найти по адресу https://olm.operatorframework.io/docs/concepts/crds/clusterserviceversion/.

В следующем разделе мы развернем в Kubernetes систему мониторинга Prometheus с использованием специального оператора и OLM, чтобы продемонстрировать возможности обоих подходов к управлению приложениями.

Мониторинг кластеров Kubernetes с помощью Prometheus и Grafana

Успех в поддержании работоспособности и производительности приложений Kubernetes зависит от множества факторов. Одним из них является создание надежной и отказоустойчивой среды. В решении этой задачи могут помочь такие инструменты мониторинга, как *Prometheus* и *Grafana*. Prometheus работает «за кулисами», собирая и сохраняя данные о состоянии кластера Kubernetes. Grafana визуализирует эти метрики в наглядной и доступной для понимания форме, позволяя вам получить более полное представление о состоянии и поведении приложений и инфраструктуры.

На приведенной ниже схеме (взятой из официальной документации) показана общая архитектура компонентов Prometheus (рис. 14.9).

Традиционный способ развертывания мониторингового стека, включающего Prometheus и Grafana, является весьма трудоемким. Он требует самостоятельного написания множества YAML-манифестов с учетом зависимостей и корректного порядка установки, что довольно сложно и чревато ошибками. Использование операторов Prometheus и Grafana упрощает эту задачу.

Рис. 14.9. Архитектура Prometheus и некоторых компонентов его экосистемы (источник: https://prometheus.io/docs/introduction/overview/)

Для развертывания всего мониторингового стека (включая операторы и экземпляры Prometheus и Grafana) также можно использовать Helm-чарт, например предоставляемый проектом kube-prometheus-stack (https://artifacthub.io/packages/helm/prometheus-community/kube-prometheus-stack). Однако в следующем подразделе мы рассмотрим альтернативный подход и развернем этот стек с помощью компонента OLM.

Далее мы рассмотрим процесс установки Prometheus и Grafana в кластере Kubernetes, чтобы обеспечить полноценный мониторинг приложений и гарантировать их стабильную работу.

Установка компонента Operator Lifecycle Manager

Для установки операторов с использованием OLM необходимо предварительно установить в кластер сам этот компонент. Сделать это можно несколькими способами — с помощью утилиты `operator-sdk`, через Helm-чарты или путем применения соответствующих YAML-манифестов. В рамках данного упражнения мы воспользуемся установочным сценарием `install.sh`:

```
$ curl -sL https://github.com/operator-framework/operator-lifecycle-manager/
releases/download/v0.28.0/install.sh | bash -s v0.28.0
```

Дождитесь завершения работы сценария, который настроит кластер Kubernetes с OLM. После этого убедитесь в том, что в пространстве имен `olm` запущены необходимые поды, выполнив команду:

```
$ kubectl get pods -n olm
NAME                                READY   STATUS    RESTARTS   AGE
catalog-operator-9f6dc8c87-rt569    1/1     Running   0          36s
olm-operator-6bccddc987-nrlkm       1/1     Running   0          36s
operatorhubio-catalog-6l8pw         0/1     Running   0          21s
packageserver-6df47456b9-8fdt7      1/1     Running   0          24s
packageserver-6df47456b9-lrvzp      1/1     Running   0          24s
```

Выведите сведения об объектах ClusterServiceVersion (CSV) с помощью команды `kubectl get csv`:

```
$ kubectl get csv -n olm
NAME            DISPLAY         VERSION   REPLACES   PHASE
packageserver   Package Server  0.28.0               Succeeded
```

Если в выводе отображается статус `Succeeded`, значит, компонент OLM успешно установлен и готов к управлению операторами. В следующем подразделе мы установим с его помощью операторы Prometheus и Grafana.

Установка операторов Prometheus и Grafana с использованием OLM

После настройки компонента OLM установка операторов не составляет особого труда. Как правило, необходимые для этого команды и инструкции можно найти на странице соответствующего оператора на сайте operatorhub.io.

Чтобы установить оператор Prometheus (https://operatorhub.io/operator/prometheus) с помощью OLM, выполните следующие шаги.

1. Найдите Prometheus Operator на сайте OperatorHub:

```
$ kubectl get packagemanifests | grep prometheus
ack-prometheusservice-controller    Community Operators   12m
prometheus                          Community Operators   12m
prometheus-exporter-operator        Community Operators   12m
```

2. Установите его с помощью OLM:

```
$ kubectl create -f https://operatorhub.io/install/prometheus.yaml
subscription.operators.coreos.com/my-prometheus created
```

На странице https://operatorhub.io/install/prometheus.yaml приведен базовый YAML-манифест для создания подписки (Subscription). При необходимости вы можете создать локальные YAML-файлы с дополнительными параметрами конфигурации.

3. Подождите несколько минут и убедитесь в том, что оператор Prometheus успешно развернут:

```
$ kubectl get csv -n operators
NAME                          DISPLAY              VERSION  REPLACES   PHASE
prometheusoperator.v0.70.0    Prometheus Operator  0.70.0
prometheusoperator.v0.65.1    Succeeded
```

4. Удостоверьтесь в том, что поды оператора Prometheus находятся в состоянии `Running`:

```
$ kubectl get pods -n operators
NAME                                  READY   STATUS    RESTARTS   AGE
prometheus-operator-84f9b76686-2j27n  1/1     Running   0          87s
```

5. Найдите и установите оператор Grafana через OLM, ориентируясь на описанные выше шаги:

```
$ kubectl create -f https://operatorhub.io/install/grafana-operator.yaml
```

6. Выведите список объектов CRD, регистрируемых в кластере при установке операторов:

```
$ kubectl get crd
NAME                                                  CREATED AT
alertmanagerconfigs.monitoring.coreos.com             2024-10-18T09:14:03Z
alertmanagers.monitoring.coreos.com                   2024-10-18T09:14:04Z
catalogsources.operators.coreos.com                   2024-10-18T09:10:00Z
clusterserviceversions.operators.coreos.com           2024-10-18T09:10:00Z
grafanaalertrulegroups.grafana.integreatly.org        2024-10-18T09:25:19Z
... <фрагмент опущен для краткости> ...
```

7. Вы увидите несколько CRD, созданных в кластере Kubernetes.

После развертывания операторов Prometheus и Grafana можно переходить к созданию их экземпляров и настройке мониторингового стека для наблюдения за состоянием кластера Kubernetes. Этот процесс описан в следующем подразделе.

Настройка экземпляров Prometheus и Grafana с помощью операторов

Для настройки новых экземпляров Prometheus и Grafana мы будем использовать стандартные YAML-манифесты, основанные на конфигурациях CRD. Все инструкции и YAML-файлы, необходимые для выполнения этого упражнения, находятся в каталоге `Chapter14` репозитория GitHub.

Следуйте приведенной ниже инструкции, чтобы настроить мониторинговый стек, состоящий из Prometheus и Grafana, в Kubernetes.

1. Создайте пространство имен для развертывания системы мониторинга (см. файл `monitoring-ns.yaml`):

```
$ kubectl apply -f monitoring-ns.yaml
namespace/monitoring created
```

2. Настройте ServiceAccount с соответствующей ролью и разрешениями RBAC (см. файл `monitoring-sa.yaml`):

```
$ kubectl apply -f monitoring-sa.yaml
serviceaccount/prometheus created
role.rbac.authorization.k8s.io/prometheus-role created
rolebinding.rbac.authorization.k8s.io/prometheus-rolebinding created
```

3. Подготовьте YAML-манифест для нового экземпляра Prometheus (см. файл `prometheus-instance.yaml`):

```
apiVersion: monitoring.coreos.com/v1
kind: Prometheus
metadata:
  name: example-prometheus
  namespace: monitoring
spec:
  replicas: 2
  serviceAccountName: prometheus
  serviceMonitorSelector:
    matchLabels:
      app.kubernetes.io/name: node-exporter
```

Обратите внимание на строку kind: Prometheus в этом манифесте — она указывает на использование CRD. Оператор Prometheus распознает этот тип ресурса и выполнит все необходимые действия для создания развертывания.

4. Создайте экземпляр Prometheus, применив конфигурацию к кластеру:

```
$ kubectl apply -f prometheus-instance.yaml
prometheus.monitoring.coreos.com/example-prometheus created
```

5. Аналогичным способом разверните экземпляр Grafana с помощью соответствующего оператора:

```
# grafana-instance.yaml
apiVersion: grafana.integreatly.org/v1beta1
kind: Grafana
```

```
metadata:
  labels:
    dashboards: grafana-a
    folders: grafana-a
  name: grafana-a
  namespace: monitoring
spec:
  config:
    auth:
      disable_login_form: 'false'
    log:
      mode: console
    security:
      admin_password: start
      admin_user: root
```

В данном примере пароли и параметры конфигурации указаны в YAML-манифесте в открытом виде. В рабочей среде такие чувствительные данные следует хранить в объектах Kubernetes Secrets.

6. Примените YAML-манифест, чтобы создать экземпляр Grafana:

```
$ kubectl apply -f grafana-instance.yaml
grafana.grafana.integreatly.org/grafana-a created
```

7. Проверьте список созданных объектов, чтобы убедиться в том, что операторы Prometheus и Grafana успешно развернули все необходимые ресурсы в пространстве имен `monitoring`:

```
$ kubectl get pod,svc,sts -n monitoring
NAME                                      READY   STATUS    RESTARTS   AGE
pod/grafana-a-deployment-69f8999f8-82zbv  1/1     Running   0          17m
pod/node-exporter-n7tlb                   1/1     Running   0          93s
pod/prometheus-example-prometheus-0       2/2     Running   0          20m
pod/prometheus-example-prometheus-1       2/2     Running   0          20m

NAME                           TYPE        CLUSTER-IP       EXTERNAL-IP
PORT(S)     AGE
service/grafana-a-service      ClusterIP   10.107.212.241   <none>
3000/TCP    17m
service/prometheus-operated    ClusterIP   None             <none>
9090/TCP    20m

NAME                                             READY   AGE
statefulset.apps/prometheus-example-prometheus   2/2     20m
```

8. Вывод команд показывает, что операторы создали необходимые ресурсы Kubernetes на основе CRD. Обратите также внимание на то, что были созданы сервисы Prometheus и Grafana, доступ к которым мы получим на следующем этапе.

В качестве демонстрации мы настроим Node Exporter в кластере и визуализируем его данные с помощью Grafana. Node Exporter — это один из экспортеров Prometheus, который собирает детальную статистику хоста, включая данные об оборудовании и операционной системе, уровень загрузки процессора, объем потребляемой памяти и другие системные метрики. Он работает на каждом узле Kubernetes (или на физических и виртуальных серверах) как отдельный сервис и предоставляет эти метрики через конечную точку HTTP. Система Prometheus собирает эти данные, позволяя отслеживать состояние и производительность узлов. Это дает администраторам возможность анализировать уровень использования ресурсов и оперативно выявлять проблемы в инфраструктуре.

Чтобы запустить экземпляр экспортера на каждом узле, его следует развернуть в виде объекта DaemonSet. В этом случае Prometheus сможет собирать системные метрики со всех узлов кластера, обеспечивая комплексное наблюдение за производительностью системы. Создайте Node Exporter с помощью файла `node-exporter-daemonset.yaml`:

```
$ kubectl apply -f node-exporter-daemonset.yaml
daemonset.apps/node-exporter created
```

Затем создайте сервис, позволяющий системе Prometheus обнаруживать и опрашивать все поды Node Exporter, работающие на узлах кластера, обеспечивая централизованный сбор данных. Для создания этого сервиса используйте YAML-манифест `node-exporter-svc.yaml`:

```
$ kubectl apply -f node-exporter-svc.yaml
service/node-exporter created
```

Создайте объект serviceMonitor, который указывает системе Prometheus, какие сервисы и как именно необходимо опрашивать. В этой конфигурации задаются целевой сервис, интервалы опроса, метки и другие параметры, обеспечивающие согласованность сбора данных без участия администратора. Для создания объекта serviceMonitor выполните следующую команду:

```
$ kubectl apply -f servicemonitor-instance.yaml
servicemonitor.monitoring.coreos.com/node-exporter created
```

Теперь пора проверить развертывание мониторингового стека в кластере Kubernetes.

Убедимся в том, что система Prometheus успешно получает метрики. В одном из терминалов выполните команду `kubectl port-forward`, чтобы открыть доступ к сервису Prometheus (по окончании тестирования остановите переадресацию портов, нажав Ctrl+C):

```
$ kubectl port-forward -n monitoring svc/prometheus-operated 9091:9090
Forwarding from 127.0.0.1:9091 -> 9090
Forwarding from [::1]:9091 -> 9090
```

После этого откройте браузер и перейдите по адресу http://localhost:9091/targets. Убедитесь в том, что Node Exporter присутствует среди целей Prometheus (рис. 14.10).

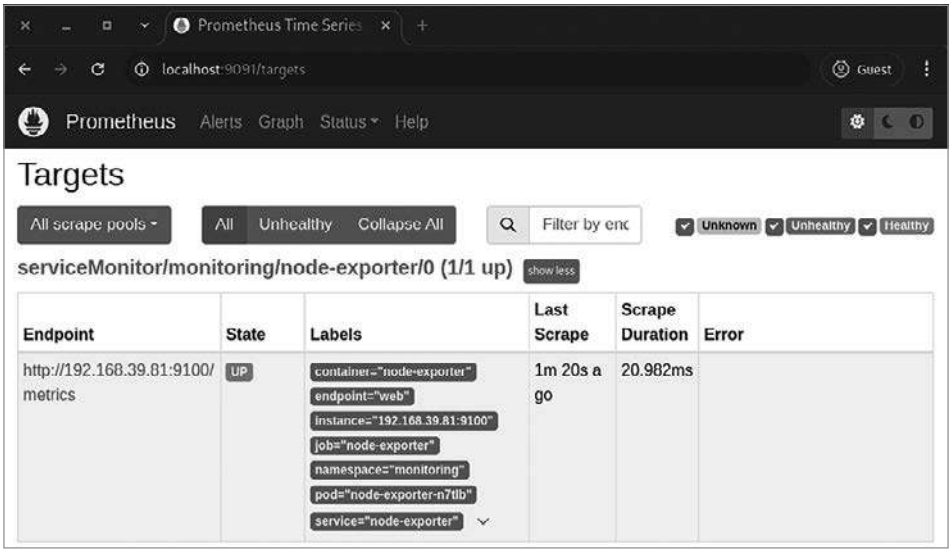

Рис. 14.10. Портал Prometheus с Node Exporter, видимым для узлов Kubernetes

Как видите, Prometheus успешно получает метрики узлов.

Теперь визуализируем получаемые данные с помощью инструмента Grafana. Откройте терминал и выполните команду `kubectl port-forward` в отношении сервиса Grafana:

```
$ kubectl port-forward -n monitoring service/grafana-a-service 3000:3000
Forwarding from 127.0.0.1:3000 -> 3000
Forwarding from [::1]:3000 -> 3000
```

Затем откройте браузер и перейдите по адресу http://localhost:3000 (рис. 14.11).

Рис. 14.11. Портал Grafana

Войдите в систему Grafana, используя учетные данные, указанные в файле `grafana-instance.yaml` или в объекте Secret (если вы использовали его).

После входа вы увидите панель, используемую по умолчанию. Чтобы отобразить метрики узлов, нам нужно настроить собственную панель.

В меню слева выберите пункт Connections ▸ Data sources (Соединения ▸ Источники данных), как показано на рис. 14.12.

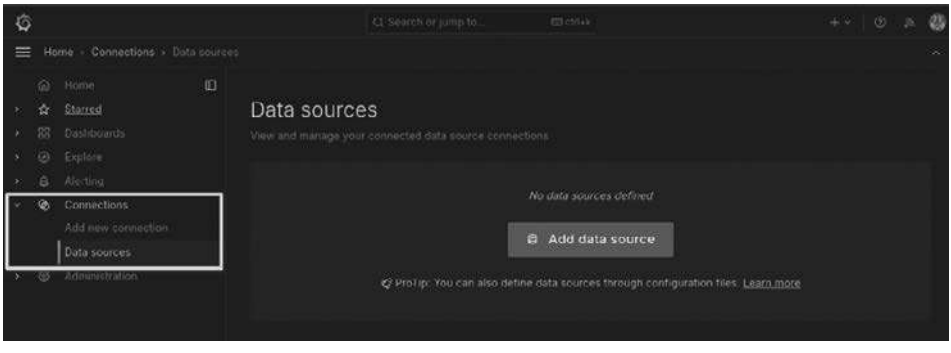

Рис. 14.12. Добавление источника данных в Grafana

В открывшемся окне выберите Prometheus в качестве источника данных и укажите URL Prometheus, как показано ниже. Не забудьте указать полное доменное имя (FQDN), например: http://prometheus-operated.monitoring.svc.cluster.local:9090 (подробнее о сервисах и FQDN можно прочитать в главе 8) (рис. 14.13).

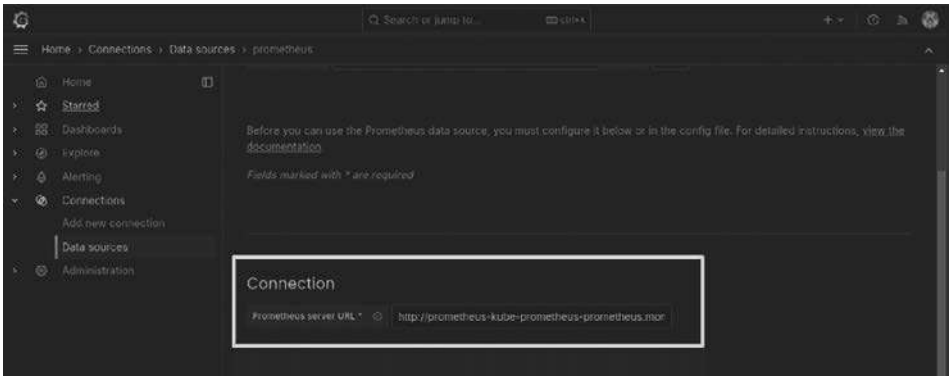

Рис. 14.13. Настройка источника данных в Grafana

Нажмите кнопку Save & test (Сохранить и протестировать) внизу страницы. В случае успешной настройки вы увидите соответствующее сообщение, показанное на рис. 14.14. (Если возникнет ошибка, проверьте указанный URL-адрес Prometheus, включая полное доменное имя (FQDN) и номер порта.)

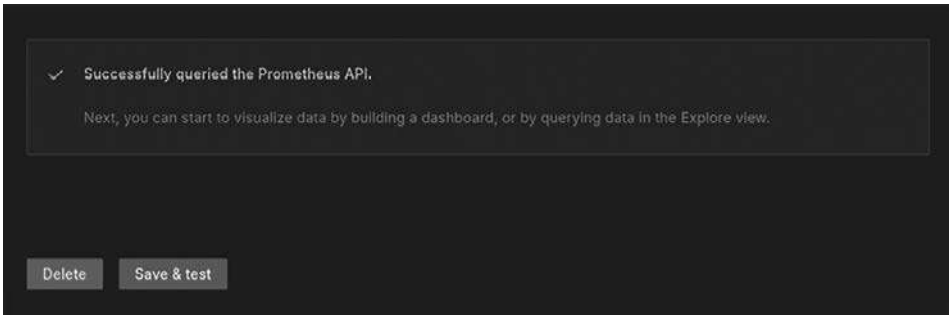

Рис. 14.14. Результат успешной настройки источника данных Grafana

После подключения источника данных необходимо создать панель для визуализации метрик. Вы можете создать ее вручную или импортировать готовую панель с заранее настроенными параметрами. Для этого перейдите на сайт https://grafana.com/grafana/dashboards/, найдите панель Node Exporter Full и нажмите кнопку Copy ID to clipboard (Скопировать ID в буфер обмена) (рис. 14.15).

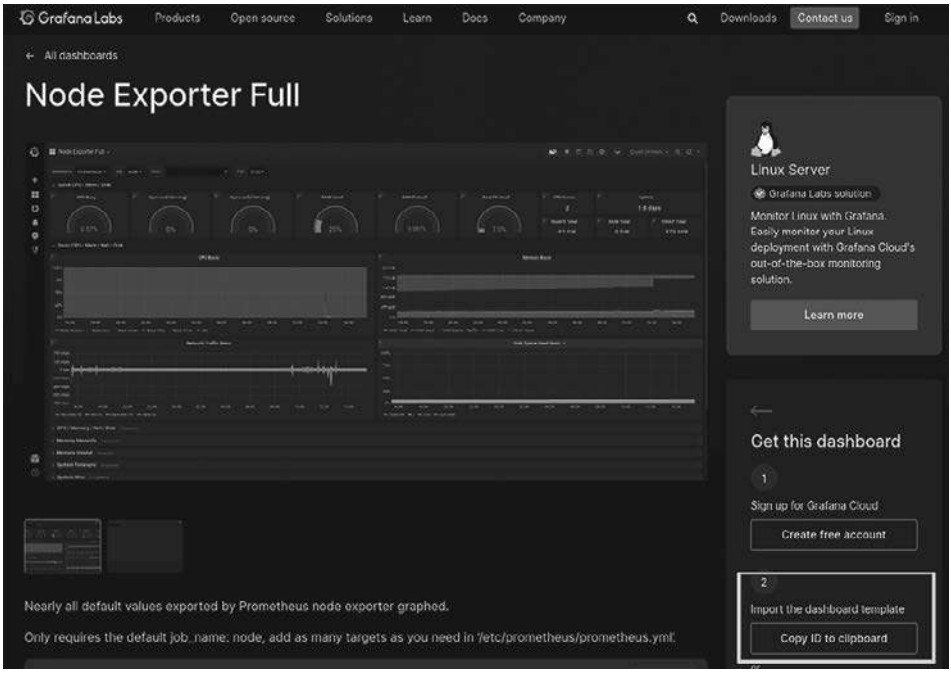

Рис. 14.15. Копирование идентификатора панели Grafana для последующего импорта

Вернитесь в веб-интерфейс Grafana и выберите пункт меню Dashboards ▸ New ▸ Import (Информационные панели ▸ Создать ▸ Импортировать), как показано на рис. 14.16.

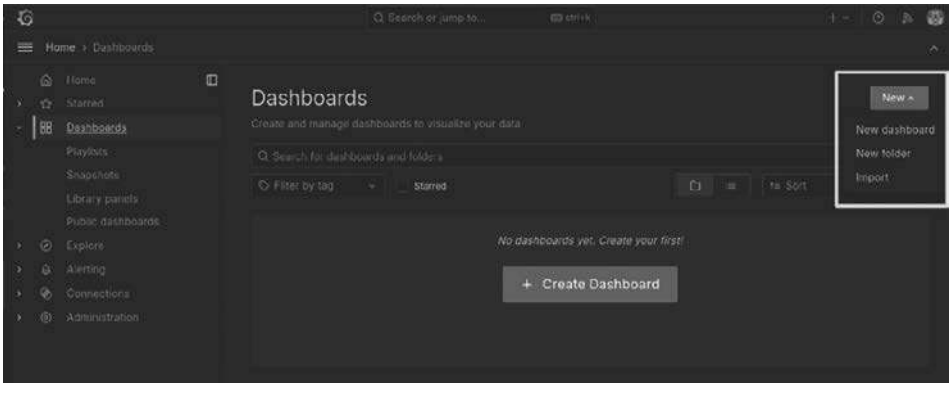

Рис. 14.16. Импорт новой панели в Grafana

Введите ранее скопированный идентификатор панели Node Exporter Full и нажмите кнопку Load (Загрузить) (рис. 14.17).

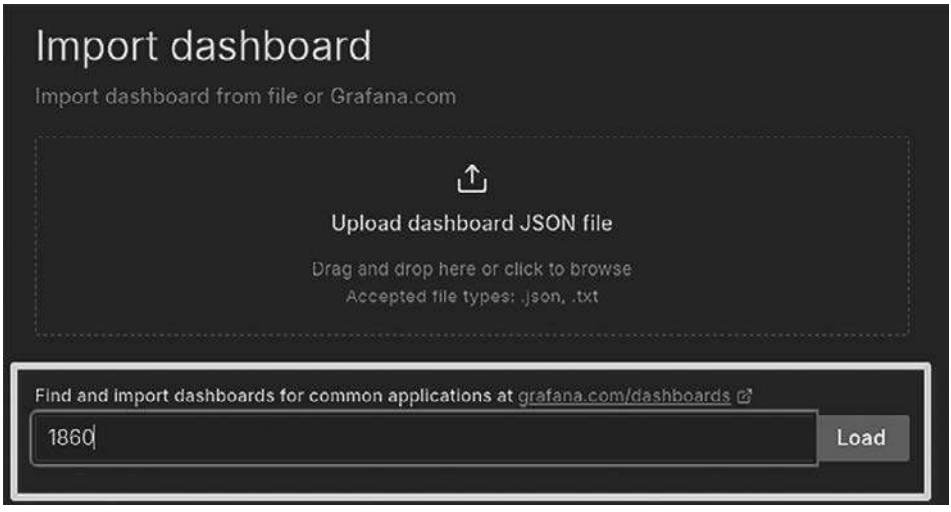

Рис. 14.17. Ввод идентификатора панели для ее импорта в Grafana

На следующем экране выберите Prometheus в качестве источника данных (настроенного ранее) и нажмите кнопку Import (Импорт) (рис. 14.18).

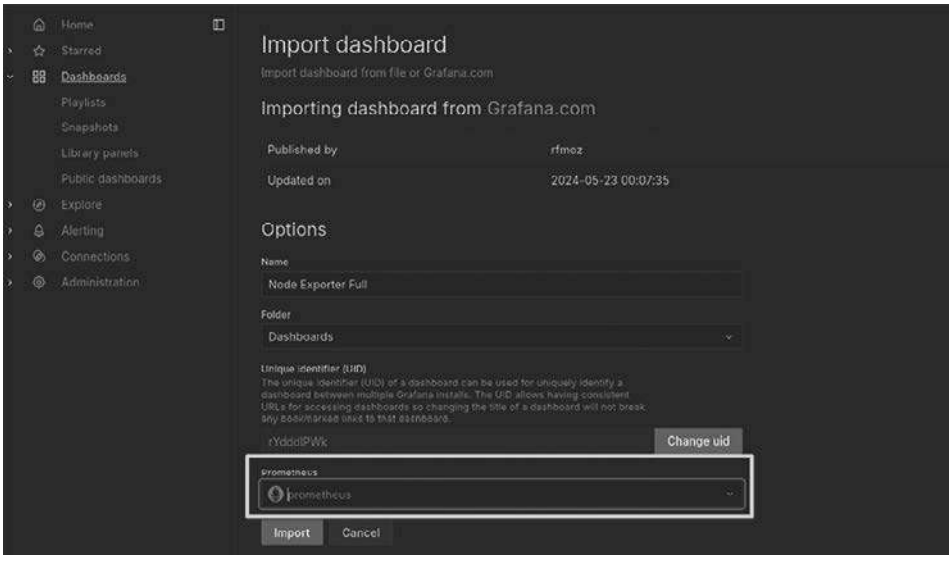

Рис. 14.18. Завершение импорта панели в Grafana

Перед вами появится информационная панель с набором готовых графиков и виджетов, показанная ниже (рис. 14.19). Теперь вы можете исследовать состояние кластера, отслеживая такие параметры, как уровень загрузки процессора, объем потребляемой памяти, сетевой трафик и многое другое.

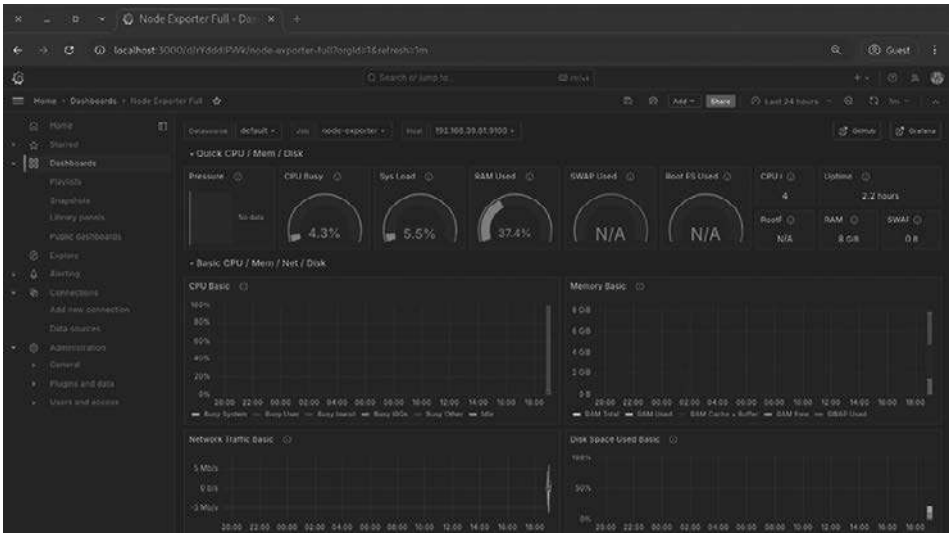

Рис. 14.19. Панель Node Exporter Full, импортированная в Grafana

Итак, вы успешно развернули стек Prometheus и Grafana в своем кластере Kubernetes, включили Node Exporter и настроили визуализацию системных метрик.

На этом мы завершаем главу. Как видите, использование Helm-чартов и операторов упрощает развертывание многокомпонентных решений в Kubernetes, обеспечивая множество преимуществ как в среде разработки, так и в рабочей среде.

Резюме

В этой главе мы подробно обсудили работу с инструментом Helm, Helm-чартами и операторами Kubernetes. Вы узнали, в чем заключается цель менеджеров пакетов и каким образом Helm выполняет функции такого менеджера в экосистеме Kubernetes. Мы показали, как установить Helm на локальную машину и развернуть тестовый чарт WordPress, чтобы убедиться в корректности установки. Затем мы разобрали структуру Helm-чартов и показали, как можно настроить YAML-шаблоны внутри чарта с помощью значений, предоставляемых пользователем. После этого мы продемонстрировали процесс установки популярных решений в кластере Kubernetes с использованием Helm. Мы установили

веб-интерфейс Kubernetes (Dashboard) и рассмотрели его основные компоненты. Далее мы рассмотрели операторы Kubernetes и обсудили связанные с ними компоненты, включая определения пользовательских ресурсов (CRD). В завершение развернули мониторинговый стек, включающий Prometheus и Grafana, как с помощью Helm-чартов, так и с применением операторов.

В следующей части книги вы получите все необходимые знания для эффективного развертывания кластеров Kubernetes в различных облачных средах. Мы начнем с развертывания кластеров на платформе Google Kubernetes Engine (GKE).

Дополнительные источники информации

- Сайт инструмента Helm: https://helm.sh/.
- Веб-интерфейс Kubernetes Dashboard: https://kubernetes.io/docs/tasks/access-application-cluster/web-ui-dashboard/.
- Получение доступа к Kubernetes Dashboard: https://github.com/kubernetes/dashboard/blob/master/docs/user/accessing-dashboard/README.md.
- Развертывание и получение доступа к Kubernetes Dashboard: https://kubernetes.io/docs/tasks/access-application-cluster/web-ui-dashboard/.
- Инструкция по созданию пользователя для Dashboard: https://github.com/kubernetes/dashboard/blob/master/docs/user/access-control/creating-sample-user.md.
- Разработка оператора Kubernetes за шесть шагов: https://developers.redhat.com/articles/2021/09/07/build-kubernetes-operator-six-steps.
- Пользовательские ресурсы: https://kubernetes.io/docs/concepts/extend-kubernetes/api-extension/custom-resources/.

Для более глубокого изучения Helm и Helm-чартов рекомендуем книгу издательства Packt Publishing: *Block A., Dewey A.* Learn Helm (https://www.packtpub.com/product/learn-helm/9781839214295).

Более подробную информацию о системах Elasticsearch и Prometheus можно найти в следующих книгах Packt Publishing:

- *Andhavarapu A.* Learning Elasticsearch (https://www.packtpub.com/product/learning-elasticsearch/9781787128453).
- *Bastos J., Araujo P.* Hands-On Infrastructure Monitoring with Prometheus (https://www.packtpub.com/product/hands-on-infrastructure-monitoring-with-prometheus/9781789612349).

15 Запуск кластера на платформе Google Cloud Platform (GCP) с помощью Google Kubernetes Engine (GKE)

В этой главе мы развернем кластер Kubernetes на облачной платформе *Google Cloud Platform (GCP)*. Еще две публичные облачные платформы будут рассмотрены в последующих главах.

По ходу чтения вы создадите учетную запись в Google Cloud Platform, запустите рабочую нагрузку Kubernetes с помощью сервиса *Google Kubernetes Engine (GKE)* и получите представление о его возможностях.

Технические требования

Для выполнения инструкций, приведенных в этой главе, вам потребуется учетная запись Google Cloud Platform с указанным способом оплаты.

Следование инструкциям, приведенным в этой главе, сопряжено с финансовыми расходами. Поэтому крайне важно удалить все созданные ресурсы после завершения работы. Указанные в тексте цены были актуальны на момент написания этой главы; перед созданием ресурсов рекомендуется проверить текущие тарифы.

Знакомство с Google Cloud Platform и Google Kubernetes Engine

Прежде чем приступить к регистрации учетной записи Google Cloud Platform и установке инструментов, необходимых для запуска кластера на базе GKE, давайте кратко обсудим сам сервис Google Kubernetes Engine и историю его появления.

Далее в главе для обозначения Google Cloud Platform и Google Kubernetes Engine мы будем использовать сокращения GCP и GKE.

Платформа Google Cloud Platform

Из «большой тройки» публичных облачных провайдеров Google Cloud Platform (GCP) является самым молодым. В двух следующих главах мы рассмотрим *Amazon Web Services (AWS)* и *Microsoft Azure*.

Выход Google на рынок публичных облачных технологий происходил не так, как в случае двух других провайдеров. В апреле 2008 года компания запустила бета-версию Google App Engine — первого компонента своей облачной платформы. Этот сервис (по-прежнему существующий на момент написания книги) позволяет разработчикам развертывать приложения в управляемых Google средах выполнения, поддерживающих такие языки, как PHP, Java, Ruby, Python, Node.js и C#, а также язык программирования Go, исходный код которого был открыт в 2009 году.

Следующим сервисом в составе GCP стал *Google Cloud Storage*, представленный в мае 2010 года, за которым последовали *Google BigQuery* и предварительная версия Prediction API. В октябре 2011 года был запущен сервис *Google Cloud SQL*, а в июне 2012 года — предварительная версия *Google Compute Engine*.

Таким образом, за четыре года компания Google создала набор ключевых компонентов, которые составляют основу ее публичной облачной платформы.

Тем не менее большинство сервисов в то время все еще находилось в статусе предварительной версии. Лишь в 2013 году они стали *общедоступными*, что позволило безопасно использовать их для широкомасштабного запуска рабочих нагрузок. Ключевым фактором стало появление *Соглашения об уровне обслуживания* (SLA), которое является обязательным условием для внедрения подобных сервисов как малыми, так и крупными компаниями. И все это произошло за год до запуска Kubernetes.

В конце 2014 года Google выпустила первую альфа-версию Google Kubernetes Engine (GKE).

Сервис Google Kubernetes Engine

Поскольку система Kubernetes была разработана в компании Google, накопившей обширный опыт масштабного запуска контейнерных рабочих нагрузок в рамках проекта Borg, неудивительно, что именно Google одной из первых предложила собственный управляемый сервис Kubernetes — Google Kubernetes Engine (GKE).

Версия Kubernetes v1 была выпущена и передана под управление организации *Cloud Native Computing Foundation (CNCF)* в июле 2015 года. Спустя всего месяц сервис GKE получил статус общедоступного.

Сервис GKE позволяет запускать и управлять сертифицированным CNCF кластером Kubernetes, использующим вычислительные, сетевые ресурсы и хранилища GCP, а также обеспечивает его глубокую интеграцию со встроенными средствами мониторинга, аутентификации и управления доступом.

Различия между облачными реализациями Kubernetes и рекомендации по их выбору будут подробно рассмотрены в конце главы 17.

Теперь, когда вы познакомились с историей появления сервиса, перейдем к регистрации учетной записи и установке инструментов, необходимых для развертывания кластера.

Подготовка локальной среды

Первым делом необходимо получить доступ к платформе GCP. Для этого нужно либо создать новую учетную запись, либо войти в уже существующую. Рассмотрим оба варианта.

Чтобы зарегистрироваться в Google Cloud Platform, перейдите по адресу https://cloud.google.com/. На открывшейся странице вы увидите интерфейс, подобный изображенному на рис. 15.1.

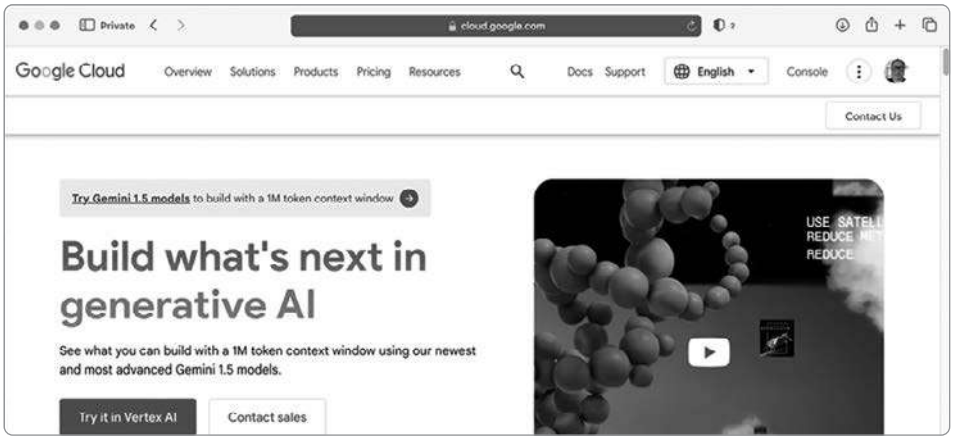

Рис. 15.1. Домашняя страница Google Cloud

Если вы пользуетесь сервисами Google, например Gmail, YouTube или смартфоном под управлением ОС Android, у вас уже есть учетная запись Google, которую можно использовать для получения доступа к системе. В этом примере один из авторов использует свою учетную запись, что видно по аватару в правом верхнем углу страницы.

Во время написания этой книги компания Google предоставляла кредиты на сумму 300 долларов США, действительные в течение 90 дней. Чтобы воспользоваться этим предложением, необходимо указать действующий способ оплаты. Это позволяет доказать, что регистрацию проходит не бот. После окончания срока действия или исчерпания кредитов вы сможете перейти на платный тарифный план. Подробнее об этом предложении можно узнать здесь: https://cloud.google.com/free/docs/free-cloud-features.

Чтобы активизировать бесплатные кредиты, перейдите по ссылке https://cloud.google.com/free/docs/free-cloud-features#try-it-for-yourself и нажмите кнопку Get started for free (Попробовать бесплатно). Если вы соответствуете условиям программы, ознакомьтесь с правилами и условиями использования и следуйте инструкциям на экране. После регистрации вы будете перенаправлены в консоль GCP. Если у вас уже есть учетная запись, войдите в консоль Google Cloud по адресу https://console.cloud.google.com/.

Теперь создадим проект для развертывания наших ресурсов.

Создание проекта

В экосистеме GCP все ресурсы размещаются внутри проектов. Если вы только что зарегистрировали учетную запись, проект с именем My First Project был создан автоматически.

Если вы используете уже существующую учетную запись, создайте новый проект специально для запуска кластера GKE. Для этого щелкните на меню Select (Выбрать), расположенном на верхней панели, справа от логотипа GCP в левом верхнем углу страницы.

Откроется список всех доступных проектов, позволяющий выбрать существующий или создать новый. Для создания нового проекта выполните следующие шаги.

1. Нажмите кнопку NEW PROJECT (Создать проект).
2. Укажите необходимые параметры.
   - Введите имя проекта.
   - Выберите организацию, к которой будет привязан проект.

- Выберите местоположение — это может быть организация или папка, в которой будет храниться проект.

3. После заполнения всех полей нажмите кнопку CREATE (Создать) (рис. 15.2).

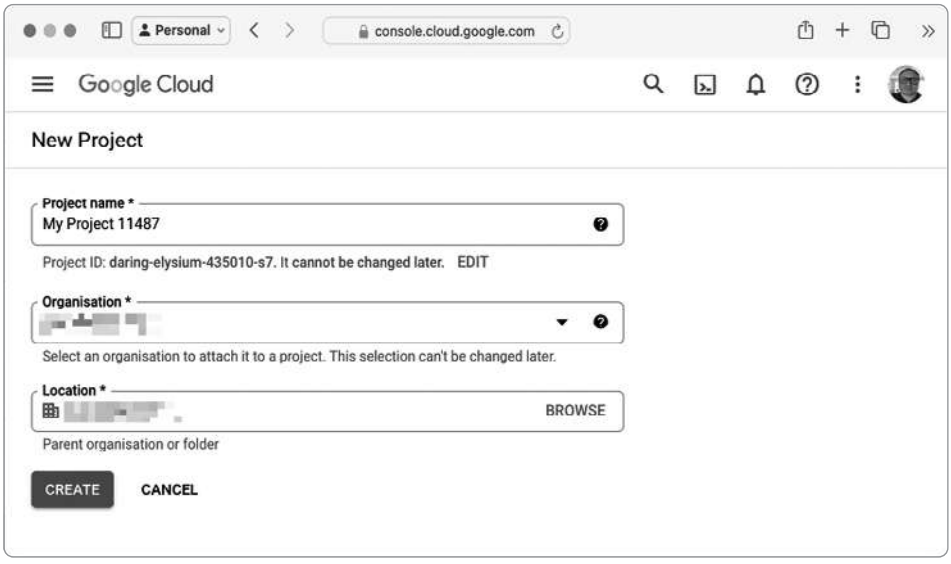

Рис. 15.2. Создание нового проекта

Теперь, когда у нас есть пространство для размещения создаваемых ресурсов, установим интерфейс командной строки GCP.

Установка интерфейса командной строки GCP

В этом подразделе описан процесс установки *интерфейса командной строки* (Command-Line Interface, CLI) GCP на локальной машине. Если вы не можете или не хотите устанавливать его локально, не беспокойтесь, его можно запускать непосредственно в облаке. Начнем с рассмотрения процесса установки в операционной системе macOS.

Установка в macOS

Если вы, как и мы, часто работаете в терминале macOS, скорее всего, вы уже устанавливали и использовали пакетный менеджер Homebrew.

Актуальные инструкции по установке Homebrew приведены на официальном сайте проекта: https://brew.sh/.

После инсталляции Homebrew установить интерфейс командной строки GCP можно с помощью следующей команды:

```
$ brew install --cask google-cloud-sdk
```

Вы можете проверить корректность установки, выполнив команду:

```
$ gcloud version
```

Если установка прошла успешно, вы увидите вывод, аналогичный приведенному на рис. 15.3.

Рис. 15.3. Результат проверки версии CLI GCP в macOS

Если при установке возникнут проблемы, обратитесь к документации, ссылки на которую приведены выше. Дополнительную информацию об установке CLI GCP можно получить с помощью команды:

```
$ brew info --cask google-cloud-sdk
```

После установки интерфейса командной строки GCP вы можете перейти к пункту «Инициализация».

Установка в Windows

Установить интерфейс командной строки GCP в Windows можно несколькими способами. Первый сводится к открытию сеанса PowerShell и выполнению следующей команды для установки CLI GCP через Chocolatey.

Как и Homebrew в macOS, Chocolatey представляет собой пакетный менеджер, обеспечивающий удобный и единообразный способ установки широкого спектра программ в Windows с помощью оболочки PowerShell и специального синтаксиса. Это избавляет от необходимости разбираться со множеством различных способов установки, предусмотренных в Windows. Более подробную информацию можно найти на официальном сайте проекта: https://chocolatey.org/.

Если менеджер Chocolatey у вас еще не установлен, выполните следующую команду в оболочке PowerShell, запущенной от имени администратора:

```
$ Set-ExecutionPolicy Bypass -Scope Process -Force;
  [System.Net.ServicePointManager]::SecurityProtocol =
    [System.Net.ServicePointManager]::SecurityProtocol -bor 3072;
  iex ((New-Object System.Net.WebClient).DownloadString(
    'https://chocolatey.org/install.ps1'
  ))
```

После установки Chocolatey (или если он уже установлен) выполните следующую команду для установки GCP CLI:

```
$ choco install --ignore-checksum gcloudsdk
```

Альтернативный способ предполагает загрузку установочного файла по ссылке: https://dl.google.com/dl/cloudsdk/channels/rapid/GoogleCloudSDKInstaller.exe. После загрузки запустите установщик и следуйте инструкциям на экране.

По завершении установки откройте новое окно PowerShell и выполните команду:

```
$ gcloud version
```

Если все прошло успешно, вы увидите результат, аналогичный выводу, показанному в предыдущем пункте. После установки интерфейса командной строки GCP вы можете перейти к пункту «Инициализация».

Установка в Linux

Хотя пакеты Google Cloud CLI доступны для большинства дистрибутивов Linux, описание установки через все возможные пакетные менеджеры заняло бы слишком много места. Поэтому воспользуемся универсальным глобальным установочным сценарием, предоставленным Google.

Для запуска установщика выполните команду:

```
$ curl https://sdk.cloud.google.com | bash
```

Во время установки сценарий задаст несколько вопросов. В большинстве случаев можно отвечать `Yes`. После завершения установки выполните следующую команду для перезапуска сеанса и применения изменений, внесенных установщиком:

```
$ exec -l $SHELL
```

Получить информацию о версии установленного интерфейса командной строки GCP можно с помощью команды:

```
$ gcloud version
```

Как видите, несмотря на различия в методах установки, во всех трех операционных системах используется одна и та же команда `gcloud`, дающая одинаковые результаты. Все последующие команды, приведенные в этой главе, применимы ко всем операционным системам.

Оболочка Cloud Shell

Установить GCP CLI также можно через встроенную оболочку Cloud Shell, доступную прямо из консоли Google Cloud. Чтобы открыть ее, щелкните на значке Shell (Оболочка), находящемся справа в верхнем меню консоли.

После инициализации вы увидите веб-интерфейс, похожий на терминал, в котором можно выполнить команду:

```
$ gcloud version
```

Вывод этой команды немного отличается, поскольку Cloud Shell уже включает полный набор инструментов Google Cloud. Тем не менее отображенные версии будут соответствовать тем, которые устанавливались локально (рис. 15.4).

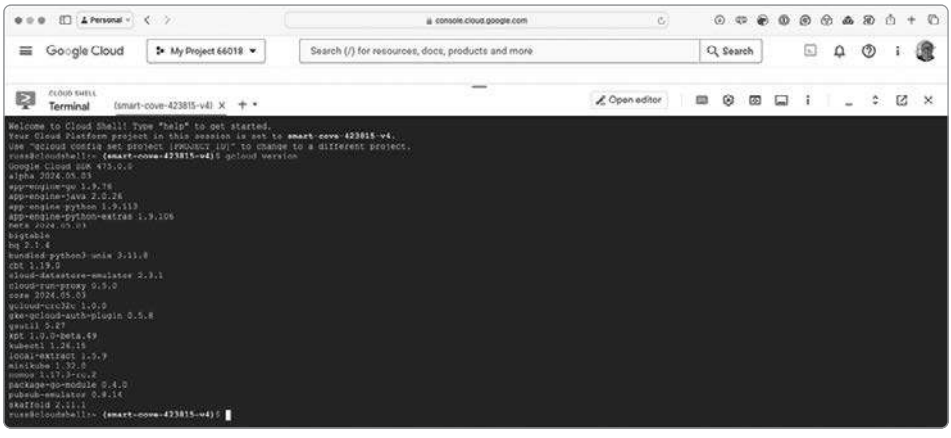

Рис. 15.4. Использование Google Cloud Shell

Если вы используете Google Cloud Shell, можете пропустить следующий пункт, так как инициализация уже была выполнена автоматически.

Инициализация

Если вы установили клиент CLI GCP локально, вам необходимо связать его со своей учетной записью GCP. Для этого выполните следующую команду:

```
$ gcloud init
```

После ее запуска программа проведет краткую сетевую диагностику, чтобы убедиться в наличии сетевого соединения, необходимого для работы клиента. После успешной проверки отобразится приглашение:

```
You must log in to continue. Would you like to log in (Y/n)?
```

Введите `Y`, чтобы открыть окно браузера. Если окно не открылось автоматически, скопируйте предоставленный URL-адрес и вставьте его в адресную строку. После выбора учетной записи, которую вы хотите использовать, появится экран с перечнем разрешений, запрашиваемых клиентом (рис. 15.5).

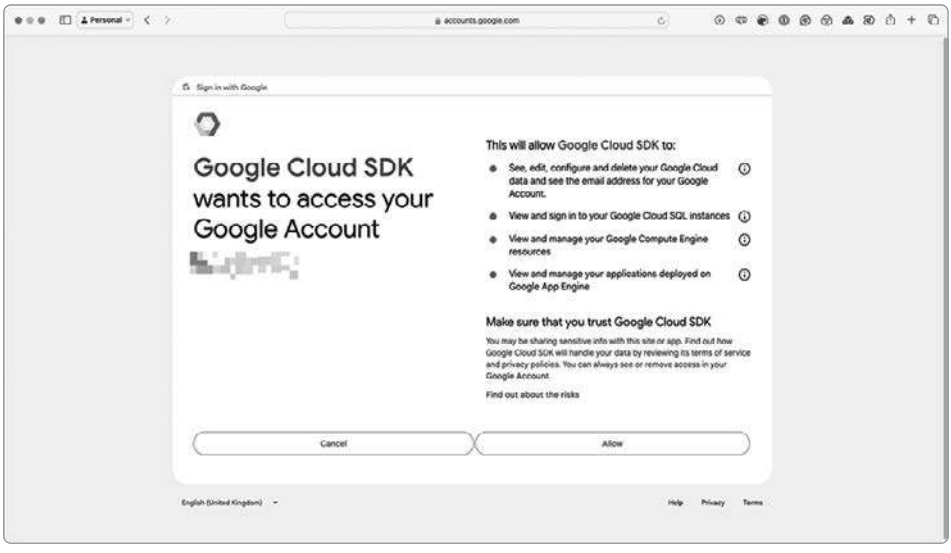

Рис. 15.5. Просмотр запрашиваемых разрешений

Если вы согласны предоставить указанные разрешения, нажмите кнопку Allow (Разрешить). После этого в терминале появится подтверждение с указанием пользователя, от имени которого был выполнен вход. Далее система предложит вам выбрать облачный проект для дальнейшей работы. В списке будут отображены только уникальные идентификаторы проектов (Project ID), а не описательные имена, заданные ранее в консоли Google Cloud. Если у вас несколько проектов, выберите нужный.

Вы можете изменить проект, используемый клиентом, выполнив команду:

```
$ gcloud config set project PROJECT_ID
```

Вместо `PROJECT_ID` введите уникальный идентификатор проекта, на который хотите переключиться.

После установки интерфейса командной строки Google Cloud и настройки учетной записи можно приступить к запуску кластера GKE.

Запуск кластера GKE

Поскольку процесс создания кластера занимает несколько минут, сначала запустим команду для его инициализации, а затем обсудим то, что происходит во время развертывания.

Перед запуском кластера необходимо активизировать сервис `container.googleapis.com`:

```
$ gcloud services enable container.googleapis.com
```

Затем нужно установить плагин, позволяющий проходить аутентификацию в кластере с помощью `kubectl`. Для этого выполните команду:

```
$ gcloud components install gke-gcloud-auth-plugin
```

После активизации сервиса и установки плагина мы можем создать двухузловой кластер `myfirstgkecluster` в одной зоне доступности региона `us-central1` (Central US) с помощью следующей команды:

```
$ gcloud container clusters create myfirstgkecluster --num-nodes=2 --zone=us-
central1-a
```

Через несколько минут вы увидите вывод, аналогичный приведенному на рис. 15.6.

Рис. 15.6. Запуск кластера

После завершения процесса развертывания вы сможете перейти по URL-адресу, указанному в выводе команды, и увидеть свой кластер в консоли Google Cloud (рис. 15.7).

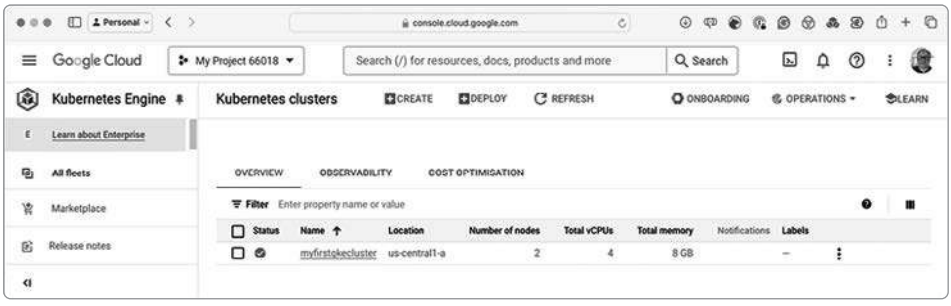

Рис. 15.7. Отображение кластера в консоли Google Cloud

Консоль Google Cloud доступна по адресу https://console.cloud.google.com/.

Теперь, когда наш кластер создан и запущен, можно перейти к развертыванию рабочей нагрузки и изучению интерфейса консоли Google Cloud.

Развертывание рабочей нагрузки и взаимодействие с кластером

При запуске кластера с помощью команды:

```
$ gcloud container clusters create myfirstgkecluster --num-nodes=2 --zone=us-
central1-a
```

в выводе появится следующая строка:

```
kubeconfig entry generated for myfirstgkecluster.
```

Как вы, вероятно, уже догадались, эта строка означает, что все данные, необходимые для подключения вашего локального клиента `kubectl` к только что развернутому кластеру GKE, были автоматически загружены.

Вы можете убедиться в этом с помощью команды:

```
$ kubectl get nodes
```

В результате выполнения этой команды вы должны увидеть два узла, имена которых начинаются с префикса gke, как показано на рис. 15.8.

Рис. 15.8. Использование kubectl для получения списка узлов

Если вывод подтверждает наличие узлов кластера GKE и текущая конфигурация `kubectl` вас устраивает, вы можете перейти сразу к пункту «Запуск примерной рабочей нагрузки» следующего подраздела.

Ссылка на официальную документацию по GKE приведена в разделе «Дополнительные источники информации» в конце этой главы.

Настройка локального клиента

Если вам необходимо подключить к своему кластеру другую установку `kubectl`, для этого можно использовать специальную команду интерфейса командной строки GCP.

Следующие инструкции предполагают, что CLI GCP уже установлен и настроен. Если это не так, вернитесь к подразделу «Установка интерфейса командной строки GCP» ранее в этой главе.

Чтобы загрузить учетные данные и настроить `kubectl` для работы с кластером, выполните команду:

```
$ gcloud container clusters get-credentials myfirstgkecluster --zone=us-
central1-a
```

Если вы хотите переключиться на другую конфигурацию (называемую контекстом), используйте следующие команды. Чтобы отобразить текущий контекст, выполните команду:

```
$ kubectl config current-context
```

Чтобы просмотреть список всех доступных контекстов, выполните команду:

```
$ kubectl config get-contexts -o name
```

Определив имя нужного контекста, вы можете переключиться на него с помощью следующей команды, заменив `CONTEXT_NAME` соответствующим значением:

```
$ kubectl config use-context CONTEXT_NAME
```

Теперь, когда ваш клиент `kubectl` настроен для взаимодействия с кластером GKE, можно переходить к развертыванию примерной рабочей нагрузки.

Запуск примерной рабочей нагрузки

В качестве примера мы развернем приложение Guestbook, основанное на связке PHP/Redis. Этот пример используется в официальной документации по Kubernetes. Первым шагом является создание объектов Deployment и Service для Redis Leader. Для этого выполните следующие команды:

```
$ kubectl apply -f https://raw.githubusercontent.com/GoogleCloudPlatform/
kubernetes-engine-samples/main/quickstarts/guestbook/redis-leader-
deployment.yaml
$ kubectl apply -f https://raw.githubusercontent.com/GoogleCloudPlatform/
kubernetes-engine-samples/main/quickstarts/guestbook/redis-leader-
service.yaml
```

Затем необходимо создать объекты Deployment и Service для Redis Follower:

```
$ kubectl apply -f https://raw.githubusercontent.com/GoogleCloudPlatform/
kubernetes-engine-samples/main/quickstarts/guestbook/redis-follower-
deployment.yaml
$ kubectl apply -f https://raw.githubusercontent.com/GoogleCloudPlatform/
kubernetes-engine-samples/main/quickstarts/guestbook/redis-follower-
service.yaml
```

После запуска обоих компонентов Redis можно развернуть фронтенд, то есть само приложение Guestbook. Для этого примените следующие манифесты:

```
$ kubectl apply -f https://raw.githubusercontent.com/GoogleCloudPlatform/
kubernetes-engine-samples/main/quickstarts/guestbook/frontend-deployment.yaml
$ kubectl apply -f https://raw.githubusercontent.com/GoogleCloudPlatform/
kubernetes-engine-samples/main/quickstarts/guestbook/frontend-service.yaml
```

Через несколько минут выполните следующую команду, чтобы получить информацию о только что созданном сервисе:

```
$ kubectl get service frontend
```

В выводе команды будет указан внешний IP-адрес (`EXTERNAL-IP`), как показано на рис. 15.9.

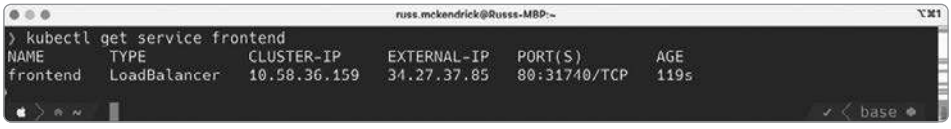

Рис. 15.9. Получение информации о сервисе frontend

После запуска приложения скопируйте значение `EXTERNAL-IP` и введите его в адресную строку браузера. Используйте адрес http://EXTERNAL-IP, а не https://EXTERNAL-IP, потому что приложение Guestbook прослушивает порт `80`, а не `443`.

В окне браузера откроется простое веб-приложение Guestbook. Попробуйте добавить несколько сообщений (рис. 15.10).

Рис. 15.10. Приложение Guestbook с тестовыми сообщениями

Теперь, когда наша рабочая нагрузка запущена, мы можем обсудить возможности Google Cloud Console.

Изучение консоли Google Cloud

Мы уже видели наш кластер в консоли Google Cloud, поэтому перейдем к разделу Workloads (Рабочие нагрузки), щелкнув на соответствующей ссылке в левом боковом меню раздела Kubernetes Engine (Движок Kubernetes) (рис. 15.11).

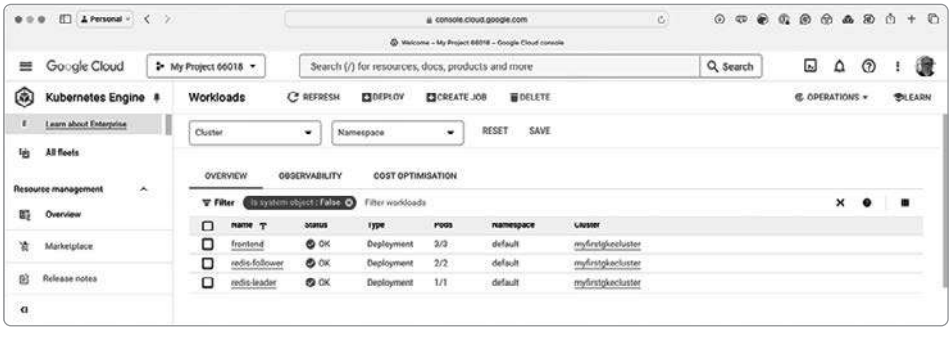

Рис. 15.11. Просмотр рабочей нагрузки в консоли

Как видно на приведенном снимке экрана, в списке отображаются три объекта Deployment с указанием пространства имен, в котором они находятся, и кластера, которому они принадлежат.

Если бы у нас было несколько кластеров с разными пространствами имен и развертываниями, мы могли бы воспользоваться функцией фильтрации для выбора рабочих нагрузок GKE.

Раздел Workloads

Если щелкнуть на развертывании frontend, откроется страница с подробной информацией об этом объекте (рис. 15.12).

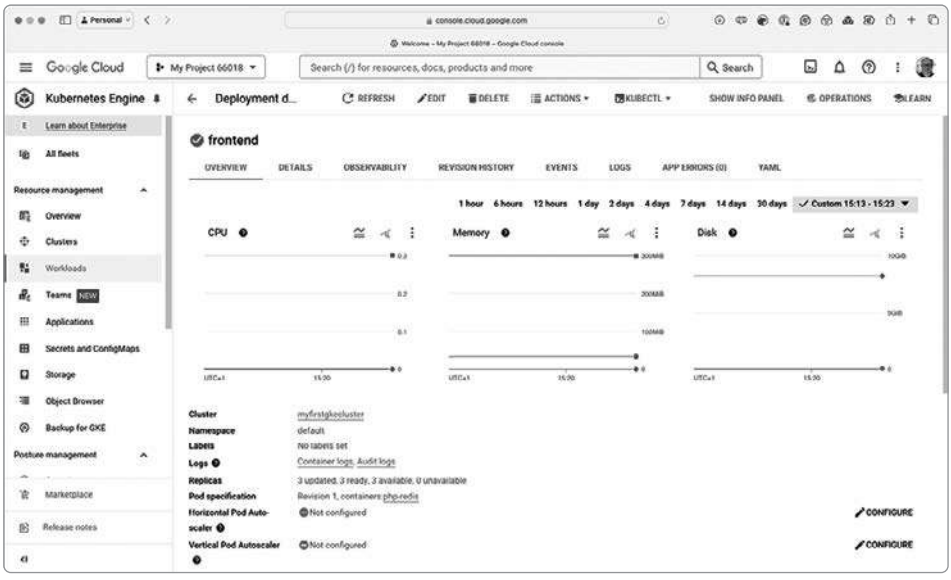

Рис. 15.12. Обзор развертывания frontend

На этой странице можно получить детальную информацию о развертывании, используя следующие вкладки.

- OVERVIEW (Обзор) — предоставляет общую информацию о выбранной рабочей нагрузке. Здесь отображаются данные об уровне использования ресурсов ЦП, памяти и дискового пространства, а также другие метрики.
- DETAILS (Сведения) — содержит дополнительную информацию о среде и самой рабочей нагрузке, включая время создания, аннотации, метки, параметры реплик, стратегию обновления и сведения о подах.
- OBSERVABILITY (Наблюдаемость) — позволяет просматривать метрики инфраструктуры, на которой выполняется рабочая нагрузка.

- REVISION HISTORY (История ревизий) — содержит список всех ревизий рабочей нагрузки. Это особенно полезно, если развертывания обновляются часто.
- EVENTS (События) — отображает все события, связанные с рабочей нагрузкой, включая масштабирование, создание и удаление подов, а также возможные ошибки.
- LOGS (Журналы) — содержит логи всех контейнеров, входящих в под.
- APP ERRORS (Ошибки приложения) — показывает ошибки, зафиксированные на уровне приложения.
- YAML — позволяет экспортировать YAML-манифест, содержащий полную конфигурацию развертывания.

Эта информация доступна для всех развертываний во всех кластерах GKE, созданных в рамках проекта.

Раздел Gateways, Services and Ingress

Как можно догадаться из названия, в разделе Gateways, Services and Ingress (Шлюзы, сервисы и входящий трафик) отображаются все сервисы, развернутые во всех кластерах GKE вашего проекта. Чтобы открыть этот раздел, выберите соответствующий пункт в меню консоли.

По умолчанию откроется вкладка GATEWAYS (Шлюзы), однако, поскольку мы еще не создали ни одного шлюза, перейдите на вкладку SERVICES (Сервисы). После загрузки вы увидите экран, подобный показанному на рис. 15.13.

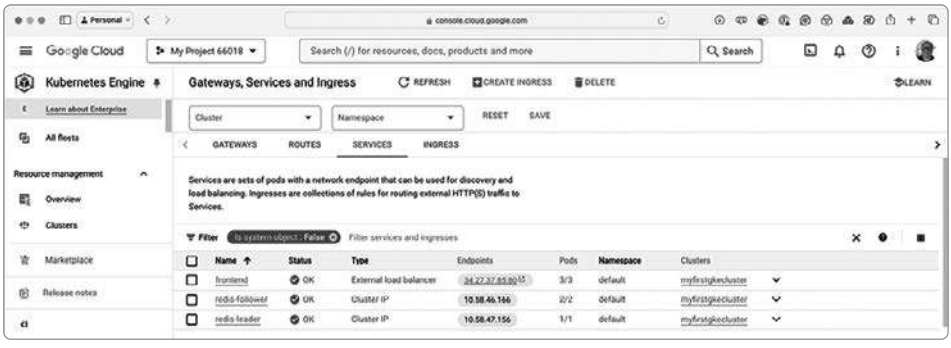

Рис. 15.13. Просмотр сервисов в консоли

Как видите, в списке отображаются три созданных нами сервиса. Сервис frontend имеет тип External load balancer и публичный IP-адрес, в то время как сервисы redis-leader и redis-follower относятся к типу Cluster IP.

Это связано с тем, что из трех сервисов только frontend должен быть доступен извне, так как именно он взаимодействует с двумя сервисами Redis.

При запуске внешнего балансировщика нагрузки в публичном облаке планировщик Kubernetes автоматически обратился к API Google Cloud, запустил балансировщик и организовал его взаимодействие с узлами нашего кластера, открыв доступ к рабочей нагрузке. Данное развертывание доступно извне на порте `80`, а трафик внутри кластера перенаправляется на порт узлов `31740`.

При выборе любого из трех запущенных сервисов в консоли можно получить подробную информацию, содержащуюся на следующих вкладках.

- OVERVIEW (Обзор) — содержит сводную информацию о конфигурации и использовании сервиса.
- DETAILS (Сведения) — предоставляет дополнительные данные о сервисе и ссылку на соответствующий ресурс балансировщика нагрузки, запущенный в рамках проекта Google Cloud.
- EVENTS (События) — отображает события, связанные с работой сервиса, что весьма полезно для диагностики.
- LOGS (Журналы) — дублирует логи, представленные в разделе Workloads (Рабочие нагрузки).
- APP ERRORS (Ошибки приложения) — содержит сведения о зафиксированных ошибках приложения.
- YAML — позволяет экспортировать полную конфигурацию сервиса в виде YAML-манифеста.

Мы рассмотрели базовый сценарий запуска рабочей нагрузки, однако GKE предусматривает множество других параметров и функций. Давайте вкратце разберем возможности, скрывающиеся за остальными пунктами в левом меню.

Другие функции GKE

Платформа GKE поддерживает два типа функций: стандартные и корпоративные. Ниже перечислены стандартные функции, доступные в обычной версии GKE. Корпоративные функции, доступные в составе GKE Enterprise, мы вкратце обсудим в конце этого пункта.

- Clusters (Кластеры) — этот раздел позволяет управлять кластерами. Здесь вы увидите кластер, который мы развернули с помощью CLI GCP, и сможете создать новые, используя веб-интерфейс.
- Workloads (Рабочие нагрузки) — как уже было сказано, этот раздел содержит все рабочие нагрузки, развернутые в ваших кластерах.

- Applications (Приложения) — показывает приложения, установленные из каталога Google Cloud Marketplace (о котором чуть позже).
- Secrets and ConfigMaps (Объекты Secret и ConfigMap) — этот раздел позволяет безопасно управлять объектами Secret и ConfigMap.
- Storage (Хранилище) — позволяет просматривать настроенные объекты PersistentVolumeClaim и управлять классами хранилища (StorageClass).
- Object Browser (Обозреватель объектов) — своеобразный проводник по API Kubernetes и GCP. Он позволяет просматривать доступные API и конечные точки, а также ресурсы и объекты, развернутые в рамках вашего проекта.

Следующие сервисы обеспечивают интеграцию GKE с нативными сервисами Google Cloud.

- Backup for GKE (Резервное копирование для GKE) — позволяет использовать сервис резервного копирования Google Cloud (не входящий в состав GKE) для создания резервных копий рабочих нагрузок и приложений, размещенных в кластере GKE. Лично я в своей практике исхожу из того, что данные в рабочих нагрузках Kubernetes, развернутых в облаке, являются эфемерными. Для запуска служб, нуждающихся в резервном копировании, таких как базы данных, мы используем облачные сервисы с гарантированным SLA, развернутые за пределами моего кластера. Однако если по какой-то причине этот вариант вам не подходит, данный сервис станет отличным решением.
- Gateways, Services and Ingress (Шлюзы, сервисы и входящий трафик) — этот раздел был рассмотрен ранее.
- Network Function Optimizer (Оптимизатор сетевой функции) — позволяет подключить рабочие нагрузки GKE к частным сетям других ресурсов Google Cloud. Это расширяет сеть кластера, позволяя его компонентам использовать приватные подключения к таким сервисам, как облачные базы данных.
- Marketplace (Маркетплейс) — здесь представлен каталог готовых к установке приложений для GKE. Они публикуются различными поставщиками ПО и включают как бесплатные, так и коммерческие решения.

Как отмечалось выше, перечисленные функции доступны в стандартной версии GKE. Ее использование аналогично самостоятельному развертыванию Kubernetes за исключением того, что в данном случае за развертывание узлов и управление ресурсами отвечает Google.

Корпоративная версия GKE Enterprise включает дополнительные функции, востребованные в крупных организациях. Среди них — управляемые сервисные сетки, а также средства обеспечения безопасности, соответствия нормативным требованиям и управления доступом на основе ролей (RBAC). Эти функции предоставляются за дополнительную плату, но являются ключевыми для предприятий, работающих в строго регулируемых отраслях.

Удаление кластера GKE

После завершения работы с кластером его можно удалить, выполнив следующие команды. Первая удаляет созданный ранее сервис, а вторая — сам кластер:

```
$ kubectl delete service frontend
$ gcloud container clusters delete myfirstgkecluster --zone=us-central1-a
```

Процесс удаления кластера займет несколько минут. Очень важно дождаться его полного завершения.

Вместе с кластером должны быть удалены ресурсы, созданные рабочими нагрузками, например балансировщики нагрузки. Тем не менее, чтобы избежать неожиданных расходов, рекомендуется проверить консоль Google Cloud на предмет наличия осиротевших сервисов или ресурсов.

Ранее мы использовали флаг `--zone=us-central1-a`, который указывает на размещение кластера в одной зоне доступности региона US Central. Теперь рассмотрим другие варианты конфигурации кластеров.

Дополнительные сведения об узлах кластера

В конце предыдущего раздела упоминались зоны доступности и регионы. Перед рассмотрением вариантов развертывания кластеров следует разобраться с этими понятиями.

- *Region* (регион) — состоит из нескольких зон. Зоны внутри одного региона соединены сетью с низкой задержкой, что позволяет развертывать высокодоступные и отказоустойчивые рабочие нагрузки.
- *Availability zone* (зона доступности) — зону можно рассматривать как отдельный центр обработки данных внутри региона. Каждая зона имеет собственную сеть и энергоснабжение. Это означает, что, если одна зона выйдет из строя, а ваша рабочая нагрузка распределена между несколькими зонами, ее работа не будет нарушена.

Важно помнить, что не все типы машин доступны в каждой зоне региона. Поэтому перед развертыванием рабочей нагрузки рекомендуется проверить их доступность.

Согласно рекомендациям Google для достижения оптимальной производительности и отказоустойчивости следует распределять рабочие нагрузки по максимально возможному числу зон внутри одного региона.

Однако рабочие нагрузки можно распределить и между несколькими регионами, используя механизм Multi Cluster Ingress. В этом случае нужно учесть увеличение задержки при взаимодействии между регионами, особенно для таких общих сервисов, как базы данных. Ссылка на источник, посвященный Multi Cluster Ingress, приведена в разделе «Дополнительные источники информации» в конце главы.

Теперь вернемся к примеру, рассмотренному в начале главы:

```
$ gcloud container clusters create myfirstgkecluster --num-nodes=2 --zone=us-
central1-a
```

Команда создает два узла в одной зоне региона `us-central1`, а именно в зоне `a`, которую мы указали с помощью флага `--zone`.

Заменим его на флаг `--region`:

```
$ gcloud container clusters create myfirstgkecluster --num-nodes=2 --region=us-
central1
```

После запуска кластера выполните команду:

```
$ kubectl get nodes
```

Вывод будет примерно таким, как показано на рис. 15.14.

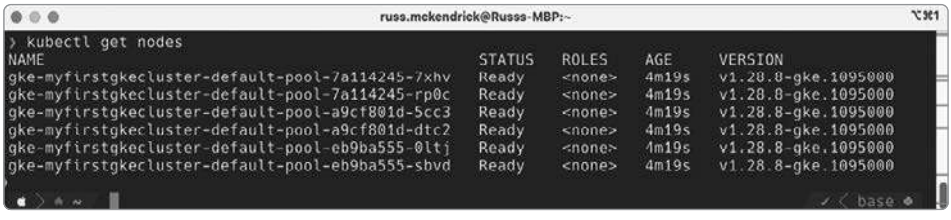

Рис. 15.14. Просмотр узлов, работающих в регионе

Как видите, в каждой зоне региона размещено по два узла, а значит, кластер содержит шесть узлов. Это объясняется тем, что при указании региона по умолчанию кластер распределяется по трем зонам. Такое поведение можно изменить, указав конкретные зоны с помощью параметра `--node-locations`.

Команда для создания такого кластера с нуля будет выглядеть следующим образом — обратите внимание на список зон, разделенных запятыми:

```
$ gcloud container clusters create myfirstgkecluster \
  --num-nodes=2 \
  --region=us-central1 \
  --node-locations us-central1-a,us-central1-b,us-central1-c,us-central1-f
```

Мы по-прежнему используем регион `us-central1`, но теперь кластер развертывается в четырех зонах: `a`, `b`, `c` и `f`.

Поскольку кластер уже запущен, мы можем просто обновить его конфигурацию, добавив новую зону и два дополнительных узла:

```
$ gcloud container clusters update myfirstgkecluster \
  --min-nodes=2 \
  --region=us-central1 \
  --node-locations us-central1-a,us-central1-b,us-central1-c,us-central1-f
```

После выполнения команды `kubectl get nodes` вы увидите результат, аналогичный приведенному на рис. 15.15.

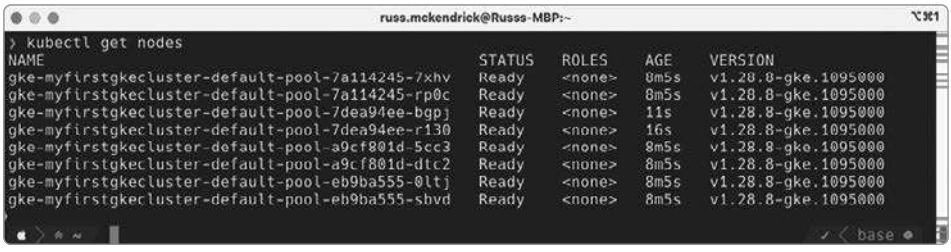

Рис. 15.15. Кластер, работающий в четырех зонах доступности

Как видите, система Google существенно упрощает процесс развертывания кластеров в нескольких зонах. Это позволяет с минимальными усилиями создавать полностью избыточные и отказоустойчивые кластеры. Если учесть сложность, с которой пришлось бы столкнуться при реализации аналогичной архитектуры вручную с использованием виртуальных машин и сетей, удобство такого подхода становится особенно очевидным.

Чтобы удалить кластер, развернутый с флагом `--region`, используйте команду:

```
$ gcloud container clusters delete myfirstgkecluster --region=us-central1
```

На момент написания этой книги двухузловой кластер, созданный в начале главы, обходился примерно в 55 долларов США в месяц, а стоимость восьмиузлового кластера, развернутого в этом примере, составляла около 347 долларов США в месяц. Для получения актуальных расчетов воспользуйтесь калькулятором стоимости услуг Google Cloud, ссылка на который приведена в разделе «Дополнительные источники информации».

На этом мы завершаем рассмотрение возможностей сервиса Google Kubernetes Engine. Перед тем как перейти к обсуждению следующего публичного облачного провайдера, подведем итоги.

Резюме

В этой главе мы рассмотрели историю возникновения Google Cloud Platform (GCP) и сервиса Google Kubernetes Engine (GKE), а затем пошагово разобрали процесс регистрации учетной записи и установки интерфейса командной строки Google Cloud.

После этого мы запустили простой двухузловой кластер с помощью единственной команды, развернули в нем рабочую нагрузку и повзаимодействовали с ней через `kubectl` и консоль Google Cloud.

Затем, используя всего одну команду, мы повторно развернули кластер, задействовав несколько зон доступности. В результате мы получили полностью избыточное и высокодоступное восьмиузловое кластерное решение, работающее в четырех зонах.

Можно с уверенностью сказать, что компания Google проделала отличную работу, сделав процесс развертывания и управления сложной инфраструктурой максимально простым и быстрым.

После развертывания рабочих нагрузок вы можете управлять ими так же, как если бы кластер находился в другой среде, поскольку работа в GCP не требует никаких особых изменений в подходах к управлению.

В следующей главе мы развернем кластер Kubernetes на платформе AWS с использованием управляемого сервиса Amazon Elastic Kubernetes Service (EKS) от компании Amazon.

Дополнительные источники информации

Ниже приведены ссылки на источники, содержащие дополнительную информацию по темам, рассмотренным в этой главе.

- Сервис Google Kubernetes Engine: https://cloud.google.com/kubernetes-engine/.
- Документация по Google Kubernetes Engine: https://cloud.google.com/kubernetes-engine/docs.
- Калькулятор стоимости услуг Google Cloud: https://cloud.google.com/products/calculator.
- Приложение Guestbook: https://github.com/GoogleCloudPlatform/kubernetes-engine-samples/tree/main/quickstarts/guestbook.
- Комикс о Kubernetes от Google Cloud: https://cloud.google.com/kubernetes-engine/kubernetes-comic.
- Региональные кластеры: https://cloud.google.com/kubernetes-engine/docs/concepts/regional-clusters.
- Механизм Multi Cluster Ingress: https://cloud.google.com/kubernetes-engine/docs/concepts/multi-cluster-ingress.

16 Запуск кластера на платформе Amazon Web Services с помощью Amazon Elastic Kubernetes Service

В предыдущей главе мы запустили кластер Kubernetes в общедоступном облаке, познакомившись с одной из трех крупнейших публичных облачных платформ — *Google Cloud Platform (GCP)*.

Теперь пришло время поговорить о сервисе *Amazon Elastic Kubernetes Service (EKS)* и платформе *Amazon Web Services (AWS)*.

В этой главе вы узнаете, как создать учетную запись AWS, установить необходимые инструменты в macOS, Windows и Linux, а также научитесь запускать кластеры Amazon EKS и взаимодействовать с ними.

Технические требования

Для выполнения инструкций, приведенных в этой главе, вам потребуется учетная запись AWS с указанным способом оплаты.

Следование инструкциям, приведенным в этой главе, сопряжено с финансовыми расходами. Поэтому крайне важно удалить все созданные ресурсы после завершения работы. Указанные в тексте цены были актуальны на момент написания главы; перед созданием ресурсов рекомендуется проверить текущие тарифы.

Знакомство с Amazon Web Services и Amazon Elastic Kubernetes Service

Amazon Web Services (AWS) — это одна из первых публичных облачных платформ. На момент написания книги (июнь 2024 года) она лидирует по доле рынка, которая составляет 31 %. За ней следует Microsoft Azure с 25 %, а третье место занимает Google Cloud Platform (GCP) с 11 %.

Платформа Amazon Web Services

Как вы, вероятно, уже догадались, платформа AWS принадлежит компании Amazon. Ее история началась в 2000 году, когда Amazon экспериментировала с облачными сервисами, создавая *интерфейсы прикладного программирования (API)* для своих розничных партнеров. Эти эксперименты в итоге привели к появлению AWS — одного из мировых лидеров в области облачных вычислений, чьи решения сейчас используют как малые, так и крупные предприятия, работающие в самых разных отраслях.

В ходе своих экспериментов компания Amazon осознала необходимость в создании более совершенной и стандартизированной инфраструктурной платформы — не только для размещения уже существующих сервисов, но и для обеспечения их масштабируемости по мере роста числа розничных подразделений, активно использующих эти программные решения.

В 2003 году Крис Пинкэм и Бенджамин Блэк подготовили технический документ, который был одобрен Джеффом Безосом в 2004 году. В нем описывалась инфраструктурная платформа, позволяющая программно разворачивать вычислительные ресурсы и ресурсы хранилища.

Первое публичное упоминание AWS состоялось в конце 2004 года. Однако тогда этот термин обозначал набор инструментов и API, предназначенных для взаимодействия сторонних разработчиков с розничным каталогом товаров Amazon, а не то, что сегодня понимается под облачной платформой Amazon Web Services.

Полноценный запуск AWS состоялся лишь в 2006 году. В марте был выпущен сервис *Simple Storage Service (S3)*[1], позволявший разработчикам записывать и предоставлять отдельные файлы, используя веб-API вместо локальной или удаленной файловой системы.

Следующим стал *Amazon Simple Queue Service (SQS)* — распределенная система обмена сообщениями, которая также могла управляться и использоваться разработчиками посредством API. Этот сервис входил в первоначальный набор инструментов AWS.

Наконец, в том же 2006 году для существующих клиентов AWS вышла бета-версия *Amazon Elastic Compute Cloud (Amazon EC2)*, позволявшая программно запускать вычислительные ресурсы Amazon EC2 с помощью API, разработанных Amazon.

Эти сервисы легли в основу публичной облачной платформы Amazon, услуги которой компания могла не только использовать для собственных нужд, но и предлагать другим организациям и частным пользователям вроде нас с вами.

---

[1] Есть совместимые открытые аналоги, которые можно развернуть на своих ресурсах самостоятельно, к примеру openminio (https://github.com/openminio/openminio).

В середине 2024 года на платформе AWS было доступно уже более 200 сервисов, работающих более чем в 125 физических центрах обработки данных, расположенных в 39 регионах, совокупная площадь которых превышает 3,5 млн м^2.

Работа всех этих сервисов основана на ключевых принципах, изложенных в документе 2003 года. Каждый сервис является программно определяемым, то есть для его запуска, настройки, использования и завершения работы достаточно выполнить простые API-запросы.

Вероятно, вы заметили, что названия сервисов начинаются с префиксов Amazon или AWS. Это не случайно: сервисы с префиксом Amazon обычно представляют собой самостоятельные решения, а сервисы с префиксом AWS, как правило, предназначены для интеграции с другими сервисами Amazon.

Времена, когда развертывание сервиса требовало оформления заказа, сборки и передачи продукта заказчику, ушли в прошлое. То, что раньше занимало недели или месяцы, сегодня делается за считаные секунды.

Поскольку обсудить все 200 с лишним сервисов нереально (для этого понадобилась бы целая серия книг), в этой главе мы сосредоточимся на рассмотрении только одного из них.

Сервис Amazon Elastic Kubernetes Service

Хотя компания Amazon была пионером на рынке публичных облачных вычислений, собственный управляемый сервис Kubernetes она запустила одной из последних. Сервис EKS (Amazon Elastic Kubernetes Service) был анонсирован в конце 2017 года и стал общедоступным в июне 2018 года, начав свою работу в регионах US East (Северная Вирджиния) и US West (Орегон).

Данный сервис тесно интегрирован с экосистемой AWS и использует ее возможности, в числе которых:

- *AWS Identity and Access Management (IAM)* — служба управления идентификацией и доступом, позволяющая контролировать как пользовательский, так и программный доступ к другим сервисам AWS;
- *Amazon Route 53* — служба *системы доменных имен* (DNS), предоставляемая Amazon. Сервис Amazon EKS может использовать ее как источник DNS-записей для кластера, что упрощает обнаружение сервисов и управление маршрутизацией внутри него;
- *Amazon Elastic Block Store (EBS)* — предоставляет постоянное блочное хранилище для контейнеров, работающих в кластерах EKS, и вычислительных экземпляров EC2;
- *EC2 Auto Scaling* — технология автоматического масштабирования кластера, позволяющая динамически изменять количество экземпляров EC2;

- *Multi-Availability Zones (AZs)* — полезная функция, позволяющая повысить отказоустойчивость и доступность развертывания за счет распределения управляющего слоя Amazon EKS и узлов кластера по нескольким зонам доступности внутри одного региона.

Перед запуском кластера Amazon EKS нам нужно установить и настроить несколько вспомогательных инструментов.

Подготовка локальной среды

Нам предстоит установить два инструмента командной строки. Однако перед этим следует кратко рассмотреть процесс регистрации новой учетной записи AWS. Если у вас уже есть учетная запись AWS, этот шаг можно пропустить и сразу перейти к подразделу «Установка интерфейса командной строки AWS».

Регистрация учетной записи AWS

Процесс регистрации учетной записи AWS достаточно прост.

1. Перейдите на сайт https://aws.amazon.com/ и нажмите кнопку Create an AWS account (Создать учетную запись AWS) в правом верхнем углу страницы.

Amazon предоставляет новым пользователям бесплатный доступ к ряду сервисов на 12 месяцев в рамках программы AWS Free Tier, подробнее о которой можно узнать по адресу https://aws.amazon.com/free/.

2. Заполните начальную форму, указав адрес электронной почты, который будет использоваться для восстановления доступа к учетной записи и выполнения базовых административных действий. Задайте имя учетной записи AWS. Его можно будет изменить после регистрации в разделе с настройками. Затем вам потребуется подтвердить адрес электронной почты. После подтверждения вам будет предложено задать пароль для учетной записи суперпользователя (root).
3. После ввода пароля нажмите кнопку Continue (Продолжить) и следуйте инструкциям на экране. Процесс включает пять шагов, в ходе которых вам потребуется подтвердить платежные данные и личность (это делается через автоматический телефонный звонок).

После создания и активизации учетной записи вы получите уведомление о готовности к работе с сервисами AWS. Обычно оно приходит сразу, но в некоторых случаях активизация аккаунта может занять до 48 часов.

Теперь установим инструменты командной строки, необходимые для запуска кластера Amazon EKS.

Установка интерфейса командной строки AWS

Следующим шагом является установка *интерфейса командной строки (CLI)* AWS. Как и в предыдущей главе, мы рассмотрим процесс его установки в Windows, Linux и macOS. Начнем с macOS.

Установка в macOS

Для установки AWS CLI в macOS с помощью Homebrew достаточно выполнить одну простую команду:

```
$ brew install awscli
```

После завершения установки выполните команду:

```
$ aws –version
```

В выводе отобразится номер версии AWS CLI вместе с информацией о вспомогательных службах, необходимых для его работы, как показано на следующем снимке экрана (рис. 16.1).

Рис. 16.1. Проверка версии AWS CLI

После успешной установки вы можете перейти к подразделу «Настройка интерфейса командной строки AWS».

Установка в Linux

Хотя пакеты AWS CLI доступны для большинства дистрибутивов Linux, самым простым способом его установки являются загрузка и запуск официального установщика.

В представленной ниже инструкции предполагается, что пакеты curl и unzip уже установлены. Если у вас их нет, установите их с помощью пакетного менеджера вашего дистрибутива. Например, в Ubuntu для этого можно использовать команду sudo apt-get install unzip curl.

Чтобы загрузить и установить AWS CLI, выполните следующие команды:

```
$ curl "https://awscli.amazonaws.com/awscli-exe-linux-x86_64.zip" -o "awscliv2.
zip"
$ unzip awscliv2.zip
$ sudo ./aws/install
```

После установки вы можете выполнить команду `aws --version`, чтобы получить результат, аналогичный показанному в пункте «Установка в macOS». Далее мы обсудим установку AWS CLI в Windows.

Установка в Windows

В системе Windows AWS CLI можно установить с помощью пакетного менеджера, как и в macOS. Мы воспользуемся менеджером Chocolatey и следующей командой:

```
$ choco install awscli
```

После установки и выполнения приведенной ниже команды вы получите такой же вывод, как в случае с macOS, если не считать различий в версиях операционной системы и языка Python:

```
$ aws --version
```

После успешной установки интерфейса командной строки можно приступать к его настройке.

Настройка интерфейса командной строки AWS

После установки AWS CLI и проверки его работоспособности с помощью команды `aws --version` необходимо связать локальную установку CLI с вашей учетной записью AWS. Для этого войдите в консоль управления AWS по адресу http://console.aws.amazon.com/.

После входа в консоль введите IAM в поле поиска, расположенное в левом верхнем углу страницы рядом с кнопкой Services (Сервисы). Затем выберите ссылку IAM Identity Center в результатах поиска, чтобы перейти на страницу Центра идентификации IAM.

Нам нужно создать пользователя с правом на получение программного доступа. Для этого выполните следующие шаги.

1. В зависимости от даты создания или уровня доступа вашей учетной записи AWS вам может потребоваться включить сервис IAM Identity Center (Центр идентификации IAM). В случае одного из авторов учетная запись

AWS используется для личных проектов, поэтому после нажатия кнопки Enable (Включить) он выбрал вариант Enable in only this AWS account (Включить только в этой учетной записи AWS) вместо рекомендуемого Enable with AWS Organizations (Включить с AWS Organizations). Он сделал это, потому что ему не нужно управлять несколькими учетными записями AWS в рамках одной организации. Следуйте инструкциям на экране, чтобы активизировать сервис.

2. После активизации сервиса IAM Identity Center (Центр идентификации IAM) необходимо создать пользователя, имеющего только программный доступ. Для этого вернитесь к строке поиска в верхнем меню, снова введите IAM, но на этот раз выберите пункт Manage access to AWS resources (Управление доступом к ресурсам AWS). После загрузки страницы выберите пункт Users (Пользователи) в разделе Access Management (Управление доступом) меню слева и нажмите кнопку Create user (Создать пользователя).
3. Введите имя пользователя ekscluster и убедитесь, что флажок Provide user access to the AWS Management Console — *optional* (Предоставить пользователю доступ к консоли управления AWS — *не обязательно*) снят. Затем нажмите кнопку Next (Далее) (рис. 16.2). Мы обсудим данную опцию после запуска кластера.

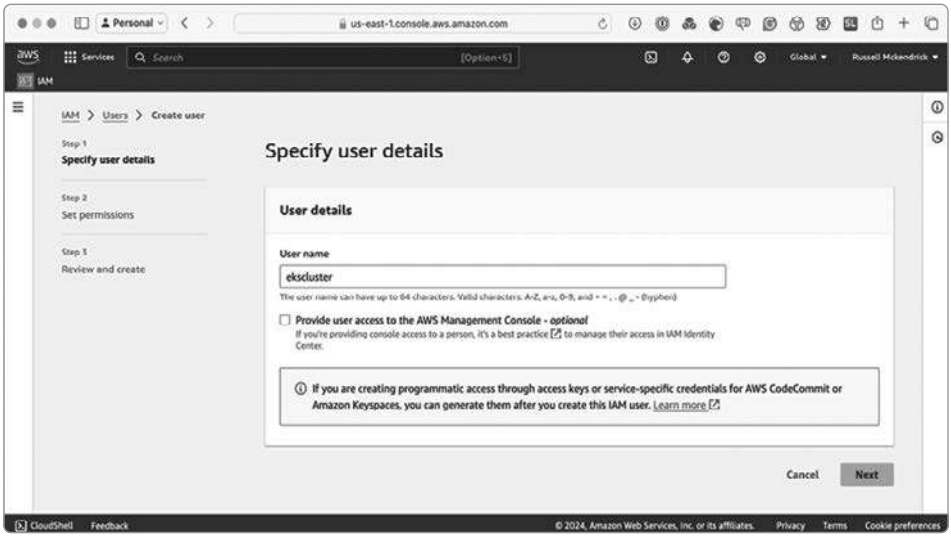

Рис. 16.2. Добавление пользователя

4. Вместо создания новой группы мы предоставим пользователю доступ на основе существующей политики. Для этого выберите Attach existing policies directly (Присоединить существующие политики напрямую), отметьте политику AdministratorAccess и нажмите кнопку Next (Далее) (рис. 16.3).

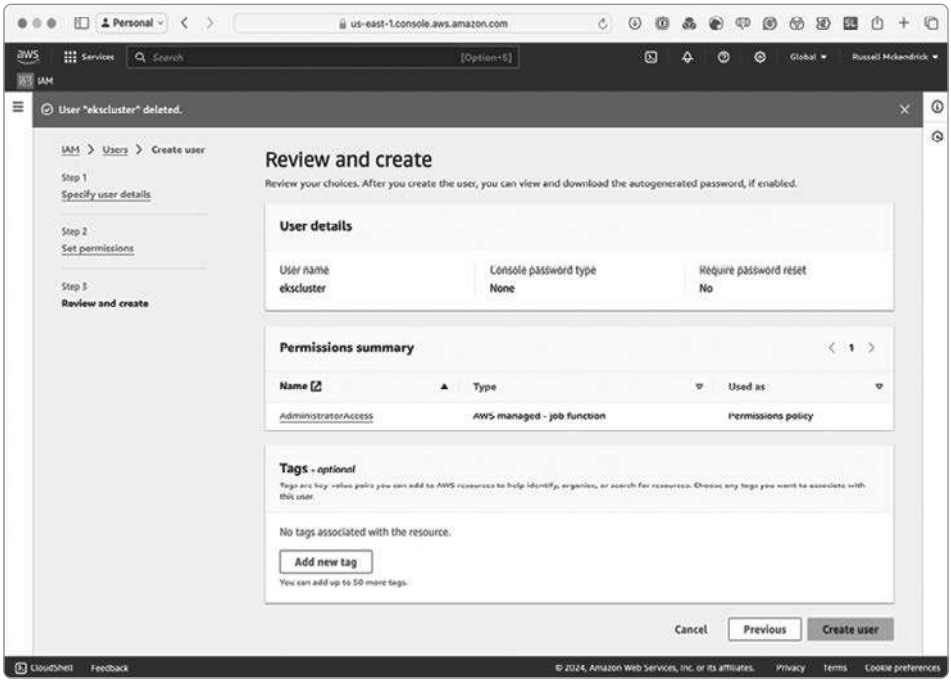

Рис. 16.3. Предоставление прав доступа

Поскольку мы проводим тестирование не в рабочей среде и планируем удалить созданного пользователя в конце главы, мы можем выбрать политику, предоставляющую достаточно широкие права доступа. Если же вы развертываете кластер в среде, приближенной к рабочей, то для корректной настройки политик и разрешений рекомендуется обратиться к официальной документации AWS.

5. После проверки введенных данных нажмите кнопку Create user (Создать пользователя). Затем выберите пользователя ekscluster из списка, перейдите на вкладку Security credentials (Учетные данные для обеспечения безопасности) и нажмите кнопку Create access key (Создать ключ доступа) в разделе Access keys (Ключи доступа). Выберите вариант Command Line Interface (CLI) (Интерфейс командной строки) и продолжите создание ключа, последовательно нажав кнопки Next (Далее) и Create access key (Создать ключ доступа). После генерации ключа нажмите кнопку Download .csv file (Скачать CSV-файл), чтобы скачать файл с учетными данными, и завершите процесс нажатием кнопки Done (Готово).

Сохраните загруженный CSV-файл в надежном месте, так как он содержит ключи доступа к вашей учетной записи AWS.

Вернитесь в терминал и создайте профиль по умолчанию, выполнив команду:

```
$ aws configure
```

Программа последовательно запросит такие параметры:

- AWS access key identifier (ID) — идентификатор ключа доступа из загруженного CSV-файла;
- AWS secret access key — секретный ключ доступа из того же файла;
- Default region name — регион по умолчанию; мы указали `us-west-2`;
- Default output format — формат вывода по умолчанию; мы оставили это поле пустым.

Чтобы проверить корректность настроек, выполните команду:

```
$ aws ec2 describe-regions
```

В результате будет выведен список доступных регионов AWS. Пример вывода показан ниже (рис. 16.4).

Рис. 16.4. Проверка работы AWS CLI

Теперь, когда AWS CLI установлен и настроен, установим второй инструмент командной строки, необходимый для запуска кластера Amazon EKS.

Установка eksctl — официального CLI для Amazon EKS

Хотя запустить кластер Amazon EKS можно и с помощью AWS CLI, этот процесс довольно сложен и состоит из множества шагов. Чтобы упростить задачу, компания Weaveworks разработала инструмент командной строки `eksctl`, который автоматически создает шаблон AWS CloudFormation, а затем развертывает на его основе кластер.

К сожалению, в начале 2024 года Weaveworks прекратила коммерческую деятельность. Но перед этим компания передала управление проектом команде AWS.

AWS CloudFormation — это сервис Amazon для управления инфраструктурой как кодом (Infrastructure as Code, IaC), который позволяет описывать ресурсы AWS в виде конфигурационных шаблонов, чтобы затем развертывать их в разных учетных записях или многократно в рамках одной и той же учетной записи. Это особенно полезно в ситуациях, когда необходимо регулярно воссоздавать окружение, например, в рамках процесса непрерывной интеграции (continuous integration, CI).

Как вы, вероятно, уже догадались, установка этого инструмента в macOS и Windows выполняется по схеме, описанной ранее. В macOS выполните команду:

```
$ brew install eksctl
```

В Windows выполните команду:

```
$ choco install eksctl
```

В Linux процесс установки `eksctl` требует выполнения следующих команд:

```
$ PLATFORM=$(uname -s)_$(uname -m)
$ curl -sLO "https://github.com/eksctl-io/eksctl/releases/latest/download/\
eksctl_$PLATFORM.tar.gz"
$ tar -xzf eksctl_$PLATFORM.tar.gz -C /tmp && rm eksctl_$PLATFORM.tar.gz
$ sudo mv /tmp/eksctl /usr/local/bin
```

После установки проверьте версию утилиты с помощью команды:

```
$ eksctl version
```

Теперь мы готовы к запуску кластера Amazon EKS.

Запуск кластера Amazon EKS

После установки всех необходимых инструментов можно приступить к развертыванию кластера Amazon EKS. Когда кластер будет готов, мы сможем взаимодействовать с ним и запускать рабочие нагрузки, так же как мы делали это в главе 15, посвященной Google Kubernetes Engine (GKE).

Поскольку нам нужен лишь тестовый кластер Amazon EKS для выполнения команд, мы воспользуемся настройками по умолчанию, встроенными в ути-

литу `eksctl`. В результате будет создан кластер Amazon EKS со следующими характеристиками.

- Регион: *us-west-1*.
- Рабочие узлы: два узла, развернутые на экземплярах (инстансах) типа *m5.large*.
- Образ (AMI): официальный *AMI (Amazon Machine Image)* для AWS EKS.
- Сеть: *Amazon Virtual Private Cloud (VPC)*.
- Имя кластера: генерируется автоматически.

Чтобы запустить кластер, выполните команду:

```
$ eksctl create cluster
```

Процесс создания кластера может занять до 30 минут. В это время можно отвлечься на чашечку кофе или на проверку электронной почты.

После запуска команды в терминале отобразится базовая информация о версии `eksctl` и используемом регионе:

```
[i] eksctl version 0.180.0-dev+763027060.2024-05-29T21:36:10Z
[i] using region us-west-2
```

Затем вы увидите информацию о сети и зонах доступности (AZ), в которых будут развернуты ресурсы:

```
[i] setting availability zones to [us-west-2d us-west-2b us-west-2c]
[i] subnets for us-west-2d - public:192.168.0.0/19 private:192.168.96.0/19
[i] subnets for us-west-2b - public:192.168.32.0/19 private:192.168.128.0/19
[i] subnets for us-west-2c - public:192.168.64.0/19 private:192.168.160.0/19
```

Далее появятся сведения о версии используемого AMI-образа и поддерживаемой им версии Kubernetes:

```
[i] nodegroup "ng-11c87ff4" will use "[AmazonLinux2/1.29]"
[i] using Kubernetes version 1.29
```

Затем начнется процесс развертывания кластера:

```
[i] creating EKS cluster "hilarious-wardrobe-1717847351" in "us-west-2" region
with managed nodes
[i] will create 2 separate CloudFormation stacks for cluster itself and the
initial managed nodegroup
[i] if you encounter any issues, check CloudFormation console or try
```

```
'eksctl utils describe-stacks --region=us-west-2 --cluster=hilarious-
wardrobe-1717847351'
[i] Kubernetes API endpoint access will use default of {publicAccess=true,
privateAccess=false} for cluster "hilarious-wardrobe-1717847351" in "us-west-2"
```

В этом примере кластер получил случайное имя `hilarious-wardrobe-1717847351`, которое используется на протяжении всего процесса сборки. По умолчанию ведение журналов в сервисе CloudWatch отключено:

```
[i] CloudWatch logging will not be enabled for cluster
"hilariouswardrobe-1717847351" in "us-west-2"
[i] you can enable it with 'eksctl utils update-cluster-logging --enable-types={
SPECIFY-YOUR-LOG-TYPES-HERE (e.g. all)} --region=us-west-2
--cluster=hilarious-wardrobe-1717847351'
```

После этого начинается основной этап — создание управляющего слоя и развертывание узлов:

```
[i] 2 sequential tasks: { create cluster control plane
"hilariouswardrobe-1717847351",
     2 sequential sub-tasks: { wait for control plane to become ready,
                               create managed nodegroup "ng-11c87ff4", } }
[i] building cluster stack "eksctl-hilarious-wardrobe-1717847351-cluster"
[i] waiting for CloudFormation stack "eksctl-hilarious-wardrobe-1717847351-
cluster"
[i] building managed nodegroup stack "eksctl-hilarious-wardrobe-1717847351-
nodegroup-ng-11c87ff4"
[i] deploying stack "eksctl-hilarious-wardrobe-1717847351-nodegroup-ng-11c87ff4"
[i] waiting for CloudFormation stack "eksctl-hilarious-wardrobe-1717847351-
nodegroup-ng-11c87ff4"
[i] waiting for the control plane to become ready
```

Когда все ресурсы будут созданы, программа автоматически загрузит учетные данные кластера и настроит утилиту `kubectl`:

```
[✔] saved kubeconfig as "/Users/russ.mckendrick/.kube/config"
[i] no tasks
[✔] all EKS cluster resources for "hilarious-wardrobe-1717847351" have been
created
```

Финальным этапом является ожидание перехода рабочих узлов в состояние готовности:

```
2024-06-08 13:03:34 [✔] created 0 nodegroup(s) in cluster
"hilariouswardrobe-1717847351"

2024-06-08 13:03:35 [i] node "ip-192-168-34-120.us-west-2.compute.internal"
is ready
```

```
2024-06-08 13:03:35 [i] node "ip-192-168-67-233.us-west-2.compute.internal"
is ready

2024-06-08 13:03:35 [i] waiting for at least 2 node(s) to become ready
in "ng-11c87ff4"
2024-06-08 13:03:35 [i] nodegroup "ng-11c87ff4" has 2 node(s)

2024-06-08 13:03:35 [i] node "ip-192-168-34-120.us-west-2.compute.internal"
is ready
2024-06-08 13:03:35 [i] node "ip-192-168-67-233.us-west-2.compute.internal"
is ready

2024-06-08 13:03:35 [✔] created 1 managed nodegroup(s) in cluster
"hilariouswardrobe-1717847351"
```

Когда оба узла зарегистрируются и будут готовы к работе, система выведет сообщение о том, что кластер полностью развернут и готов к использованию:

```
2024-06-08 13:03:36 [i] kubectl command should work with "/Users/russ.
mckendrick/.kube/config", try 'kubectl get nodes'
2024-06-08 13:03:36 [✔] EKS cluster "hilarious-wardrobe-1717847351"
in "us-west-2" region is ready
```

После этого можно выполнить команду `kubectl get nodes`. В результате должен отобразиться список из двух узлов, составляющих кластер, как показано на следующем изображении (рис. 16.5).

Рис. 16.5. Просмотр списка узлов, составляющих кластер Amazon EKS

Теперь, когда кластер успешно создан и запущен, можно перейти к развертыванию той же рабочей нагрузки, что и в примере с кластером *Google Kubernetes Engine (GKE)*.

Развертывание рабочей нагрузки и взаимодействие с кластером

В главе 15 мы использовали пример Guestbook из репозитория GCP GKE на GitHub. В этом разделе мы сначала развернем данное приложение, а затем изучим веб-консоль AWS. Итак, приступим к развертыванию Guestbook.

Развертывание рабочей нагрузки

Хотя наш кластер запущен на платформе AWS с помощью сервиса EKS, для развертывания рабочей нагрузки мы будем использовать те же YAML-манифесты, что и в случае с GKE, применив локальную утилиту `kubectl`. Для этого выполните следующие шаги.

1. Сначала создайте объекты Deployment и Service для Redis Leader, выполнив две команды:

```
$ kubectl apply -f https://raw.githubusercontent.com/GoogleCloudPlatform/
kubernetes-engine-samples/main/quickstarts/guestbook/redis-leader-
deployment.yaml
$ kubectl apply -f https://raw.githubusercontent.com/GoogleCloudPlatform/
kubernetes-engine-samples/main/quickstarts/guestbook/redis-leader-service.
yaml
```

2. Затем создайте объекты Deployment и Service для Redis Follower:

```
$ kubectl apply -f https://raw.githubusercontent.com/GoogleCloudPlatform/
kubernetes-engine-samples/main/quickstarts/guestbook/redis-follower-
deployment.yaml
$ kubectl apply -f https://raw.githubusercontent.com/GoogleCloudPlatform/
kubernetes-engine-samples/main/quickstarts/guestbook/redis-follower-service.
yaml
```

3. После запуска Redis Leader и Follower создайте развертывание и сервис `frontend`, выполнив следующие команды:

```
$ kubectl apply -f https://raw.githubusercontent.com/GoogleCloudPlatform/
kubernetes-engine-samples/main/quickstarts/guestbook/frontend-deployment.
yaml
$ kubectl apply -f https://raw.githubusercontent.com/GoogleCloudPlatform/
kubernetes-engine-samples/main/quickstarts/guestbook/frontend-service.yaml
```

4. Через несколько минут можно будет выполнить следующую команду для получения сведений о запущенном сервисе. Ее вывод должен содержать информацию о способе получения доступа к рабочей нагрузке:

```
$ kubectl get service frontend
```

На этот раз результат немного отличается от вывода, который мы получили при запуске рабочей нагрузки в GKE (рис. 16.6).

Как видите, вместо *IP-адреса* мы получаем *URL-адрес*. Скопируйте его в адресную строку браузера.

Поскольку мы использовали те же команды и конфигурационные файлы, вы увидите уже знакомое веб-приложение *Guestbook*, изображенное на рис. 16.7.

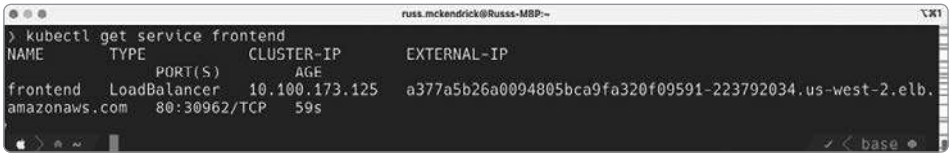

Рис. 16.6. Получение информации о сервисе frontend

Рис. 16.7. Приложение Guestbook с несколькими записями

Теперь, когда рабочая нагрузка запущена, посмотрим, какие сведения о нашем кластере можно получить через консоль AWS.

Изучение консоли AWS

В этом разделе мы исследуем развернутую рабочую нагрузку с помощью консоли AWS. Сначала войдите в консоль по адресу https://console.aws.amazon.com/. После входа выберите пункт US West (Oregon) us-west-2 в списке регионов, расположенном в правом верхнем углу экрана рядом с вашим именем пользователя (рис. 16.8).

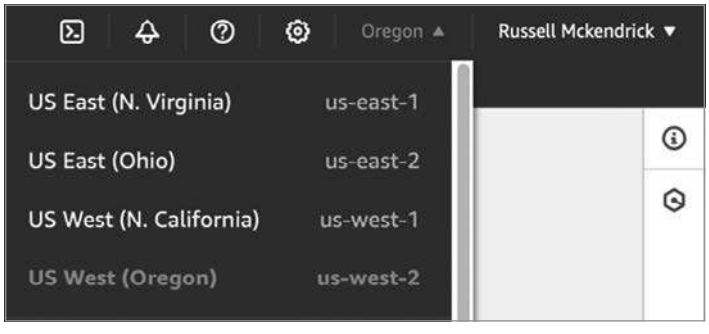

Рис. 16.8. Выбор нужного региона

После выбора региона введите в поисковой строке вверху справа Elastic Kubernetes Service и выберите этот сервис (он должен быть указан первым в результатах поиска), чтобы перейти на страницу EKS для заданного региона (рис. 16.9).

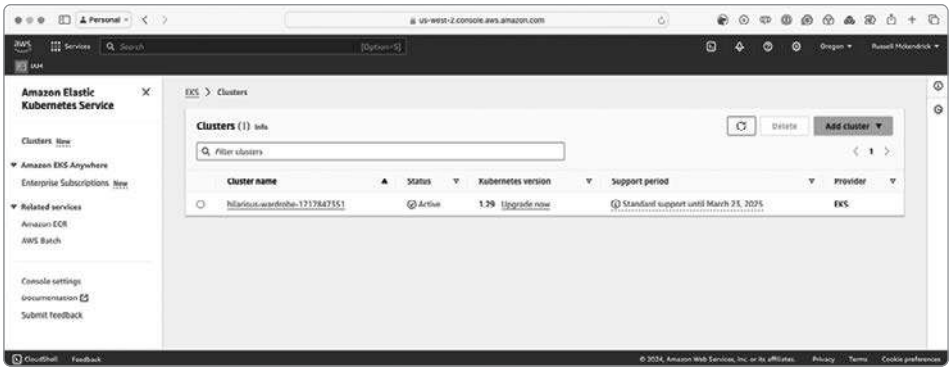

Рис. 16.9. Наш кластер EKS в консоли AWS

Все выглядит нормально, однако если вы щелкнете на имени своего кластера, то увидите страницу, похожую на приведенную рис. 16.10.

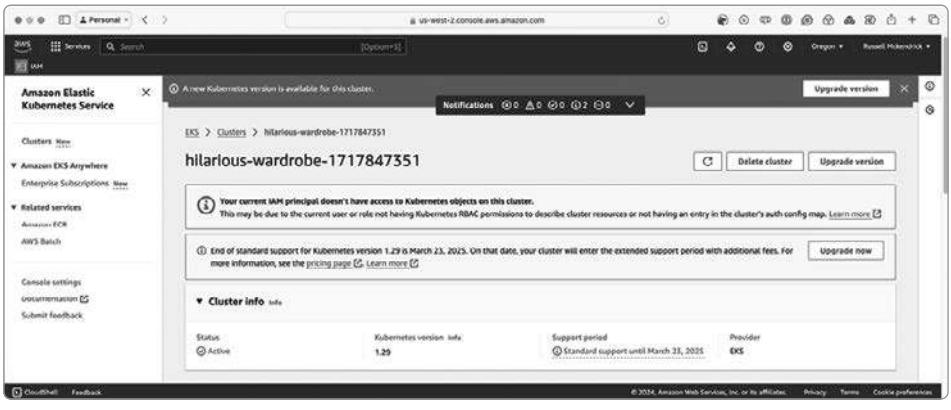

Рис. 16.10. Доступ запрещен!

Давайте разберемся в причине этого ограничения. Вероятно, вы думаете: «Я же вошел в систему от имени основного пользователя, который должен иметь полный доступ».

Однако дело в том, что при создании нашего кластера инструмент `eksctl` предоставил права на взаимодействие с кластером через сервисы AWS пользователю ekscluster (которого мы создали ранее в главе). Другими словами, AWS CLI был

настроен на работу от имени этого пользователя, а не того, от имени которого вы вошли в консоль.

Это означает, что для просмотра рабочих нагрузок и других ресурсов в консоли AWS необходимо войти в систему от имени пользователя ekscluster. Чтобы это сделать, вернитесь в раздел IAM, перейдите на страницу Users (Пользователи) и выберите пользователя ekscluster. Откройте вкладку Security credentials (Учетные данные для обеспечения безопасности) и нажмите кнопку Enable console access (Предоставить доступ через консоль) (рис. 16.11).

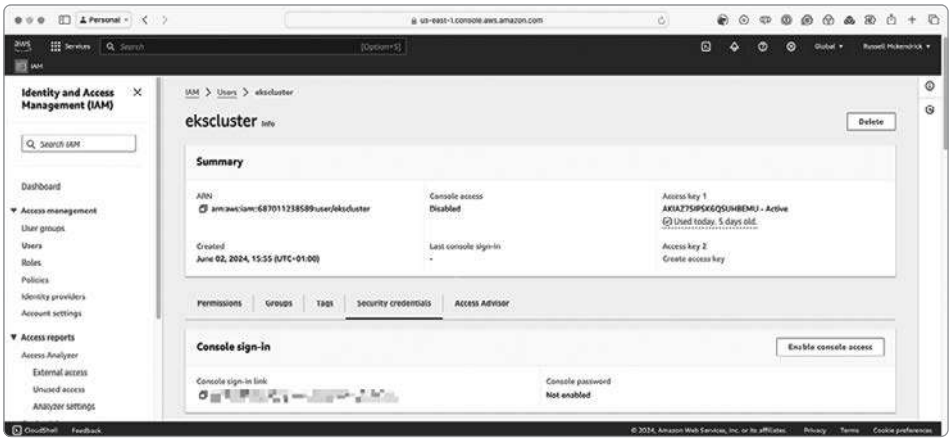

Рис. 16.11. Предоставление доступа через консоль пользователю ekscluster

Выберите вариант Autogenerated password (Автоматически сгенерированный пароль), предоставьте доступ и скачайте CSV-файл с учетными данными, как делали это ранее.

После этого выйдите из консоли AWS, откройте загруженный CSV-файл и перейдите по указанному в нем URL-адресу, позволяющему IAM-пользователям (вроде созданного нами ранее) получать доступ к вашей учетной записи. Введите имя пользователя и пароль из файла CSV.

После входа в консоль вернитесь на страницу EKS и снова выберите свой кластер. На этот раз сообщение об отсутствии прав доступа не появится.

При первом открытии страницы кластера в консоли вы увидите несколько вкладок.

- Overview (Обзор) — содержит сведения о версии Kubernetes, конечной точке, статусе кластера, дате и времени его создания и т. д.
- Resources (Ресурсы) — отображает информацию об узлах, подах, пространствах имен и рабочих нагрузках.

- Compute (Вычисления) — содержит сведения об узлах, группах узлов и связанных с кластером профилях Fargate.
- Networking (Сеть) — содержит информацию о конфигурации VPC.
- Add-ons (Дополнения) — содержит список установленных и доступных для кластера дополнений.
- Access (Доступ) — отображает роли IAM, словарь конфигурации (ConfigMap) `aws-auth` и привязки ролей Kubernetes RBAC.
- Observability (Наблюдаемость) — позволяет настраивать и просматривать логи, метрики и последние события.
- Upgrade insights (Обновления) — показывает доступные версии Kubernetes и результаты проверки совместимости.
- Update history (История обновлений) — отображает историю обновлений кластера и групп узлов.
- Tags (Теги) — отображает и позволяет управлять тегами, связанными с кластером EKS.

Сведения об узлах кластера показаны на рис. 16.12.

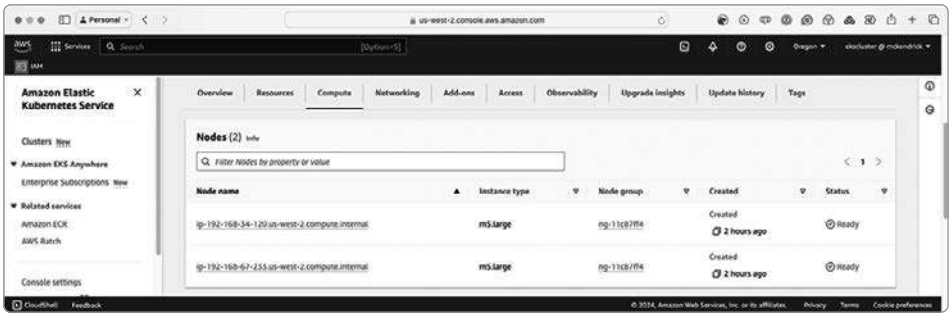

Рис. 16.12. Просмотр сведений о двух узлах кластера на вкладке Compute (Вычисления)

Чтобы увидеть запущенную нами рабочую нагрузку, откройте вкладку Resources (Ресурсы), выберите пункт Deployments (Развертывания) и отфильтруйте результаты по рабочему пространству default (рис. 16.13).

При щелчке на конкретном развертывании отображается информация о нем, включая сведения о подах, конфигурации и т. д. Однако в процессе исследования консоли AWS вы можете заметить, что она предоставляет лишь основную информацию о сервисах, без графиков, журналов и расширенных сведений о рабочих нагрузках, так как ее возможности ограничены отображением данных, получаемых напрямую от системы Kubernetes.

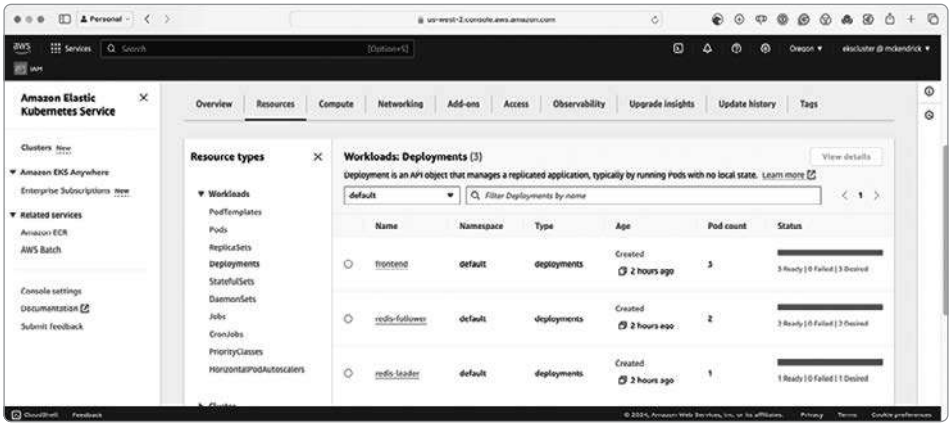

Рис. 16.13. Просмотр запущенной рабочей нагрузки

Если перейти со страницы сервиса EKS в раздел EC2 консоли AWS, можно увидеть два запущенных узла, как показано на рис. 16.14.

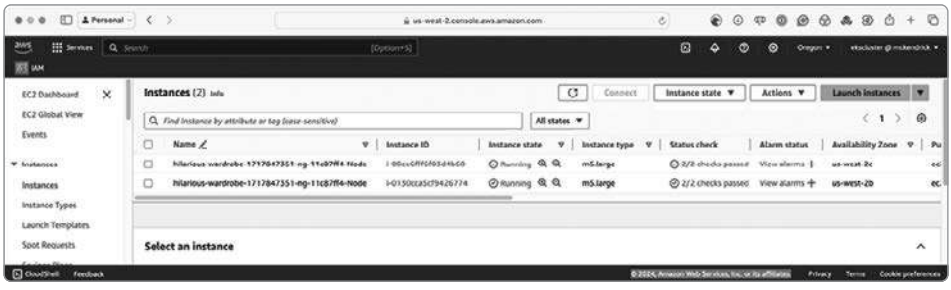

Рис. 16.14. Просмотр вычислительных экземпляров EC2

Здесь содержатся подробные сведения о вычислительных экземплярах, например уровень использования ресурсов процессора, оперативной памяти и сети. Однако эта информация относится к самим экземплярам EC2, а не к запущенным рабочим нагрузкам Kubernetes.

Если выбрать пункт Load Balancing ▸ Load Balancers (Балансировка нагрузки ▸ Балансировщики нагрузки) в меню слева, вы увидите эластичный балансировщик нагрузки, который был автоматически создан и настроен при развертывании фронтенд-сервиса (рис. 16.15).

Еще одним сервисом AWS, который мы используем, является AWS CloudFormation. Чтобы перейти к нему, введите CloudFormation в строке поиска сервисов и откройте соответствующую страницу.

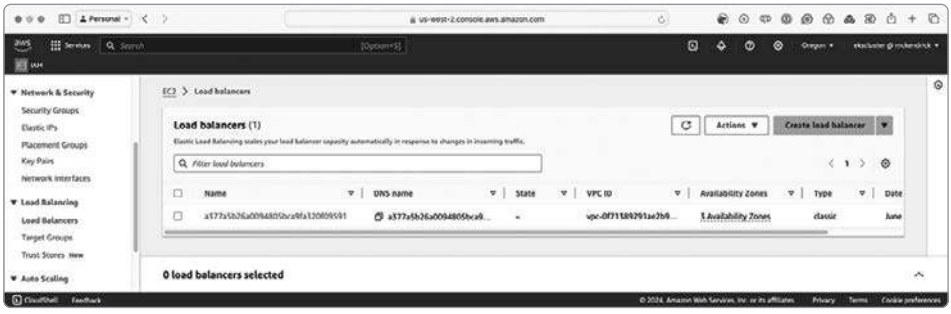

Рис. 16.15. Просмотр созданного балансировщика нагрузки

На ней вы увидите два стека: один — для узлов EKS (двух экземпляров EC2), второй — для самого кластера EKS, то есть управляющего слоя Kubernetes (рис. 16.16).

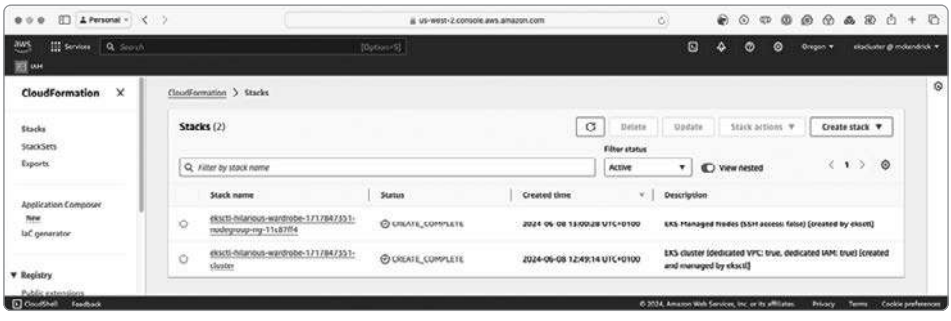

Рис. 16.16. Два стека, составляющих кластер

При выборе любого стека отобразятся подробности о процессе его запуска, включая полный список ресурсов, созданных при развертывании кластера Amazon EKS с помощью `eksctl`.

Выбранный шаблон можно просмотреть в визуальном редакторе. Там же доступен исходный шаблон CloudFormation, сгенерированный `eksctl`, который представляет собой довольно сложный JSON-файл. Если нажать кнопку View in Application Composer (Посмотреть в Application Composer), откроется более наглядное визуальное представление стека, как показано на рис. 16.17.

На этом обзор возможностей консоли AWS можно завершить. Несмотря на то что запуск кластера EKS через `eksctl` относительно прост, уровень его интеграции с консолью AWS оставляет желать лучшего — особенно по сравнению с тем, что мы видели в предыдущей главе при работе с GKE.

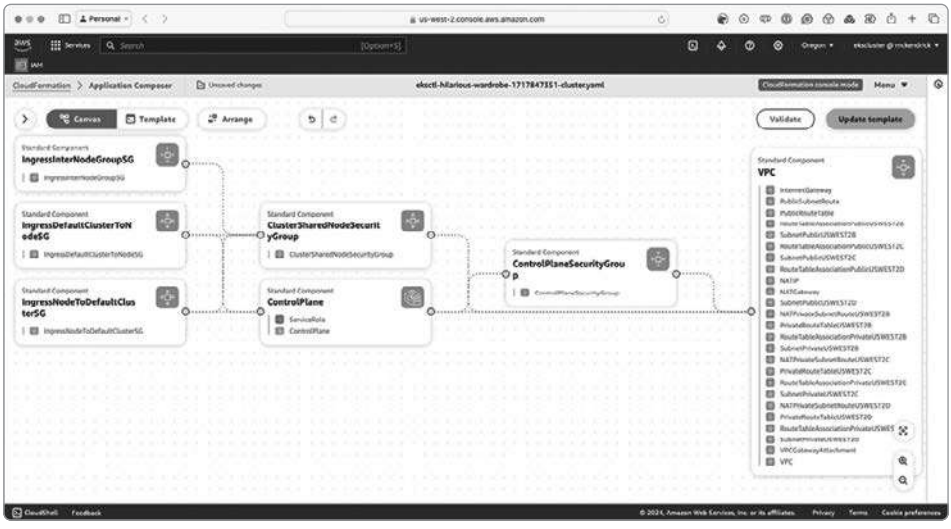

Рис. 16.17. Просмотр шаблона CloudFormation в Application Composer

Мы смогли просмотреть сведения о развернутой рабочей нагрузке, но возможности взаимодействия с ней оказались ограниченными. А мониторинг кластера был сведен к изучению базовых метрик, предоставляемых сервисом Amazon EC2.

После завершения работы кластер Amazon EKS можно удалить.

Удаление кластера Amazon EKS

Удалить кластер можно с помощью следующей команды (не забудьте подставить имя вашего кластера):

```
$ eksctl delete cluster --name hilarious-wardrobe-1717847351
```

Процесс удаления кластера занимает заметно меньше времени, чем его создание (примерно 5 минут).

Как и при развертывании, утилита `eksctl` отображает подробную информацию о выполняемых действиях:

```
[i] deleting EKS cluster "hilarious-wardrobe-1717847351"
[i] will drain 0 unmanaged nodegroup(s) in cluster "hilarious-
wardrobe-1717847351"
[i] starting parallel draining, max in-flight of 1
[i] deleted 0 Fargate profile(s)
```

На первом этапе обновляется локальный файл конфигурации `kubectl`:

```
[✔] kubeconfig has been updated
```

Затем завершается работа всех ресурсов, созданных в процессе развертывания рабочих нагрузок:

```
[i] cleaning up AWS load balancers created by Kubernetes objects of Kind
Service or Ingress
```

После этого удаляются два стека AWS CloudFormation, что влечет за собой автоматическое удаление всех созданных ими ресурсов и настроек:

```
[i] 2 sequential tasks: { delete nodegroup "ng-11c87ff4",
                          delete cluster control plane "hilarious-
wardrobe-1717847351" [async] }
[i] will delete stack "eksctl-hilarious-wardrobe-1717847351-nodegroup-ng-
11c87ff4"
[i] waiting for stack "eksctl-hilarious-wardrobe-1717847351-nodegroup-ng-
11c87ff4" to get deleted
[i] waiting for CloudFormation stack "eksctl-hilarious-wardrobe-1717847351-
nodegroup-ng-11c87ff4"
[i] will delete stack "eksctl-hilarious-wardrobe-1717847351-cluster"
[✔] all cluster resources were deleted
```

На этом процесс удаления кластера завершается.

После удаления кластера обязательно проверьте разделы EC2, EKS и CloudFormation консоли AWS, чтобы убедиться, что все ресурсы действительно удалены. За оставшиеся «осиротевшие» или неактивные ресурсы может продолжать взиматься плата. Хотя подобные ситуации случаются редко, лучше убедиться в полном удалении ресурсов заранее, чем получить неожиданный счет в конце месяца.

Теперь подсчитаем примерную стоимость эксплуатации нашего кластера Amazon EKS в течение месяца.

В данном случае существуют два основных типа затрат.

- Стоимость самого кластера Amazon EKS. За использование каждого созданного кластера EKS взимается 0,10 доллара США в час. При этом один кластер может включать несколько групп узлов, поэтому обычно нет необходимости создавать более одного кластера на регион. Таким образом,

ежемесячная стоимость использования кластера Amazon EKS составляет примерно 73 доллара США.

- Стоимость используемых кластером ресурсов AWS. В частности, каждый экземпляр EC2, выступающий в роли узла кластера, стоил бы около 70 долларов США. Таким образом, сумма ежемесячных расходов на эксплуатацию кластера составила бы примерно 213 долларов США. Это ориентировочная сумма, поскольку дополнительно взимается плата за сетевой трафик и использование сервиса *Elastic Load Balancing (ELB)*, что слегка увеличивает итоговую стоимость.

Ссылка на страницу с актуальной информацией о ценах приведена в разделе «Дополнительные источники информации» в конце главы.

Резюме

В этой главе мы обсудили историю платформы AWS и сервиса Amazon EKS, а затем рассмотрели процесс регистрации учетной записи, а также установки и настройки двух инструментов командной строки, необходимых для запуска кластера Amazon EKS.

После развертывания кластера мы запустили ту же рабочую нагрузку, что и при создании кластера GKE. Нам не потребовалось вносить никаких изменений, чтобы адаптировать ее к другому облачному провайдеру. Более того, система автоматически создала балансировщик нагрузки с использованием встроенного сервиса AWS, избавив нас от необходимости вручную указывать соответствующие параметры в конфигурации.

Тем не менее мы отметили, что уровень интеграции Amazon EKS с консолью AWS является значительно более низким, чем у аналогичного сервиса Google. Вы также узнали, что для удобного развертывания кластера необходимо установить отдельный инструмент `eksctl`, поскольку выполнение той же операции через AWS CLI предполагает гораздо больше шагов — около восьми — при условии, что конфигурация VPC и роли IAM были созданы заранее.

Такой уровень сложности и менее тесная интеграция с интерфейсом AWS делают эксплуатацию кластеров Kubernetes в Amazon EKS не самой удобной по сравнению с использованием других провайдеров. Процесс работы кажется не таким цельным, как, например, в случае с Google.

В следующей главе мы познакомимся с последним из трех крупнейших публичных облачных провайдеров, развернув кластер на платформе Microsoft Azure с помощью сервиса *Azure Kubernetes Service (AKS)*.

Дополнительные источники информации

Ниже приведены ссылки на ресурсы и инструменты, упомянутые в этой главе.

- Платформа AWS: https://aws.amazon.com/.
- Сервис Amazon EKS: https://aws.amazon.com/eks/.
- Инструмент AWS CLI: https://aws.amazon.com/cli/.
- Инструмент `eksctl`: https://eksctl.io/.
- Информация о поддержке `eksctl`: https://github.com/aws/containers-roadmap/issues/2280.
- Официальная документация: https://docs.aws.amazon.com/eks/latest/userguide/what-is-eks.html.
- Стоимость использования Amazon EKS: https://aws.amazon.com/eks/pricing/.

17 Запуск кластера на платформе Microsoft Azure с помощью Azure Kubernetes Service

Последним из публичных облачных сервисов Kubernetes, который мы рассмотрим, является *Azure Kubernetes Service (AKS)*, предоставляемый Microsoft Azure — одной из трех крупнейших публичных облачных платформ (две другие, Google Cloud Platform и Amazon Web Services, мы обсудили в главах 15 и 16 соответственно).

В ходе изучения этой главы вы настроите локальную среду и установите инструменты, необходимые для взаимодействия с вашей учетной записью Microsoft Azure и запуска кластера AKS.

После создания кластера мы развернем ту же рабочую нагрузку, что и в предыдущих главах, а затем оценим уровень интеграции кластера AKS с порталом Microsoft Azure.

В завершение главы мы сравним все три рассмотренные платформы и выберем наиболее предпочтительный сервис.

Технические требования

Для выполнения инструкций, приведенных в этой главе, вам потребуется учетная запись Microsoft Azure с указанным способом оплаты.

Следование инструкциям, приведенным в этой главе, сопряжено с финансовыми расходами. Поэтому крайне важно удалить все созданные ресурсы после завершения работы. Указанные в тексте цены были актуальны на момент написания этой главы; перед созданием ресурсов рекомендуется проверить текущие тарифы.

Знакомство с Microsoft Azure и Azure Kubernetes Service

Прежде чем перейти к установке необходимых инструментов, давайте познакомимся с историей сервисов, о которых пойдет речь в этой главе.

Платформа Microsoft Azure

В 2008 году компания Microsoft официально объявила о запуске нового сервиса под названием Windows Azure.

Он был частью проекта, известного внутри компании как Project Red Dog, который разрабатывался с 2004 года и был направлен на предоставление услуг центра обработки данных с помощью ключевых компонентов Windows.

На конференции для разработчиков 2008 года Microsoft представила пять основных компонентов новой платформы.

- *Microsoft SQL Data Services* — облачная версия сервиса Microsoft SQL Database, реализованная по модели *PaaS* (Platform as a Service — платформа как услуга) и призванная избавить пользователей от сложностей, связанных с самостоятельным размещением и обслуживанием SQL-серверов.
- *Microsoft .NET Services* — еще один PaaS-сервис, позволяющий разработчикам развертывать свои приложения на базе .NET в среде выполнения, управляемой Microsoft.
- *Microsoft SharePoint* — облачная версия популярного корпоративного продукта, реализованная по модели *SaaS* (Software as a Service — программное обеспечение как услуга).
- *Microsoft Dynamics* — SaaS-версия системы управления взаимоотношениями с клиентами (CRM) от Microsoft.
- *Windows Azure* — решение, реализованное по модели *IaaS* (Infrastructure as a Service — инфраструктура как услуга), аналогичное решениям других облачных провайдеров. Оно позволяло пользователям создавать виртуальные машины, настраивать хранилища данных и сетевые службы для поддержки запущенных рабочих нагрузок.

Все эти сервисы были построены на основе операционной системы Red Dog, от которой проект и получил свое название. Это была специализированная версия Windows со встроенным облачным уровнем.

В 2014 году компания переименовала Windows Azure в Microsoft Azure, чтобы подчеркнуть название операционной системы, лежащей в основе облачных сервисов, а также тот факт, что в среде Azure уже выполнялось множество рабочих нагрузок на базе Linux. В рамках этого объявления новый генеральный директор

Microsoft Сатья Наделла представил ставший знаменитым (или печально известным, в зависимости от точки зрения) слайд с надписью «Microsoft ❤ Linux».

Мы говорим «знаменитый» или «печально известный», потому что бывший генеральный директор компании Стив Балмер однажды заявил:

> *«Linux — это рак, который пожирает все, чего касается, в смысле интеллектуальной собственности».*

Поэтому данный шаг Microsoft стал разворотом на 180 градусов и многих удивил.

В 2020 году стало известно, что более 50 % всех виртуальных ядер в Azure выполняют Linux-нагрузки, а около 60 % образов в Azure Marketplace основаны на этой ОС. Такой результат оказался следствием открытой политики Microsoft в отношении Linux и проектов с открытым исходным кодом, таких как Kubernetes, что в итоге привело к появлению ее собственного нативного сервиса Azure Kubernetes Service (AKS).

Сервис Azure Kubernetes Service

Изначально компания Microsoft запустила сервис контейнеризации под названием *Azure Container Service (ACS)*. Он позволял пользователям развертывать контейнерные рабочие нагрузки, выбирая один из трех доступных оркестраторов: Docker Swarm, DC/OS или Kubernetes, каждый из которых предоставлял собственное решение для управления кластером контейнеров.

Вскоре стало очевидно, что среди этих оркестраторов самым популярным является Kubernetes, и ACS был постепенно заменен AKS (Azure Kubernetes Service) — сервисом, полностью основанным на Kubernetes и соответствующим стандартам CNCF. Этот переход занял около двух лет: сервис AKS стал общедоступным в 2018 году, а ACS окончательно прекратил работу в начале 2020 года.

Сервис AKS тесно интегрирован с другими ключевыми компонентами Microsoft Azure, включая Azure Active Directory и Policies. Помимо AKS, Microsoft развивает и другие решения для работы с контейнерами. Одним из новейших является Azure Container Apps.

Azure Container Apps — это бессерверная платформа, которая позволяет пользователям запускать контейнеризованные приложения, избавляя их от необходимости управлять инфраструктурой. Среди ее возможностей можно выделить: динамическое масштабирование в зависимости от трафика, событий, загрузки процессора или памяти; поддержку входящего трафика, передаваемого по протоколам HTTPS и TCP; интеграцию с Dapr; автоматическое масштабирование; возможность использования встроенных инструментов Azure для управления, безопасного хранения секретов, внутреннего обнаружения сервисов и разделения трафика между версиями приложения при внедрении новых релизов. Приложения могут запускать контейнеры из любых реестров и легко

интегрироваться с другими сервисами Azure. При этом контейнеризованные рабочие нагрузки также можно развертывать и в Azure App Services, что иногда вызывает путаницу при выборе подходящего сервиса.

Тем не менее вместо рассмотрения всех доступных вариантов запуска контейнерных приложений в Microsoft Azure имеет смысл сразу перейти к практике и установить инструменты, необходимые для создания и управления кластером AKS.

Подготовка локальной среды

Прежде чем приступить к развертыванию кластера, необходимо выполнить несколько предварительных действий. Прежде всего вам понадобится среда для его запуска, так что, если у вас еще нет учетной записи Microsoft Azure, самое время ее создать.

Создание бесплатной учетной записи Microsoft Azure

Зарегистрировать бесплатную учетную запись Azure можно по адресу https://azure.microsoft.com/free/ (рис. 17.1).

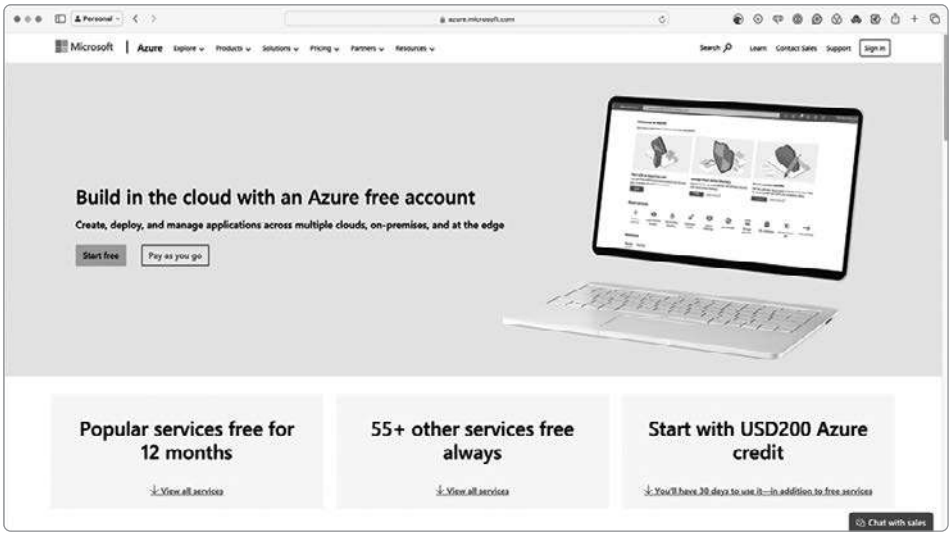

Рис. 17.1. Просмотр бесплатных сервисов Microsoft Azure

На момент написания книги бесплатная учетная запись предоставляла 12-месячный доступ к популярным сервисам, кредит в размере 200 долларов США для изучения и тестирования различных сервисов Azure и доступ к более чем 55 бесплатным сервисам.

Нажмите кнопку Start free (Попробовать Azure бесплатно) и следуйте инструкциям на экране. Процесс регистрации занимает примерно 15 минут. Для завершения регистрации и активации бесплатного аккаунта потребуется указать действительные данные банковской карты (кредитной или дебетовой).

После создания учетной записи вам нужно будет установить интерфейс командной строки (CLI) Azure.

Установка интерфейса командной строки Azure

Компания Microsoft предоставляет мощный кросс-платформенный инструмент командной строки для управления ресурсами Microsoft Azure — Azure CLI. Процесс его установки в macOS, Linux и Windows является максимально простым.

Установка в macOS

Если вы следовали инструкциям из предыдущих глав, то, вероятно, уже догадались, что для установки Azure CLI в macOS мы будем использовать менеджер пакетов Homebrew.

Выполните следующую команду:

```
$ brew install azure-cli
```

После завершения установки проверьте версию утилиты:

```
$ az --version
```

В результате вы увидите вывод, аналогичный показанному на рис. 17.2.

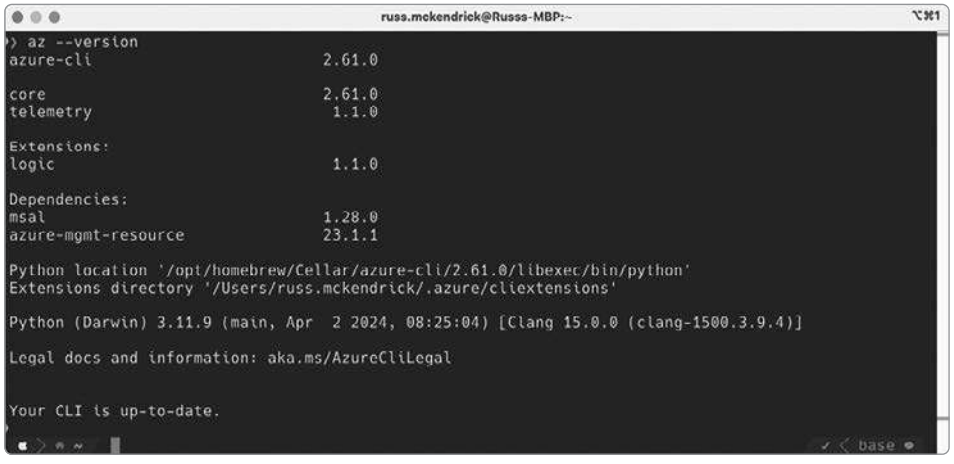

Рис. 17.2. Проверка версии Azure CLI в macOS

После успешной установки вы можете перейти к пункту «Настройка интерфейса командной строки Azure».

Установка в ОС Linux

Microsoft предоставляет универсальный установочный сценарий, подходящий для самых распространенных дистрибутивов Linux. Чтобы запустить его, выполните команду:

```
$ curl -L https://aka.ms/InstallAzureCli | bash
```

Эта команда загрузит, установит и настроит все необходимое для работы Azure CLI в выбранном дистрибутиве Linux. После завершения установки необходимо перезапустить сеанс, то есть выйти из системы и войти снова.

В некоторых дистрибутивах для этого достаточно выполнить следующую команду:

```
$ source ~/.profile
```

После перезапуска сеанса проверьте версию установленной утилиты:

```
$ az --version
```

Вывод будет практически идентичен результату, полученному при установке CLI в macOS, за исключением информации об операционной системе. После успешной установки вы можете перейти к пункту «Настройка интерфейса командной строки Azure».

Установка в Windows

Установить Azure CLI в Windows можно несколькими способами. Первый заключается в том, чтобы загрузить установщик по адресу https://aka.ms/installazurecliwindows и запустить его двойным щелчком кнопкой мыши.

Второй способ предполагает выполнение следующей команды в оболочке PowerShell для загрузки и установки упомянутого выше установщика:

```
$ ProgressPreference = 'SilentlyContinue';
Invoke-WebRequest -Uri https://aka.ms/installazurecliwindows
-OutFile .\AzureCLI.msi;
Start-Process msiexec.exe -Wait -ArgumentList '/I AzureCLI.msi /quiet';
Remove-Item .\AzureCLI.msi
```

Установить Azure CLI также можно через пакетный менеджер *Chocolatey*:

```
$ choco install azure-cli
```

Независимо от выбранного способа после установки проверьте версию программы:

```
$ az --version
```

Вывод будет практически идентичен результату, полученному при установке CLI в macOS. Теперь, когда Azure CLI установлен, можно приступать к его настройке.

Настройка интерфейса командной строки Azure

Чтобы настроить Azure CLI, выполните команду:

```
$ az login
```

В браузере откроется окно авторизации. После успешного входа Azure CLI будет настроен на работу с учетной записью пользователя, от имени которого вы вошли в систему.

Если при авторизации возникают проблемы или вы работаете в среде без графического интерфейса (например, на удаленном сервере Linux), выполните следующую команду, чтобы получить URL-адрес и уникальный код авторизации:

```
$ az login --use-device-code
```

После успешного входа в систему в консоли отобразится информация об учетной записи Azure. Чтобы просмотреть эти сведения повторно, используйте команду:

```
$ az account show
```

Если по какой-либо причине вы не можете установить Azure CLI локально, это не проблема — в портале Azure доступна веб-версия терминала с CLI, которую мы рассмотрим далее.

Доступ к оболочке Azure Cloud Shell

Чтобы открыть оболочку Azure Cloud Shell, перейдите по адресу https://portal.azure.com/ и войдите в систему, используя свои учетные данные. После входа щелкните на значке Cloud Shell на верхней панели рядом с центральным полем поиска.

При запуске оболочки вам будет предложено подключить хранилище. Поскольку мы планируем использовать утилиту `kubectl`, нам нужно обеспечить сохранение конфигурации между сеансами. Выберите вариант Mount storage

account (Подключить учетную запись хранения), а затем укажите свою подписку, как показано на рис. 17.3.

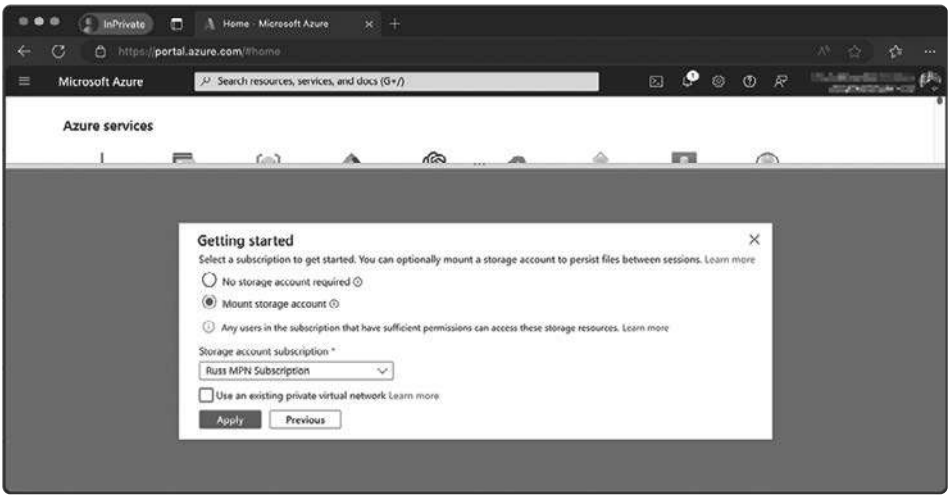

Рис. 17.3. Выбор варианта подключения хранилища в Cloud Shell

После нажатия кнопки Apply (Применить) откроется диалоговое окно с тремя вариантами. Выберите пункт We will create a storage account for you (Мы создадим учетную запись хранения за вас) (рис. 17.4).

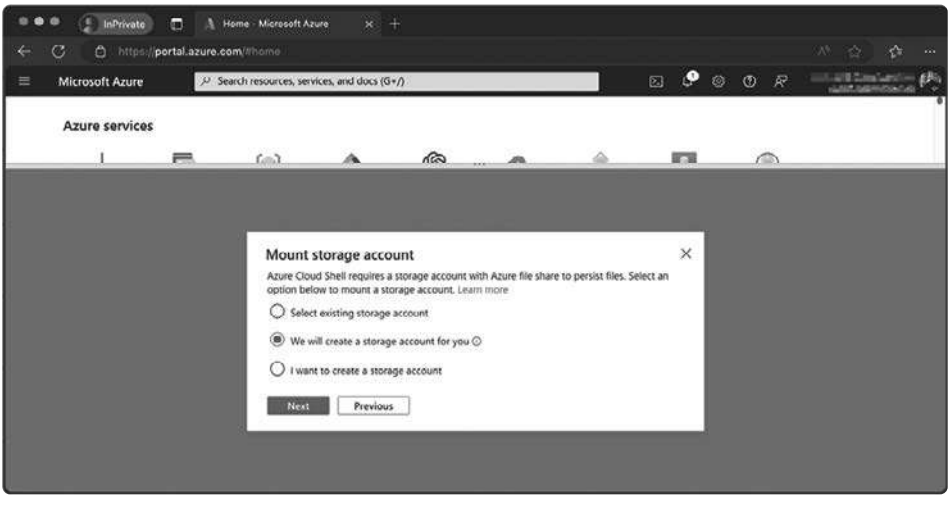

Рис. 17.4. Выбор способа создания учетной записи хранения

Примерно через минуту запустится Cloud Shell, и вы увидите приглашение командной строки (рис. 17.5).

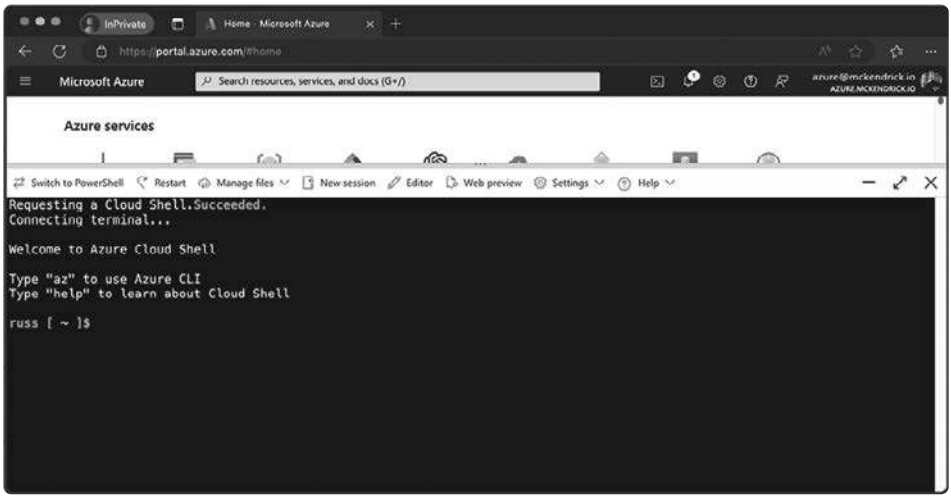

Рис. 17.5. Вход выполнен, оболочка готова к работе

Выполните следующую команду, чтобы проверить версию Azure CLI, как мы это делали при локальной установке:

```
$ az --version
```

Команду `az login` выполнять не требуется, так как при запуске Cloud Shell авторизация в Azure происходит автоматически в фоновом режиме.

Теперь, когда у нас есть настроенный инструмент Azure CLI, можно переходить к запуску кластера AKS.

Запуск кластера AKS

Для запуска кластера AKS достаточно выполнить две команды.

Первая команда создает группу ресурсов Azure:

```
$ az group create --name rg-myfirstakscluster-eus --location eastus -o table
```

В этом примере создается группа ресурсов `rg-myfirstakscluster-eus` в регионе `eastus`, а параметр `-o table` форматирует вывод в виде таблицы (по умолчанию Azure CLI использует формат JSON).

Группа ресурсов (resource group) — это логический контейнер, объединяющий связанные ресурсы Azure. Сервисы, запущенные внутри группы, могут наследовать такие параметры, как настройки механизма управления доступом на основе ролей (RBAC), блокировки и регион запуска.

После успешного создания группы ресурсов вы увидите подтверждение в виде таблицы (рис. 17.6).

Рис. 17.6. Создание группы ресурсов

Теперь, когда у нас есть контейнер для ресурсов, можно запустить сам кластер AKS, выполнив следующую команду:

```
$ az aks create --resource-group rg-myfirstakscluster-eus \
    --name aksmyfirstakscluster-eus --node-count 2 \
    --enable-addons monitoring --generate-sshkeys -o yaml
```

Запуск и настройка кластера занимают около 5 минут. Пока выполняется этот процесс, разберем параметры команды `az aks create`.

- `--resource-group` — группа ресурсов, в которой будет запущен кластер. Кластер унаследует регион своей группы ресурсов. В нашем примере используется созданная ранее группа `rg-myfirstakscluster-eus`, поэтому кластер будет создан в регионе `eastus`.
- `--name` — означает имя запускаемого кластера. В нашем случае это `aksmyfirstakscluster-eus`.
- `--node-count` — количество узлов в кластере, в данном случае два. На момент написания книги для узлов по умолчанию использовались виртуальные машины типа Standard_DS2_v2, предоставляющие каждому узлу два виртуальных процессора (vCPU) и 7 Гбайт оперативной памяти.
- `--enable-addons` — список дополнений, включаемых при создании кластера. В данном примере мы активизируем только дополнение `monitoring`.
- `--generate-ssh-keys` — создает открытый и закрытый SSH-ключи для доступа к кластеру.
- `-o` — определяет формат вывода команды. На этот раз мы выбрали формат YAML, так как он воспринимается легче, чем JSON или таблица.

После запуска кластера вы увидите примерно следующее (рис. 17.7).

Рис. 17.7. Результат запуска кластера

Как видите, вывод содержит множество данных, однако анализировать их вручную не придется, так как мы будем взаимодействовать с кластером через Azure CLI и портал Azure, а не путем отправки запросов к API Azure Resource Manager.

Теперь, когда кластер создан, необходимо настроить локальный клиент `kubectl` для взаимодействия с ним.

Для этого выполните команду:

```
$ az aks get-credentials --resource-group rg-myfirstakscluster-eus \
    --name aksmyfirstakscluster-eus
```

Появится сообщение о загрузке учетных данных и настройке `kubectl` (рис. 17.8).

Рис. 17.8. Загрузка учетных данных для доступа к кластеру и настройка kubectl

Затем, когда кластер запущен и клиент `kubectl` настроен, можно вывести список узлов кластера с помощью уже знакомой вам команды:

```
$ kubectl get nodes
```

Результат ее выполнения показан на рис. 17.9.

Теперь можно развернуть тестовую рабочую нагрузку Guestbook, которую мы уже использовали в предыдущих двух главах для проверки работоспособности кластеров.

Рис. 17.9. Проверка доступности узлов кластера

Развертывание рабочей нагрузки и взаимодействие с кластером

Здесь мы воспользуемся той же рабочей нагрузкой, что и в главах 15 и 16, поэтому приведем только сами команды без их детального разбора.

Развертывание рабочей нагрузки

Начнем с создания объектов Deployment и Service для Redis Leader:

```
$ kubectl apply -f https://raw.githubusercontent.com/GoogleCloudPlatform/
kubernetes-engine-samples/main/quickstarts/guestbook/redis-leader
-deployment.yaml
$ kubectl apply -f https://raw.githubusercontent.com/GoogleCloudPlatform/
kubernetes-engine-samples/main/quickstarts/guestbook/redis-leader-service.yaml
```

Затем создадим те же объекты для Redis Follower:

```
$ kubectl apply -f https://raw.githubusercontent.com/GoogleCloudPlatform/
kubernetes-engine-samples/main/quickstarts/guestbook/redis-follower
-deployment.yaml
$ kubectl apply -f https://raw.githubusercontent.com/GoogleCloudPlatform/
kubernetes-engine-samples/main/quickstarts/guestbook/redis-follower
-service.yaml
```

Наконец, развернем фронтенд с помощью следующих команд:

```
$ kubectl apply -f https://raw.githubusercontent.com/GoogleCloudPlatform/
kubernetes-engine-samples/main/quickstarts/guestbook/frontend
-deployment.yaml
$ kubectl apply -f https://raw.githubusercontent.com/GoogleCloudPlatform/
kubernetes-engine-samples/main/quickstarts/guestbook/frontend-service.yaml
```

Через несколько минут можно получить информацию о сервисе `frontend`, выполнив команду:

```
$ kubectl get service frontend
```

Как и в предыдущих главах, вывод этой команды будет содержать публичный IP-адрес, по которому можно получить доступ к приложению Guestbook (рис. 17.10).

Рис. 17.10. Получение информации о сервисе frontend

Введите этот IP-адрес в браузере. Используйте префикс `http://`, поскольку SSL-сертификат не настроен. В результате откроется веб-интерфейс приложения Guestbook (рис. 17.11).

Рис. 17.11. Веб-интерфейс приложения Guestbook

Теперь, когда рабочая нагрузка успешно запущена, давайте изучим портал Azure.

Изучение портала Azure

Если вы еще не вошли в систему, перейдите по адресу https://portal.azure.com/ и авторизуйтесь. После входа введите Kubernetes в поле поиска в верхней части страницы.

В списке отобразится пункт Kubernetes services — выберите его, чтобы открыть список всех сервисов Kubernetes, развернутых в рамках подписок, к которым у вас есть доступ (рис. 17.12).

Щелкните на кластере `aks-myfirstakscluster-eus`, чтобы перейти на страницу с обзорной информацией о нашем кластере и запущенной рабочей нагрузке (рис. 17.13).

Рис. 17.12. Список доступных сервисов Kubernetes

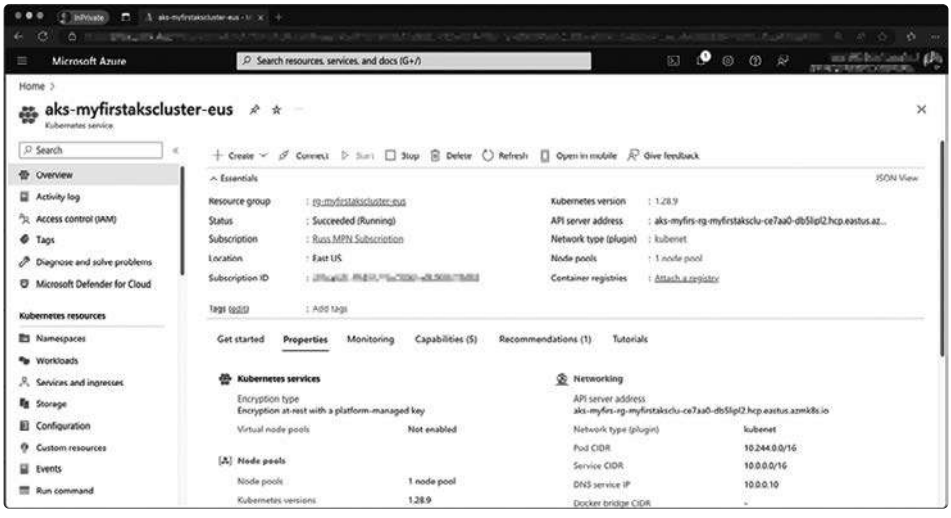

Рис. 17.13. Страница с обзорной информацией о кластере

В меню слева под заголовком Kubernetes resources (Ресурсы Kubernetes) вы увидите несколько разделов, которые мы рассмотрим далее.

Раздел Namespaces (Пространства имен)

В этом разделе отображаются все активные пространства имен кластера. Поскольку при запуске рабочей нагрузки мы не указали собственное пространство имен, все созданные развертывания и сервисы будут размещены в пространстве `default`.

Помимо `default`, в кластере по умолчанию присутствуют и системные пространства имен — `kube-node-lease`, `kube-public` и `kube-system`. Изменять их не рекомендуется.

Если щелкнуть на пространстве имен `default`, откроется страница Overview (Обзор), где можно просмотреть и отредактировать YAML-манифест, определяющий пространство имен, ознакомиться с журналом событий и настроить сервисную сетку. В нашем тестовом развертывании сервисная сетка отсутствует.

Раздел Workloads (Рабочие нагрузки)

В этом разделе отображается информация о запущенных рабочих нагрузках. На следующем снимке экрана показан список рабочих нагрузок, развернутых в пространстве имен `default` (рис. 17.14).

Рис. 17.14. Просмотр запущенных рабочих нагрузок

Если щелкнуть на конкретном развертывании, откроется страница с подробной информацией о нем. Например, при выборе развертывания `frontend` появится следующая страница (рис. 17.15).

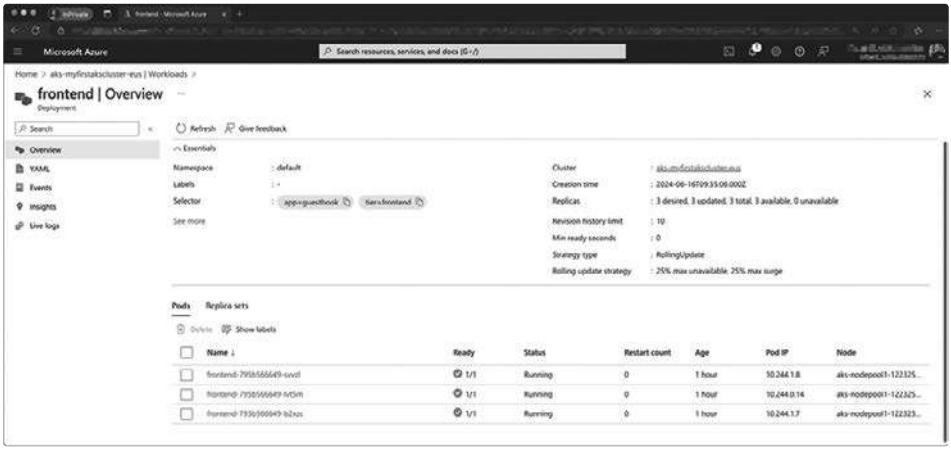

Рис. 17.15. Просмотр подробных сведений о развертывании

Как видите, в меню слева появилось несколько дополнительных разделов. Помимо YAML (YAML-манифест) и Events (События), стал доступен пункт Insights (Аналитика), к которому мы вернемся чуть позже.

В разделе Live logs (Потоковые журналы) можно выбрать конкретный под и просматривать его журналы в режиме реального времени (рис. 17.16).

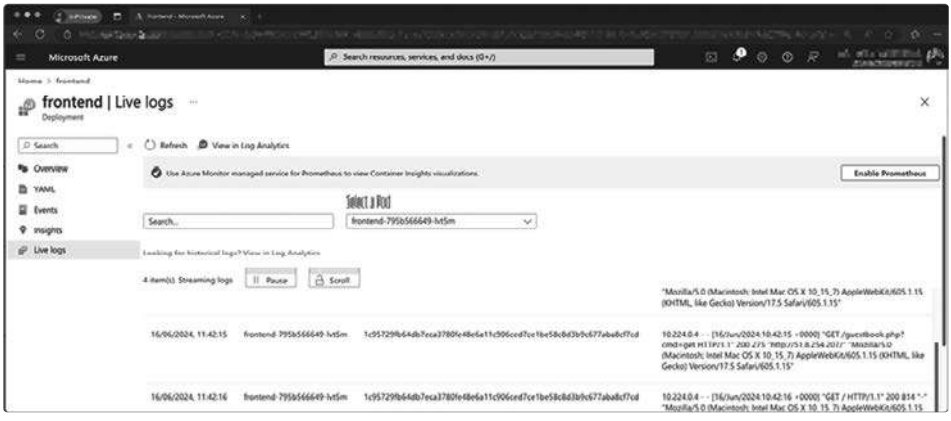

Рис. 17.16. Просмотр журналов пода в реальном времени

Если вернуться в раздел Workloads (Рабочие нагрузки) и открыть вкладку Pods (Поды), отобразится список всех запущенных подов, входящих в состав рабочей нагрузки. Для каждого пода указаны IP-адрес и имя узла, на котором он запущен (рис. 17.17).

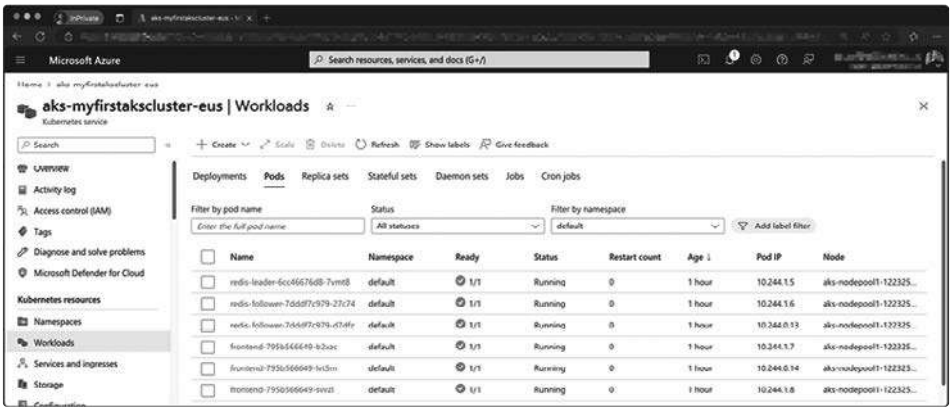

Рис. 17.17. Список запущенных подов

При выборе конкретного пода открывается страница с обзорной информацией о нем, YAML-манифестом и журналом событий.

На вкладке Replica sets (Наборы реплик) перечислены все наборы реплик, созданные в рамках развертывания рабочей нагрузки. Щелкнув на любом из них, вы получите привычный набор вкладок, включая Overview (Обзор), YAML (YAML-манифест) и Events (События) (рис. 17.18).

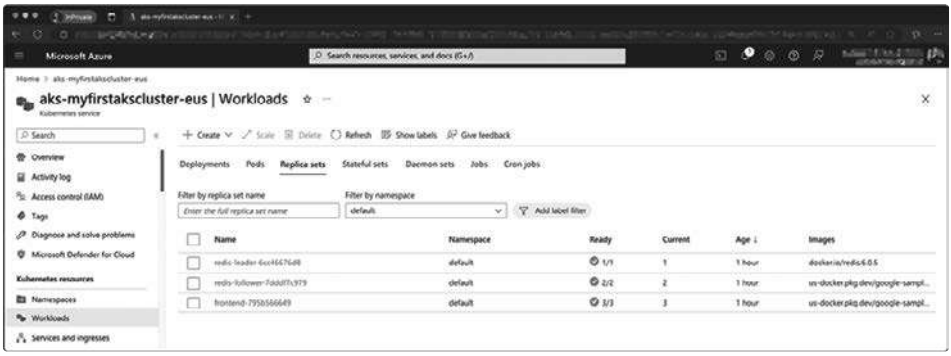

Рис. 17.18. Список наборов реплик

Следующая вкладка — Stateful sets (Наборы с сохранением состояния). Ни в нашем, ни в системных пространствах имен Microsoft таких объектов нет, поэтому здесь пока нечего просматривать. Однако, если бы они присутствовали, данная страница содержала бы те же типы сведений, что и другие вкладки раздела Workloads (Рабочие нагрузки).

Далее следует вкладка Daemon sets (Наборы демонов). В нашем пространстве таких объектов нет, но в кластере присутствуют системные объекты DaemonSet, развернутые Microsoft, которые при желании можно изучить.

На последних двух вкладках Jobs (Задания) и Cron jobs (Запланированные задания) отображаются сведения обо всех объектах Job и CronJob, развернутых внутри кластера.

Раздел Services and ingresses (Сервисы и объекты Ingress)

В этом разделе отображается список всех сервисов, развернутых в кластере. Как видно на рис. 17.19, здесь указаны используемые сервисом кластерные IP-адреса (Cluster IP) и настроенные пользователем внешние IP-адреса (External IP).

Щелкнув на любом из сервисов, вы увидите уже знакомый интерфейс, позволяющий подробно изучить конфигурацию выбранного объекта.

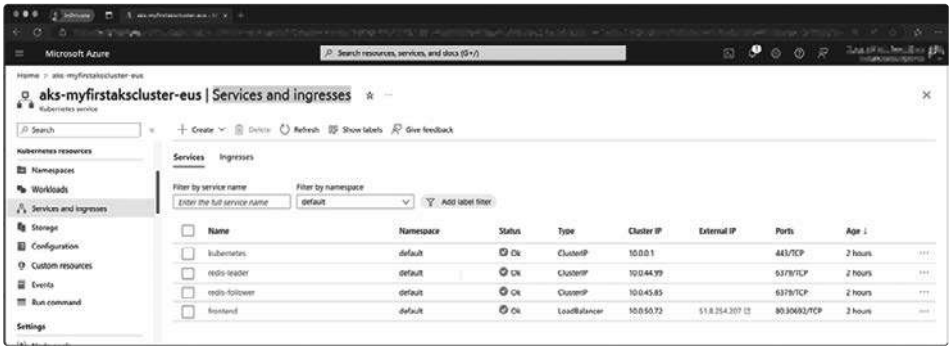

Рис. 17.19. Просмотр списка сервисов

Раздел Storage (Хранилище)

В этом разделе можно управлять параметрами постоянного хранилища, настроенного в кластере. В нашем тестовом примере постоянное хранилище не используется, поэтому здесь почти ничего не отображается.

Раздел Configuration (Конфигурация)

Здесь можно просматривать и редактировать объекты ConfigMap и Secret, созданные в кластере. Поскольку в нашей рабочей нагрузке подобные объекты не были настроены, в данном случае список состоит только из системных ресурсов самого кластера, изменять которые настоятельно не рекомендуется.

Раздел Custom resources (Пользовательские ресурсы)

Этот раздел позволяет управлять пользовательскими ресурсами, подключенными к вашему кластеру.

Раздел Events (События)

Здесь отображаются события кластера в режиме реального времени. Они позволяют отслеживать состояние компонентов и устранять возможные проблемы с кластером и приложениями.

Раздел Run command (Выполнение команд)

Эта полезная функция позволяет выполнять команды `kubectl` прямо с портала Azure, не запуская оболочку Cloud Shell и не настраивая локальную установку `kubectl` (рис. 17.20).

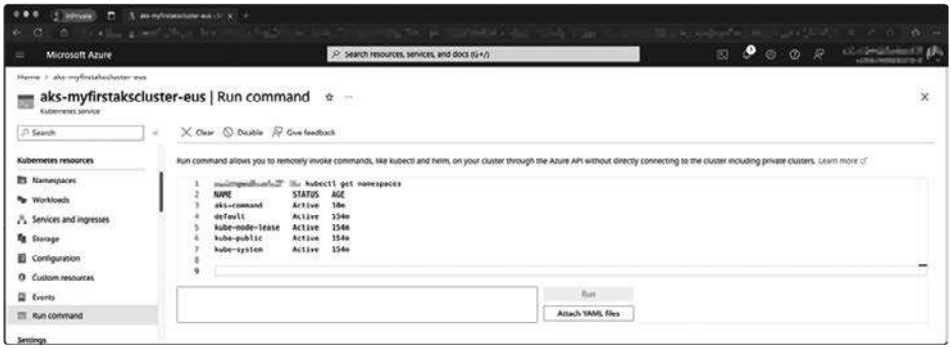

Рис. 17.20. Работа с кластером путем выполнения команд через портал Azure

В разделе меню Settings (Настройки) содержатся следующие пункты.

Раздел Node pools (Пулы узлов)

Здесь отображается информация о пуле узлов и предоставляется возможность обновить версию Kubernetes, работающую в этом пуле. Данная функция становится доступной только после обновления версии Kubernetes в управляющем слое.

Кроме того, в этом разделе можно масштабировать пул узлов или добавить новый. На рис. 17.21 показаны параметры масштабирования.

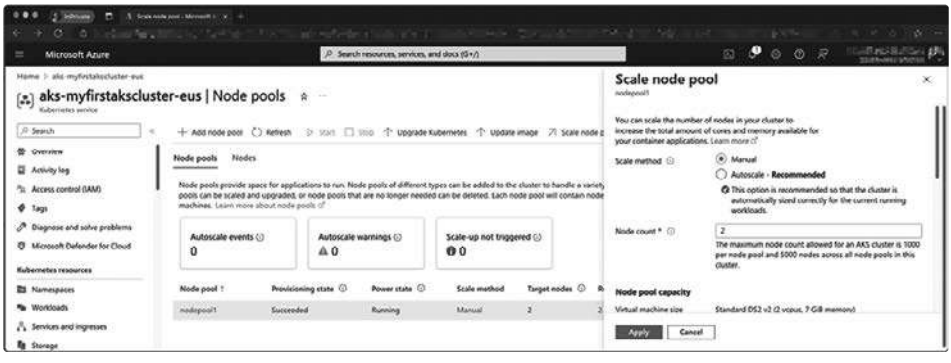

Рис. 17.21. Просмотр параметров масштабирования

На вкладке Nodes (Узлы) можно просмотреть информацию о каждом узле (рис. 17.22).

Рис. 17.22. Просмотр информации об узлах

Раздел Cluster configuration (Конфигурация кластера)

Как было сказано выше, обновить версию Kubernetes в пулах узлов можно только после обновления управляющего слоя — данная операция выполняется именно здесь. Управляющий слой контролируется Microsoft и изолирован от пользовательских пулов узлов.

Управляющий слой Kubernetes поддерживает обратную совместимость в пределах трех релизов, поэтому обновление обычно возможно только до версии, не более чем на три релиза новее текущей.

Раздел Application scaling (Масштабирование приложений)

В этом разделе можно активизировать механизм *Kubernetes Event-Driven Autoscaler (KEDA)*, который динамически масштабирует рабочие нагрузки в ответ на внешние события. На момент написания книги при настройке сервиса через портал Azure поддерживались следующие источники событий: Azure Service Bus, Cron, память (Memory) и процессор (CPU).

Раздел Networking (Сеть)

Здесь можно управлять сетевыми настройками вашего кластера.

Раздел Extensions + applications (Расширения и приложения)

В этом разделе представлены приложения Microsoft и сторонние приложения из каталога Azure Marketplace, которые можно развернуть в кластере AKS и управлять ими прямо через портал.

Раздел Backup (Резервное копирование)

Нативный сервис резервного копирования Azure теперь поддерживает создание резервных копий рабочих нагрузок Kubernetes и данных приложений. Все соответствующие операции можно выполнять в этом разделе.

Раздел Other options (Прочие параметры)

Раздел меню Settings (Настройки) также содержит дополнительные параметры для настройки различных аспектов кластера и его интеграции с другими нативными сервисами Azure. Ссылка на документацию по AKS, содержащую более подробные сведения, доступна в разделе «Дополнительные источники информации» в конце главы.

Раздел Insights (Аналитика)

Еще одной частью портала Azure, на которую стоит обратить внимание, является раздел Insights (Аналитика), расположенный в меню Monitoring (Мониторинг). Как вы помните, при создании кластера мы активизировали дополнение `monitoring` с помощью флага `--enable-addons monitoring`.

Эта функция активизирует встроенную систему мониторинга Microsoft, которая отправляет данные ресурсов кластера в сервис Azure Log Analytics. После этого собранные данные визуализируются в разделе Insights (Аналитика). Для большинства сервисов Azure предусмотрена вкладка Insights (Аналитика), а собранная здесь информация может использоваться службой Azure Monitor для генерации оповещений (рис. 17.23).

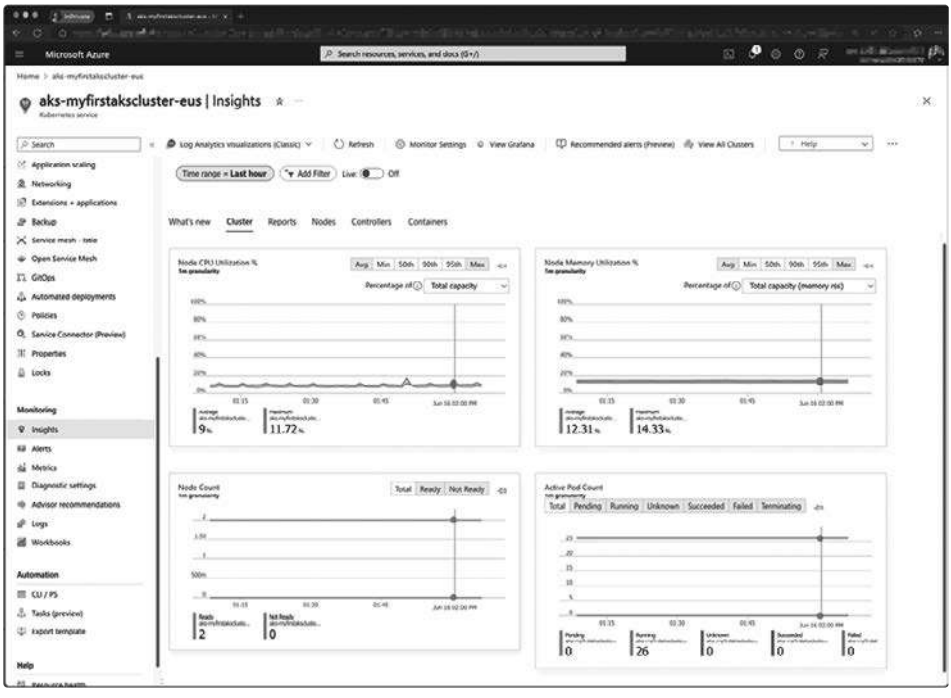

Рис. 17.23. Страница Insights (Аналитика)

На странице Insights (Аналитика) представлено несколько вкладок.

- Cluster (Кластер) — эта вкладка содержит общую информацию о кластере, включая уровень использования ресурсов процессора и памяти, количество узлов и подов.
- Reports (Отчеты) — готовые отчеты о результатах мониторинга узлов (производительность), ресурсов (доступность), сети и затрат. По мере развития сервиса сюда будут добавляться новые типы отчетов.
- Nodes (Узлы) — сводная информация об узлах.
- Controllers (Контроллеры) — сведения о контроллерах (например, ReplicaSet и DaemonSet), запущенных в кластере.
- Containers (Контейнеры) — данные обо всех контейнерах, работающих в подах.

Может показаться, что часть информации дублируется, и это действительно так, однако раздел Insights (Аналитика) позволяет быстро получить полную картину состояния кластера, не переходя по множеству страниц и не используя фильтры.

Прежде чем удалять кластер, мы рекомендуем изучить интерфейс портала и просмотреть все доступные разделы, чтобы лучше понять уровень интеграции вашего кластера AKS с Azure.

Удаление кластера AKS

Последним шагом является удаление кластера. Для этого вернитесь в Azure CLI и выполните следующую команду:

```
$ az aks delete --resource-group rg-myfirstakscluster-eus --name
aksmyfirstakscluster-eus
```

Система запросит подтверждение. Введите слово `yes`, чтобы начать процесс удаления.

Эта операция занимает примерно 5 минут. Следует отметить, что приведенная выше команда удаляет только сам кластер, но не группу ресурсов, в которой он размещен. Чтобы удалить ее, выполните следующую команду:

```
$ az group delete --name rg-myfirstakscluster-eus
```

От вас снова потребуется подтверждение — введите `yes`.

Теперь оценим стоимость эксплуатации нашего кластера.

В отличие от двух других облачных провайдеров, о которых мы говорили в предыдущих главах, Microsoft Azure позволяет бесплатно управлять кластерами

при запуске рабочих нагрузок непроизводственного уровня — плата взимается только за потребление вычислительных ресурсов.

Стоимость использования двух экземпляров типа Standard_DS2_v2, запущенных в регионе US East, составила бы около 213 долларов США в месяц. При выборе стандартного варианта управления кластером общая стоимость составила бы 286 долларов США в месяц.

Чтобы сэкономить, можно выбрать более новые поколения виртуальных машин с аналогичными характеристиками, например, выполнив следующие команды:

```
$ az group create --name rg-myfirstakscluster-eus --location eastus -o table
$ az aks create --resource-group rg-myfirstakscluster-eus \
    --name aksmyfirstakscluster-eus --node-count 2 \
    --enable-addons monitoring --generate-sshkeys \
    --node-vm-size standard_ds3_v2 -o yaml
```

В результате будет создан двухузловой кластер с четырьмя виртуальными процессорами (vCPU) и 16 Гбайт оперативной памяти. Стоимость его эксплуатации при условии выбора тарифа для нерабочих (non-production) сред составит около 140 долларов США в месяц.

Сравнение трех публичных облачных платформ

В завершение этой главы сравним три рассмотренные облачные платформы (табл. 17.1).

Здесь следует отметить несколько ключевых моментов.

- GKE обычно лидирует по уровню поддержки последних версий Kubernetes и предоставляет наиболее удобные механизмы автоматического обновления.
- AKS считается самой дружественной к пользователю платформой, особенно теми, кто уже использует другие сервисы экосистемы Azure.
- В EKS взимается отдельная плата за управляющий слой, тогда как в GKE и AKS пользователь платит только за рабочие узлы.
- Все три сервиса тесно интегрированы со своими облачными экосистемами.
- GKE часто хвалят за передовые возможности и производительность, это объясняется тем, что именно компания Google является разработчиком Kubernetes.
- Каждая платформа имеет свои преимущества. У GKE они заключаются в высокой производительности и широком функционале, у EKS — в тесной

интеграции с экосистемой AWS, у AKS — в простоте использования и тесной интеграции с Azure.

- Оптимальный выбор зависит от вашей инфраструктуры, требований и степени знакомства с экосистемой конкретного облачного провайдера.

Таблица 17.1. Сравнение трех публичных облачных платформ

Характеристика	Google Kubernetes Engine (GKE)	Amazon Elastic Kubernetes Service (EKS)	Microsoft AKS
Поддержка версий Kubernetes	Последние версии, частые обновления	Немного отстает в плане поддержки версий	Последние версии, частые обновления
Автоматическое обновление	Автоматически для управляющего слоя и узлов	По запросу для управляющего слоя, вручную для узлов	По запросу для управляющего слоя и узлов
Простота использования	Высокая — интуитивный интерфейс	Средняя — сложная настройка	Высокая — интуитивный интерфейс
Интеграция с облачными сервисами	Тесная интеграция с сервисами GCP	Тесная интеграция с сервисами AWS	Тесная интеграция с сервисами Azure
Масштабируемость	Хорошая, поддерживает автомасштабирование	Хорошая, поддерживает автомасштабирование	Хорошая, поддерживает автомасштабирование
Безопасность	Высокий уровень, интеграция со средствами обеспечения безопасности GCP	Высокий уровень, интеграция со средствами обеспечения безопасности AWS	Высокий уровень, интеграция со средствами обеспечения безопасности Azure
Стоимость	Бесплатный управляющий слой, плата за узлы	$0,10 в час за управляющий слой и плата за узлы	Бесплатный управляющий слой, плата за узлы
Поддержка многозонных кластеров	Да	Да	Да
Поддержка частных кластеров	Да	Да	Да
Поддержка бессерверных вычислений	Да, с использованием сервиса Cloud Run for Anthos	Да, с использованием сервиса Fargate for EKS	Да, с использованием виртуальных узлов AKS

Как видите, выбор между этими тремя сервисами во многом определяется тем, в какой облачной среде вы запускаете свои рабочие нагрузки. Как было продемонстрировано в последних трех главах, после запуска кластеров опыт взаимодействия с ними практически одинаков.

Резюме

В этой главе мы поговорили о том, как появилась платформа Microsoft Azure, проследили историю развития контейнерных сервисов Microsoft и увидели, как компания пришла к созданию Azure Kubernetes Service (AKS).

Затем мы зарегистрировали учетную запись Azure, установили Azure CLI и запустили собственный кластер AKS. После этого мы развернули ту же рабочую нагрузку, что и в главах, посвященных GKE и Amazon EKS.

После успешного запуска приложения мы перешли на портал Azure, чтобы изучить средства мониторинга и аналитики, а также возможности для управления кластером.

В завершение мы удалили созданные ресурсы и оценили стоимость эксплуатации рассмотренного кластера.

Из трех публичных облачных сервисов на первое место мы поставили бы Microsoft Azure AKS, поскольку он отличается наиболее продуманным и богатым функционалом, а также простотой использования. На второе место — сервис GKE от Google, о котором шла речь в главе 15. Он тоже хорошо реализован, однако компании стоит пересмотреть политику ценообразования, чтобы составить достойную конкуренцию предложению Microsoft.

На последнее место мы поставили бы сервис AWS, описанный в главе 16, так как он производит впечатление менее доработанного по сравнению с решениями Microsoft и Google. Работа с ним напоминает не органичное взаимодействие с экосистемой AWS, а, скорее, запуск самостоятельного кластера Kubernetes в инфраструктуре Amazon.

Если отбросить личные впечатления, то главный вывод, который мы сделали, развернув кластеры Kubernetes на трех различных облачных платформах, заключается в следующем: если не считать процесса запуска кластера и настройки клиента `kubectl`, опыт работы с разными сервисами является практически идентичным и вашей рабочей нагрузке безразлично, где именно она выполняется. Сегодня различия между провайдерами уже не играют решающей роли. Всего несколько лет назад подобный уровень универсальности казался недостижимым, а по-настоящему облачно-независимые рабочие нагрузки были лишь несбыточной мечтой.

В следующей главе мы обсудим вопросы безопасности в Kubernetes, включая аутентификацию и авторизацию, контроллеры допуска, сетевые политики и другие важные аспекты.

Дополнительные источники информации

Ниже приведены ссылки на источники и инструменты, упомянутые в этой главе.

- Платформа Microsoft Azure: https://azure.microsoft.com/.
- Сервис AKS: https://azure.microsoft.com/services/kubernetes-service/.
- Официальная документация по AKS: https://docs.microsoft.com/en-us/azure/aks/.
- Статья *Microsoft Loves Linux*: https://www.microsoft.com/en-us/windows-server/blog/2015/05/06/microsoft-loves-linux/.
- Статья *Ballmer: Linux is a cancer*: https://www.theregister.com/2001/06/02/ballmer_linux_is_a_cancer/.
- Статья *Ballmer: I may have called Linux a cancer but now I love it*: https://www.zdnet.com/article/ballmer-i-may-have-called-linux-a-cancer-but-now-i-love-it/.
- Статья *Linux is Most Used OS in Microsoft Azure — over 50 percent of VM cores*: https://build5nines.com/linux-is-most-used-os-in-microsoft-azure-over-50-percent-fo-vm-cores/.

18 Безопасность в системе Kubernetes

Аутентификация и авторизация являются краеугольными камнями современных программных систем, обеспечивающими управление идентификацией и доступом. Эти два понятия нередко путают, хотя на деле они представляют собой совершенно разные процессы. Аутентификация предполагает проверку подлинности пользователя, обычно выполняемую с помощью таких механизмов, как имя пользователя и пароль. Авторизация, в свою очередь, связана с определением того, к каким ресурсам может обращаться уже аутентифицированный пользователь и что он может делать в системе. Таким образом, аутентификация всегда предшествует авторизации. Kubernetes расширяет эту модель с помощью механизма *управления доступом на основе ролей* (Role-Based Access Control, RBAC). Он позволяет администратору определять роли с заданными правами и назначать их конкретным пользователям, тем самым реализуя принцип наименьших привилегий и обеспечивая детализированный контроль доступа.

Помимо средств для управления идентификацией и доступом, Kubernetes предоставляет целый ряд других механизмов защиты, предназначенных для дополнительного повышения уровня безопасности компонентов системы. Будучи наиболее зрелым и широко применяемым оркестратором контейнеров, Kubernetes уделяет особое внимание защите кластеров, узлов, контейнеров, сетей и приложений, снижая риски на каждом из этих уровней.

В этой главе мы рассмотрим основные принципы обеспечения безопасности в системе Kubernetes, включая различные способы аутентификации с использованием клиентских сертификатов X.509 и токенов OpenID Connect. В особых случаях, например при интеграции с LDAP, Kubernetes поддерживает и другие варианты, включая возможность использования аутентифицирующих прокси и вебхуков. Мы также подробно разберем модель *RBAC* данной платформы, которая предоставляет администраторам контроль над доступом к ресурсам кластера, а также позволяет управлять пользователями, группами и учетными записями сервисов.

Кроме того, в этой главе будет представлена такая продвинутая функция Kubernetes, как контроллеры допуска. Эти компоненты обеспечивают применение политик безопасности для проверки ресурса на этапе его допуска в кластер.

Контроллеры допуска создают дополнительный уровень защиты, управляя запросами к ресурсам и обеспечивая соблюдение установленных политик при их создании и модификации.

Особое внимание будет уделено защите подов и контейнеров, поскольку они представляют собой рабочие среды приложений, которые могут взаимодействовать с конфиденциальными данными. Kubernetes предоставляет ряд параметров `securityContext`, позволяющих администраторам задавать конкретные настройки безопасности для контейнеров. В частности, это позволяет принудительно запускать контейнеры от имени непривилегированного пользователя. Не менее важным аспектом является сетевая безопасность, поэтому мы также поговорим о том, как механизм NetworkPolicy обеспечивает изоляцию и защиту сетевых взаимодействий между подами внутри кластера за счет детального контроля над потоками трафика.

Затем мы перейдем к вопросам безопасности среды выполнения контейнеров. Мы рассмотрим такие альтернативные среды выполнения, как gVisor и Kata Containers, создающие дополнительные границы безопасности. Первая использует ядро в пользовательском пространстве для перехвата системных вызовов, а вторая — легковесную виртуальную машину, выделяемую для каждого контейнера. Такой подход сочетает скорость контейнеров с уровнем безопасности, характерным для виртуальных машин.

Наконец, мы обсудим способы защиты учетных данных приватных реестров контейнеров, которые играют ключевую роль в обеспечении безопасности образов внутри кластера. Вы узнаете, как Kubernetes обрабатывает эти данные, предоставляя доступ к ним только авторизованным компонентам. В ходе изучения этой главы вы получите углубленное понимание продвинутых механизмов обеспечения безопасности, предусмотренных в Kubernetes. Вы узнаете, как усилить защиту своего кластера, минимизировать риски и обезопасить себя от потенциальных уязвимостей. Освоив описанные техники, вы сможете эффективно защищать развертывания Kubernetes на всех уровнях — от управления идентификацией до изоляции среды выполнения — и повышать надежность контейнеризованных приложений.

Технические требования

Для работы с этой главой вам потребуется:

- развернутый кластер Kubernetes. Мы рекомендуем отдать предпочтение многоузловому или облачному кластеру;
- интерфейс командной строки `kubectl`, установленный на локальной машине и настроенный для управления кластером Kubernetes.

Инструкции по созданию кластера Kubernetes и настройке клиента `kubectl` приведены в главах 2 и 3.

В предыдущих главах (15, 16 и 17) рассматривался процесс развертывания полнофункциональных кластеров Kubernetes в различных облачных средах и установки соответствующих CLI-инструментов для управления ими.

Примеры кода для этой главы можно загрузить из официального репозитория GitHub по адресу https://github.com/PacktPublishing/The-Kubernetes-Bible-Second-Edition/tree/main/Chapter18.

Аутентификация и авторизация: управление пользовательским доступом

Эти механизмы составляют основу системы управления доступом, обеспечивая использование ресурсов Kubernetes только аутентифицированными и авторизованными пользователями. Аутентификация связана с подтверждением подлинности пользователя, а авторизация — с определением того, какие действия этот пользователь может выполнять в пределах кластера. Kubernetes поддерживает различные методы аутентификации, включая сертификаты X.509, протокол OpenID Connect и токены. В сочетании с процедурой проверки подлинности механизм RBAC предоставляет детализированный контроль над правами пользователей, что позволяет администраторам эффективно управлять обширным набором разрешений. Более подробно эта тема будет рассмотрена чуть позже.

А в следующем разделе мы поговорим об аутентификации.

Аутентификация и управление пользователями

API-сервер Kubernetes предоставляет REST-интерфейсы для управления кластером и выступает в роли фронтенда, обеспечивающего доступ к его разделяемому состоянию. Все операции с кластером, инициируемые как пользователями, так и внутренними компонентами, осуществляются через этот API-сервер.

Далее мы рассмотрим принцип работы механизма аутентификации в Kubernetes.

Процесс аутентификации в Kubernetes

Подобно режимному объекту, кластер Kubernetes нуждается в надежных мерах безопасности для защиты своих ресурсов. Эта защита реализуется с помощью многоуровневого подхода, задействующего несколько ключевых компонентов, показанных на следующей схеме (рис. 18.1).

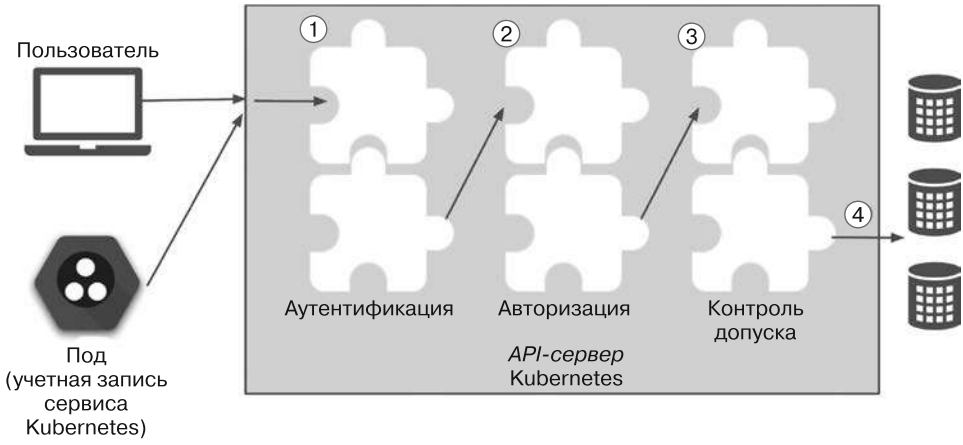

Рис. 18.1. Запрос к API Kubernetes проходит несколько этапов (источник: https://kubernetes.io/docs/concepts/security/controlling-access/)

- *Аутентификация*. Это первая линия обороны, предназначенная для проверки подлинности пользователя, пытающегося получить доступ к серверу API Kubernetes. Ее можно уподобить охраннику, проверяющему документы на входе. Для прохождения процесса аутентификации пользователи могут применять пароли, токены или специальные сертификаты, подтверждающие их право доступа.
- *Авторизация*. После подтверждения подлинности система определяет, какие действия пользователь может выполнять в кластере. Это можно сравнить с назначением разных уровней доступа: одни пользователи могут только просматривать ресурсы, другие — изменять их, а третьи — создавать новые, но лишь в определенных областях.
- *Контроль допуска*. Этот этап обеспечивает дополнительный уровень проверки, подобно сканеру безопасности на входе. Модули контроля допуска анализируют входящие запросы, проверяя их на соответствие заданным политикам безопасности. Они могут изменять запросы для соблюдения определенных правил или отклонять их, если они представляют угрозу.
- *Аудит*. Подобно журналу посещений режимного объекта, система аудита в Kubernetes фиксирует все действия в кластере, выполняемые как пользователями и приложениями, так и компонентами управляющего слоя. Эти журналы играют ключевую роль при мониторинге подозрительной активности и поддержании безопасной среды.

Совместная работа этих механизмов формирует многоуровневую систему защиты, предоставляющую доступ к кластеру только авторизованным пользова-

телям и гарантирующую соответствие их действий установленным политикам безопасности.

В следующем подразделе мы рассмотрим механизм аутентификации более подробно.

Аутентификация в API Kubernetes

Аутентификация в API Kubernetes гарантирует то, что только авторизованные пользователи и сервисы смогут взаимодействовать с ресурсами кластера. Каждый входящий запрос проходит через цепочку настроенных модулей аутентификации.

Все запросы к API делятся на две категории:

- запросы от обычного внешнего пользователя или учетной записи сервиса (ServiceAccount), определенной в кластере Kubernetes;
- *анонимные* запросы, если конфигурация кластера их допускает.

Тип запроса определяется на этапе *аутентификации*. В качестве входных данных процесса проверки используется весь HTTP-запрос, однако обычно анализу подвергаются только заголовки запроса или клиентский сертификат. Аутентификация выполняется с помощью специальных модулей, активизированных в конфигурации кластера. Если активизировано несколько модулей, они выполняются поочередно, пока один из них не подтвердит запрос. В случае непрохождения аутентификации API-сервер возвращает код HTTP-ответа `401` (Unauthorized) или обрабатывает запрос как анонимный, если такие запросы являются допустимыми.

Анонимные запросы автоматически сопоставляются со специальным именем пользователя system:anonymous и группой system:unauthenticated. Это позволяет настраивать правила авторизации для таких запросов точно так же, как для обычных пользователей или учетных записей сервисов.

Все внутренние и внешние операции с кластером осуществляются через API-сервер Kubernetes и, следовательно, требуют аутентификации. Это правило распространяется на действия внутренних компонентов кластера и подов, обращающиеся к API-серверу. Запросы, которые вы, как внешний пользователь, отправляете через `kubectl` или напрямую в API, тоже проходят эту процедуру.

- *Обычные пользователи*. Управление такими пользователями осуществляется *вне* кластера. На данный момент в Kubernetes нет встроенных объектов для их представления. Внешнее управление пользователями может быть реализовано простыми (хотя и *не*рекомендуемыми) методами, например

с помощью статических файлов с учетными данными, которые при запуске передаются API-серверу через параметр `--token-auth-file`, указанный в манифесте пода `/etc/kubernetes/manifests/kube-apiserver.yaml` на узлах управляющего слоя. В рабочих средах рекомендуется задействовать таких *поставщиков услуг идентификации* (Identity Providers, IdP), как *Google*, *GitHub*, *Azure Active Directory (AAD)* или *AWS IAM*. Для интеграции Kubernetes с этими системами применяется механизм *OpenID Connect* (*OIDC*, https://openid.net/connect/), обеспечивающий бесшовную аутентификацию. Следует помнить, что учетные записи обычных пользователей в Kubernetes являются глобальными и не ограничиваются пространствами имен.

- *Учетные записи сервисов*. Эти учетные записи управляются самим кластером и представлены объектами типа ServiceAccount. Их можно создавать и изменять так же, как любые другие ресурсы Kubernetes, — с помощью утилиты `kubectl` или YAML-манифестов. Объекты ServiceAccount предназначены для процессов, работающих внутри кластера, например в подах или компонентах управления. Учетные данные для них создаются в виде объектов Secret (вручную или через API TokenRequest) и монтируются в поды, что позволяет контейнерным процессам использовать их для обращения к API-серверу. Процесс, прошедший аутентификацию с помощью токена `ServiceAccount`, определяется как пользователь с именем `system:serviceaccount:<пространство_имен>:<имя_ServiceAccount>`. Обратите внимание на то, что учетные записи сервисов ограничены пространством имен.

Как видите, для управления пользователями в Kubernetes предусмотрено несколько подходов, что позволяет адаптировать систему под нужды различных организаций. Главное то, что после прохождения аутентификации запрос либо отклоняется (в некоторых случаях обрабатывается как анонимный), либо ассоциируется с конкретным источником. В случае обычных пользователей атрибут `username` предоставляется внешней системой, а в случае учетных записей сервисов — формируется по шаблону: `system:serviceaccount:<пространство_имен>:<имя_ServiceAccount>`. Помимо `username`, запросу присваиваются и другие атрибуты, например `uid` (идентификатор пользователя), `groups` (группы) и `extra` (дополнительные поля). Эта информация используется на этапе авторизации с помощью механизма RBAC, который мы обсудим в следующих подразделах.

Далее мы рассмотрим методы аутентификации, поддерживаемые в Kubernetes.

Методы аутентификации в Kubernetes

Механизмы аутентификации позволяют контролировать доступ к API-серверу Kubernetes. Для проверки подлинности пользователей и сервисов можно задействовать несколько стратегий аутентификации, каждая из которых подходит

для определенных сценариев и уровней безопасности. Эти стратегии опираются на токены и сертификаты, подтверждающие подлинность пользователей и приложений, взаимодействующих с кластером. Преимущество API-сервера Kubernetes заключается в поддержке нескольких механизмов аутентификации, что позволяет комбинировать различные подходы в рамках одного кластера. В следующих разделах мы рассмотрим наиболее распространенные методы аутентификации, включая статические файлы с токенами, токены ServiceAccount, клиентские сертификаты X.509 и токены OpenID Connect.

Статические файлы с токенами. Это самый простой метод аутентификации, предусмотренный в Kubernetes для управления обычными пользователями. По своей сути он напоминает использование файлов `/etc/shadow` и `/etc/passwd` в системах Unix/Linux. Однако следует подчеркнуть, что этот метод *не* рекомендуется использовать, так как он считается *небезопасным* для кластеров, развернутых в рабочей среде.

Такой подход предполагает создание CSV-файла, в котором каждая строка имеет следующий формат:

```
token,user,uid,"group1,group2,group3"
```

При запуске API-сервера Kubernetes этот файл передается ему через параметр `--token-auth-file`, указанный в статическом манифесте пода `/etc/kubernetes/manifests/kube-apiserver.yaml` на узлах управляющего слоя (или на мастер-узлах):

```
# /etc/kubernetes/manifests/kube-apiserver.yaml
... <фрагмент опущен для краткости> ...
spec:
  containers:
  - command:
    - kube-apiserver
    - --advertise-address=192.168.59.154
    - --allow-privileged=true
    - --authorization-mode=Node,RBAC
    - --token-auth-file=/etc/kubernetes/user-tokens.csv
    - --client-ca-file=/var/lib/minikube/certs/ca.crt
... <фрагмент опущен для краткости> ...
```

Для аутентификации на API-сервере необходимо использовать стандартную *схему HTTP-аутентификации Bearer*, добавив в запрос следующий заголовок:

```
Authorization: Bearer <токен>
```

Получив запрос, API-сервер Kubernetes сопоставит указанный токен с содержимым статического файла и присвоит пользователю атрибуты из соответствующей записи.

При работе с `kubectl` необходимо внести изменения в файл конфигурации `kubeconfig`, выполнив команду:

```
$ kubectl config set-credentials <контекстный_пользователь> --token=<токен>
```

После этого нужно создать контекст с этим пользователем и активизировать его для отправки запросов с помощью команды `kubectl config use-context`.

До выхода версии 1.19 в Kubernetes существовал механизм, позволяющий использовать схему HTTP-аутентификации Basic и файл, передаваемый API-серверу через параметр --basic-auth-file. Однако по соображениям безопасности этот метод больше не поддерживается.

Принцип работы этого метода аутентификации представлен на следующей схеме (рис. 18.2).

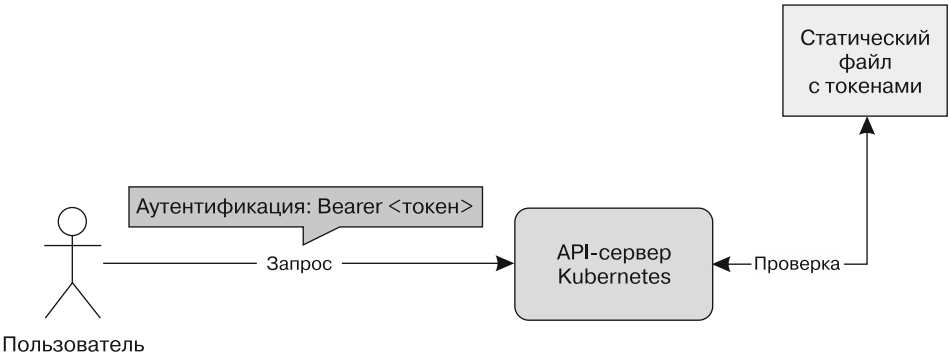

Рис. 18.2. Аутентификация в Kubernetes с помощью статического файла с токенами

Теперь перечислим преимущества и недостатки метода аутентификации на основе статического файла с токенами.

Преимущества:

- простая настройка;
- понятный принцип работы.

Недостатки:

- низкий уровень безопасности, поскольку компрометация файла с токенами приводит к утечке данных всех пользователей кластера;
- необходимость в ручном управлении пользователями;

- добавление или удаление пользователей требует перезапуска API-сервера Kubernetes;
- ротация токенов также требует перезапуска API-сервера;
- в высокодоступных кластерах синхронизация файла с токенами между несколькими управляющими узлами требует дополнительных усилий.

В целом этот метод подходит для среды разработки и знакомства с основами аутентификации в Kubernetes, но не рекомендуется для использования в рабочей среде. Далее мы рассмотрим процесс аутентификации пользователей с помощью токенов ServiceAccount.

Токены ServiceAccount. Хотя объекты ServiceAccount (учетные записи сервисов) предназначены для идентификации процессов, выполняющихся внутри подов, или компонентов кластера, их можно использовать и для аутентификации внешних запросов.

ServiceAccount — это объект Kubernetes, которым можно управлять так же, как и другими ресурсами кластера, — с помощью `kubectl` или прямых HTTP-запросов к API-серверу. Токены ServiceAccount представляют собой объекты *JSON Web Token (JWT)* и генерируются по запросу или с помощью команды `kubectl create token`.

Каждое пространство имен Kubernetes содержит предварительно созданную учетную запись сервиса default. Поды, для которых явно не указана учетная запись сервиса, автоматически наследуют эту учетную запись по умолчанию для авторизации внутри кластера. Вы можете выяснить, какая учетная запись сервиса используется подом, выполнив команду kubectl get pods/ <имя_пода> -o yaml и просмотрев значение поля spec.serviceAccountName.

Обычно при определении пода можно указать учетную запись сервиса, которая должна использоваться для процессов внутри контейнера. Это делается через поле `.spec.serviceAccountName` в спецификации пода. JWT-токен внедряется в контейнер, после чего выполняющийся внутри процесс может использовать его для аутентификации на API-сервере Kubernetes по схеме Bearer. Это требуется только в том случае, если контейнеру необходимо взаимодействовать с API Kubernetes, например, для обнаружения других подов в кластере. Принцип работы этого метода аутентификации представлен на следующей схеме (рис. 18.3).

Эта схема также объясняет, почему токены ServiceAccount могут применяться для аутентификации внешних запросов: API-сервер Kubernetes не анализирует источник запроса, его интересует лишь токен, переданный в его заголовке. Этот токен можно использовать как в командах `kubectl`, так и в прямых HTTP-запросах к API-серверу. Тем не менее использование токенов ServiceAccount для аутентификации внешних запросов не является рекомендуемой практикой.

Такой подход допустим лишь в отдельных случаях, например при невозможности использования услуг внешнего провайдера для аутентификации обычных пользователей.

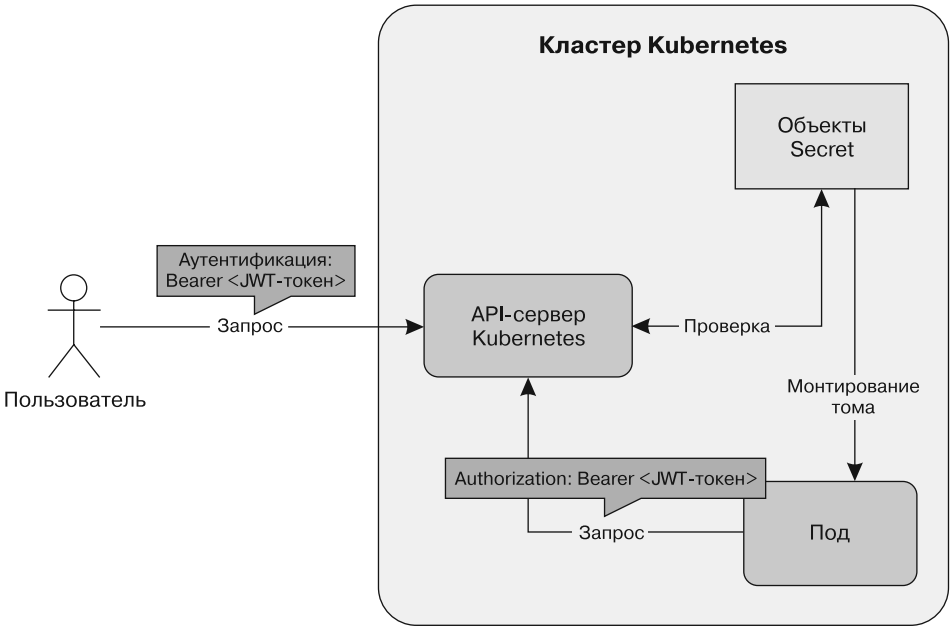

Рис. 18.3. Аутентификация в Kubernetes с помощью токенов ServiceAccount

До выхода версии 1.22 система Kubernetes автоматически создавала учетные данные API для объектов ServiceAccount с использованием объектов Secret. Эти объекты содержали токены, которые поды могли монтировать для получения доступа. Однако такой подход имел ряд ограничений.

- *Статические токены*. Токены хранились в объекте Secret в виде открытого текста, что представляло угрозу безопасности в случае компрометации.
- *Ограниченная степень контроля*. Срок действия и разрешения токенов было сложно контролировать.

Начиная с версии 1.22, в Kubernetes используется более безопасный механизм. Теперь поды получают токены напрямую с помощью API *TokenRequest*. Эти токены:

- *являются короткоживущими*. Токены имеют ограниченный срок жизни, что уменьшает последствия возможной утечки;
- *автоматически монтируются в поды*. Токены автоматически монтируются в виде томов, что устраняет потребность в заранее созданных объектах Secret.

Хотя автоматическое монтирование является предпочтительным вариантом, при необходимости вы можете вручную создавать объекты Secret для токенов учетных записей сервисов. Это может быть полезно в тех случаях, когда токену требуется более длительный срок действия, однако в большинстве ситуаций для повышения уровня безопасности рекомендуется выбирать автоматическое монтирование токенов.

Как мы уже выяснили, Kubernetes автоматически монтирует учетные данные API объекта ServiceAccount в поды для упрощения доступа к кластеру. Чтобы отключить это поведение и управлять токенами другим способом, задайте параметр automountServiceAccountToken: false в манифесте ServiceAccount или в спецификации пода. Это значение применяется ко всем подам, использующим данную учетную запись сервиса, если только оно не переопределено в конфигурации конкретного пода. При одновременном указании параметра в обоих местах приоритет имеет настройка пода. Подробнее об этом параметре можно узнать в официальной документации по адресу https://kubernetes.io/docs/tasks/configure-pod-container/configure-service-account/#opt-out-of-api-credential-automounting.

Далее мы покажем, как создавать и управлять объектами ServiceAccount, а также продемонстрируем способ применения JWT-токенов для аутентификации при работе с `kubectl`. Это даст вам некоторое представление о механизме RBAC, с которым вы познакомитесь чуть позже. А пока выполните следующие шаги.

1. Создайте YAML-манифест для нового пространства имен и объекта ServiceAccount. Мы настроим механизм RBAC таким образом, чтобы эта учетная запись могла просматривать поды только в данном пространстве имен:

```
# 01_serviceaccount/example-sa-ns.yaml
---
apiVersion: v1
kind: Namespace
metadata:
  name: example-ns

---
apiVersion: v1
kind: ServiceAccount
metadata:
  name: example-sa
  namespace: example-ns
```

   Учтите, что те же ресурсы можно создать и *императивным* способом с помощью команды `kubectl create serviceaccount example-sa`.

2. Создайте YAML-манифест для объекта `Role` с именем `pod-reader` в пространстве имен `example-ns`. Эта роль предоставит вам права на получение (`get`), отслеживание (`watch`) и вывод списка (`list`) подов в этом пространстве

имен. Файл `01_serviceaccount/pod-reader-role.yaml` должен содержать следующий манифест:

```
# 01_serviceaccount/pod-reader-role.yaml
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  namespace: example-ns
  name: pod-reader
rules:
- apiGroups: [""]
  resources: ["pods"]
  verbs: ["get", "watch", "list"]
```

3. Создайте YAML-манифест для объекта `RoleBinding` с именем `read-pods`. Этот объект *связывает* созданную роль `pod-reader` с учетной записью сервиса `example-sa`. В результате учетная запись получает доступ к подам только для чтения и никаких других прав. Файл `01_serviceaccount/read-pods-rolebinding.yaml` должен содержать следующий манифест:

```
# 01_serviceaccount/read-pods-rolebinding.yaml
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: read-pods
  namespace: example-ns
subjects:
- kind: ServiceAccount
  name: example-sa
  namespace: example-ns
roleRef:
  kind: Role
  name: pod-reader
  apiGroup: rbac.authorization.k8s.io
```

4. Примените все созданные манифесты к кластеру с помощью команды `kubectl apply -f`:

```
$ kubectl apply -f 01_serviceaccount/
namespace/example-ns created
serviceaccount/example-sa created
role.rbac.authorization.k8s.io/pod-reader created
rolebinding.rbac.authorization.k8s.io/read-pods created
```

5. Создайте токен для учетной записи сервиса:

```
$ kubectl create token example-sa -n example-ns
```

Скопируйте JWT-токен из вывода команды — он понадобится для аутентификации от имени этой учетной записи. При желании вы можете просмотреть содержимое токена на сайте https://jwt.io, как показано на рис. 18.4.

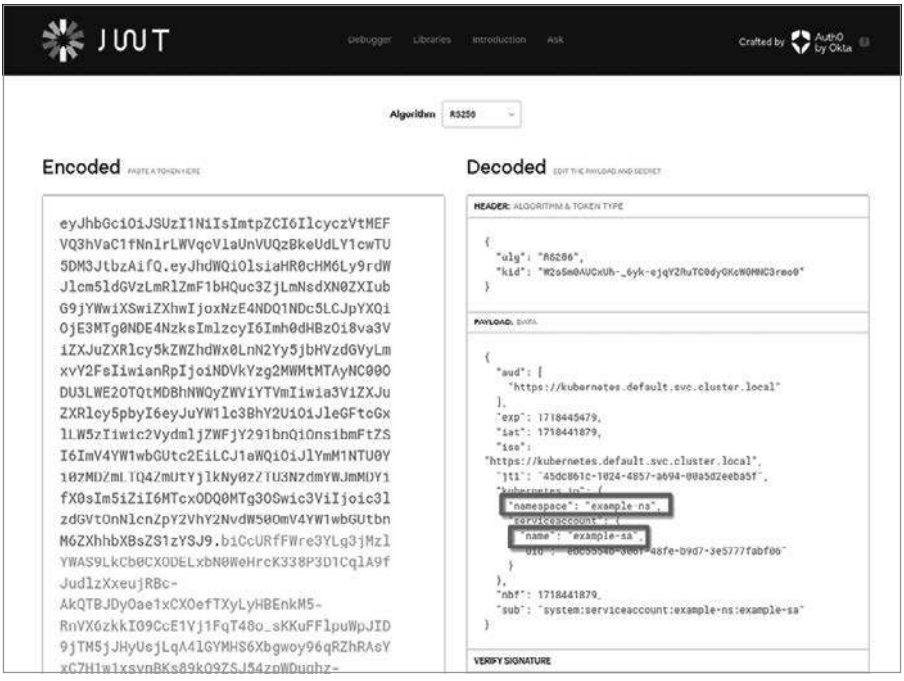

Рис. 18.4. Проверка содержимого JWT-токена для объекта ServiceAccount

Как видите, JWT-токен соответствует объекту `example-sa` в пространстве имен `example-ns`. Кроме того, в поле `sub` раздела `PAYLOAD` указано фактическое имя пользователя, которое будет использоваться в Kubernetes: `system:serviceaccount:example-ns:example-sa`.

6. Теперь настройте `kubeconfig` для использования этого токена. Сначала создайте нового пользователя с помощью команды:

```
$ kubectl config set-credentials example-sa --token=<ваш_токен>
User "example-sa" set.
```

где `example-sa` — это имя созданного объекта ServiceAccount, а *<ваш_токен>* — скопированное ранее значение токена.

7. Создайте новый контекст для этого пользователя в файле `kubeconfig`. Сначала узнайте имя текущего кластера с помощью команды `kubectl config view`, а затем создайте контекст, выполнив команду `kubectl config set-context`:

```
$ kubectl config set-context <имя_нового_контекста> --user=<новый_
пользователь> --cluster=<имя_кластера>
```

Например, следующая команда создает контекст `example-sa-context`, использующий кластер `minikube` и пользователя `example-sa`:

```
$ kubectl config set-context example-sa-context --user=example-sa \
    --cluster=minikube
Context "example-sa-context" created.
```

8. Прежде чем переключиться на новый контекст, создайте простой под `nginx` в пространстве имен `example-ns`. Скопируйте пример манифеста `Chapter18/references/sa-demo-nginx-pod.yaml` в файл `Chapter18/01_serviceaccount/nginx-pod.yaml` и примените конфигурацию:

```
$ cp references/sa-demo-nginx-pod.yaml 01_serviceaccount/nginx-pod.yaml
$ kubectl apply -f 01_serviceaccount/nginx-pod.yaml
pod/nginx-pod created
$ kubectl get po -n example-ns
NAME         READY   STATUS    RESTARTS   AGE
nginx-pod    1/1     Running   0          12m
```

9. Перед переключением контекста вы также можете проверить имя текущего контекста, чтобы позднее переключиться обратно. Для этого используйте следующую команду:

```
$ kubectl config current-context
minikube
```

10. Переключитесь на новый контекст:

```
$ kubectl config use-context example-sa-context
Switched to context "example-sa-context".
```

11. Проверьте, чьи учетные данные используются[1]:

```
$ kubectl auth whoami
ATTRIBUTE                  VALUE
Username                   system:serviceaccount:example-ns:example-sa
UID                        ebc5554b-306f-48fe-b9d7-3e5777fabf06
Groups                     [system:serviceaccounts system:
                           serviceaccounts:example-ns system:authenticated]
Extra: authentication.kubernetes.io/credential-id  [JTI=45dc861c-1024-4857-
a694-00a5d2eeba5f]
```

12. Теперь можно убедиться в том, что аутентификация выполняется успешно, а роли RBAC предоставляют доступ к подам в пространстве имен `example-ns` только для чтения. Для начала выведите список подов:

---

[1] Команда kubectl auth whoami пока находится на экспериментальной стадии. В зависимости от версии kubectl может быть такой: kubectl alpha auth whoami.

```
$ kubectl get po -n example-ns
NAME         READY   STATUS    RESTARTS   AGE
nginx-pod    1/1     Running   0          18m
```

13. Как видите, результат соответствует ожиданиям! Теперь попробуйте получить список подов из пространства имен `kube-system`:

```
$ kubectl get pods -n kube-system
Error from server (Forbidden): pods is forbidden: User
"system:serviceaccount:example-ns:example-sa" \
cannot list resource "pods" in API group "" in the namespace "kube-system"
```

14. Аутентификация прошла успешно, однако механизм RBAC правомерно запретил выполнение операции. В завершение попробуйте получить список сервисов:

```
$ kubectl get svc -n example-ns
Error from server (Forbidden): services is forbidden: User
"system:serviceaccount:example-ns:example-sa" \
cannot list resource "services" in API group "" in the namespace "example-ns"
```

Этот результат также является ожидаемым, поскольку в механизме RBAC не заданы правила, разрешающие этому объекту ServiceAccount просматривать ресурсы типа Service в пространстве имен `example-ns`.

Итак, мы успешно аутентифицировались с помощью токена ServiceAccount и проверили свои привилегии. Теперь можно переключиться на прежний контекст с помощью команды `kubectl config use-context <имя_контекста>`.

Описанную процедуру настройки контекста kubectl с bearer-токеном можно использовать и при аутентификации с помощью статического файла с токенами.

Ниже перечислены преимущества и недостатки метода аутентификации с помощью токенов ServiceAccount.

Преимущества:

- простота настройки и использования;
- управление осуществляется кластером Kubernetes, что избавляет от потребности во внешних поставщиках услуг аутентификации;
- объекты ServiceAccount ограничены пространством имен.

Недостаток состоит в том, что объекты ServiceAccount предназначены для идентификации процессов, выполняющихся внутри подов, и предоставления им возможности использовать механизм RBAC системы Kubernetes. *Использование токенов ServiceAccount обычными пользователями не является хорошей практикой.*

В целом применение токенов ServiceAccount для внешней аутентификации допустимо только в средах разработки и тестирования при невозможности использования соответствующих услуг внешнего провайдера. Применять этот метод в рабочей среде не рекомендуется по соображениям безопасности. Далее мы поговорим об использовании клиентских сертификатов X.509 для аутентификации в API Kubernetes.

Клиентские сертификаты X.509. Использование клиентских сертификатов X.509 является одним из отраслевых стандартов в системах аутентификации. Однако важно обеспечить надежное управление процессами подписания, отзыва и ротации сертификатов. В противном случае можно столкнуться с теми же проблемами безопасности, что и при использовании токенов ServiceAccount. Подробнее о сертификатах X.509 и связанных с ними процессах можно узнать по адресу https://www.ssl.com/faqs/what-is-an-x-509-certificate/.

Применение этого метода в Kubernetes выглядит следующим образом.

1. API-сервер Kubernetes запускается с параметром `--client-ca-file`. Этот параметр указывает на файл с информацией о *центре сертификации* (CA, Certification authority), который используется для проверки подлинности клиентских сертификатов, предоставленных при аутентификации. Вы можете задать пользовательский CA-сертификат или использовать CA-сертификат по умолчанию, созданный при развертывании кластера. Например, при использовании кластера `minikube` можно увидеть, что в конфигурации `kube-apiserver` уже задан CA-сертификат по умолчанию:

```
# /etc/kubernetes/manifests/kube-apiserver.yaml
... <фрагмент опущен для краткости> ...
spec:
  containers:
  - command:
    - kube-apiserver
    - --advertise-address=192.168.59.154
    - --allow-privileged=true
    - --authorization-mode=Node,RBAC
    - --client-ca-file=/var/lib/minikube/certs/ca.crt
... <фрагмент опущен для краткости> ...
```

2. Для аутентификации на API-сервере пользователи должны запросить у центра сертификации клиентский сертификат X.509. Этот процесс должен быть защищенным и контролируемым. После успешной аутентификации содержимое поля сертификата `CN` (Common Name) используется в качестве атрибута `username` (имя пользователя). Начиная с версии Kubernetes 1.19, для управления запросами на подпись сертификатов можно использовать API Certificates. Подробнее об этом можно узнать в официальной документации по адресу https://kubernetes.io/docs/reference/access-authn-authz/certificate-signing-requests/.

3. Пользователь должен предоставить клиентский сертификат при аутентификации на API-сервере. API-сервер проверяет этот сертификат на соответствие корневому центру сертификации. В зависимости от результата запрос либо успешно проходит процедуру аутентификации, либо отклоняется. Если вы используете кластер `minikube`, то фактически уже применяете метод аутентификации на основе сертификатов, как показано в следующем примере:

```
$ kubectl config view -o json | jq '.users[]'
{
  "name": "example-sa",
  "user": {
    "token": "REDACTED"
  }
}
{
  "name": "minikube",
  "user": {
    "client-certificate": "/home/iamgini/.minikube/profiles/minikube/
     client.crt",
    "client-key": "/home/iamgini/.minikube/profiles/minikube/client.key"
  }
}
```

При работе с `kubectl` пользователи могут настроить этот метод аутентификации в файле `kubeconfig` с помощью команды `kubectl config set-credentials`, как было показано ранее. На рис. 18.5 представлена схема, обобщающая описанный выше процесс.

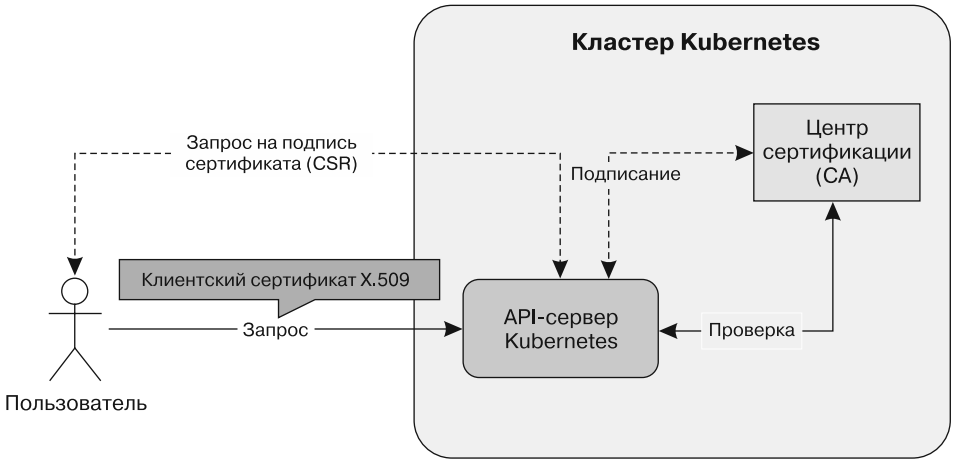

Рис. 18.5. Аутентификация в Kubernetes с использованием клиентских сертификатов X.509

Следует отметить, что на схеме изображен случай, когда исходный запрос на подпись сертификата (CSR, Certificate Signing Request) обрабатывается API Certificate внутри кластера Kubernetes. Однако это не является обязательным — центр сертификации может быть внешним по отношению к кластеру, а API-сервер Kubernetes может использовать копию файла CA в формате `.pem` для проверки клиентских сертификатов.

В следующем упражнении мы настроим аутентификацию в Kubernetes с использованием клиентских сертификатов.

1. Создайте закрытый ключ с помощью команды `openssl`:

```
$ openssl genrsa -out iamgini.key 2048
```

2. Сформируйте запрос на подпись сертификата (CSR):

```
$ openssl req -new -key iamgini.key -out iamgini.csr -subj "/CN=iamgini/
O=web1/O=frontend"
```

3. Просмотрите содержимое CSR и закодируйте его в формат base64:

```
$ cat iamgini.csr | base64 -w 0
```

4. Теперь нам нужно создать ресурс `CertificateSigningRequest` с помощью *API Certificates*. Для этого понадобится YAML-файл `csr.yaml` со следующим содержимым:

```
# csr.yaml
apiVersion: certificates.k8s.io/v1
kind: CertificateSigningRequest
metadata:
  name: iamgini
spec:
  request: <содержимое CSR, закодированное на шаге 3>
  signerName: kubernetes.io/kube-apiserver-client
  usages:
    - client auth
```

5. Примените манифест для создания CSR:

```
$ kubectl apply -f csr.yaml
certificatesigningrequest.certificates.k8s.io/iamgini created
```

6. Теперь администраторы (или пользователи с привилегией `certificatesigningrequests`) могут просматривать ресурсы CSR:

```
$ kubectl get csr
NAME     AGE   SIGNERNAME                              REQUESTOR
REQUESTEDDURATION   CONDITION
iamgini   25s  kubernetes.io/kube-apiserver-client   minikube-user <none>
Pending
```

7. Проверьте и одобрите CSR следующим образом:

```
$ kubectl certificate approve iamgini
certificatesigningrequest.certificates.k8s.io/iamgini approved
```

8. После одобрения CSR извлеките из него данные сертификата и сохраните их в файл `iamgini.crt` с помощью следующей команды:

```
$ kubectl get csr iamgini -o json | jq -r '.status.certificate' | base64
--decode > iamgini.crt
```

9. Итак, у нас есть закрытый ключ и сертификат (файл `.csr` можно удалить — он нам больше не понадобится):

```
$ ls iamgini.*
iamgini.crt  iamgini.csr  iamgini.key
```

10. Теперь настроим `kubeconfig`, добавив нового пользователя и контекст. Создайте запись о пользователе в `kubeconfig` следующим образом (не забудьте указать полный путь к файлам ключа и сертификата):

```
$ kubectl config set-credentials iamgini \
    --client-key=/full-path/iamgini.key \
    --client-certificate=/full-path/iamgini.crt
User "iamgini" set.
```

11. Создайте контекст, связанный с новым пользователем:

```
$ kubectl config set-context iamgini --cluster=minikube --user=iamgini
Context "iamgini" created.
```

12. После обновления `kubeconfig` можно проверить доступ. Для этого переключите контекст с помощью следующей команды:

```
$ kubectl config use-context iamgini
Switched to context "iamgini".
```

13. Проверьте текущий контекст и подключение:

```
$ kubectl auth whoami
ATTRIBUTE   VALUE
Username    iamgini
Groups      [web1 frontend system:authenticated]
```

Итак, мы успешно настроили аутентификацию на основе сертификата X.509. Однако учтите, что созданный нами пользователь не сможет выполнять никакие операции до тех пор, пока вы не настроите соответствующие ресурсы RBAC.

Ниже перечислены преимущества и недостатки использования сертификатов X.509 для прохождения аутентификации.

Преимущества:

- более высокая безопасность по сравнению с использованием токенов ServiceAccount и статических файлов с токенами;
- сертификаты не хранятся внутри кластера, что исключает риск компрометации сразу всех учетных данных. Клиентские сертификаты X.509 можно использовать для учетных записей с повышенными привилегиями;
- при необходимости сертификаты можно отозвать, что особенно важно при возникновении инцидентов.

Недостатки:

- сертификаты имеют ограниченный срок действия. В простых сценариях разработки это может быть неудобно, но в рабочих средах это является преимуществом с точки зрения безопасности. Однако помните, что сертификат необходимо хранить в безопасном месте, поскольку механизм аутентификации на основе файлов представляет угрозу безопасности, так как файл может быть похищен и использован для получения несанкционированного доступа;
- необходимость контролировать сроки действия, отзыв и ротацию сертификатов. Эти процессы следует автоматизировать, чтобы быстрее реагировать на инциденты;
- использование клиентских сертификатов для аутентификации в браузере может создавать трудности, например, при попытке входа в веб-интерфейс Kubernetes Dashboard.

В целом аутентификация с использованием клиентских сертификатов X.509 — это безопасное решение, однако для того, чтобы воспользоваться всеми его преимуществами, необходимо грамотно управлять сертификатами. Далее мы обсудим токены OpenID Connect — рекомендуемый метод аутентификации для облачных сред.

Токены OpenID Connect. Используя протокол *OpenID Connect (OIDC)*, вы можете реализовать механизм *единого входа* (Single Sign-On, SSO) для вашего кластера Kubernetes (а также для других ресурсов вашей организации). OIDC представляет собой уровень аутентификации, работающий поверх протокола OAuth 2.0, который позволяет сторонним приложениям проверять подлинность конечного пользователя и получать базовую информацию о его профиле. OIDC использует JWT-токены, которые можно получить с помощью потоков аутентификации, соответствующих спецификации OAuth 2.0. Основная проблема при использовании OIDC для аутентификации в Kubernetes заключается в ограниченной доступности OpenID-провайдеров. Преимуществом *управляемых* облачных Kubernetes-кластеров (Amazon EKS, AKS и GKE) является

интеграция с нативным OpenID-провайдером, которая доступна по умолчанию или активируется минимальными изменениями конфигурации. Иными словами, вам не нужно вручную настраивать API-сервер Kubernetes на работу с выбранным OpenID-провайдером, так как соответствующая функция уже встроена в управляемое решение. Подробнее о протоколе OIDC можно узнать на официальной веб-странице проекта: https://openid.net.

Для более детального изучения потоков аутентификации, например, в контексте Azure Active Directory (AAD) обратитесь к документации Microsoft: https://docs.microsoft.com/en-us/azure/active-directory/develop/v2-protocols-oidc.

На следующей схеме показаны основные этапы процесса аутентификации в Kubernetes с использованием OIDC (рис. 18.6).

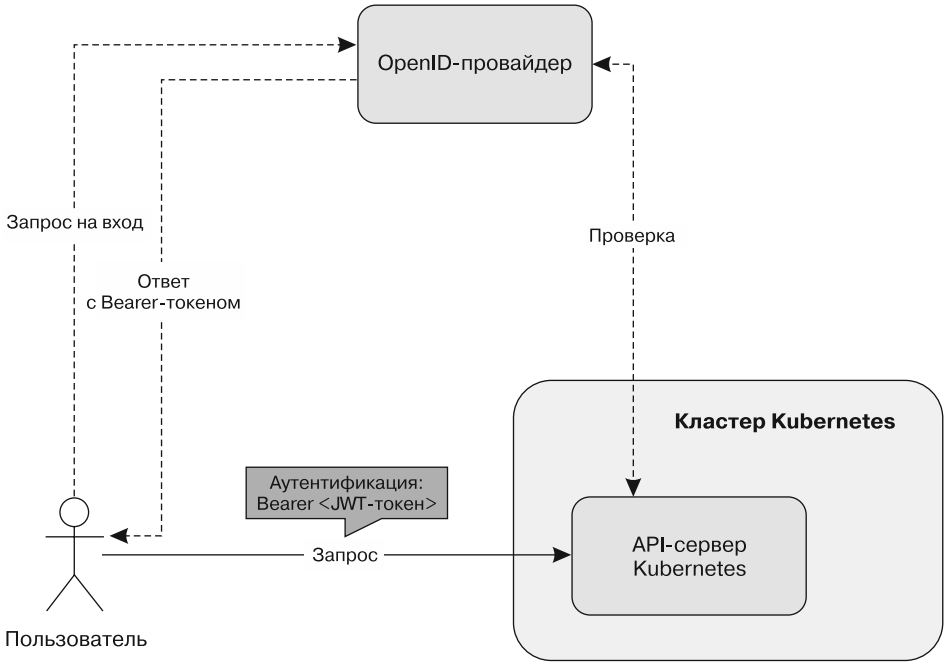

Рис. 18.6. Аутентификация в Kubernetes с использованием OIDC

Суть заключается в том, что именно OpenID-провайдер отвечает за реализацию единого входа (SSO) и управление bearer-токенами, а API-сервер Kubernetes проверяет полученные токены, сверяя их с данными OpenID-провайдера.

Ниже перечислены преимущества и недостатки использования OIDC для прохождения аутентификации.

Преимущества:

- реализация механизма единого входа (SSO), который можно использовать с другими сервисами организации;
- у большинства поставщиков облачных услуг есть встроенные OpenID-провайдеры, которые легко интегрируются с их управляемыми Kubernetes-сервисами;
- этот механизм можно использовать и с другими OpenID-провайдерами, а также в локальных развертываниях, правда это потребует чуть более сложной настройки;
- безопасное и масштабируемое решение.

Недостатки:

- в Kubernetes отсутствует веб-интерфейс, в котором можно было бы инициировать процесс аутентификации, поэтому учетные данные приходится запрашивать вручную у IdP-сервиса. В управляемых облачных Kubernetes-кластерах эту проблему можно решить с помощью вспомогательных инструментов, которые автоматически формируют файл `kubeconfig` с актуальными учетными данными;
- токены OIDC могут быть отозваны IdP-сервисом, если он поддерживает такую функцию. Это позволяет аннулировать токен до истечения его срока действия, например, при компрометации учетной записи пользователя. Однако не все IdP-сервисы поддерживают эту возможность, а сама система Kubernetes не имеет встроенного механизма отзыва токенов.

Использование OIDC в Kubernetes. В Kubernetes отсутствует встроенный OpenID-провайдер, поэтому для аутентификации используются внешние решения — сервисы поставщиков облачных услуг или самостоятельные инструменты. Как уже было сказано, наиболее популярные облачные платформы (AWS, GCP и Azure) предусматривают нативную поддержку OIDC в своих управляемых Kubernetes-сервисах, что значительно упрощает реализацию механизма единого входа. Для локальных или самоуправляемых Kubernetes-кластеров организации могут настроить собственный IdP-сервис с помощью таких инструментов, как Dex, Keycloak, UAA или OpenUnison.

Требования, предъявляемые к IdP-сервису в Kubernetes. Чтобы IdP-сервис мог корректно работать с Kubernetes, он должен соответствовать следующим требованиям.

- *Поддержка механизма OIDC Discovery*. Этот механизм упрощает процесс конфигурации, предоставляя всю необходимую информацию о конечных точках IdP-сервиса и его открытых ключах. Kubernetes считывает открытые ключи IdP с конечной точки OIDC Discovery и использует их для проверки подлинности OIDC-токенов.

- *Соответствие требованиям протокола TLS*. IdP-сервис должен использовать актуальные, неустаревшие шифры TLS (Transport Layer Security — безопасность транспортного уровня), поскольку от этого зависит безопасность передачи конфиденциальных данных аутентификации.
- *Наличие сертификата, подписанного центром сертификации*. Независимо от того, используете ли вы коммерческий центр сертификации или самоподписанный сертификат, флаг `CA` в сертификате IdP-сервиса должен быть установлен в значение `TRUE`. Это необходимо потому, что Kubernetes использует TLS-клиент, реализованный на языке Go, который требует выполнения этого условия, чтобы доверять сертификатам IdP-сервиса при проверке пользовательских токенов.

При самостоятельном развертывании IdP-сервиса (без участия коммерческого CA) можно использовать такие инструменты, как сценарий gencert из проекта Dex, позволяющий создавать корректный CA-сертификат вместе с ключом подписи.

Ниже перечислены популярные OIDC-провайдеры для Kubernetes.

- *Dex* — один из самых легковесных и популярных IdP-сервисов с открытым исходным кодом, специально разработанный для использования в среде Kubernetes. Он поддерживает протокол OIDC и хорошо вписывается в ожидаемую Kubernetes модель аутентификации. Dex интегрируется с такими внешними IdP-сервисами, как LDAP, GitHub и Google, что делает его хорошим выбором для организаций, использующих сложные сценарии идентификации.
- *Keycloak* — IdP-сервис с открытым исходным кодом и широкой поддержкой протоколов OIDC и SAML, предоставляющий такие расширенные корпоративные функции, как федеративная идентификация и управление доступом на основе ролей (RBAC). Keycloak подходит для случаев, требующих повышенного контроля или глубокой кастомизации механизма аутентификации.
- *OpenUnison* — еще один IdP-сервис, оптимизированный для Kubernetes. Поддерживает нативную интеграцию с механизмом RBAC Kubernetes и федеративную идентификацию. Подходит предприятиям, которым требуется готовая, но настраиваемая система аутентификации для защиты Kubernetes-кластеров.
- *Cloud Foundry User Account and Authentication (UAA)* — многофункциональный IdP-сервис с открытым исходным кодом, изначально разработанный как часть платформы Cloud Foundry. Он поддерживает протокол OIDC и интеграцию с облачными платформами и корпоративными системами аутентификации, поэтому хорошо подходит для сложных Kubernetes-развертываний, работающих в гибридных облачных средах.

Настройка OIDC в API-сервере Kubernetes. Включение поддержки OIDC в Kubernetes требует настройки API-сервера с использованием ряда параметров.

- `--oidc-issuer-url` — URL-адрес OIDC-провайдера, используемый системой Kubernetes для проверки подлинности токенов.
- `--oidc-client-id` — идентификатор клиента, используемый системой Kubernetes при аутентификации в IdP-сервисе.
- `--oidc-username-claim` — указывает, какое утверждение (claim) в токене сопоставляется с именем пользователя Kubernetes.
- `--oidc-groups-claim` — сопоставляет группы в IdP-сервисе с группами Kubernetes для управления ролями RBAC.

Более подробные инструкции по настройке конкретных OIDC-провайдеров можно найти в официальных источниках: https://dexidp.io/docs/guides/kubernetes/ и https://kubernetes.io/docs/reference/access-authn-authz/authentication/.

В целом OIDC представляет собой предпочтительный метод аутентификации в Kubernetes, особенно при развертывании кластеров производственного уровня в облачных средах.

Другие методы аутентификации. Kubernetes поддерживает несколько дополнительных методов аутентификации, предназначенных для более сложных сценариев, например для интеграции с LDAP или Kerberos. Одним из них является *аутентификация с помощью прокси-сервера*.

При использовании аутентифицирующего прокси, установленного перед API-сервером Kubernetes, этот API-сервер можно настроить таким образом, чтобы он извлекал информацию о пользователе из определенных HTTP-заголовков. Иными словами, прокси выполняет аутентификацию пользователя, а затем передает полученные данные вместе с запросом в виде дополнительных заголовков.

Подробнее об этом методе можно узнать в официальной документации по адресу https://kubernetes.io/docs/reference/access-authn-authz/authentication/#authenticating-proxy.

Еще одним методом является *аутентификация токенов с помощью вебхуков*, при которой API-сервер Kubernetes использует внешний сервис для проверки bearer-токенов. В рамках этого подхода API-сервер отправляет данные в виде объекта TokenReview через HTTP-запрос типа `POST`. Внешний сервис выполняет проверку и возвращает объект TokenReview с результатами верификации.

Подробнее об этом методе можно узнать в официальной документации по адресу https://kubernetes.io/docs/reference/access-authn-authz/authentication/#webhook-token-authentication.

Kubernetes также поддерживает аутентификацию с помощью *bootstrap-токенов* (или токенов начальной загрузки). Однако они используются не для обычной аутентификации пользователей, а для добавления узлов в кластер. Bootstrap-токены представляют собой особый тип объектов Secret, упрощающий процесс подключения новых узлов. Эти токены хранятся в пространстве имен `kube-system` и имеют короткий срок действия. С их помощью API-сервер может аутентифицировать `kubelet`-процессы (выполняемые на узлах) при первичном подключении, что существенно упрощает процедуру добавления новых узлов и создания кластера с нуля. Bootstrap-токены можно использовать как с инструментом `kubeadm`, так и без него. Они также беспрепятственно интегрируются с механизмом Kubelet TLS Bootstrapping, обеспечивая безопасный обмен данными. Подробнее об аутентификации с использованием bootstrap-токенов и механизме TLS Bootstrapping можно узнать в документации: https://kubernetes.io/docs/reference/access-authn-authz/bootstrap-tokens.

В целом методы аутентификации с помощью прокси-сервера и вебхуков применяются только в особых случаях, когда необходима интеграция с существующими в организации IdP-сервисами, которые не поддерживаются в Kubernetes по умолчанию.

В следующем подразделе мы поговорим об авторизации и управлении доступом на основе ролей (RBAC).

Авторизация и знакомство с механизмом RBAC

Для обеспечения безопасности в Kubernetes используются два ключевых процесса: *аутентификация* и *авторизация*. Аутентификация — это подтверждение подлинности пользователя, пытающегося получить доступ к системе, гарантирующее то, что он является именно тем, за кого себя выдает. На этом этапе обычно проверяются учетные данные, например имя пользователя и пароль или токены.

После успешной аутентификации выполняется авторизация, определяющая то, какие действия пользователь может выполнять в системе. В Kubernetes решение о предоставлении доступа принимает API-сервер. Он анализирует личность пользователя (подтвержденную при аутентификации) в совокупности с другими параметрами запроса, такими как запрашиваемая конечная точка API или операция. Затем на основе заранее определенных политик или данных внешних сервисов модули авторизации разрешают или запрещают выполнение запроса.

Таким образом, аутентификация отвечает за установление подлинности пользователя, а авторизация — за проверку его прав на выполнение запрошенного действия.

Контроль доступа на основе содержимого полей конкретных объектов осуществляют контроллеры допуска. Их срабатывание происходит после этапа авторизации и только в том случае, если выполнение запроса было разрешено. Подробнее о контроллерах допуска мы поговорим далее в этой главе.

Результат аутентификации запроса API-сервером Kubernetes выражается в наборе дополнительных атрибутов, таких как *пользователь*, *группа*, *глагол API-запроса*, *HTTP-метод* и др. Затем эти атрибуты передаются модулям авторизации, которые на их основе определяют, имеет ли пользователь право выполнить запрашиваемое действие. Если какой-либо из модулей отклоняет запрос, пользователь получает код HTTP-ответа `403 (Forbidden)`.

Важно учитывать различия между кодами HTTP-ответов. Код 401 (Unauthorized) означает, что система не распознала пользователя, например, из-за неверных учетных данных или отсутствия соответствующей учетной записи. Код 403 (Forbidden) означает, что пользователь успешно аутентифицирован, но не имеет прав на выполнение запрошенного действия. Это различие особенно полезно учитывать при диагностике проблем с доступом к кластеру Kubernetes.

В Kubernetes предусмотрено несколько режимов авторизации, которые задаются с помощью параметра `--authorization-mode` при запуске API-сервера:

```
# /etc/kubernetes/manifests/kube-apiserver.yaml
...<фрагмент опущен для краткости>...
spec:
  containers:
  - command:
    - kube-apiserver
    - --advertise-address=192.168.59.154
    - --allow-privileged=true
    - --authorization-mode=Node,RBAC
...<фрагмент опущен для краткости>...
```

Kubernetes поддерживает следующие режимы авторизации.

- *RBAC* (Role-Based Access Control — управление доступом на основе ролей). Позволяет управлять доступом с помощью ролей и привилегий. RBAC — это один из отраслевых стандартов управления доступом, применяемый не только в Kubernetes. Роли определяют набор разрешений, который можно назначить пользователям или группам. Это обеспечивает гибкое управление доступом в соответствии с *принципом наименьших привилегий*. Например, можно создать роль, предоставляющую доступ только к определенным файлам или ресурсам, а затем связать эту роль с пользователями

или группами. В Kubernetes это делается с помощью объектов *RoleBinding* и *ClusterRoleBinding*. Один пользователь может иметь несколько ролей, а одна роль может быть назначена нескольким пользователям. Важно отметить, что механизм RBAC в Kubernetes построен на модели *явного разрешения*. Иными словами, по умолчанию запрещено все, и вы должны явно задать правила, которые *разрешают* те или иные действия.

- *ABAC* (Attribute-Based Access Control — управление доступом на основе атрибутов). Это часть общей парадигмы контроля доступа, применяемая не только в Kubernetes и основанная на политиках, учитывающих атрибуты пользователя, ресурса и среды. Данный подход обеспечивает очень детализированный контроль доступа. Например, он позволяет задать правило, разрешающее пользователю доступ к файлу только при одновременном выполнении трех условий: если у пользователя есть допуск к конфиденциальным данным (атрибут пользователя), если владельцем файла является Майк (атрибут ресурса) и если запрос поступает из внутренней сети (атрибут среды). Таким образом, политика представляет собой набор атрибутов, которые должны присутствовать одновременно, чтобы действие было разрешено. В Kubernetes она реализуется с помощью объекта *Policy*. Например, можно определить политику, разрешающую аутентифицированному пользователю по имени `Mike` читать любые поды в пространстве имен `default`. Чтобы предоставить те же права пользователю `Bob`, необходимо создать для него отдельный объект *Policy*.
- *Node*. Специальный режим авторизации, используемый для проверки запросов к API, поступающих от компонента `kubelet` внутри кластера.
- *Webhook*. В этом режиме API-сервер отправляет внешнему сервису объект *SubjectAccessReview* через HTTP-запрос типа `POST`. Сервис проверяет запрос и возвращает объект SubjectAccessReview с результатом проверки (доступ разрешен/доступ запрещен). В зависимости от ответа API-сервер либо выполняет запрос, либо возвращает код `403`. Такой подход часто используется при интеграции с корпоративными системами управления доступом.
- *AlwaysAllow*. Разрешает выполнение всех запросов без ограничений. Используется исключительно в тестовых средах по соображениям безопасности.
- *AlwaysDeny*. Блокирует выполнение всех запросов. Применяется только в тестовых целях, например для проверки поведения системы при полном запрете доступа.

В настоящее время режим RBAC считается отраслевым стандартом благодаря своей гибкости и простоте управления. Поэтому в следующих разделах мы сосредоточимся именно на нем.

Режим RBAC в Kubernetes

Использование механизма RBAC в Kubernetes основано на следующих типах API-ресурсов, входящих в группу `rbac.authorization.k8s.io`.

- *Role* и *ClusterRole* — определяют набор разрешений. Каждое правило (`rule`) внутри объекта Role описывает, какие операции допустимы в отношении тех или иных API-ресурсов. Различие между ними заключается в области действия: Role — это объект, ограниченный пространством имен, а ClusterRole является глобальным.
- *RoleBinding* и *ClusterRoleBinding* — связывают пользователей (их группы или учетные записи сервисов) с определенной ролью. RoleBinding действует в пределах одного пространства имен, а ClusterRoleBinding — на уровне всего кластера. При этом ClusterRoleBinding работает только с ClusterRole, а RoleBinding может работать как с Role, так и с ClusterRole.

Управление этими объектами осуществляется так же, как и другими ресурсами Kubernetes, — через утилиту `kubectl` и YAML-манифесты.

Теперь продемонстрируем принцип работы механизма RBAC на практике. В предыдущем разделе мы рассмотрели базовую настройку RBAC для учетной записи сервиса, которая использовалась для аутентификации при работе с `kubectl`. В следующем примере мы разберем немного иной сценарий: мы создадим под, который работает под управлением *выделенной* учетной записи сервиса и периодически запрашивает у API-сервера Kubernetes список подов. В целом использование выделенных учетных записей сервиса для подов — это хорошая практика, позволяющая придерживаться принципа наименьших привилегий. Например, если вашему поду требуется только получать список подов в кластере и не нужно создавать новые поды, то соответствующий объект ServiceAccount должен иметь роль, разрешающую выполнение только операции `list`. Для выполнения необходимой настройки сделайте следующее.

1. Чтобы создать выделенное пространство имен для объектов, подготовьте YAML-манифест:

```
# 02_rbac/rbac-demo-ns.yaml
---
apiVersion: v1
kind: Namespace
metadata:
  name: rbac-demo-ns
```

   Примените этот манифест с помощью команды:

```
$ kubectl apply -f 02_rbac/rbac-demo-ns.yaml
namespace/rbac-demo-ns created
```

2. Создайте тестовый под `nginx` в этом пространстве имен, используя файл `02_rbac/nginx-pod.yaml`:

```
$ kubectl apply -f 02_rbac/nginx-pod.yaml
pod/nginx-pod created
```

Обратите внимание на то, что под nginx сам по себе здесь ничего не делает; позднее нам понадобится под pod-logger-app для получения сведений о поде nginx в пространстве имен rbac-demo-ns.

3. Создайте объект ServiceAccount `pod-logger`. Для этого подготовьте манифест `pod-logger-serviceaccount.yaml`:

```
# 02_rbac/pod-logger-serviceaccount.yaml
apiVersion: v1
kind: ServiceAccount
metadata:
  name: pod-logger
  namespace: rbac-demo-ns
```

И примените его к кластеру с помощью команды:

```
$ kubectl apply -f 02_rbac/pod-logger-serviceaccount.yaml
serviceaccount/pod-logger created
```

4. Создайте роль `pod-reader`, разрешающую только операции `get`, `watch` и `list` над ресурсами `pods` в REST API Kubernetes. Фактически это соответствует работе с конечной точкой API `/api/v1/namespaces/rbac-demo-ns/pods`. Значение `apiGroups: [""]` указывает на использование основной (`core`) группы API. Структура манифеста `pod-reader-role.yaml` выглядит следующим образом:

```
# 02_rbac/pod-reader-role.yaml
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  namespace: rbac-demo-ns
  name: pod-reader
rules:
  - apiGroups: [""]
    resources: ["pods"]
    verbs: ["get", "watch", "list"]
```

5. Примените манифест к кластеру с помощью следующей команды:

```
$ kubectl apply -f 02_rbac/pod-reader-role.yaml
role.rbac.authorization.k8s.io/pod-reader created
```

6. Обычно на этом этапе создается объект RoleBinding, связывающий учетную запись сервиса с ролью. Однако, чтобы сделать пример более наглядным, мы сначала создадим под, работающий под учетной записью сервиса `pod-logger`. Этот под не сможет запрашивать список подов через API, поскольку у него отсутствуют соответствующие права (помните, что механизм RBAC по умолчанию запрещает все действия). Создайте YAML-манифест `pod-logger-app.yaml` для пода с именем `pod-logger-app`, который запускается без участия дополнительных контроллеров:

```
# 02_rbac/pod-logger-app.yaml
apiVersion: v1
kind: Pod
metadata:
  name: pod-logger-app
  namespace: rbac-demo-ns
spec:
  serviceAccountName: pod-logger
  containers:
    - name: logger
      image: quay.io/iamgini/k8sutils:debian12
      command:
      - /bin/sh
      - -c
      - |
        SERVICEACCOUNT=/var/run/secrets/kubernetes.io/serviceaccount
        TOKEN=$(cat ${SERVICEACCOUNT}/token)
        while true
        do
          echo "Querying Kubernetes API Server for Pods in default
          namespace..."
          curl --cacert $SERVICEACCOUNT/ca.crt \
             --header "Authorization: Bearer $TOKEN" \
             -X GET https://kubernetes.default.svc.cluster.local/api/v1/
             namespaces/rbac-demo-ns/pods
          sleep 10
        done
```

Здесь самыми важными являются два поля: `.spec.serviceAccountName`, в котором указана учетная запись сервиса, от имени которой должен запускаться под, и `command` — команда в определении контейнера, которую мы переопределили для периодического опрашивания API-сервера Kubernetes.

7. Примените манифест `02_rbac/pod-logger-app.yaml`, чтобы создать под:

```
$ kubectl apply -f 02_rbac/pod-logger-app.yaml
pod/pod-logger-app created

$ kubectl get po -n rbac-demo-ns
NAME             READY   STATUS    RESTARTS   AGE
nginx-pod        1/1     Running   0          15m
pod-logger-app   1/1     Running   0          9s
```

8. В результате назначения учетной записи сервиса `pod-logger` соответствующий объект Secret с JWT-токеном будет смонтирован в файловую систему контейнера по пути `/var/run/secrets/kubernetes.io/serviceaccount/token`. Убедиться в этом можно с помощью команды `kubectl exec`:

```
$ kubectl exec -it -n rbac-demo-ns pod-logger-app -- bash
root@pod-logger-app:/# ls -l /var/run/secrets/kubernetes.io/serviceaccount/
total 0
lrwxrwxrwx 1 root root 13 Jul 14 03:33 ca.crt      -> ..data/ca.crt
lrwxrwxrwx 1 root root 16 Jul 14 03:33 namespace   -> ..data/namespace
lrwxrwxrwx 1 root root 12 Jul 14 03:33 token       -> ..data/token
```

9. Переопределенные команды выполняются в бесконечном цикле в оболочке Linux (например, bash) с десятисекундными интервалами. В рамках каждой итерации мы обращаемся к конечной точке API Kubernetes (https://kubernetes/api/v1/namespaces/rbac-demo-ns/pods), чтобы получить список подов в пространстве имен `rbac-demo-ns`, с помощью HTTP-запроса типа `GET` и утилиты `curl`. Для прохождения аутентификации мы передаем содержимое файла `/var/run/secrets/kubernetes.io/serviceaccount/token` в виде *bearer*-токена в заголовке запроса `Authorization`. Кроме того, мы передаем путь к CA-сертификату для проверки подлинности удаленного сервера с помощью аргумента `cacert`. Этот сертификат автоматически добавляется в путь `/var/run/secrets/kubernetes.io/serviceaccount/ca.crt` средой выполнения Kubernetes. При просмотре логов вы увидите множество сообщений с кодом HTTP-ответа `403 (Forbidden)`. Это объясняется тем, что для данной учетной записи сервиса пока не создан объект RoleBinding, связывающий ее с ролью `pod-reader`.
10. Просмотрите логи пода `pod-logger-app` с помощью следующей команды:

```
$ kubectl logs -n rbac-demo-ns pod-logger-app -f
Querying Kubernetes API Server for Pods in rbac-demo-ns namespace...
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100   336  100   336    0     0   25611      0 --:--:-- --:--:-- --:--:-- 25846
{
  "kind": "Status",
  "apiVersion": "v1",
  "metadata": {},
  "status": "Failure",
  "message": "pods is forbidden: User \"system:serviceaccount:rbac-demo-ns:pod-
logger\" cannot list resource \"pods\" in API group \"\" in the namespace
\"rbac-demo-ns\"",
  "reason": "Forbidden",
  "details": {
    "kind": "pods"
  },
  "code": 403
}
```

11. Переключившись на новое окно консоли (или завершив просмотр логов нажатием Ctrl+F), создайте объект RoleBinding, *связывающий* учетную запись сервиса `pod-logger` с ролью `pod-reader`. Для этого подготовьте файл `read-pods-rolebinding.yaml` со следующим содержимым:

```
# 02_rbac/read-pods-rolebinding.yaml
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: read-pods
  namespace: rbac-demo-ns
subjects:
- kind: ServiceAccount
  name: pod-logger
  namespace: rbac-demo-ns
roleRef:
  kind: Role
  name: pod-reader
  apiGroup: rbac.authorization.k8s.io
```

В манифесте RoleBinding есть три ключевых компонента: `name` — идентификатор самого объекта RoleBinding; `subjects` — список субъектов (пользователей, групп или учетных записей сервисов), которым назначается роль; `roleRef` — ссылка на роль, которая связывается с субъектами.

12. Примените манифест RoleBinding:

```
$ kubectl apply -f 02_rbac/read-pods-rolebinding.yaml
rolebinding.rbac.authorization.k8s.io/read-pods created
```

13. Теперь снова просмотрите логи пода `pod-logger-app` — вы увидите, что под смог получить список подов в пространстве имен `rbac-demo-ns`. Иными словами, запрос был успешно авторизован:

```
$ kubectl logs -n rbac-demo-ns pod-logger-app -f
...<фрагмент опущен для краткости>...
Querying Kubernetes API Server for Pods in default namespace...
  % Total    % Received % Xferd  Average Speed   Time    Time     Time
Current
                                 Dload  Upload   Total   Spent    Left   Speed
{
  "kind": "PodList",
  "apiVersion": "v1",
  "metadata": {
    "resourceVersion": "4889"
  },
  "items": [
    {
      "metadata": {
        "name": "nginx-pod",
```

```
        "namespace": "rbac-demo-ns",
        "uid": "b62b2bdb-2677-4809-a134-9d6cfa07ecad",
        ...<фрагмент опущен для краткости>...
      }
    }
  ]
}
```

14. Наконец, вы можете удалить объект RoleBinding с помощью следующей команды:

```
$ kubectl delete rolebinding read-pods -n rbac-demo-ns
```

15. Если после этого вы снова просмотрите логи пода `pod-logger-app`, то увидите, что запросы опять отклоняются (код HTTP-ответа `403`).

Итак, вы успешно применили механизм RBAC в Kubernetes для ограничения доступа пода, работающего под управлением объекта ServiceAccount, сделав так, чтобы он мог только получать список подов в заданном пространстве имен. Чтобы очистить окружение и удалить все созданные в ходе демонстрации ресурсы, достаточно удалить пространство имен `rbac-demo-ns`.

Мы подробно изучили процессы аутентификации и авторизации в Kubernetes. В следующем разделе мы рассмотрим еще один механизм обеспечения безопасности — контроллеры допуска.

Контроллеры допуска: политики и проверки безопасности

Представьте себе контрольно-пропускной пункт на стратегическом объекте. Контроллеры допуска (admission controllers) в Kubernetes выполняют аналогичную роль в вашем кластере, перехватывая запросы к API-серверу Kubernetes до создания, изменения или удаления ресурсов. Эти контроллеры проверяют (и при необходимости изменяют) запросы в соответствии с заранее заданными правилами, предоставляя допуск в систему только разрешенным и корректно настроенным ресурсам. Важно отметить, что контроллеры допуска не могут блокировать запросы на чтение объектов (то есть выполнение операций `get`, `watch` и `list`).

Некоторые ключевые функции Kubernetes зависят от работы конкретных контроллеров допуска. Поэтому без правильно настроенных контроллеров допуска API-сервер Kubernetes будет неполноценным и не сможет реализовывать часть ожидаемого функционала.

Существуют два типа контроллеров допуска.

- *Валидирующие контроллеры* (validation controllers) — тщательно проверяют входящие запросы и отклоняют их при обнаружении чего-либо подозрительного или несоответствующего заданным политикам.
- *Мутирующие контроллеры* (mutation controllers) — способны изменять запрос до его сохранения. Например, они могут добавить отсутствующие аннотации, связанные с обеспечением безопасности, или скорректировать лимиты ресурсов.

Далее мы рассмотрим двухэтапный процесс работы контроллеров допуска.

Двухэтапный процесс работы контроллеров допуска

Процесс работы контроллеров допуска в Kubernetes включает два этапа, обеспечивая попадание в кластер только тех ресурсов, которые соответствуют установленным требованиям.

Следующая схема демонстрирует этот двухэтапный процесс обработки запросов в Kubernetes. На ней видно, что контроллеры допуска сначала модифицируют или обогащают входящие запросы, а затем проверяют их на соответствие обязательным требованиям и политикам безопасности.

Такой порядок действий обеспечивает согласованность, безопасность и соответствие заданным политикам перед изменением состояния кластера (рис. 18.7).

Далее приведено описание каждого из этапов.

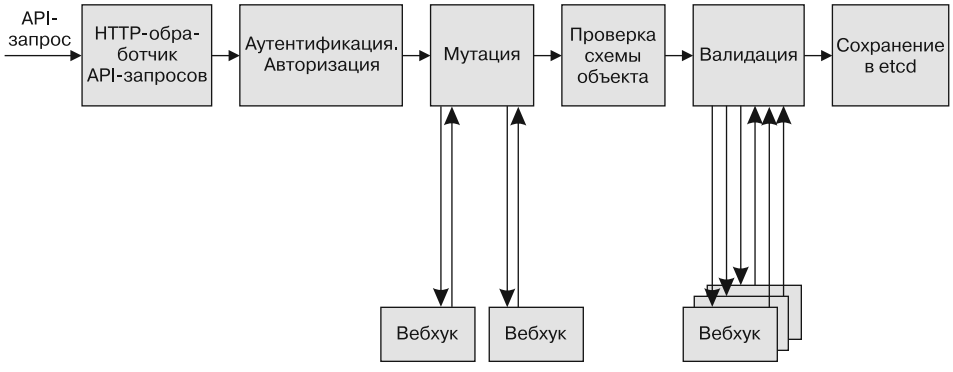

Рис. 18.7. Процесс обработки API-запроса контроллерами допуска (источник изображения: https://kubernetes.io/blog/2019/03/21/a-guide-to-kubernetes-admission-controllers/)

Мутация

На этом этапе мутирующие контроллеры изменяют входящие API-запросы, чтобы привести их в соответствие с политиками кластера перед дальнейшей обработкой. Такие контроллеры действуют как своеобразные «формовщики»: они не только обеспечивают согласованность запросов с установленными правилами, но и могут автоматически добавлять или корректировать параметры, например оставлять значения по умолчанию или задавать необходимые метки безопасности. Это позволяет гарантировать соблюдение политик и поддерживать единообразие конфигураций без ручного вмешательства.

Примеры действий, выполняемых на данном этапе:

- добавление недостающих меток безопасности к подам;
- корректировка запросов и лимитов ресурсов для подов в соответствии с заранее заданными правилами;
- внедрение контейнеров-прицепов для реализации дополнительных функций.

Валидация

На этапе валидации механизм контроллеров допуска завершает обработку, начатую на предыдущем шаге. На этом этапе контроллеры тщательно проверяют входящие запросы, которые могли быть изменены другими контроллерами. Их часто называют «стражами», поскольку они контролируют строгое соответствие запросов политикам и требованиям безопасности кластера. Данный этап играет ключевую роль в предотвращении ошибок конфигурации и несанкционированных изменений, обеспечивая целостность и безопасность кластера за счет отклонения запросов, не отвечающих установленным критериям.

Примеры действий, выполняемых на данном этапе:

- одобрение запроса, удовлетворяющего заданным критериям (например, квотам ресурсов или стандартам безопасности);
- отклонение запроса, нарушающего заданные политики, с предоставлением информативного сообщения об ошибке.

В следующем разделе мы обсудим способы включения и отключения контроллеров допуска в Kubernetes.

Включение и отключение контроллеров допуска

Чтобы узнать, какие контроллеры допуска включены в данный момент, необходимо изучить конфигурацию *API-сервера Kubernetes* путем просмотра конфигурационного файла, в котором он определен. Как правило, этот файл находится в системных каталогах конфигураций либо управляется соответствующим

менеджером. В этом файле следует найти параметр `--enable-admission-plugins`, определяющий список активных контроллеров допуска.

Например, в среде `minikube` вы можете подключиться к виртуальной машине с помощью команды `minikube ssh`. После входа найдите и изучите файл `kube-apiserver.yaml`, который обычно расположен по пути `/etc/kubernetes/manifests/`. Чтобы просмотреть его содержимое, используйте команду: `sudo cat /etc/kubernetes/manifests/kube-apiserver.yaml` и найдите параметр `--enable-admission-plugins`:

```
$ minikube ssh 'sudo grep -- "--enable-admission-plugins" /etc/kubernetes/
manifests/kube-apiserver.yaml'
- --enable-admission-plugins=NamespaceLifecycle,LimitRanger,ServiceAccount,
DefaultStorageClass,DefaultTolerationSeconds,NodeRestriction,
MutatingAdmissionWebhook,ValidatingAdmissionWebhook,ResourceQuota
```

Чтобы изменить список включенных контроллеров, откройте этот файл в текстовом редакторе (например, nano или vi), внесите необходимые правки и сохраните их. Компонент `kubelet` отслеживает файлы манифестов и автоматически перезапускает API-сервер (пересоздает под) при обнаружении изменений в них.

Мы также можем отключить контроллеры допуска, включенные по умолчанию. Это делается следующим образом:

```
kube-apiserver --disable-admission-plugins=PodNodeSelector,AlwaysDeny ...
```

Далее мы рассмотрим контроллеры допуска, предусмотренные в Kubernetes.

Распространенные контроллеры допуска

В Kubernetes контроллеры допуска встроены в компонент `kube-apiserver` и должны настраиваться исключительно администратором кластера. Особенно важными являются два контроллера: *MutatingAdmissionWebhook* и *ValidatingAdmissionWebhook*. Они отвечают за вызов мутирующих и валидирующих вебхуков соответственно, которые настраиваются через API. Ниже перечислены наиболее распространенные контроллеры допуска.

- *Основные действия:* `AlwaysAdmit` (устарел), `AlwaysDeny` (устарел), `AlwaysPullImages`.
- *Значения по умолчанию:* `DefaultStorageClass`, `DefaultTolerationSeconds`
- *Безопасность:* `DenyEscalatingExec`, `DenyServiceExternalIPs`, `PodSecurityPolicy`, `SecurityContextDeny`.
- *Управление ресурсами:* `LimitRanger`, `ResourceQuota`, `RuntimeClass`.
- *Жизненный цикл объектов:* `NamespaceAutoProvision`, `NamespaceExists`, `NamespaceLifecycle`, `PersistentVolumeClaimResize`, `StorageObjectInUseProtection`.

- *Управление узлами:* `NodeRestriction`, `TaintNodesByCondition`.
- *Вебхуки:* `MutatingAdmissionWebhook`, `ValidatingAdmissionWebhook`.
- *Другие:* `EventRateLimit`, `LimitPodHardAntiAffinityTopology`, `OwnerReferencesPermissionEnforcement`, `PodNodeSelector` (устарел), `Priority`, `ServiceAccount`.

Контроллеры допуска обеспечивают множество преимуществ при работе с Kubernetes-кластером. В следующем разделе мы рассмотрим некоторые из них.

Преимущества контроллеров допуска

Использование контроллеров допуска в кластерах Kubernetes имеет ряд важных преимуществ, среди которых можно выделить следующие.

- *Усиление безопасности*. Принудительно применяя политики безопасности, такие как стандарты безопасности подов, контроллеры допуска защищают кластер от неавторизованных или уязвимых развертываний.
- *Автоматическое применение политик*. Контроллеры допуска способны обеспечивать соблюдение заданных правил, касающихся использования ресурсов, получения образов и других аспектов работы кластера.
- *Согласованность и стандартизация*. Контроллеры допуска обеспечивают соответствие всех ресурсов кластера установленным лучшим практикам и конфигурационным стандартам.

Подводя итог, можно сказать, что контроллеры допуска играют ключевую роль в обеспечении безопасности Kubernetes за счет двух последовательных этапов — мутации и валидации. На этапе мутации контроллеры изменяют запросы в соответствии с политиками кластера, а на этапе валидации гарантируют обработку только тех запросов, которые удовлетворяют стандартам безопасности. Вместе эти два процесса повышают общий уровень защищенности кластера, обеспечивая соблюдение правил и предотвращая несанкционированные изменения.

В следующих разделах мы поговорим о защите рабочих нагрузок в Kubernetes с помощью контекста безопасности и сетевых политик.

Защита подов и контейнеров

Защита подов и контейнеров является ключевым условием обеспечения устойчивости среды Kubernetes, поскольку именно они напрямую взаимодействуют с рабочими нагрузками и конфиденциальными данными. В следующих разделах вы узнаете, как контексты безопасности и сетевые политики позволяют реализовать строгий контроль доступа и изоляцию для повышения уровня безопасности подов и контейнеров в кластере.

Защита подов и контейнеров в Kubernetes с помощью контекста безопасности

В параметре `securityContext` (контекст безопасности) задаются настройки, определяющие то, как должен работать под или контейнер внутри кластера. Эти настройки позволяют применять лучшие практики обеспечения безопасности и уменьшать поверхность атаки за счет ограничения привилегий и контроля доступа.

Контекст безопасности повышает защищенность Kubernetes-кластера, определяя условия выполнения пода или контейнера. Задавая соответствующие параметры, вы гарантируете соблюдение принципа наименьших привилегий, снижая вероятность злонамеренной активности и допущения случайных ошибок в конфигурации.

Типичным сценарием использования контекста безопасности является запуск контейнеров от имени непривилегированного (non-root) пользователя. Это предотвращает получение контейнером избыточных разрешений и ограничивает возможный ущерб в случае его компрометации. Для дополнительной защиты кластера можно использовать файловые системы, доступные только для чтения, и осуществлять детализированное управление привилегиями.

Ключевые компоненты контекста безопасности

Ниже приведен разбор основных компонентов параметра `securityContext` с поясняющими примерами.

Пользователь и группа. Данный компонент задает идентификаторы пользователя и группы, от имени которых будут выполняться процессы внутри контейнера. Он позволяет реализовать принцип наименьших привилегий, предоставив контейнерам только те разрешения, которые необходимы им для работы. Пример определения пода с данным компонентом `securityContext` выглядит следующим образом[1]:

```
apiVersion: v1
kind: Pod
metadata:
  name: my-app
spec:
  containers:
  - name: app-container
    securityContext:
```

[1] securityContext — довольно важный параметр, который влияет в том числе на разработку. К примеру, Visual Studio дает возможность генерировать шаблоны образов контейнеров, которые позволяют минимально запустить рабочую версию кода, однако если параметр securityContext корректно настроен в кластере, разработчик не сможет запустить свое приложение, пока не адаптирует процессы и параметры запуска под непривилегированного пользователя.

```
    runAsUser: 1000   # Запуск процессов в контейнере
                      # от имени пользователя с ID 1000
    runAsGroup: 1000  # Запуск процессов в контейнере от имени группы с ID 1000
```

Привилегии Linux. Привилегии (`capabilities`) — это специальные разрешения, которые можно предоставить контейнеру в дополнение к правам обычного пользователя. Параметр `securityContext` позволяет указать, какие именно привилегии должен иметь контейнер, и тем самым реализовать определенные функции без предоставления ему полного доступа, как показано в следующем примере:

```
apiVersion: v1
kind: Pod
metadata:
  name: privileged-container
spec:
  containers:
  - name: app-container
    securityContext:
      capabilities:
        add:
        - CAP_NET_ADMIN   # Предоставление привилегий для управления сетью
```

Подробнее о привилегиях Linux можно узнать в официальной документации: https://linux-audit.com/kernel/capabilities/linux-capabilities-hardening-linux-binaries-by-removing-setuid/.

Привилегированный режим. По умолчанию контейнеры запускаются в непривилегированном режиме. Параметр `securityContext` позволяет запустить контейнер в привилегированном режиме, предоставив ему права суперпользователя. Однако из-за повышенных рисков безопасности использовать этот режим следует с осторожностью. Ниже приведен пример YAML-манифеста пода с активизированным привилегированным режимом:

```
apiVersion: v1
kind: Pod
metadata:
  name: privileged-container
spec:
  containers:
  - name: app-container
    securityContext:
      privileged: true   # Запуск контейнера в привилегированном режиме
                         # (использовать с осторожностью)
```

Файловая система, доступная только для чтения. С помощью параметра `securityContext` можно настроить контейнер так, чтобы его корневая файловая система была доступна только для чтения. Это повышает уровень безопасности, предотвращая случайные или преднамеренные изменения базовой системы:

```
apiVersion: v1
kind: Pod
metadata:
  name: read-only-container
```

```
spec:
  containers:
  - name: app-container
    securityContext:
      readOnlyRootFilesystem: true  # Монтирование корневой файловой системы
                                    # в режиме "только для чтения"
```

Существуют и другие настройки `SecurityContext`, такие как *Security Enhanced Linux (SELinux)*, *AppArmor* (https://kubernetes.io/docs/tutorials/security/apparmor/), *Seccomp* и др. Подробнее о них можно узнать в официальной документации по адресу https://kubernetes.io/docs/tasks/configure-pod-container/security-context/.

В следующем разделе мы поговорим о выборе оптимального места использования параметра `securityContext` в конфигурации.

Задание параметра SecurityContext на уровне пода и контейнера

В Kubernetes параметр `securityContext` можно задавать как на уровне пода, так и на уровне контейнера, что обеспечивает гибкость в настройке параметров безопасности для приложений.

Задание параметра SecurityContext на уровне пода. При задании параметра `securityContext` на уровне пода его настройки наследуются всеми контейнерами внутри этого пода, что обеспечивает согласованность их параметров безопасности:

```
apiVersion: v1
kind: Pod
metadata:
  name: my-pod
spec:
  securityContext:
    runAsUser: 1000    # Все контейнеры в поде запускаются
                       # от имени пользователя с ID 1000
    runAsGroup: 1000   # Все контейнеры в поде запускаются
                       # от имени группы с ID 1000
  containers:
  - name: app-container
    image: my-app-image
  - name: sidecar-container
    image: my-sidecar-image
```

Задание параметра SecurityContext на уровне контейнера. При задании параметра `securityContext` на уровне контейнера соответствующие настройки затрагивают *только* его, что позволяет задавать разные параметры безопасности для разных контейнеров одного пода:

```
apiVersion: v1
kind: Pod
metadata:
  name: my-pod
```

```
spec:
  containers:
  - name: app-container
    image: my-app-image
    securityContext:
      runAsUser: 1000               # Этот контейнер запускается от имени
                                    # пользователя с ID 1000
      capabilities:
        add: ["CAP_NET_ADMIN"]      # Контейнеру предоставляются особые привилегии
  - name: sidecar-container
    image: my-sidecar-image
    securityContext:
      runAsUser: 2000               # Этот контейнер запускается от имени
                                    # пользователя с ID 2000
      readOnlyRootFilesystem: true  # Корневая файловая система контейнера
                                    # доступна только для чтения
```

В следующем разделе мы применим параметр `SecurityContext` к поду.

Применение параметра SecurityContext к поду

В этом примере создается под с контейнером, который запускается от имени непривилегированного пользователя и группы с корневой файловой системой, доступной только для чтения:

```
# pod-with-security-context.yaml
apiVersion: v1
kind: Pod
metadata:
  name: security-context-demo
spec:
  containers:
  - name: app-container
    image: nginx:latest
    securityContext:
      runAsUser: 1000               # Запуск процессов в контейнере
                                    # от имени пользователя с ID 1000.
      runAsGroup: 1000              # Запуск процессов в контейнере
                                    # от имени группы с ID 1000.
      readOnlyRootFilesystem: true # Монтирование корневой файловой системы
                                    # в режиме только для чтения
    volumeMounts:
    - name: html-volume
      mountPath: /usr/share/nginx/html
  volumes:
  - name: html-volume
    emptyDir: {}                    # Том, предоставляющий пространство для записи
```

В этом YAML-манифесте стоит обратить внимание на следующие параметры.

- `runAsUser` и `runAsGroup` — гарантируют запуск контейнера от имени конкретных пользователя и группы, не имеющих прав суперпользователя, в соответствии с принципом наименьших привилегий.

- `readOnlyRootFilesystem` — монтирует корневую файловую систему контейнера в режиме «только для чтения», что предотвращает случайные или намеренные изменения базовой системы.

Создайте под с помощью приведенного YAML-манифеста следующим образом:

```
$ kubectl apply -f pod-with-security-context.yaml
pod/security-context-demo created
```

После создания пода выполните несколько команд внутри контейнера, чтобы проверить действие настроенного параметра `securityContext`:

```
$ kubectl exec -it security-context-demo -- /bin/sh
~ $ id
uid=1000 gid=1000 groups=1000
~ $ touch /testfile
touch: /testfile: Read-only file system
```

Вы увидите сообщение об ошибке `Read-only file system` — это вполне ожидаемо.

Чтобы узнать больше о параметре `securityContext` в Kubernetes, обратитесь к официальной документации: https://kubernetes.io/docs/tasks/configure-pod-container/security-context/.

В следующем разделе мы обсудим управление сетевыми потоками в Kubernetes с использованием объекта *NetworkPolicy*, позволяющего создавать своеобразные сетевые брандмауэры, предотвращающие нежелательные взаимодействия между подами.

Защита подов с помощью объекта NetworkPolicy

Объект *NetworkPolicy* (сетевая политика) — это последний ресурс, который мы изучим в этой главе для получения целостного представления о механизмах обеспечения безопасности в Kubernetes. Данный объект позволяет реализовать сетевой брандмауэр прямо внутри кластера.

Причины для использования объекта NetworkPolicy

При управлении реальными рабочими нагрузками Kubernetes вам приходится развертывать в рабочей среде все больше приложений, которые должны обмениваться данными друг с другом.

Организация сетевого взаимодействия между компонентами — это одна из ключевых задач микросервисной архитектуры. Большая часть этого взаимодействия происходит по сети, которую необходимо защищать с помощью брандмауэров (межсетевых экранов).

В Kubernetes роль сетевого брандмауэра выполняет объект NetworkPolicy. Представьте, что вам нужно разрешить доступ к поду `nginx` на порте `80` только с определенного IP-адреса и заблокировать весь остальной трафик. Для этого можно создать объект NetworkPolicy и подключить его к нужному поду.

Объект NetworkPolicy позволяет:

- задавать правила управления входящим (ingress) и исходящим (egress) сетевым трафиком (правила ingress/egress) на основе блоков *CIDR* (Classless Inter-Domain Routing — бесклассовая междоменная маршрутизация);
- задавать правила ingress/egress на основе меток и селекторов подов (так же как было показано в контексте обсуждения взаимодействия сервисов и подов);
- задавать правила ingress/egress на основе пространств имен (этот механизм мы рассмотрим в следующей главе).

Для работы NetworkPolicy в Kubernetes-кластере должен быть установлен CNI-плагин, поддерживающий эту функциональность. CNI-плагины обычно не устанавливаются в Kubernetes по умолчанию. Если вы используете `minikube` в учебных целях, хорошая новость состоит в том, что данный инструмент интегрируется с Calico — CNI-плагином, который поддерживает функциональность NetworkPolicy. Чтобы задействовать ее, вам достаточно пересоздать кластер `minikube` следующим образом:

```
$ minikube start --network-plugin=cni --cni=calico --container-
runtime=containerd
```

Если вы работаете с Kubernetes в облачной среде, рекомендуем ознакомиться с документацией конкретного облачного провайдера, чтобы узнать о доступных CNI-плагинах и наличии поддержки NetworkPolicy.

Изначальная неизолированность подов

В Kubernetes поды по умолчанию не изолированы и могут взаимодействовать друг с другом без каких-либо ограничений.

Без использования сетевой политики поды остаются полностью доступными для всего остального трафика. Однако после привязки к поду объекта NetworkPolicy к нему начинают применяться правила, описанные в этом объекте.

Чтобы два пода, для которых заданы сетевые политики, могли обмениваться данными, их взаимодействие должно быть разрешено в обоих направлениях. Это означает, что под А должен иметь правило Egress, разрешающее отправку трафика к поду Б, а под Б должен иметь правило Ingress, разрешающее прием трафика от пода А. При невыполнении одного из этих условий трафик будет заблокирован. Следующая схема иллюстрирует этот принцип (рис. 18.8).

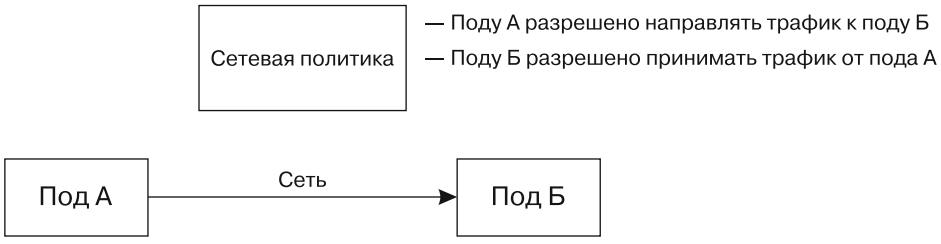

Рис. 18.8. Один из подов не работает, но сервис продолжает направлять к нему трафик

При поиске причин неисправностей сетевую политику (объект NetworkPolicy) необходимо проверять в первую очередь, поскольку она часто приводит к проблемам с соединением. Теперь давайте настроим такую политику для двух подов, используя метки и селекторы.

Настройка NetworkPolicy с использованием меток и селекторов

Для начала создадим два пода `nginx`. Чтобы наглядно продемонстрировать изоляцию, в этом примере мы будем использовать два отдельных пространства имен (см. главу 6).

Реализация полной изоляции сетевого взаимодействия внутри одного пространства имен может быть проблематичной и иметь нежелательные побочные эффекты. Перед применением подобных ограничений тщательно оцените свои потребности и возможные последствия.

Теперь создадим пространства имен и два пода с разными метками, чтобы упростить их выбор в качестве целей при настройке сетевой политики.

YAML-манифест для пространства имен `web1` с подом `nginx1` выглядит так:

```
# web1-app.yaml
---
apiVersion: v1
kind: Namespace
metadata:
  name: web1
  labels:
    project: web1

---
apiVersion: v1
kind: Pod
metadata:
  name: nginx1
  namespace: web1
```

```
  labels:
    app: nginx1
spec:
  containers:
    - name: nginx1
      image: nginx
```

А YAML-манифест для пространства имен `web2` с подом `nginx2` — так:

```
# web2-app.yaml
---
apiVersion: v1
kind: Namespace
metadata:
  name: web2
  labels:
    project: web2

---
apiVersion: v1
kind: Pod
metadata:
  name: nginx2
  namespace: web2
  labels:
    app: nginx2
spec:
  containers:
    - name: nginx2
      image: nginx
```

Как видите, в приведенных выше фрагментах кода мы используем пространства имен `web1` и `web2` вместо стандартного `default`.

Теперь создадим соответствующие ресурсы и проверим поды:

```
$ kubectl apply -f web-app1.yaml
namespace/web1 created
pod/nginx1 created

$ kubectl apply -f web-app2.yaml
namespace/web2 created
pod/nginx2 created

$ kubectl get po -o wide -n web1
NAME     READY   STATUS    RESTARTS   AGE   IP             NODE       NOMINATED
NODE   READINESS GATES
nginx1   1/1     Running   0          3m    10.244.120.71  minikube   <none>
<none>

$ kubectl get po -o wide -n web2
NAME     READY   STATUS    RESTARTS   AGE     IP             NODE
NOMINATED NODE   READINESS GATES
nginx2   1/1     Running   0          2m53s   10.244.120.72  minikube   <none>
<none>
```

После создания подов с уникальными метками в разных пространствах имен мы можем получить их IP-адреса с помощью флага `-o wide`. Далее следует проверить доступность, отправив `curl`-запрос с пода `nginx1` на под `nginx2`. Этот тест должен пройти успешно, так как сетевые политики еще не настроены, а маршрутизация трафика разрешена по умолчанию. Пример команды приведен ниже (IP-адрес `10.244.120.72` принадлежит поду `nginx2` в пространстве имен `web2`):

```
$ kubectl -n web1 exec nginx1 -- curl 10.244.120.72
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left   Speed
100   615  100   615    0     0   698k      0 --:--:-- --:--:-- --:--:--  600k

<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
...<фрагмент опущен для краткости>...
```

Как видите, мы получили домашнюю страницу сервера Nginx от пода `nginx2`.

Теперь явно заблокируем весь входящий трафик, поступающий в пространство имен `web2`. Для этого создадим сетевую политику по умолчанию:

```
# default-deny-ingress.yaml
---
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: default-deny-ingress
spec:
  podSelector: {}
  policyTypes:
    - Ingress
  ingress: []
```

В этом YAML-манифесте важно обратить внимание на следующие параметры.

- `podSelector: {}` — выбирает все поды в пространстве имен, независимо от их меток. Это означает, что данная сетевая политика *будет применяться ко всем подам в пространстве имен*.
- `policyTypes: - Ingress` — задает тип политики; в нашем случае Ingress. Это означает, что данная политика будет регулировать входящий трафик, направляемый на выбранные поды.
- `ingress: []` — список правил Ingress. В данном случае он пуст; это означает, что весь входящий трафик, направляемый на выбранные поды, будет заблокирован по умолчанию.

Применим эту политику к пространству имен `web2`, чтобы заблокировать весь входящий трафик:

```
$ kubectl apply -f default-deny-ingress.yaml -n web2
networkpolicy.networking.k8s.io/default-deny-ingress created
```

Теперь еще раз попробуем обратиться к поду `nginx2` из пода `nginx1`:

```
$ kubectl -n web1 exec nginx1 -- curl 10.244.120.72
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left   Speed
  0     0    0     0    0     0      0       0 --:--:--  0:02:15 --:--:--     0
curl: (28) Failed to connect to 10.244.120.72 port 80 after 135435 ms: Couldn't
connect to server
command terminated with exit code 28
```

Как видите, весь трафик, направляемый к подам в пространстве имен `web2`, блокируется сетевой политикой `default-deny-ingress`.

Теперь настроим сетевую политику для пода `nginx2` в пространстве имен `web2`, чтобы явно разрешить прием трафика от пода `nginx1` в пространстве имен `web1`. Для этого создадим YAML-манифест со следующим содержимым:

```
# allow-from-web1-netpol.yaml
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: allow-from-web1-netpol
  namespace: web2
spec:
  podSelector:
    matchLabels:
      app: nginx2
  policyTypes:
    - Ingress
  ingress:
    - from:
        - namespaceSelector:
            matchLabels:
              project: web1
        - podSelector:
            matchLabels:
              app: nginx1
      ports:
        - protocol: TCP
          port: 80
```

Обратите внимание на то, что в поле `namespaceSelector.matchLabels` используется метка `project: web1`, которую мы заранее назначили пространству имен `web1` именно для этой цели. Применим YAML-манифест с помощью команды:

```
$ kubectl apply -f allow-from-web1-netpol.yaml
networkpolicy.networking.k8s.io/nginx2-networkpolicy created
```

Теперь выполним тот же `curl`-запрос, что и раньше:

```
$ kubectl -n web1 exec nginx1 -- curl 10.244.120.72
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left   Speed
100   615  100   615    0     0   1280k      0 --:--:-- --:--:-- --:--:--   600k
<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
...<фрагмент опущен для краткости>...
```

Как видите, все работает так же, как и до применения сетевых политик. Это объясняется тем, что:

- поду `nginx2` теперь явно разрешено принимать входящий трафик, поступающий на порт `80` от пода `nginx1` из пространства имен `web1`, при этом весь остальной трафик по-прежнему заблокирован;
- для пода `nginx1` сетевая политика не задана, а значит, отправка всего исходящего трафика разрешена по умолчанию.

Запомните: если для пода не определена сетевая политика, весь его входящий и исходящий трафик разрешен по умолчанию.

Мы настоятельно рекомендуем всегда использовать объект NetworkPolicy вместе с подами. Также учтите, что объекты NetworkPolicy можно применять для создания брандмауэров на основе блоков CIDR, что бывает особенно полезно при наличии доступа к подам кластера извне. Однако для настройки брандмауэров между подами предпочтительнее использовать метки и селекторы, как и при настройке сервисов.

В следующем разделе мы поговорим о защите каналов связи между компонентами Kubernetes с помощью TLS-сертификатов. Вы узнаете, как TLS-сертификаты обеспечивают конфиденциальность передаваемых данных и безопасную коммуникацию между различными частями экосистемы Kubernetes.

Защита взаимодействий компонентов Kubernetes с помощью TLS-сертификатов

Безопасное взаимодействие компонентов Kubernetes имеет критически важное значение. Протоколы *TLS* (Transport Layer Security — безопасность транспортного уровня) и *SSL* (Secure Sockets Layer — уровень защищенных сокетов) играют ключевую роль в шифровании передаваемых данных и установлении доверия между сервисами.

При использовании протокола *mTLS* (TLS with mutual authentication — TLS с взаимной аутентификацией) клиент и сервер подтверждают подлинность друг друга с помощью цифровых сертификатов, выданных доверенным центром сертификации (CA). Это создает дополнительный уровень защиты, предотвращая несанкционированный доступ и обеспечивая целостность данных.

Ниже перечислены примеры использования TLS-сертификатов в Kubernetes.

- *Взаимодействие API-сервера с etcd*. API-сервер, центральный компонент управляющего слоя, взаимодействует с `etcd`, распределенным хранилищем данных в формате «ключ — значение», для управления состоянием кластера. Использование протокола mTLS при организации взаимодействия между API-сервером и хранилищем `etcd` позволяет защитить критически важные данные кластера от перехвата и подмены.
- *Взаимодействие Ingress-контроллера с сервисами*. Ingress-контроллер — это единая точка входа для внешнего трафика, которая перенаправляет запросы к бэкенд-сервисам. Применение протокола mTLS при организации взаимодействий между Ingress-контроллером и сервисами гарантирует перенаправление трафика только к авторизованным сервисам, снижая риск компрометации.
- *Внутреннее взаимодействие сервисов*. Сервисы внутри кластера также могут использовать mTLS для безопасного обмена данными. Это особенно важно для сервисов, работающих с конфиденциальной информацией или требующих строгой аутентификации.
- *Сервисные сетки (например, Istio)*. Подобные системы предоставляют расширенные возможности для управления трафиком и обеспечения безопасности, включая автоматическое применение mTLS при организации взаимодействий между микросервисами. Это значительно упрощает защиту межсервисного взаимодействия, избавляя разработчиков от необходимости интегрировать TLS в код приложений.
- *Балансировщики нагрузки*. Приложения, развернутые за балансировщиком нагрузки, могут использовать протокол TLS для защиты взаимодействия между балансировщиком и backend-сервисами. В такой конфигурации данные остаются зашифрованными на всем пути следования.
- *Набор протоколов IPSec*. Еще один механизм обеспечения безопасности предусматривает активизацию IPSec внутри кластера Kubernetes для шифрования сетевого трафика между узлами. Это может быть полезно для защиты данных в облачных средах или при их передаче между различными дата-центрами.

Используя TLS-сертификаты и протокол mTLS, администраторы Kubernetes могут существенно повысить безопасность кластера. Такой подход обеспечивает

шифрование каналов связи, подтверждение подлинности взаимодействующих компонентов и снижает риски, связанные с несанкционированным доступом к данным или их подменой.

В следующем подразделе вы узнаете, как можно повысить безопасность контейнеров с помощью таких специальных сред выполнения, как *gVisor* и *Kata Containers*.

Защита контейнеров с помощью gVisor и Kata Containers

Традиционные контейнеры используют ядро операционной системы хоста вместе с другими приложениями. Это создает риски для безопасности, так как уязвимость в одном контейнере может привести к компрометации всей базовой системы. *gVisor* и *Kata Containers* представляют собой альтернативные реализации среды выполнения контейнеров, которые ставят безопасность на первое место. Мы обсудим их в следующих разделах.

gVisor

gVisor (Guest Virtual Machine Supervisor) — это легковесная виртуальная машина, реализованная в пользовательском пространстве. Она выполняет роль песочницы для каждого контейнера, изолируя его от хостового ядра и других контейнеров.

На рис. 18.9 показана общая схема архитектуры gVisor.

Виртуализируя функции ядра для каждого контейнера, gVisor предотвращает влияние уязвимостей контейнера на хостовую систему. Он создает надежную границу, изолируя контейнер, даже если тот оказывается скомпрометированным. gVisor особенно хорошо подходит для сред, требующих максимальной изоляции, правда, его использование ведет к увеличению потребления ресурсов.

Kata Containers

Среда Kata Containers использует легковесные виртуальные машины (ВМ), схожие с традиционными ВМ, но оптимизированные для запуска контейнерных рабочих нагрузок. Kata Containers предоставляет изолированную среду выполнения, помещая каждый контейнер внутрь отдельной легковесной виртуальной машины, что обеспечивает более высокий уровень безопасности по сравнению со стандартными контейнерами, при этом не оказывая заметного влияния на производительность.

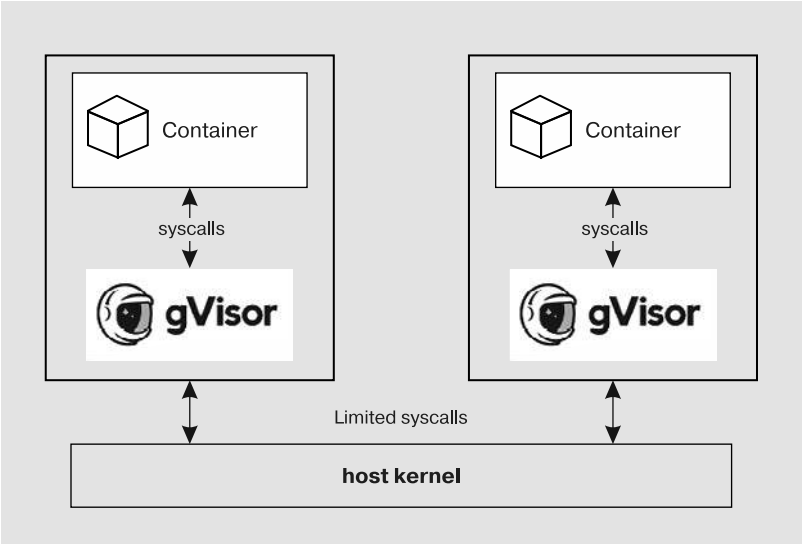

Рис. 18.9. Архитектура gVisor

На рис. 18.10 показано, чем архитектура Kata Containers отличается от традиционных контейнеров.

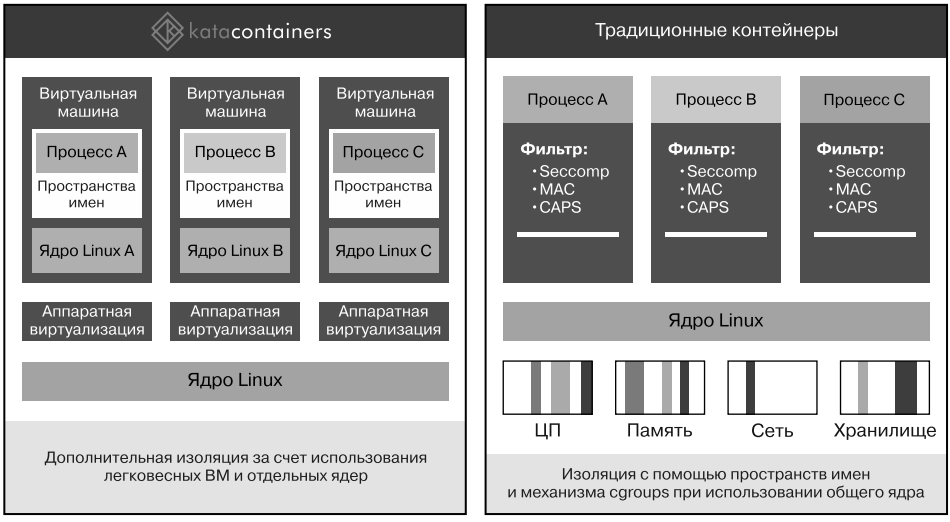

Рис. 18.10. Сравнение Kata Containers с традиционными контейнерами (источник изображения: https://katacontainers.io/learn/)

Kata Containers рекомендуется использовать в ситуациях, когда необходимо сочетать высокий уровень безопасности с оптимальной производительностью, особенно при работе с ресурсоемкими приложениями.

Использование ресурса RuntimeClass в профилях безопасности

В Kubernetes можно указать среду выполнения контейнеров с помощью поля `runtimeClassName` в спецификации пода. Вот пример такой конфигурации:

```
apiVersion: v1
kind: Pod
metadata:
  name: secure-pod
spec:
  runtimeClassName: kata-containers   # Указывает Kata Containers
                                      # в качестве среды выполнения
  containers:
  - name: my-app
    image: my-secure-image
```

Эта настройка указывает Kubernetes использовать среду выполнения Kata Containers для обеспечения лучшей изоляции и повышенной безопасности.

В этой главе мы рассмотрели множество важных аспектов обеспечения безопасности в Kubernetes. В завершение поговорим о получении доступа к приватным реестрам и образам контейнеров.

Управление секретами и учетными данными реестров

В Kubernetes для скачивания образов контейнеров из приватных репозиториев требуются соответствующие учетные данные. Без них поды не могут получить доступ к образам, размещенным в этих реестрах. Безопасное управление такими учетными данными позволяет гарантировать то, что только авторизованные поды смогут запрашивать и использовать конкретные образы контейнеров.

Команда `kubectl create secret docker-registry` упрощает управление учетными данными реестра контейнеров в Kubernetes. Она обеспечивает безопасность за счет шифрования секретов в состоянии покоя, что делает их доступными только для авторизованных узлов. Такой подход устраняет сложность, присущую ручным методам управления, минимизирует количество ошибок и повышает операционную эффективность. Кроме того, он легко интегрируется со спецификациями подов, упрощая настройку параметра `imagePullSecrets` для их аутентификации в приватных реестрах контейнеров.

Создание секрета для реестра Docker с помощью kubectl

Ниже показан способ создания секрета для реестра Docker и его использования в конфигурации пода:

```
$ kubectl create secret docker-registry my-registry-secret \
  --docker-server=your-registry.com \
  --docker-username=your_username \
  --docker-password=your_password \
  --docker-email=your-email@example.com
```

Замените `your-registry.com`, `your_username`, `your_password` и `your-email@example.com` своими учетными данными.

Теперь обновите YAML-манифест пода, чтобы использовать созданный секрет для скачивания образов из приватного реестра:

```
apiVersion: v1
kind: Pod
metadata:
  name: my-pod
spec:
  containers:
  - name: my-container
    image: your-registry.com/your-image:tag
  imagePullSecrets:
  - name: my-registry-secret
```

Убедитесь в том, что значение `my-registry-secret` совпадает с именем, которое вы использовали при создании секрета для реестра Docker[1].

При создании пода образ будет скачан из приватного реестра, а параметры `imagePullSecrets` — использованы в качестве учетных данных для аутентификации.

На этом мы завершаем эту большую главу, посвященную безопасности в Kubernetes.

Резюме

В этой главе мы рассмотрели механизмы *аутентификации* и *авторизации* в Kubernetes. Мы начали с обзора доступных методов аутентификации и обсудили способ использования токенов ServiceAccount для внешней аутентификации. Затем мы сосредоточились на механизме RBAC. Вы узнали, как использовать

[1] Стоит также убедиться, что под и секрет созданы в одном пространстве имен.

объекты Role, ClusterRole, RoleBinding и ClusterRoleBinding для управления правами доступа к кластеру. Мы также разобрали практический пример использования RBAC для учетных записей сервисов, создав под, который может получать список подов в кластере через API Kubernetes, соблюдая принцип наименьших привилегий.

Далее мы изучили контроллеры допуска и варианты их применения для усиления защиты кластера. Мы также познакомились с параметром `SecurityContext` и его компонентами. Затем мы рассмотрели объект NetworkPolicy, позволяющий контролировать сетевой трафик между подами и играющий роль сетевого брандмауэра, встроенного в кластер. В рамках обсуждения темы безопасности контейнеров мы изучили такие альтернативные среды выполнения, как Kata Containers и gVisor. Наконец, мы обсудили способ получения доступа к приватным реестрам контейнеров с помощью соответствующих учетных данных. В следующей главе мы подробно рассмотрим продвинутые методы планирования подов.

Дополнительные источники информации

- Управление доступом к API Kubernetes: https://kubernetes.io/docs/concepts/security/controlling-access.
- Управление учетными записями сервисов: https://kubernetes.io/docs/reference/access-authn-authz/service-accounts-admin/.
- Настройка учетных записей сервисов для подов: https://kubernetes.io/docs/tasks/configure-pod-container/configure-service-account/.
- Сертификаты и запросы на подпись сертификатов: https://kubernetes.io/docs/reference/access-authn-authz/certificate-signing-requests/.
- Авторизация: https://kubernetes.io/docs/reference/access-authn-authz/authorization/.
- Протокол OpenID Connect: https://openid.net/developers/how-connect-works/.
- Контроллеры допуска: https://kubernetes.io/docs/reference/access-authn-authz/admission-controllers.
- Контекст безопасности: https://kubernetes.io/docs/tasks/configure-pod-container/security-context/.

19 Продвинутые методы планирования подов

В главе 2 мы рассмотрели принципы работы планировщика Kubernetes (`kube-scheduler`), который является компонентом управляющего слоя и играет ключевую роль в кластере. Если вкратце, то он отвечает за назначение контейнерных рабочих нагрузок (подов Kubernetes) работоспособным узлам, соответствующим заданным требованиям.

В этой главе вы научитесь управлять критериями распределения подов между узлами кластера. Особое внимание мы уделим *правилам предпочтения узлов*, *ограничениям* и *допускам*. Мы также поговорим о *политиках планирования*, позволяющих компоненту `kube-scheduler` гибко приоритизировать распределяемые поды. Все эти механизмы играют важную роль при эксплуатации производственных кластеров в облачных средах.

Технические требования

Для работы с этой главой вам потребуется:

- *многоузловой* кластер Kubernetes. Подобный кластер значительно упрощает понимание таких концепций, как правила предпочтения узлов, ограничения и допуски;
- интерфейс командной строки `kubectl`, установленный на локальной машине и настроенный для управления кластером Kubernetes.

Инструкции по созданию кластера Kubernetes и настройке клиента `kubectl` приведены в главах 2 и 3. В главах 15–17 рассматривался процесс развертывания полнофункциональных кластеров Kubernetes в различных облачных средах.

Примеры кода для этой главы можно загрузить из официального репозитория GitHub по адресу https://github.com/PacktPublishing/The-Kubernetes-Bible-Second-Edition/tree/main/Chapter19.

Обзор компонента kube-scheduler

Планировщик `kube-scheduler` — это ключевой компонент управляющего слоя кластеров Kubernetes. Его основная задача состоит в том, чтобы распределять контейнерные рабочие нагрузки (поды) и *назначать* их корректно функционирующим вычислительным узлам, удовлетворяющим определенным требованиям. Напомним, что под — это группа, состоящая из одного или нескольких контейнеров, с общей сетью и хранилищем, а также наименьшая *единица развертывания* в системе Kubernetes. Как правило, для управления подами используются такие контроллеры, как объекты Deployment и StatefulSet, однако именно `kube-scheduler` в итоге назначает созданные поды конкретным узлам.

В управляемых облачных кластерах, например AKS (Azure Kubernetes Service) или EKS (Amazon Elastic Kubernetes Service), доступ к управляющему слою и узлам-контроллерам, как правило, отсутствует, поскольку ими управляет облачный провайдер. Это означает, что вы не можете напрямую взаимодействовать с компонентом kube-scheduler или изменять его конфигурацию (например, политики планирования). Тем не менее вы можете полностью контролировать параметры подов, влияющих на процесс их распределения.

Компонент `kube-scheduler` регулярно обращается к *API-серверу Kubernetes* (`kube-apiserver`), чтобы получить список подов, которые еще не были назначены конкретному узлу. После создания под помечается как *незапланированный* (unscheduled). Это означает, что для его выполнения еще не выбран вычислительный узел. Такой незапланированный под фиксируется в хранилище `etcd` без указания узла, на котором он должен выполняться. В результате ни один активный агент `kubelet` не получает информацию о необходимости запуска этого пода, поэтому ни один контейнер, определенный в спецификации пода, не будет выполнен.

Объект Pod, хранящийся в `etcd`, имеет свойство `nodeName`. Как следует из названия, в этом свойстве должно быть указано имя вычислительного узла, на котором будет размещен под. Если это свойство заполнено, под считается *запланированным* (scheduled), а если нет, значит, под находится в состоянии *ожидания* (pending).

Нам необходимо найти способ заполнения значения `nodeName` — и именно эту задачу выполняет `kube-scheduler`. Он периодически опрашивает компонент `kube-apiserver`, находит объекты Pod с пустым свойством `nodeName` и запускает алгоритм выбора подходящего узла. После того как выбор сделан, `kube-scheduler` обновляет поле `nodeName` путем отправки соответствующего запроса API-серверу. При выборе узла планировщик учитывает как свои внутренние политики, так и критерии, заданные пользователем в спецификации пода. Наконец, агент `kubelet` выбранного узла обнаруживает новый запланированный под и приступает к его запуску. Этот принцип показан на рис. 19.1.

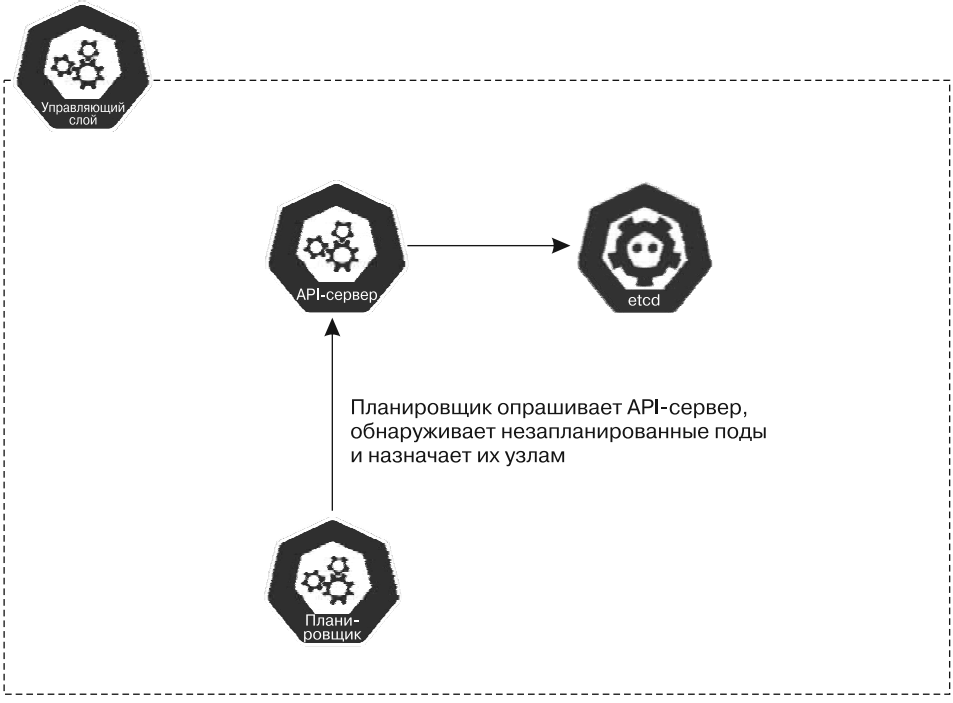

Рис. 19.1. Взаимодействие компонентов kube-scheduler и kube-apiserver

Процесс планирования пода выполняется в два этапа.

- *Фильтрация*. Компонент `kube-scheduler` определяет набор узлов, способных запустить данный под. На этом этапе он проверяет фактическое состояние узлов и сверяет их с требованиями к ресурсам и другими критериями, указанными в спецификации пода. Если подходящих узлов не обнаруживается, под не может быть запланирован и остается в состоянии ожидания.
- *Оценка*. Компонент `kube-scheduler` присваивает каждому узлу оценку на основе *политик планирования* и назначает под узлу с наивысшей оценкой. О политиках планирования мы подробно поговорим далее в главе.

Планировщик `kube-scheduler` учитывает критерии и параметры конфигурации, заданные в спецификации пода. Благодаря этому вы можете управлять процессом выбора узла для размещения пода. Для этого достаточно задать специальные ограничения или указать предпочтительные узлы. Обычно Kubernetes весьма эффективно распределяет поды между узлами без использования явных ограничений, предотвращая дефицит ресурсов. Однако иногда возникает необходимость повлиять на их размещение, например гарантировать запуск пода на узле с SSD-накопителем или разместить вместе те поды, которые активно взаимодействуют и должны находиться в одной зоне доступности.

Корректность решений планировщика гарантируется только в момент первоначального назначения пода узлу. После запуска пода компонент kube-scheduler не выполняет повторное планирование на протяжении всего времени работы пода (которое может длиться дни или даже месяцы). Таким образом, даже если под перестанет соответствовать узлу согласно вашим правилам, он продолжит работу. Повторное планирование произойдет только в случае завершения работы этого пода и создания нового[1].

Для управления распределением подов в Kubernetes можно использовать следующие механизмы:

- поле `nodeSelector` для сопоставления с метками узлов;
- правила предпочтения (affinity) и антипредпочтения (anti-affinity) узлов;
- поле `nodeName`;
- ограничения распределения подов по топологии (pod topology spread constraints);
- ограничения (taint) и допуски (toleration).

В следующих разделах мы подробно рассмотрим эти механизмы. Прежде чем переходить к практическим упражнениям по планированию подов, убедитесь в наличии многоузлового кластера Kubernetes.

В нашем случае будет использоваться кластер, развернутый с помощью `minikube`, как показано ниже (вы можете изменить драйвер на kvm2, Docker или Podman):

```
$ minikube start \
    --driver=virtualbox \
    --nodes 3 \
    --cni calico \
    --cpus=2 \
    --memory=2g \
    --kubernetes-version=v1.30.0 \
    --container-runtime=containerd
```

Аргумент `--nodes=3` указывает `minikube` развернуть кластер Kubernetes с тремя узлами, причем первый будет управляющим узлом, а второй и третий — вычислительными:

```
$ kubectl get nodes
NAME           STATUS   ROLES           AGE     VERSION
minikube       Ready    control-plane   3m34s   v1.30.0
```

[1] Для решения этой проблемы есть инструмент kubernetes-sigs/descheduler, который может быть установлен как Job/CronJob или Deployment в существующий кластер.

```
minikube-m02   Ready    <none>          2m34s   v1.30.0
minikube-m03   Ready    <none>          87s     v1.30.0
```

Если вы используете другой кластер Kubernetes, можете пропустить этап развертывания кластера `minikube`.

Теперь поговорим о правилах предпочтения узлов, а также о параметрах `nodeName` и `nodeSelector`.

Правила предпочтения узлов

Чтобы понять, как работают правила предпочтения узлов в Kubernetes, сначала нужно рассмотреть наиболее простые способы планирования подов, основанные на использовании полей `nodeName` и `nodeSelector` в их спецификации.

Использование поля nodeName в спецификации подов

Как мы уже отмечали, каждый объект Pod предусматривает поле `nodeName`, которое обычно заполняется самим планировщиком `kube-scheduler`. Тем не менее это свойство можно задать явно в YAML-манифесте при создании пода или в контроллере, использующем шаблон пода. Это самый простой способ статического назначения пода конкретному узлу, однако использовать его, как правило, *не рекомендуется* по причине негибкости и невозможности масштабирования. Если имя узла со временем изменится, вы рискуете столкнуться с нехваткой ресурсов на выбранном узле.

Иногда явное указание значения nodeName может оказаться полезным для запуска пода на конкретном узле в диагностических целях.

Принципы планирования подов будут демонстрироваться на примере простого объекта Deployment (см. главу 11), управляющего *пятью* репликами подов с веб-сервером `nginx`.

Перед использованием поля `nodeName` в манифесте Deployment следует выяснить, какие узлы присутствуют в кластере, чтобы понимать, как именно распределяются поды и как можно повлиять на этот процесс. Для получения списка узлов используйте команду `kubectl get nodes`:

```
$ kubectl get nodes
NAME           STATUS   ROLES           AGE     VERSION
minikube       Ready    control-plane   3m34s   v1.30.0
minikube-m02   Ready    <none>          2m34s   v1.30.0
minikube-m03   Ready    <none>          87s     v1.30.0
```

В нашем примере используется трехузловой кластер (не забудьте указать корректные имена узлов своего кластера в манифесте). Для удобства будем называть `minikube` узлом 1, `minikube-m02` — узлом 2, а `minikube-m03` — узлом 3.

В качестве демонстрации назначим все пять подов `nginx` узлу `minikube-m02`. Для этого создайте YAML-манифест `01_nodename/nginx-deployment.yaml` со следующим содержимым:

```
# 01_nodename/nginx-deployment.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-app
spec:
  replicas: 5
  selector:
    matchLabels:
      app: nginx
      environment: test
  template:
    metadata:
      labels:
        app: nginx
        environment: test
    spec:
      containers:
        - name: nginx
          image: nginx:1.17
          ports:
          - containerPort: 80
```

Примените этот манифест следующим образом:

```
$ kubectl apply -f 01_nodename/nginx-deployment.yaml
deployment.apps/nginx-app created
```

Объект Deployment создаст пять реплик пода. Просмотрите поды и имена узлов с помощью команды `kubectl get pods`, настроив ее вывод следующим образом:

```
$ kubectl get pods --namespace default \
    --output=custom-columns="NAME:.metadata.name,STATUS:.status.phase,NODE:.
spec.nodeName"
NAME                              STATUS    NODE
nginx-app-7b547cfd87-4g9qx        Running   minikube
nginx-app-7b547cfd87-m76l2        Running   minikube-m02
nginx-app-7b547cfd87-mjf78        Running   minikube-m03
nginx-app-7b547cfd87-vvrgk        Running   minikube-m02
nginx-app-7b547cfd87-w7jcw        Running   minikube-m03
```

Как видите, по умолчанию поды распределяются равномерно: узел 1 получил один под, а узлы 2 и 3 — по два пода. Это результат работы стандартных политик планировщика `kube-scheduler`, отвечающих за фильтрацию и оценку узлов.

Если вы управляете кластером Kubernetes самостоятельно, то можете просмотреть журналы пода kube-scheduler с помощью команды kubectl logs или непосредственно на узлах управляющего слоя — в файле /var/log/kube-scheduler.log. В некоторых случаях для получения достаточной информации бывает необходимо предварительно повысить уровень детализации журналов (verbosity). Подробнее об этом можно узнать здесь: https://kubernetes.io/docs/reference/command-line-tools-reference/kube-scheduler/.

На данном этапе шаблон пода в разделе `.spec.template.spec` не содержит никаких настроек, влияющих на процесс распределения реплик пода.

Давайте *принудительно* назначим все поды объекта Deployment *узлу 2* (в нашем случае это `minikube-m02`), используя поле `nodeName` в шаблоне пода. Измените манифест `nginx-deployment.yaml`, указав корректное имя узла *вашего* кластера:

```
# 01_nodename/nginx-deployment.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-app
spec:
  ...
  template:
    ...
    spec:
      nodeName: minikube-m02
...<фрагмент опущен для краткости>...
```

Обратите внимание на строку `nodeName: minikube-m02` — с ее помощью мы явно указываем, что поды `nginx` должны запускаться именно на узле `minikube-m02`.

Примените манифест к кластеру, используя команду `kubectl apply -f`:

```
$ kubectl apply -f 01_nodename/nginx-deployment.yaml
deployment.apps/nginx-app created
```

Теперь снова проверьте состояние подов и узлы, которым они назначены:

```
$ kubectl get pods --namespace default \
    --output=custom-columns="NAME:.metadata.name,STATUS:.status.phase,NODE:.
spec.nodeName"
NAME                              STATUS    NODE
nginx-app-85b577894f-8tqj7        Running   minikube-m02
nginx-app-85b577894f-9c6hd        Running   minikube-m02
nginx-app-85b577894f-fldxx        Running   minikube-m02
nginx-app-85b577894f-jrnjc        Running   minikube-m02
nginx-app-85b577894f-vs7c5        Running   minikube-m02
```

Как и ожидалось, теперь *все пять* подов работают на *узле 2* (`minikube-m02`). Учтите, что это новые поды: изменение шаблона пода в спецификации Deployment

инициирует внутренний процесс обновления, при котором создается новый объект ReplicaSet, а количество реплик в старом постепенно уменьшается до нуля, как было объяснено в главе 11.

Таким образом, мы фактически обошли kube-scheduler. Если вы просмотрите список событий одного из подов с помощью команды kubectl describe pod, то заметите, что в столбце Reason отсутствуют записи Scheduled.

Далее мы рассмотрим еще один простой способ планирования подов с помощью поля `nodeSelector`.

Использование поля nodeSelector в спецификации подов

В спецификации пода есть поле `.spec.nodeSelector`, которое позволяет назначать этот под только узлам, имеющим определенные метки. Данный механизм работает подобно *селекторам меток* в объектах Deployment и StatefulSet, однако он поддерживает только простые проверки на *равенство*, но не позволяет реализовать сложную логику на основе *множеств*.

Этот подход особенно полезен в следующих сценариях.

- *Гибридные кластеры*. Указав конкретную ОС в качестве критерия выбора узла, можно гарантировать то, что контейнеры Windows будут запускаться на узлах Windows, а контейнеры Linux — на узлах Linux.
- *Распределение ресурсов*. Поды можно назначать узлам, обладающим определенными ресурсами (ЦП, память, хранилище), оптимизируя их использование.
- *Аппаратные требования*. Поды, нуждающиеся в специализированном оборудовании (например, в GPU), можно назначать только узлам, обладающим соответствующими возможностями.
- *Зоны безопасности*. Мы можем определить зоны безопасности с помощью меток и использовать поле `nodeSelector` для размещения подов только в определенных зонах.

Каждый узел Kubernetes по умолчанию имеет набор стандартных меток, среди которых:

- `kubernetes.io/arch` — архитектура процессора узла, например `amd64` или `arm` (также существует в виде `beta.kubernetes.io/arch`);
- `kubernetes.io/os` — операционная система узла, например `linux` или `windows` (также существует в виде `beta.kubernetes.io/os`);
- `node-role.kubernetes.io/control-plane` — роль узла в кластере Kubernetes.

Если вы просмотрите метки любого узла, то увидите, что их довольно много. В нашем примере некоторые метки являются специфическими для кластеров `minikube`:

```
$ kubectl describe nodes minikube
Name:           minikube
Roles:          control-plane
Labels:         beta.kubernetes.io/arch=amd64
                beta.kubernetes.io/os=linux
                kubernetes.io/arch=amd64
                kubernetes.io/hostname=minikube
                kubernetes.io/os=linux
                minikube.k8s.io/commit=5883c09216182566a63dff4c326a6fc9ed2982ff
                minikube.k8s.io/name=minikube
                minikube.k8s.io/primary=true
                minikube.k8s.io/updated_at=2024_07_21T16_40_25_0700
                minikube.k8s.io/version=v1.33.1
                node-role.kubernetes.io/control-plane=
                node.kubernetes.io/exclude-from-external-load-balancers=
...<фрагмент опущен для краткости>...
```

Разумеется, вы можете определять *собственные* метки для узлов и использовать их для управления процессом распределения подов. Однако помните, что ресурсы Kubernetes следует снабжать семантическими метками, которые отражают их сущность, а не специальными метками, предназначенными исключительно для целей планирования. Чтобы понять, как это делается, выполните следующие шаги.

1. Используйте команду `kubectl label nodes`, чтобы добавить метку `node-type=superfast` для узла 1 и узла 2 в кластере:

```
$ kubectl label nodes minikube-m02 node-type=superfast
node/minikube-m02 labeled
$ kubectl label nodes minikube-m03 node-type=superfast
node/minikube-m03 labeled
```

2. Проверьте метки узлов с помощью следующей команды:

```
$ kubectl get nodes --show-labels | grep superfast
minikube-m02   Ready   <none>   2m59s   v1.31.0   beta.kubernetes.io/
arch=amd64,beta.kubernetes.io/os=linux,kubernetes.io/arch=amd64,kubernetes.
io/hostname=minikube-m02,kubernetes.io/os=linux,minikube.k8s.io/commit=5883
c09216182566a63dff4c326a6fc9ed2982ff,minikube.k8s.io/name=minikube,minikube.
k8s.io/primary=false,minikube.k8s.io/updated_at=2024_10_13T15_26_06_0700,
minikube.k8s.io/version=v1.33.1,node-type=superfast
minikube-m03   Ready   <none>   2m30s   v1.31.0   beta.kubernetes.io/
arch=amd64,beta.kubernetes.io/os=linux,kubernetes.io/arch=amd64,kubernetes.
io/hostname=minikube-m03,kubernetes.io/os=linux,minikube.k8s.io/commit=
5883c09216182566a63dff4c326a6fc9ed2982ff,minikube.k8s.io/
name=minikube,minikube.k8s.io/primary=false,minikube.k8s.io/updated_at=
2024_10_13T15_26_34_0700,minikube.k8s.io/version=v1.33.1,node-type=superfast
```

3. Измените манифест `./nginx-deployment.yaml` (или создайте отдельный файл `02_nodeselector/nginx-deployment.yaml`), указав `node-type: superfast` в качестве селектора узла в шаблоне пода:

```
# 02_nodeselector/nginx-deployment.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-app
spec:
  ...
  template:
    ...
    spec:
      nodeSelector:
        node-type: superfast
...<фрагмент опущен для краткости>...
```

4. Примените манифест к кластеру с помощью команды `kubectl apply -f 02_nodeselector/nginx deployment.yaml` и снова проверьте статус подов и узлы, которым они назначены. Возможно, вам придется дождаться завершения обновления объекта Deployment:

```
$ kubectl get pods --namespace default \
    --output=custom-columns="NAME:.metadata.name,STATUS:.status.phase,
NODE:.spec.nodeName"

NAME                          STATUS    NODE
nginx-app-6c5b8b758-2dcsc     Running   minikube-m02
nginx-app-6c5b8b758-48c5t     Running   minikube-m03
nginx-app-6c5b8b758-pfmvg     Running   minikube-m03
nginx-app-6c5b8b758-v6rhj     Running   minikube-m02
nginx-app-6c5b8b758-zqvqm     Running   minikube-m02
```

Как видите, теперь поды назначены узлам `minikube-m02` и `minikube-m03`, имеющим метку `node-type=superfast` (узел `minikube-m02` получил три пода, а `minikube-m03` — два).

5. Если изменить манифест `./nginx-deployment.yaml`, указав в качестве селектора узла в шаблоне пода значение `node-type: slow`, которое не назначено ни одному узлу в кластере, то поды не будут запланированы и развертывание зависнет. Отредактируйте манифест следующим образом (или создайте новый файл `02_nodeselector/nginx-deployment-slow.yaml`):

```
# 02_nodeselector/nginx-deployment.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-app
```

```
spec:
  ...
  template:
    ...
    spec:
      nodeSelector:
        node-type: slow
...<фрагмент опущен для краткости>...
```

6. Примените манифест к кластеру:

```
$ kubectl apply -f 02_nodeselector/nginx-deployment-slow.yaml
```

Проверьте статус подов и узлы, которым они назначены:

```
$ kubectl get pods --namespace default \
    --output=custom-columns="NAME:.metadata.name,STATUS:.status.phase,
NODE:.spec.nodeName"
NAME                          STATUS    NODE
nginx-app-6c5b8b758-48c5t     Running   minikube-m03
nginx-app-6c5b8b758-pfmvg     Running   minikube-m03
nginx-app-6c5b8b758-v6rhj     Running   minikube-m02
nginx-app-6c5b8b758-zqvqm     Running   minikube-m02
nginx-app-9cc8544f4-7dwcd     Pending   <none>
nginx-app-9cc8544f4-cz947     Pending   <none>
nginx-app-9cc8544f4-lfqqj     Pending   <none>
```

Причина, по которой три новых пода находятся в состоянии ожидания, а четыре старых пода продолжают работать, заключается в стандартных настройках обновления объекта Deployment. По умолчанию значение параметра `maxSurge` равно 25 % от числа реплик пода (при этом абсолютное значение округляется *в бóльшую сторону*), поэтому в нашем случае можно создать два дополнительных пода сверх желаемых пяти. В итоге в кластере оказывается семь подов. Значение параметра `maxUnavailable` также равно 25 % от числа реплик (но его абсолютное значение округляется *в меньшую сторону*). Это означает, что один из пяти подов может быть недоступным. Другими словами, четыре пода должны находиться в состоянии `Running`. А поскольку новые поды, находящиеся в состоянии `Pending`, не могут быть назначены узлу, объект Deployment не может продолжить обновление. В такой ситуации можно выполнить откат к предыдущей версии объекта Deployment или изменить значение `nodeSelector` на то, которое соответствует существующим узлам. Альтернативным вариантом является добавление нового узла с нужной меткой или присвоение недостающих меток уже существующим узлам без выполнения отката.

Далее мы обсудим более продвинутый механизм планирования подов, а именно правила *предпочтения узлов*.

Использование поля nodeAffinity в спецификации подов

Концепция предпочтения узлов (node affinity) расширяет подход, основанный на использовании поля `nodeSelector`, предлагая более выразительный язык для описания желательных или нежелательных узлов для назначения пода. В обычной речи под «предпочтением» подразумевается «естественная склонность или симпатия», и это очень точно передает суть правил предпочтения узлов, позволяющих вам контролировать то, к каким узлам ваш под будет «тяготеть», а от каких — «отталкиваться».

Использование правил предпочтения узлов, которые задаются в поле `.spec.affinity.nodeAffinity`, дает следующие преимущества.

- *Более выразительные правила сопоставления подов и узлов.* Например, для проверки меток можно использовать операторы `In`, `NotIn`, `Exists`, `DoesNotExist`, `Gt` и `Lt`.
- *По аналогии с проверкой совместимости с узлами (nodeAffinity) можно использовать правила совместимости между подами (inter-Pod affinity) и несовместимости (anti-affinity).* Правила несовместимости (`podAntiAffinity`) имеют противоположный эффект, «отталкивая» поды от узлов. С помощью этих правил можно сделать так, чтобы поды притягивались к узлам или отталкивались от узлов, на которых *уже выполняются* определенные поды. Это особенно полезно, если вы хотите разместить поды вместе для уменьшения задержек (или, соответственно, разделить их).
- *Возможность задавать мягкие правила совместимости и несовместимости (предпочтения/антипредпочтения).* При использовании таких правил планировщик может назначить под узлу, даже если тот им не соответствует. Мягкие правила задаются в поле `preferredDuringSchedulingIgnoredDuringExecution`, а жесткие — в поле `requiredDuringSchedulingIgnoredDuringExecution`.
- *Поддержка весов для мягких правил.* Можно определить несколько правил с разными значениями веса. Планировщик будет учитывать эти веса наряду с другими параметрами при выборе узла.

Несмотря на отсутствие отдельного поля для задания правил антипредпочтения узлов в спецификации пода, нужного эффекта можно добиться, используя операторы NotIn и DoesNotExist. С их помощью можно сделать так, чтобы поды «отталкивались» от узлов с определенными метками, в том числе мягким способом. Подробнее об этом можно узнать в документации: https://kubernetes.io/docs/concepts/scheduling-eviction/assign-pod-node/#affinity-and-anti-affinity.

Сценарии использования правил предпочтения и антипредпочтения узлов, по сути, *неограниченны*. При наличии достаточного количества меток можно выразить практически любые требования. Например, можно задать правило: «размещать под только на узле Windows с процессором Intel и премиальным

хранилищем в регионе West Europe при условии, что на этом узле не выполняются поды MySQL». Или: «желательно не размещать под в зоне доступности 1, но если выбора нет, зона 1 допустима».

Для демонстрации принципа работы правил предпочтения узлов мы смоделируем следующие требования для нашего объекта Deployment: «*Стараться* размещать поды только на узлах с меткой `node-type`, имеющей значение `fast` или `superfast`, а если это невозможно — использовать любой узел, *за исключением* узла с меткой `node-type = extremelyslow`». Для этого нам понадобятся:

- *мягкое правило предпочтения узлов* типа `preferredDuringSchedulingIgnoredDuringExecution` для сопоставления подов с узлами `fast` и `superfast`;
- *жесткое правило предпочтения узлов* типа `requiredDuringSchedulingIgnoredDuringExecution`, строго исключающее узлы с меткой `node-type = extremelyslow`. Для реализации эффекта антипредпочтения нам потребуется оператор `NotIn`.

Узлы нашего кластера изначально будут иметь следующие метки:

- узел 1: `slow`;
- узел 2: `fast`;
- узел 3: `superfast`.

Согласно описанным выше требованиям поды Deployment должны быть размещены на узле 2 и узле 3, если этому не препятствует нехватка ресурсов ЦП или памяти. В этом случае допускается их назначение узлу 1, поскольку мы используем мягкое правило предпочтения узлов.

Затем мы изменим метки узлов следующим образом:

- узел 1: `slow`;
- узел 2: `extremelyslow`;
- узел 3: `extremelyslow`.

После этого нам нужно будет повторно развернуть объект Deployment (например, уменьшить число реплик до нуля и снова увеличить его до исходного значения либо выполнить команду `kubectl rollout restart`), чтобы заново распределить поды. Согласно нашим требованиям планировщик `kube-scheduler` должен назначить все поды узлу 1 (поскольку он по-прежнему считается допустимым в соответствии с мягким правилом) и *строго* избегать узлов 2 и 3. Если ресурсов узла 1 окажется недостаточно, поды останутся в состоянии `Pending`.

Для решения задачи повторного планирования уже работающих подов (при котором kube-scheduler заново оценивает их соответствие текущему состоянию кластера) в экосистеме Kubernetes существует экспериментальный проект под названием Descheduler. Подробнее о нем можно узнать здесь: https://github.com/kubernetes-sigs/descheduler.

Для выполнения демонстрации сделайте следующее.

1. Используйте команду `kubectl label nodes`, чтобы присвоить метку `node-type=slow` узлу 1, `node-type=fast` — узлу 2 и `node-type=superfast` — узлу 3:

```
$ kubectl label nodes --overwrite minikube node-type=slow
node/minikube labeled

$ kubectl label nodes --overwrite minikube-m02 node-type=fast
node/minikube-m02 labeled

$ kubectl label nodes --overwrite minikube-m03 node-type=superfast
node/minikube-m03 not labeled
# Обратите внимание, что данная метка уже существовала
```

2. Отредактируйте манифест `03_affinity/nginx-deployment.yaml` и задайте мягкое правило предпочтения узлов следующим образом:

```
# 03_affinity/nginx-deployment.yaml
...
spec:
  ...
  template:
    ...
    spec:
      affinity:
        nodeAffinity:
          requiredDuringSchedulingIgnoredDuringExecution:
            nodeSelectorTerms:
            - matchExpressions:
              - key: node-type
                operator: NotIn
                values:
                - extremelyslow
          preferredDuringSchedulingIgnoredDuringExecution:
          - weight: 1
            preference:
              matchExpressions:
              - key: node-type
                operator: In
                values:
                - fast
                - superfast
...<фрагмент опущен для краткости>...
```

Как видите, мы использовали `nodeAffinity` (но не `podAffinity` или `podAntiAffinity`) с параметром `preferredDuringSchedulingIgnoredDuringExecution`, определяющим одно мягкое правило: метка `node-type` должна иметь значение `fast` или `superfast`. Это означает, что при нехватке ресурсов на таких узлах поды могут быть назначены другим узлам. Дополнительно мы указали одно жесткое правило антипредпочтения (`requiredDuringSchedulingIgnoredDuringExecution`), которое требует, что-

бы узел *не* имел метку `node-type = extremelyslow`. Полную спецификацию поля `.spec.affinity` для подов можно найти в официальной документации по адресу https://kubernetes.io/docs/concepts/scheduling-eviction/assign-pod-node/#affinity-and-anti-affinity.

3. Примените манифест к кластеру с помощью команды `kubectl apply -f 03_affinity/nginx-deployment.yaml`. Затем снова проверьте статус подов и узлы, которым они назначены. Возможно, вам придется дождаться завершения обновления объекта Deployment:

```
$ kubectl get pods --namespace default \
    --output=custom-columns="NAME:.metadata.name,STATUS:.status.phase,NODE:.
spec.nodeName"
NAME                          STATUS    NODE
nginx-app-7766c596cc-4d4sl    Running   minikube-m02
nginx-app-7766c596cc-4h6k6    Running   minikube-m03
nginx-app-7766c596cc-ksld5    Running   minikube-m03
nginx-app-7766c596cc-nw9hx    Running   minikube-m02
nginx-app-7766c596cc-tmwhm    Running   minikube-m03
```

Наши правила были заданы так, чтобы предпочтение отдавалось узлам, имеющим метку `node-type=fast` или `node-type=superfast`, и, как видите, поды были назначены только узлам 2 и 3.

Далее мы проведем эксперимент, демонстрирующий совместное использование мягкого правила предпочтения узлов с жестким правилом их антипредпочтения. Мы изменим метки узлов, как описано во введении, повторно развернем объект Deployment и посмотрим, что произойдет. Выполните следующие шаги:

1. Используйте команду `kubectl label nodes`, чтобы присвоить метку `node-type=slow` узлу 1, а также метку `node-type=extremelyslow` узлу 2 и узлу 3.

```
$ kubectl label nodes --overwrite minikube node-type=slow
node/minikube not labeled
# Обратите внимание, что данная метка уже существовала

$ kubectl label nodes --overwrite minikube-m02 node-type=extremelyslow
node/minikube-m02 labeled

$ kubectl label nodes --overwrite minikube-m03 node-type=extremelyslow
node/minikube-m03 labeled
```

2. Если выполнить команду `kubectl get pods` на этом этапе, в распределении подов не будет видно никаких изменений. Как было сказано выше, назначение пода узлу фиксируется в момент планирования и не меняется вплоть до перезапуска пода. Чтобы принудительно перезапустить поды, можно было бы сократить количество реплик подов объекта Deployment до нуля, а затем снова увеличить его до пяти. Однако гораздо проще использовать императивную команду `kubectl rollout restart`. Преимущество этого подхода состоит

в том, что объект Deployment остается доступным, а поды перезапускаются постепенно, без уменьшения числа доступных экземпляров. Выполните следующую команду:

```
$ kubectl rollout restart deployment nginx-app
deployment.apps/nginx-app restarted
```

3. Снова проверьте статус подов и узлы, которым они назначены. Возможно, вам придется дождаться завершения обновления объекта Deployment:

```
$ kubectl get pods --namespace default \
    --output=custom-columns="NAME:.metadata.name,STATUS:.status.phase,NODE:.
spec.nodeName"
NAME                          STATUS    NODE
nginx-app-7d8c65464c-5d9cc    Running   minikube
nginx-app-7d8c65464c-b97g8    Running   minikube
nginx-app-7d8c65464c-cqwh5    Running   minikube
nginx-app-7d8c65464c-kh8bm    Running   minikube
nginx-app-7d8c65464c-xhpss    Running   mi
```

Как видите, все поды были назначены *узлу 1* с меткой `node-type=slow`. Такие узлы допустимо использовать при отсутствии более подходящих вариантов, а в данном случае узел 2 и узел 3 имеют метку `node-type=extremelyslow`, которая категорически запрещена жестким правилом антипредпочтения узлов.

Для получения еще большего контроля над процессом распределения подов можно использовать ограничения распределения подов по топологии (Pod topology spread constraints), описанные в официальной документации по адресу https://kubernetes.io/docs/concepts/workloads/pods/pod-topology-spread-constraints/.

Итак, вы успешно настроили правила предпочтения узлов для подов объекта Deployment. Далее мы рассмотрим еще один механизм планирования подов, а именно *ограничения* и *допуски*.

Выбор узлов на основе ограничений и допусков

Правила предпочтения узлов — это чрезвычайно мощный механизм управления процессом планирования подов. Однако иногда нам нужен более простой способ указания того, какие узлы должны «отталкивать» поды. Для этого в Kubernetes существуют такие более простые механизмы, как *ограничения* и *допуски*. Ограничение (taint) назначается конкретному узлу и описывает определенное неблагоприятное свойство. Чтобы под мог выполняться на узле с таким ограничением, в его спецификации должен быть указан соответствующий допуск (toleration). При этом наличие допуска у пода не означает, что узел обязательно *должен* иметь ограничение. Само понятие taint (буквально «пятно, примесь чего-то плохого

или нежелательного») хорошо отражает суть идеи: все поды по умолчанию будут избегать узла с ограничением, но мы можем наделить определенные поды способностью «допускать» или «терпеть» конкретный тип ограничения.

Если сравнить ограничения и допуски с правилами предпочтения узлов, можно увидеть, что похожие результаты могут быть достигнуты с помощью меток узлов и использования жестких или мягких правил предпочтения с оператором NotIn. Однако важное отличие состоит в том, что для ограничения можно задать эффект NoExecute, который приводит к завершению работы пода при отсутствии у него соответствующего допуска. Добиться аналогичного результата с помощью правил предпочтения узлов невозможно без ручного перезапуска пода.

Ограничения узлов задаются в формате `<ключ>=<значение>:<эффект>`. Пара «ключ — значение» *определяет* конкретное ограничение и может использоваться для более точного контроля допусков. Например, можно задать допуск, разрешающий игнорировать все ограничения с определенным ключом, независимо от значения. Несмотря на то что ограничения напоминают обычные метки, важно помнить, что они являются отдельным свойством узла и их назначение никоим образом не влияет на его метки. В рамках нашей демонстрации мы будем использовать ограничение с ключом `machine-check-exception` и значением `memory`. Разумеется, это теоретический пример, с помощью которого мы моделируем проблему с памятью на узле. Если бы нам требовалось указать на проблему с процессором или дисковой подсистемой, можно было бы задать ограничение с тем же ключом, но со значением `cpu` или `disk`.

Ограничения должны *семантически* обозначать тип проблемы, возникшей на узле. В принципе ничто не мешает вам использовать произвольные ключи и значения для создания ограничений, однако если они будут иметь логический смысл, управлять ими и определять допуски будет гораздо проще.

Ограничение может иметь различные эффекты.

- `NoSchedule` — планировщик `kube-scheduler` *не будет назначать* поды этому узлу. Аналогичного поведения можно добиться с помощью жесткого правила предпочтения узлов.
- `PreferNoSchedule` — планировщик *постарается не назначать* поды этому узлу. Аналогичного поведения можно добиться с помощью мягкого правила предпочтения узлов.
- `NoExecute` — планировщик *не будет назначать* поды этому узлу и будет *выселять* (завершать работу и пытаться запланировать заново) уже работающие на узле поды. Реализовать такое поведение с помощью правил предпочтения узлов невозможно. Учтите, что при определении допуска для ограничения данного типа можно использовать параметр `tolerationSeconds`, чтобы задать временной интервал, в течение которого под сможет игнорировать это ограничение, прежде чем будет выселен.

Kubernetes автоматически назначает ряд ограничений с эффектом `NoExecute`, отслеживая состояние узлов. Следующие ограничения встроены в систему и управляются *контроллером узла* (`NodeController`) или агентом `kubelet`.

- `node.kubernetes.io/not-ready` — назначается в тех случаях, когда состояние узла `Ready` имеет значение `False`.
- `node.kubernetes.io/unreachable` — назначается в тех случаях, когда состояние узла `Ready` имеет значение `Unknown`. Это происходит, когда компонент `NodeController` не может связаться с узлом.
- `node.kubernetes.io/memory-pressure` — узел испытывает нехватку памяти.
- `node.kubernetes.io/disk-pressure` — узел испытывает нехватку дискового пространства.
- `node.kubernetes.io/network-unavailable` — сеть на узле недоступна.
- `node.kubernetes.io/unschedulable` — узел находится в состоянии, при котором ему нельзя назначать поды.
- `node.cloudprovider.kubernetes.io/uninitialized` — используется для узлов, которые подготавливаются внешним облачным провайдером. После инициализации узла компонентом `cloud-controller-manager` это ограничение автоматически удаляется.

Для назначения ограничения узлу используется команда `kubectl taint node`:

```
$ kubectl taint node =:<имяУзла> <ключ>=<значение>:<эффект>
```

Например, если мы хотим задать для узла 1 ограничение с ключом `machine-check-exception`, значением `memory` и эффектом `NoExecute`, можем использовать следующую команду:

```
$ kubectl taint node minikube machine-check-exception=memory:NoExecute
node/minikube tainted
```

Для удаления этого ограничения используется такая же команда, только с символом `-` в конце определения:

```
$ kubectl taint node minikube machine-check-exception=memory:NoExecute-
node/minikube untainted
```

Мы также можем удалить все ограничения с определенным ключом:

```
$ kubectl taint node minikube machine-check-exception:NoExecute-
```

Чтобы нейтрализовать действие ограничения для конкретных подов, можно задать допуски в их спецификации. Иными словами, допуски позволяют подам

игнорировать ограничения и запускаться на узлах, к которым они применены. Если узел имеет несколько ограничений, под должен предусматривать допуски для них для всех. Допуски задаются в разделе `.spec.tolerations` спецификации пода и имеют следующую структуру:

```
tolerations:
- key: <ключ>
  operator: <тип_оператора>
  value: <значение>
  effect: <эффект>
```

Поле `operator` может принимать значение `Equal` или `Exists`. `Equal` требует точного совпадения ключа и значения ограничения. `Exists` означает, что достаточно совпадения ключа, а значение при этом не учитывается[1]. Для игнорирования ограничения из нашего примера допуск должен выглядеть следующим образом:

```
tolerations:
- key: machine-check-exception
  operator: Equal
  value: memory
  effect: NoExecute
```

Для корректного размещения пода можно задать несколько допусков одновременно.

При создании допуска для ограничения типа `NoExecute` можно использовать дополнительное поле `tolerationSeconds`, чтобы задать временной интервал, в течение которого под сможет игнорировать это ограничение, прежде чем будет выселен. Таким способом можно реализовать механизм частичного допуска ограничения с тайм-аутом. Учтите, что использование ограничения `NoExecute` обычно требует дополнительного задания ограничения `NoSchedule`. Это предотвращает *бесконечный цикл выселения* пода, имеющего допуск `NoExecute` с параметром `tolerationSeconds`. Причина в том, что в течение заданного количества секунд ограничение не оказывает на под никакого воздействия, в том числе *не препятствует* его назначению узлу с соответствующим ограничением.

При создании подов Kubernetes автоматически добавляет два допуска с оператором Exists для ограничений node.kubernetes.io/not-ready и node.kubernetes.io/unreachable со значением параметра tolerationSeconds, равным 300.

---

[1] В kubernetes v1.35 появились дополнительные операторы числового сравнения: Gt и Lt. Почитать о них можно в официальной документации: https://kubernetes.io/docs/concepts/scheduling-eviction/taint-and-toleration/#numeric-comparison-operators.

Теперь, когда мы разобрались с концепцией ограничений и допусков, перейдем к практическим упражнениям. Выполните следующие шаги.

1. Если у вас все еще работает объект Deployment `nginx-app` с заданными ранее правилами предпочтения узлов, то в данный момент все его поды размещены на узле 1 (`minikube`). Правила предпочтения узлов были сформулированы таким образом, что поды не могут быть назначены узлу 2 и узлу 3. Посмотрим, что произойдет, если задать для узла 1 ограничение `machine-check-exception=memory:NoExecute`:

```
$ kubectl taint node minikube machine-check-exception=memory:NoExecute
node/minikube tainted
```

2. Проверьте статус подов и узлы, которым они назначены:

```
$ kubectl get pods --namespace default \
    --output=custom-columns="NAME:.metadata.name,STATUS:.status.phase,
NODE:.spec.nodeName"
NAME                        STATUS    NODE
nginx-app-7d8c65464c-5j69n  Pending   <none>
nginx-app-7d8c65464c-c8j58  Pending   <none>
nginx-app-7d8c65464c-cnczc  Pending   <none>
nginx-app-7d8c65464c-drpdh  Pending   <none>
nginx-app-7d8c65464c-xss9b  Pending   <none>
```

   Все поды объекта Deployment находятся в состоянии `Pending`, поскольку планировщик `kube-scheduler` не смог найти узел, подходящий для их запуска.

3. Отредактируйте манифест `./nginx-deployment.yaml` (или просмотрите готовый файл `04_taints/nginx-deployment.yaml`), удалив правила предпочтения узлов (`affinity`). Вместо них определите допуск для ограничения `machine-check-exception=memory:NoExecute` с тайм-аутом 60 секунд:

```
# 04_taints/nginx-deployment.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-app
spec:
  ...
  template:
    ...
    spec:
      tolerations:
        - key: machine-check-exception
          operator: Equal
          value: memory
          effect: NoExecute
          tolerationSeconds: 60
...<фрагмент опущен для краткости>...
```

   После применения манифеста старые правила предпочтения узлов, препятствовавшие назначению подов узлам 2 и 3, перестанут действовать. Иначе

говоря, поды снова смогут назначаться этим узлам. Однако для узла 1 по-прежнему задано ограничение `machine-check-exception=memory:NoExecute`. Значит, поды *не должны* назначаться узлу 1, так как эффект `NoExecute` подразумевает и `NoSchedule`, верно? Давайте это проверим.

4. Примените манифест к кластеру с помощью команды `kubectl apply -f 04_taints/nginx-deployment.yaml` и снова проверьте статус подов и узлы, которым они назначены. Возможно, вам придется дождаться завершения обновления объекта Deployment:

```
$ kubectl get pods --namespace default \
    --output=custom-columns="NAME:.metadata.name,STATUS:.status.phase,
NODE:.spec.nodeName"
NAME                          STATUS    NODE
nginx-app-84d755f746-4zkjd    Running   minikube
nginx-app-84d755f746-58qmh    Running   minikube-m02
nginx-app-84d755f746-5h5vk    Running   minikube-m03
nginx-app-84d755f746-psmgf    Running   minikube-m02
nginx-app-84d755f746-zkbc6    Running   minikube-m03
```

Результат может показаться неожиданным. Как видите, часть подов была назначена узлу 2 и узлу 3, но узел 1 также получает поды, которые выселяются каждые 60 секунд! Причина в том, что из-за параметра `tolerationSeconds`, заданного в допуске для ограничения `NoExecute`, эффект этого ограничения полностью игнорируется на протяжении 60 секунд. Поэтому планировщик может назначить под узлу 1, даже если позднее этот под будет выселен.

5. Давайте исправим это поведение, следуя рекомендации, в соответствии с которой использование ограничения `NoExecute` должно сопровождаться заданием ограничения `NoSchedule`. Это гарантирует то, что выселенные поды не смогут быть повторно назначены узлу с таким ограничением, если только у подов не появится допуск и для этого ограничения. Назначьте узлу 1 ограничение `NoSchedule` с помощью следующей команды:

```
$ kubectl taint node minikube machine-check-exception=memory:NoSchedule
node/minikube tainted
```

6. Проверьте статус подов и узлы, которым они назначены:

```
$ kubectl get pods --namespace default \
    --output=custom-columns="NAME:.metadata.name,STATUS:.status.phase,
NODE:.spec.nodeName"
NAME                          STATUS    NODE
nginx-app-84d755f746-58qmh    Running   minikube-m02
nginx-app-84d755f746-5h5vk    Running   minikube-m03
nginx-app-84d755f746-psmgf    Running   minikube-m02
nginx-app-84d755f746-sm2cm    Running   minikube-m03
nginx-app-84d755f746-zkbc6    Running   minikube-m03
```

Как видите, поды распределены между узлом 2 и узлом 3, что нам и требовалось.

7. Теперь удалите *оба* ограничения с узла 1:

```
$ kubectl taint node minikube machine-check-exception-node/
minikube untainted
```

8. Перезапустите объект Deployment, чтобы выполнить повторное назначение подов:

```
$ kubectl rollout restart deployment nginx-app
deployment.apps/nginx-app restarted
```

9. Проверьте статус подов и узлы, которым они назначены:

```
$ kubectl get pods --namespace default \
    --output=custom-columns="NAME:.metadata.name,STATUS:.status.phase,NODE:.
spec.nodeName"
NAME                          STATUS    NODE
nginx-app-5bdd957558-fj7bk    Running   minikube-m02
nginx-app-5bdd957558-mrddn    Running   minikube-m03
nginx-app-5bdd957558-mz2pz    Running   minikube-m02
nginx-app-5bdd957558-pftz5    Running   minikube-m03
nginx-app-5bdd957558-vm6k9    Running   minikube
```

Как видите, поды снова были распределены между тремя узлами.

10. Наконец, посмотрим, как работает комбинация ограничений `NoExecute` и `NoSchedule` при использовании параметра `tolerationSeconds = 60` в допуске для ограничения `NoExecute`. Снова задайте оба ограничения для узла 1:

```
$ kubectl taint node minikube machine-check-exception=memory:NoSchedule
node/minikube tainted
$ kubectl taint node minikube machine-check-exception=memory:NoExecute
node/minikube tainted
```

11. Сразу после этого начните наблюдать за подами и узлами, которым они назначаются. Сначала вы увидите, что поды продолжают назначаться узлу 1. Однако по истечении 60 секунд вы увидите следующее:

```
$ kubectl get pods --namespace default \
    --output=custom-columns="NAME:.metadata.name,STATUS:.status.phase,NODE:.
spec.nodeName"
NAME                          STATUS    NODE
nginx-app-5bdd957558-7n42p    Running   minikube-m03
nginx-app-5bdd957558-fj7bk    Running   minikube-m02
nginx-app-5bdd957558-mrddn    Running   minikube-m03
nginx-app-5bdd957558-mz2pz    Running   minikube-m02
nginx-app-5bdd957558-pftz5    Running   minikube-m03
```

Как и ожидалось, поды были выселены через 60 секунд и бесконечного цикла выселения не возникло.

Этот пример демонстрирует более сложный сценарий использования ограничений, который невозможно воспроизвести с помощью правил предпочтения узлов.

В следующем разделе мы поговорим о статических подах.

Статические поды в Kubernetes

Статические поды предоставляют альтернативный способ управления подами внутри кластера Kubernetes. В отличие от обычных подов, которые контролируются планировщиком кластера и API-сервером, статические поды управляются непосредственно агентом `kubelet` на конкретном узле. API-сервер не знает о статических подах — он видит только зеркальные поды (mirror Pods), которые `kubelet` создает для отражения их состояния.

Перечислим ключевые особенности статических подов.

- *Привязка к узлу*. Статические поды жестко связаны с конкретным узлом и не могут быть перенесены на другие узлы кластера.
- *Управление агентом kubelet*. Агент `kubelet` выбранного узла отвечает за запуск, остановку и перезапуск статических подов.
- *Зеркальные поды*. Агент `kubelet` создает зеркальные поды на API-сервере для отражения состояния статических подов, однако управлять этими зеркальными подами через API невозможно.

Создавать статические поды можно двумя способами. Первый предполагает сохранение конфигурации в файловой системе. В этом случае вы размещаете определение пода в формате YAML или JSON в специальной папке на узле. Агент `kubelet` отслеживает эту папку и управляет подами в соответствии с обнаруженными в ней файлами. Второй способ предполагает сохранение YAML-файла с определением пода на веб-сервере. Агент `kubelet` получает URL-адрес этого файла, периодически загружает его и управляет статическими подами на основе полученной конфигурации.

Статические поды часто используются для начального развертывания таких критически важных компонентов кластера, как API-сервер или менеджер контроллеров, на каждом узле. Однако для запуска подов на каждом узле кластера обычно предпочтительнее использовать объект DaemonSet. Статические поды имеют ограничения, например, они не могут ссылаться на такие объекты Kubernetes, как Secret или ConfigMap, а также не поддерживают эфемерные контейнеры. Тем не менее они могут пригодиться в ситуациях, требующих повышенного контроля над размещением подов на отдельных узлах.

Итак, мы рассмотрели различные механизмы управления процессом планирования и размещения подов. В следующем разделе обсудим дополнительные возможности и настройки планировщика Kubernetes.

Расширенные конфигурации планировщика Kubernetes

В дополнение к стандартным настройкам планировщика система Kubernetes поддерживает ряд расширенных конфигураций, которые мы рассмотрим в этом разделе.

Конфигурация планировщика

Поведение планировщика можно настраивать с помощью специального конфигурационного файла. Этот файл определяет, как планировщик будет приоритизировать узлы при выборе места размещения подов с учетом различных критериев.

Рассмотрим ключевые понятия.

- *Профили планирования.* В конфигурационном файле можно задать несколько профилей планирования. Каждый профиль имеет собственное имя и может быть настроен с использованием индивидуального набора плагинов.
- *Плагины планировщика.* Плагины — это своеобразные строительные блоки, выполняющие определенные операции в процессе планирования. Они могут фильтровать узлы на основе доступности ресурсов, аппаратной совместимости или других критериев.
- *Точки расширения.* Это этапы процесса планирования, на которых можно «подключать» плагины. Разные плагины подходят для разных этапов, например для фильтрации неподходящих узлов и оценки подходящих.

Важное примечание: политики планирования (в версиях Kubernetes до v1.23)

До выхода версии Kubernetes v1.23 политики планирования можно было задавать с помощью флагов компонента kube-scheduler или через объект ConfigMap. Эти политики определяли то, как планировщик выбирает узлы для подов, используя предикаты (критерии фильтрации) и приоритеты (оценочные функции). В версии v1.23 этот механизм был заменен на конфигурацию планировщика. Новый подход обеспечивает значительно большую гибкость и позволяет точнее управлять процессом планирования.

Конфигурационный файл планировщика Kubernetes предоставляет ряд преимуществ при управлении размещением подов в кластере. Он обеспечивает гибкость, позволяющую настраивать поведение планировщика под ваши нужды. Например, вы можете задать приоритет для подов, нуждающихся в GPU, так, чтобы они назначались лишь тем узлам, которые обладают соответствующими ресурсами. Кроме того, вы можете разрабатывать собственные плагины для удовлетворения уникальных требований к процессу планирования. Наконец, возможность определять несколько профилей планирования позволяет создавать детализированные правила, назначая разные профили разным типам подов.

Прежде чем завершить эту главу, рассмотрим механизм ограничения узлов в процессе планирования.

Изоляция узлов и ограничение их возможностей

Kubernetes позволяет изолировать поды на определенных узлах с помощью меток узлов. Эти метки могут отражать требования безопасности или соответствия нормативам, обеспечивая назначение подов только тем узлам, которые удовлетворяют заданным критериям. Чтобы скомпрометированный узел не мог изменять свои метки в собственных интересах, в системе предусмотрен плагин допуска `NodeRestriction`, который запрещает агенту `kubelet` изменять метки с определенным префиксом (например, `node-restriction.kubernetes.io/`). Для использования этой функциональности необходимо активизировать плагин `NodeRestriction` и режим `Node authorizer`. После этого вы можете добавлять узлам метки с защищенным префиксом и использовать их в параметре `nodeSelector` в спецификации пода, чтобы гарантировать запуск подов в заранее определенной, изолированной среде.

Повышение производительности планировщика Kubernetes

В больших кластерах Kubernetes эффективность работы планировщика имеет критически важное значение. Ключевым параметром, оказывающим на нее наибольшее влияние, является `percentageOfNodesToScore`.

Параметр `percentageOfNodesToScore` определяет количество узлов, учитываемых планировщиком при поиске подходящего места для размещения пода. При более высоком значении планировщик анализирует больше узлов, что увеличивает вероятность оптимального выбора, но занимает больше времени. Напротив, более низкое значение ускоряет процесс планирования, но может привести к менее оптимальному результату в плане размещения подов. Этот параметр

можно настроить в конфигурационном файле компонента `kube-scheduler`. Допустимый диапазон значений — от 1 до 100 %. Значение по умолчанию зависит от размера кластера (50 % для кластера, состоящего из 100 узлов, 10 % для кластера, состоящего из 5000 узлов).

Чтобы задать значение `percentageOfNodesToScore`, равное 50 %, для кластера, состоящего из сотен узлов, используйте следующую конфигурацию планировщика:

```
apiVersion: kubescheduler.config.k8s.io/v1alpha1
kind: KubeSchedulerConfiguration
algorithmSource:
  provider: DefaultProvider
...
percentageOfNodesToScore: 50
```

Оптимальное значение данного параметра зависит от ваших приоритетов. Если вам нужно обеспечить максимально быстрое планирование, можно задать более низкое значение. Если же выбор оптимальных узлов важнее скорости, стоит использовать более высокое значение. Главное — не делать его слишком низким, чтобы планировщик не пропускал потенциально более подходящие узлы.

На этом мы завершаем главу, посвященную различным механизмам и стратегиям управления процессом планирования подов, а также расширенным конфигурациям планировщика Kubernetes.

Резюме

В этой главе мы рассмотрели продвинутые методы планирования подов в Kubernetes. Сначала мы описали основной принцип работы компонента `kube-scheduler` и процесс назначения подов узлам. Затем обсудили базовые механизмы планирования подов, основанные на использовании имен и селекторов узлов, после чего вы познакомились с более сложной концепцией предпочтения узлов. Мы также показали, как с помощью концепции предпочтения можно реализовать антипредпочтение узлов. После этого мы рассмотрели ограничения, задаваемые узлам, и допуски, определяемые в спецификациях подов. Вы познакомились с различными эффектами ограничений и применили эти знания при выполнении практического упражнения, в котором использовались ограничения `NoExecute` и `NoSchedule`. Наконец, мы обсудили дополнительные возможности Kubernetes, включая расширенные конфигурации планировщика, изоляцию узлов и статические поды.

В следующей главе мы поговорим об *автоматическом масштабировании* подов и узлов — функции, обеспечивающей гибкость Kubernetes при выполнении рабочих нагрузок в облачных средах.

Дополнительные источники информации

- Планировщик Kubernetes: https://kubernetes.io/docs/concepts/scheduling-eviction/kube-scheduler/.
- Назначение подов узлам: https://kubernetes.io/docs/concepts/scheduling-eviction/assign-pod-node/.
- Ограничения и допуски: https://kubernetes.io/docs/concepts/scheduling-eviction/taint-and-toleration/.
- Конфигурация планировщика: https://kubernetes.io/docs/reference/scheduling/config/.

Для получения дополнительной информации о планировании подов в Kubernetes вы можете обратиться к следующим книгам издательства Packt Publishing:

- *Baier J., Sayfan G., White J.* The Complete Kubernetes Guide (https://www.packtpub.com/en-in/product/the-complete-kubernetes-guide-9781838647346).
- *Baier J., White J.* Getting Started with Kubernetes. 3 ed. (https://www.packtpub.com/en-in/product/getting-started-with-kubernetes-9781788997263).
- *Heck J.* Kubernetes for Developers (https://www.packtpub.com/en-in/product/kubernetes-for-developers-9781788830607).

Кроме того, вы можете обратиться к официальной документации.

- Документация по Kubernetes (https://kubernetes.io/docs/home/) является наиболее актуальным источником информации об этой системе.
- Правила предпочтения узлов: https://kubernetes.io/docs/concepts/scheduling-eviction/assign-pod-node/.
- Ограничения и допуски: https://kubernetes.io/docs/concepts/scheduling-eviction/taint-and-toleration/.
- Приоритизация и вытеснение подов (в этой главе данная тема не рассматривалась): https://kubernetes.io/docs/concepts/configuration/pod-priority-preemption/.
- Расширенная настройка компонента `kube-scheduler` с использованием профилей планирования (в этой главе данная тема не рассматривалась): https://kubernetes.io/docs/reference/scheduling/config.

20 Автоматическое масштабирование подов и узлов в Kubernetes

При эксплуатации приложений в облаке наличие возможностей их *автомасштабирования* считается настоящим святым Граалем. В общих чертах под автомасштабированием понимают метод автоматического и динамического регулирования объема вычислительных ресурсов (таких как ЦП и память), доступных приложению. Цель автомасштабирования состоит в том, чтобы добавлять или удалять ресурсы в зависимости от нагрузки и активности конечных пользователей. Так, например, приложению может понадобиться больше ресурсов ЦП и памяти в дневные часы, когда пользователи наиболее активны, и значительно меньше — ночью. Аналогичным образом, если вы поддерживаете инфраструктуру платформы электронной коммерции, можете подготовиться к резкому увеличению нагрузки во время распродаж. Такой подход позволяет не только предоставлять пользователям более качественный и высокодоступный сервис, но и снижать себестоимость продаваемых товаров. Чем меньше ресурсов вы потребляете в облаке, тем меньше платите, а сэкономленные средства компания может направить на другие задачи. Это классическая ситуация, в которой выигрывают все. Разумеется, единого универсального правила на все случаи жизни не существует, поэтому эффективное автомасштабирование должно опираться на критически важные метрики и включать *прогностические механизмы*, позволяющие предсказывать будущие уровни нагрузки на основе исторических данных.

Будучи самой зрелой системой оркестрации контейнеров, Kubernetes предлагает широкий набор встроенных средств автомасштабирования. Часть этих возможностей поддерживается нативно в любом кластере Kubernetes, другие требуют установки дополнительных компонентов или определенного способа развертывания кластера. Кроме того, автомасштабирование может происходить в нескольких *измерениях*.

- *Вертикальное автомасштабирование подов* — регулирует объем ресурсов ЦП и памяти, доступных поду. Поды работают в рамках установленных лимитов, предотвращающих чрезмерное потребление ресурсов, однако эти лимиты могут требовать автоматической корректировки, а не ручного подбора оператором. Данный вариант автомасштабирования реализуется с помощью *вертикального автоскейлера* (объекта VerticalPodAutoscaler, VPA).

- *Горизонтальное автомасштабирование подов* — динамически изменяет число реплик подов объектов Deployment или StatefulSet. Эти объекты изначально предусматривают удобные механизмы масштабирования, однако регулировка количества реплик может быть автоматизирована с помощью *горизонтального* автоскейлера (объекта HorizontalPodAutoscaler, HPA).
- *Горизонтальное автомасштабирование узлов* — это еще одно измерение горизонтального масштабирования, только на уровне узлов Kubernetes. Вы можете масштабировать весь кластер, добавляя или удаляя узлы. Разумеется, для этого кластер должен быть развернут в среде, поддерживающей динамическое выделение машин, например, в облаке. Данный вариант автомасштабирования реализуется с помощью *инструмента автомасштабирования кластера* или *кластерного автоскейлера* (Cluster Autoscaler, CA), предоставляемого некоторыми облачными провайдерами.

Технические требования

Для работы с этой главой вам потребуется:

- развернутый кластер Kubernetes (желательно многоузловой);
- многоузловой кластер *GKE* (Google Kubernetes Engine), необходимый для вертикального автомасштабирования подов и автомасштабирования узлов;
- интерфейс командной строки `kubectl`, установленный на локальной машине и настроенный для управления кластером Kubernetes.

Инструкции по созданию кластера Kubernetes и настройке клиента `kubectl` приведены в главах 2 и 3.

В главах 15, 16 и 17 рассматривался процесс развертывания полнофункциональных кластеров Kubernetes в различных облачных средах и установки соответствующих CLI-инструментов для управления ими.

Примеры кода для этой главы можно загрузить из официального репозитория GitHub по адресу https://github.com/PacktPublishing/The-Kubernetes-Bible-Second-Edition/tree/main/Chapter20.

Запросы и лимиты ресурсов, доступных подам

Прежде чем перейти к теме автомасштабирования в Kubernetes, нам следует обсудить способы управления уровнем потребления *вычислительных ресурсов* (ЦП и память) контейнерами подов. Это позволяет точнее планировать потребности кластера и, что особенно важно, предотвращать ситуации, когда один контейнер потребляет все вычислительные ресурсы, лишая другие поды возможности обслуживать запросы.

При создании пода можно указать, какой объем вычислительных ресурсов *требуется* его контейнерам, а также наложить *ограничения* на их потребление. Ресурсная модель Kubernetes различает *сжимаемые* и *несжимаемые* ресурсы. Сжимаемым считается такой ресурс, который можно ограничить без серьезных последствий.

Идеальным примером служит ресурс *центрального процессора* (ЦП): если уменьшить объем ресурсов ЦП, доступный контейнеру, он продолжит работать, но чуть медленнее. Несжимаемые ресурсы нельзя ограничить без серьезных последствий. К таким ресурсам относится *память*. Если не выделить необходимый объем памяти для выполнения процесса, запущенного внутри контейнера, он завершится с ошибкой, что приведет к перезапуску контейнера.

Для управления объемом доступных ресурсов в спецификации пода можно указать два параметра.

- `requests` — определяет объем ресурса, гарантированный системой. Можно сказать, что это объем ресурса, который *требуется* контейнеру пода для корректной работы. Это значение особенно важно, поскольку при размещении подов планировщик Kubernetes ориентируется именно на параметр `requests` (а не `limits`). В частности, на основе этого значения проверяется условие `PodFitsResources` и вычисляется приоритет `BalancedResourceAllocation`.
- `limits` — определяет *максимальный* объем ресурса, который система может предоставить контейнеру. При указании обоих параметров значение `limits` должно быть больше или равно значению `requests`. В зависимости от того, является ли ресурс сжимаемым или несжимаемым, превышение лимита имеет разные последствия: в случае сжимаемых ресурсов (ЦП) это приводит к ограничению их использования, а в случае несжимаемых (память) это *может* привести к завершению работы контейнера и его перезапуску.

Задав разные значения для параметров `requests` и `limits`, мы можем разрешить оверкоммит ресурсов. Это повышает устойчивость системы к кратковременным пикам нагрузки и оптимизирует общее использование ресурсов в кластере. Данный подход работает, поскольку вероятность одновременного достижения лимитов всеми контейнерами на узле очень мала. Благодаря этому Kubernetes может более эффективно распределять доступные мощности. Это напоминает оверпровиженинг виртуальных машин или овербукинг, при котором авиакомпании продают больше билетов, чем доступно мест на борту самолета, рассчитывая на то, что не все пассажиры ими воспользуются. Таким образом, на каждом узле можно разместить больше подов, чтобы повысить эффективность использования ресурсов.

Если не задать параметр `limits`, контейнер сможет задействовать любой объем ресурсов узла. Однако их потребление можно ограничить с помощью *квот*

ресурсов и *диапазонов ограничений*, задаваемых на уровне пространства имен (см. главу 6).

В более сложных сценариях можно задать параметры requests и limits для больших страниц (huge pages) и эфемерного хранилища.

Перед обсуждением особенностей конфигурации давайте вспомним, какие единицы измерения объема ресурсов ЦП и памяти используются в Kubernetes.

- Базовой единицей измерения объема ресурсов ЦП является *KCU (Kubernetes CPU)*. Значение `1` эквивалентно одному vCPU в Azure, одному ядру в GCP или одному гиперпоточному ядру на физической машине.
- Допускаются дробные значения: например, значение `0.1` можно указать как `100m` (*милли-KCU*).
- Базовой единицей измерения объема памяти является *байт*; можно использовать стандартные приставки — `M`, `Mi`, `G`, `Gi`[1].

Чтобы задать запросы и лимиты вычислительных ресурсов для контейнеров объекта Deployment `nginx`, который мы использовали в предыдущих главах, внесите следующие изменения в YAML-манифест `resource-limit/nginx-deployment.yaml`:

```
# resource-limit/nginx-deployment.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment-example
spec:
  replicas: 5
...
  template:
    ...
    spec:
      containers:
        - name: nginx
          image: nginx:1.17
          ports:
            - containerPort: 80
          resources:
            limits:
              cpu: 200m
              memory: 60Mi
            requests:
              cpu: 100m
              memory: 50Mi
```

[1] Стоит учитывать разницу между E, P, T, G, M, k (степени 10) и Ei, Pi, Ti, Gi, Mi, Ki (степени 2). К примеру, 1k = 1000, но 1Ki = 1024.

Для каждого контейнера в поде нужно заполнить поле `.spec.template.spec.containers[*].resources`. В данном примере мы задали следующие параметры: `limits` — `200m` KCU (ресурсы ЦП) и `60Mi` (оперативная память), `requests` — `100m` KCU и `50Mi`.

После применения манифеста с помощью команды `kubectl apply -f resource-limit/nginx-deployment.yaml` вы можете выполнить команду `kubectl describe node` для описания одного из узлов, которому назначены поды нашего объекта Deployment. В выводе команды будет содержаться детальная информация о квотах и текущем распределении вычислительных ресурсов:

```
$ kubectl describe node minikube-m03
...<фрагмент опущен для краткости>...
Non-terminated Pods:          (5 in total)
  Namespace     Name                                        CPU Requests   CPU
Limits   Memory Requests   Memory Limits   Age
  ---------     ----                                        ------------   -----
-----   --------------   ------------   ---
  default       nginx-deployment-example-6d444cfd96-f5tnq     100m (5%)      200m
(10%)   50Mi (2%)         60Mi (3%)      23s
  default       nginx-deployment-example-6d444cfd96-k6j9d     100m (5%)      200m
(10%)   50Mi (2%)         60Mi (3%)      23s
  default       nginx-deployment-example-6d444cfd96-mqxxp     100m (5%)      200m
(10%)   50Mi (2%)         60Mi (3%)      23s
  kube-system   calico-node-92bdc                           250m (12%)     0
(0%)       0 (0%)            0 (0%)         6d23h
  kube-system   kube-proxy-5cd4x                            0 (0%)         0
(0%)       0 (0%)            0 (0%)         6d23h
Allocated resources:
  (Total limits may be over 100 percent, i.e., overcommitted.)
  Resource           Requests           Limits
  --------           --------           ------
  cpu                550m (27%)         600m (30%)
  memory             150Mi (7%)         180Mi (9%)
  ephemeral-storage  0 (0%)             0 (0%)
  hugepages-2Mi      0 (0%)             0 (0%)
Events: <none>
```

Теперь, опираясь на полученную информацию, вы можете поэкспериментировать, запросив для контейнера объем ресурсов ЦП, превышающий емкость одного узла в кластере. Изменим соответствующий параметр в файле `resource-limit/nginx-deployment.yaml` следующим образом:

```
...
        resources:
          limits:
            cpu: 2000m
            memory: 60Mi
          requests:
            cpu: 2000m
            memory: 50Mi
...
```

Примените измененную конфигурацию:

```
$ kubectl apply -f resource-limit/nginx-deployment.yaml
deployment.apps/nginx-deployment-example configured
```

Если вы проверите статус подов, то заметите, что новые поды остаются в состоянии ожидания (`Pending`), поскольку их невозможно разместить ни на одном подходящем узле:

```
$ kubectl get pod
NAME                                     READY   STATUS    RESTARTS   AGE
nginx-deployment-example-59b669d85f-cdptx   1/1     Running   0          52s
nginx-deployment-example-59b669d85f-hdzdf   1/1     Running   0          54s
nginx-deployment-example-59b669d85f-ktn59   1/1     Running   0          54s
nginx-deployment-example-59b669d85f-vdn87   1/1     Running   0          52s
nginx-deployment-example-69bd6d55b4-n2mzq   0/1     Pending   0          3s
nginx-deployment-example-69bd6d55b4-qb62p   0/1     Pending   0          3s
nginx-deployment-example-69bd6d55b4-w7xng   0/1     Pending   0          3s
```

Чтобы разобраться в причине этого состояния, опишите под с помощью следующей команды:

```
$ kubectl describe pod nginx-deployment-example-69bd6d55b4-n2mzq
...
Events:
  Type     Reason            Age                    From               Message
  ----     ------            ----                   ----               -------
  Warning  FailedScheduling  23m (x21 over 121m)    default-scheduler 0/3
nodes are available: 1 node(s) had untolerated taint {machine-check-exception:
memory}, 2 Insufficient cpu. preemption: 0/3 nodes are available:
1 Preemption is not helpful for scheduling, 2 No preemption victims found
for incoming pod.
```

Некоторые из рассмотренных в этой главе механизмов автомасштабирования в настоящее время находятся в стадии альфа- или бета-тестирования. Они могут быть недостаточно стабильными и не всегда подходят для использования в рабочей среде. С более надежными и готовыми к эксплуатации решениями можно ознакомиться в разделе «Альтернативные автоскейлеры для Kubernetes» далее в этой главе.

Согласно приведенному выше выводу, в кластере нет узлов, способных обеспечить работу пода, контейнеру которого требуется `2000m` KCU, поэтому такой под не может быть запланирован в данный момент.

Теперь, когда мы разобрались с управлением вычислительными ресурсами, мы можем перейти к теме автомасштабирования, начав с вертикального автомасштабирования подов.

Вертикальное автомасштабирование подов с помощью объекта VerticalPodAutoscaler

В предыдущем разделе мы вручную управляли запросами и лимитами вычислительных ресурсов. Точное задание этих значений требует весьма аккуратной *оценки*, наблюдения за метриками и проведения тестов для последующей корректировки параметров. Завышение значений параметра `requests` приводит к нерациональному использованию вычислительных ресурсов, в то время как их занижение может привести к излишне плотному размещению подов на узлах и проблемам с производительностью. Кроме того, в некоторых случаях единственным способом увеличения производительности пода является *вертикальное масштабирование*, то есть увеличение объема доступных ему вычислительных ресурсов. В случае с физическими серверами речь может идти об обновлении процессора или добавлении оперативной памяти. В случае с контейнерами все сводится к выделению им больших квот ресурсов. Разумеется, эффективность данного подхода ограничена возможностями конкретного узла. Превысить этот предел за счет вертикального масштабирования невозможно без добавления в кластер более мощных узлов.

Для решения этих задач можно использовать вертикальный автоскейлер (VPA, объект VerticalPodAutoscaler), который динамически увеличивает или уменьшает объем ресурсов ЦП и памяти, запрашиваемый контейнерами подов.

Следующая схема демонстрирует принцип вертикального масштабирования подов (рис. 20.1).

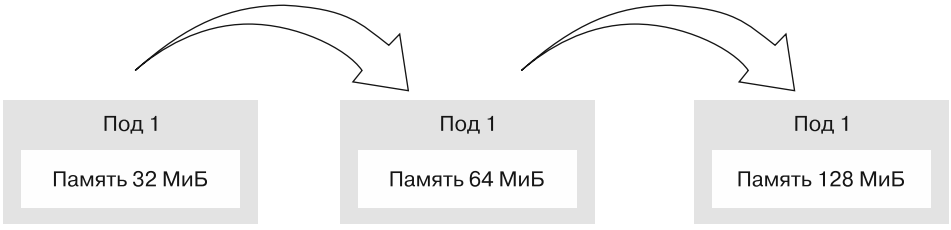

Рис. 20.1. Вертикальное масштабирование подов

Цель VPA заключается в том, чтобы согласовать объем выделенных ресурсов с *фактическими* потребностями контейнера, не полагаясь на жестко заданные значения запросов и лимитов. Кроме того, он позволяет регулировать лимиты ресурсов, поддерживая предопределенное соотношение между их запрошенным и максимально доступным объемом.

VPA создается с помощью объекта *CRD* (определение пользовательского ресурса) с именем `VerticalPodAutoscaler`. Это означает, что данный объект не входит

в стандартные группы API Kubernetes и должен устанавливаться в кластере отдельно. VPA разрабатывается в рамках проекта *autoscaler* (https://github.com/kubernetes/autoscaler) экосистемы Kubernetes.

VPA состоит из трех ключевых компонентов.

- *Recommender (рекомендатель)* — отслеживает текущее и прошлое потребление ресурсов и формирует рекомендуемые значения запросов ЦП и памяти для контейнера пода.
- *Updater (обновитель)* — выявляет поды с некорректно заданными значениями `requests` и `limits` и *удаляет* их, чтобы они могли быть пересозданы с обновленными значениями соответствующих параметров.
- *Admission plugin (плагин допуска)* — задает корректные запросы и лимиты ресурсов для новых подов или подов, пересоздаваемых их контроллером (например, объектом Deployment) на основе изменений, внесенных обновителем.

Причина, по которой обновителю приходится удалять поды и использовать плагин допуска, заключается в том, что Kubernetes не поддерживает динамическое изменение запросов и лимитов. Единственный способ применить новые параметры — завершить работу пода и создать его заново с использованием обновленных значений.

VPA можно запускать в режиме, в котором он только формирует рекомендации, не применяя их к подам. В этом случае вы увидите рекомендуемые параметры в объекте VPA, но поды останутся неизменными. В настоящее время VPA считается экспериментальным механизмом, и использование режима, предполагающего пересоздание подов, может привести к простою приложения. Ситуация должна измениться после добавления функции обновления параметров requests и limits на месте.

Некоторые Kubernetes-платформы поддерживают установку VPA «в один клик» или через оператор. Двумя хорошими примерами являются OpenShift и GKE. Реализация VPA в OpenShift описана в статье *Automatically adjust pod resource levels with the vertical pod autoscaler* (https://docs.openshift.com/container-platform/4.16/nodes/pods/nodespods-vertical-autoscaler.html).

Включение механизма InPlacePodVerticalScaling

В версии Kubernetes 1.27 появилась функция, позволяющая динамически корректировать объем ресурсов, доступных поду, без его перезапуска (in-place pod resizing), что улучшает производительность приложения и повышает эффективность использования ресурсов.

Функция изменения ресурсов пода без его перезапуска находится на стадии альфа-тестирования[1] и может быть изменена в будущих версиях Kubernetes без предварительного уведомления. Ее не следует использовать в рабочих средах ввиду возможной нестабильности. В целом любые функции, находящиеся на стадии альфа-тестирования, не считаются финальными и могут быть изменены в любой момент.

Для активизации этой функции необходимо использовать флаг `--feature-gates` на всех узлах кластера.

Это можно сделать следующим образом.

1. Отредактируйте манифест `/etc/kubernetes/manifests/kube-apiserver.yaml` (или соответствующий конфигурационный файл в вашей среде Kubernetes).
2. Добавьте флаг `--feature-gates`:

```
# /etc/kubernetes/manifests/kube-apiserver.yaml
...
  - command:
    - kube-apiserver
    ...<фрагмент опущен для краткости>...
    - --feature-gates=InPlacePodVerticalScaling=true
```

При использовании среды `minikube` добавьте этот флаг при запуске кластера:

```
$ minikube start --feature-gates=InPlacePodVerticalScaling=true
```

Далее мы поговорим о включении VPA в кластере GKE.

Включение VPA в кластере GKE

Включить вертикальный автоскейлер (VPA) в кластере *GKE* (Google Kubernetes Engine) можно с помощью следующей команды:

```
$ gcloud container clusters update <имя_кластера> --enable-vertical-pod-
autoscaling
```

Учтите, что выполнение этой операции инициирует перезапуск управляющего слоя Kubernetes.

Чтобы включить VPA при создании нового кластера, используйте дополнительный аргумент `--enable-vertical-pod autoscaling`:

```
$ gcloud container clusters create k8sforbeginners \
    --num-nodes=2 \
    --zone=us-central1-a \
    --enable-vertical-pod-autoscaling
```

[1] С выходом версии 1.35 функционал переведен в стабильные и флаг --feature-gates не требуется.

После включения в кластере GKE появится CRD-объект VPA, позволяющий управлять процессом вертикального автомасштабирования подов.

В следующем разделе описан способ включения VPA в стандартных Kubernetes-кластерах. Если вы используете иной тип кластера, следуйте инструкциям, предусмотренным для вашей платформы.

Включение VPA в других Kubernetes-кластерах

При использовании таких платформ, как AKS или EKS (а также локальных тестовых развертываний), VPA необходимо устанавливать вручную путем добавления в кластер соответствующего объекта CRD. Актуальные пошаговые инструкции доступны в репозитории GitHub: https://github.com/kubernetes/autoscaler/tree/master/vertical-pod-autoscaler#installation.

Чтобы установить VPA в свой кластер, выполните следующие действия.

1. Клонируйте репозиторий `autoscaler` (https://github.com/kubernetes/autoscaler):

```
$ git clone https://github.com/kubernetes/autoscaler
```

2. Перейдите в каталог компонента VPA:

```
$ cd autoscaler/vertical-pod-autoscaler
```

3. Инициируйте процесс установки с помощью следующей команды (предполагается, что ваш текущий контекст `kubectl` указывает на нужный кластер):

```
$ ./hack/vpa-up.sh
```

4. Эта команда создаст целый набор объектов Kubernetes. Убедитесь, что основные поды VPA успешно запущены:

```
$ kubectl get pods -n kube-system | grep vpa
vpa-admission-controller-5b64b4f4c4-vsn9j   1/1   Running   0   5m34s
vpa-recommender-54c76554b5-m7wnk            1/1   Running   0   5m34s
vpa-updater-7d5f6fbf9b-rkwlb                1/1   Running   0   5m34s
```

Если все компоненты работают, можно переходить к тестированию VPA на реальных подах.

Использование VPA

Для демонстрации нам потребуется объект Deployment с подами, которые оказывают измеримую нагрузку на ЦП. В репозитории Kubernetes `autoscaler` есть простой и удобный пример с *предсказуемым* потреблением ресурсов ЦП: https://github.com/kubernetes/autoscaler/blob/master/vertical-pod-autoscaler/examples/hamster.yaml. Мы слегка модифицируем этот пример и разберем все шаги по порядку.

Предупреждение

Эффективность работы VPA в значительной степени зависит от используемой сборки Kubernetes и степени ее зрелости. В некоторых случаях поды могут не пересоздаваться так, как ожидается, что способно привести к простоям приложения. Поэтому при полной автоматизации работы VPA и отсутствии адекватного мониторинга существует риск возникновения каскадных проблем, связанных с избыточным потреблением ресурсов и нестабильностью кластера.

Для начала подготовим объект Deployment.

1. Первым делом активизируйте сервер метрик для вашего кластера Kubernetes. Вы можете развернуть в своем кластере стандартный сервер метрик (https://github.com/kubernetes-sigs/metrics-server). При использовании кластера `minikube` активизируйте сервер метрик следующим образом:

```
$ minikube addons enable metrics-server
```

2. Создайте новое пространство имен. Для этого подготовьте YAML-манифест:

```
# vpa/vpa-demo-ns.yaml
---
apiVersion: v1
kind: Namespace
metadata:
  name: vpa-demo
  labels:
    project: vpa-demo
```

   Затем примените этот манифест с помощью команды `kubectl apply`:

```
$ kubectl apply -f vpa/vpa-demo-ns.yaml
namespace/vpa-demo created
```

3. Создайте YAML-манифест `hamster-deployment.yaml` (пример можно найти в файле `vpa/hamster-deployment.yaml`):

```
# vpa/hamster-deployment.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: hamster
  namespace: vpa-demo
spec:
  selector:
    matchLabels:
      app: hamster
  replicas: 5
  template:
    metadata:
      labels:
        app: hamster
```

```
spec:
  containers:
    - name: hamster
      image: ubuntu:20.04
      resources:
        requests:
          cpu: 100m
          memory: 50Mi
      command:
        - /bin/sh
        - -c
        - while true; do timeout 0.5s yes >/dev/null; sleep 0.5s; done
```

Команда (`command`), выполняемая внутри контейнера `ubuntu` в составе пода, циклически потребляет максимально доступный объем ресурсов ЦП в течение 0,5 секунды, а следующие 0,5 секунды бездействует. В среднем фактическое потребление ресурсов ЦП будет находиться на уровне около `500m` KCU. При этом в поле `requests` указано, что поду требуется всего `100m` KCU. Это означает, что под будет потреблять больше ресурсов, чем декларирует, но, поскольку лимиты не заданы, Kubernetes не будет ограничивать использование ЦП этим контейнером. Такая ситуация может привести к некорректным решениям планировщика Kubernetes при размещении подов.

4. Примените манифест к кластеру:

```
$ kubectl apply -f vpa/hamster-deployment.yaml
deployment.apps/hamster created
```

5. Отобразите список подов в пространстве имен `vpa-demo`:

```
$ kubectl get po -n vpa-demo
NAME                       READY   STATUS    RESTARTS   AGE
hamster-7fb7dbff7-hmzt5    1/1     Running   0          8s
hamster-7fb7dbff7-lbk9f    1/1     Running   0          8s
hamster-7fb7dbff7-ql6gd    1/1     Running   0          8s
hamster-7fb7dbff7-qmxd8    1/1     Running   0          8s
hamster-7fb7dbff7-qtrpp    1/1     Running   0          8s
```

6. Определите фактическое потребление ресурсов ЦП этими подами с помощью команды `kubectl top`:

```
$ kubectl top pod -n vpa-demo
NAME                       CPU(cores)   MEMORY(bytes)
hamster-7fb7dbff7-hmzt5    457m         0Mi
hamster-7fb7dbff7-lbk9f    489m         0Mi
hamster-7fb7dbff7-ql6gd    459m         0Mi
hamster-7fb7dbff7-qmxd8    453m         0Mi
hamster-7fb7dbff7-qtrpp    451m         0Mi
```

Как и ожидалось, уровень потребления ресурсов ЦП (`CPU`) каждым подом колеблется у отметки `500m`.

Теперь создадим объект VPA для наших подов. VPA может работать в четырех *режимах*, задаваемых в поле `.spec.updatePolicy.updateMode`.

- `Recreate` — запросы и лимиты контейнера назначаются при создании пода и в дальнейшем динамически пересчитываются. Обновление значений требует перезапуска пода, что может приводить к прерыванию работы приложения.
- `Auto` — в текущей реализации эквивалентен режиму `Recreate`, но в будущем, с появлением механизма, обеспечивающего обновления на месте, автоматически переключится на его использование[1].
- `Initial` — запросы и лимиты задаются при создании пода и впоследствии не изменяются.
- `Off` — режим «только рекомендации». Предлагаемые значения запросов и лимитов ресурсов можно посмотреть в объекте VPA, например, с помощью утилиты `kubectl`.

Сначала создадим VPA для объекта Deployment `hamster`, который будет работать в режиме `Off`, а затем включим режим `Auto`. Для этого сделайте следующее.

1. Создайте YAML-манифест VPA `vpa/hamster-vpa.yaml`:

```
# vpa/hamster-vpa.yaml
apiVersion: autoscaling.k8s.io/v1
kind: VerticalPodAutoscaler
metadata:
  name: hamster-vpa
  namespace: vpa-demo
spec:
  targetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: hamster
  updatePolicy:
    updateMode: 'Off'
  resourcePolicy:
    containerPolicies:
      - containerName: '*'
        minAllowed:
          cpu: 100m
          memory: 50Mi
        maxAllowed:
          cpu: 1
          memory: 500Mi
        controlledResources:
          - cpu
          - memory
```

[1] Развитие Kubernetes динамично, этот режим уже признан устаревшим и скоро будет удален, на смену ему приходит InPlaceOrRecreate.

Этот VPA создан для объекта Deployment `hamster`, как указано в поле `.spec.targetRef`. Режим работы `'Off'` задан в поле `.spec.updatePolicy.updateMode` (значение `'Off'` необходимо заключать в кавычки, чтобы оно не было интерпретировано как логическое). Политика для контейнера настроена в поле `.spec.resourcePolicy.containerPolicies` и допускает автоматическую корректировку запросов: для ресурсов ЦП — в диапазоне от `100m` до `1000m`, а для памяти — от `50Mi` до `500Mi`.

2. Примените манифест к кластеру:

```
$ kubectl apply -f vpa/hamster-vpa.yaml
verticalpodautoscaler.autoscaling.k8s.io/hamster-vpa created
```

3. Подождите, пока будет рассчитано первое рекомендуемое значение. После этого проверьте его, описав объект VPA:

```
$ kubectl describe vpa hamster-vpa -n vpa-demo
...<фрагмент опущен для краткости>...
Status:
  Conditions:
    Last Transition Time:  2024-08-11T09:20:44Z
    Status:                True
    Type:                  RecommendationProvided
  Recommendation:
    Container Recommendations:
      Container Name:  hamster
      Lower Bound:
        Cpu:     461m
        Memory:  262144k
      Target:
        Cpu:     587m
        Memory:  262144k
      Uncapped Target:
        Cpu:     587m
        Memory:  262144k
      Upper Bound:
        Cpu:     1
        Memory:  500Mi
Events:          <none>
```

VPA порекомендовал выделить чуть больше ожидаемых `500m` KCU и `262144k` памяти. Это логично, так как ресурсы ЦП требуются поду с небольшим запасом.

4. Теперь проверим VPA в действии и переключим его режим на `Auto`. Для этого измените файл `vpa/hamster-vpa.yaml` следующим образом:

```
# vpa/hamster-vpa.yaml
apiVersion: autoscaling.k8s.io/v1
kind: VerticalPodAutoscaler
```

```
metadata:
  name: hamster-vpa
  namespace: vpa-demo
spec:
...
  updatePolicy:
    updateMode: Auto
...
```

5. Примените обновленный манифест к кластеру:

```
$ kubectl apply -f vpa/hamster-vpa.yaml
verticalpodautoscaler.autoscaling.k8s.io/hamster-vpa configured
```

6. Через некоторое время вы заметите, как VPA перезапускает поды объекта Deployment:

```
$ kubectl get po -n vpa-demo -w
NAME                      READY   STATUS              RESTARTS   AGE
hamster-7fb7dbff7-24p89   0/1     ContainerCreating   0          2s
hamster-7fb7dbff7-6nz8f   0/1     ContainerCreating   0          2s
hamster-7fb7dbff7-hmzt5   1/1     Running             0          20m
hamster-7fb7dbff7-lbk9f   1/1     Running             0          20m
hamster-7fb7dbff7-ql6gd   1/1     Terminating         0          20m
hamster-7fb7dbff7-qmxd8   1/1     Terminating         0          20m
hamster-7fb7dbff7-qtrpp   1/1     Running             0          20m
hamster-7fb7dbff7-24p89   1/1     Running             0          2s
hamster-7fb7dbff7-6nz8f   1/1     Running             0          2s
```

7. Мы можем описать один из перезапущенных подов, чтобы увидеть текущие значения запросов ресурсов:

```
$ kubectl describe pod hamster-7fb7dbff7-24p89 -n vpa-demo
...
Annotations:
  ...<фрагмент опущен для краткости>...
  vpaObservedContainers: hamster
  vpaUpdates: Pod resources updated by hamster-vpa:
    container 0: memory request, cpu request
...
Containers:
  hamster:
    ...
    Requests:
      cpu:     587m
      memory:  262144k
...<фрагмент опущен для краткости>...
```

Как видите, запросы ресурсов ЦП и памяти у нового пода заданы в соответствии с рекомендациями VPA.

VPA не рекомендуется использовать с HPA, ориентирующимся на метрики потребления ресурсов ЦП и памяти. Однако вы можете комбинировать VPA с HPA, использующим пользовательские метрики.

Далее мы поговорим о горизонтальном автомасштабировании подов.

Горизонтальное автомасштабирование подов с помощью объекта HorizontalPodAutoscaler

Если вертикальный автоскейлер (VPA) выступает в роли оптимизатора использования ресурсов, то за реальное масштабирование объектов Deployment и StatefulSet, работающих с несколькими репликами подов, отвечает горизонтальный автоскейлер (HPA). Задача HPA заключается в том, чтобы автоматически изменять количество реплик в объекте Deployment или StatefulSet в зависимости от текущего уровня потребления ресурсов ЦП или других пользовательских метрик. С подробным описанием алгоритма, определяющего требуемое количество реплик на основе метрик, можно ознакомиться здесь: https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/#algorithm-details.

Не все приложения одинаково эффективно работают с HPA и VPA. Одни лучше масштабируются, другие могут вовсе не поддерживать автомасштабирование или даже деградировать при его использовании. Перед применением любого подхода тщательно проанализируйте поведение своего приложения.

Следующая схема иллюстрирует разницу между вертикальным и горизонтальным масштабированием (рис. 20.2).

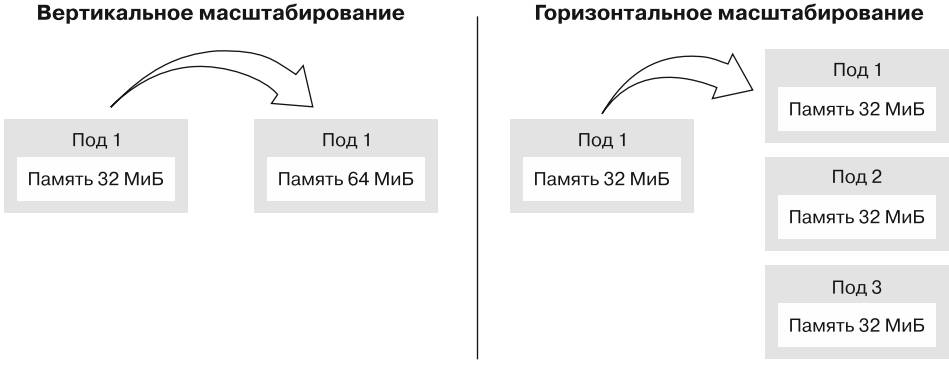

Рис. 20.2. Вертикальное и горизонтальное масштабирование подов

HPA предоставляет широкие возможности для настройки. В этой главе мы рассмотрим стандартный сценарий автомасштабирования на основе целевого уровня потребления ресурсов ЦП.

HPA является API-ресурсом из группы autoscaling. Текущая стабильная версия autoscaling/v2 поддерживает масштабирование на основе уровня потребления памяти и пользовательских метрик. При использовании версии autoscaling/v1 новые поля, появившиеся в autoscaling/v2, сохраняются в виде аннотаций.

Роль HPA состоит в том, чтобы отслеживать выбранную метрику подов, например уровень загрузки ЦП, и оценивать целесообразность изменения количества реплик. Обычно HPA вычисляет среднее значение метрики для всех подов и определяет, позволит ли добавление или удаление реплик приблизиться к целевому значению. Допустим, целевая загрузка ЦП задана на уровне 50 %. В какой-то момент из-за возросшей нагрузки поды объекта Deployment начинают потреблять 80 % ресурсов ЦП. HPA решает увеличить число реплик подов, чтобы приблизить средний показатель загрузки к заданным 50 %. Затем цикл повторяется. Таким образом, HPA стремится поддерживать средний уровень загрузки ЦП как можно ближе к целевому. Это напоминает работу контроллера с обратной связью, хорошим примером которого служит термостат, реагирующий на изменение температуры в помещении.

Компоненты HPA в Kubernetes показаны на рис. 20.3.

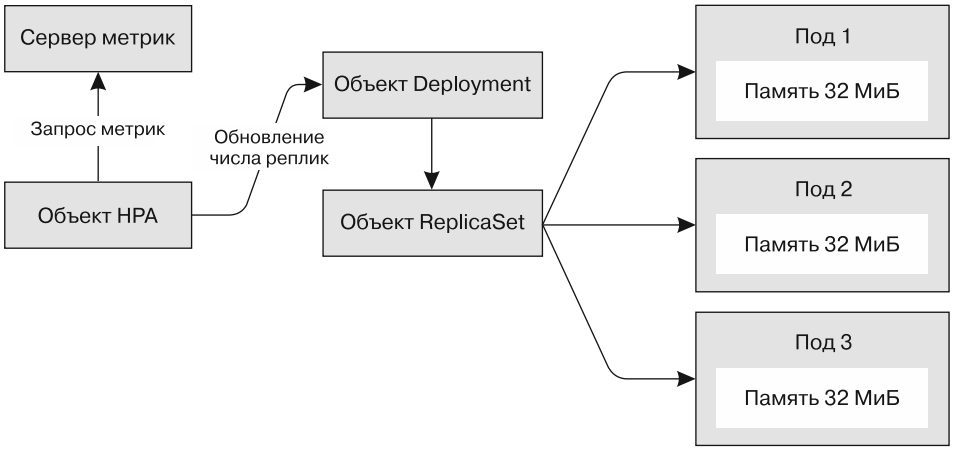

Рис. 20.3. Компоненты HPA в Kubernetes

Помимо всего прочего, HPA использует такие механизмы, как *окно стабилизации*, чтобы предотвратить слишком быстрое изменение количества реплик, то есть нежелательные резкие колебания их числа (replica flapping).

В GKE доступна функция многомерного автомасштабирования подов, находящаяся на стадии бета-тестирования. Она сочетает горизонтальное масштабирование на основе уровня потребления ресурсов ЦП и вертикальное масштабирование на основе уровня потребления памяти. Подробнее о ней можно прочитать в официальной документации: https://cloud.google.com/kubernetes-engine/docs/how-to/multidimensional-pod-autoscaling. Учтите, что данная функция подпадает под условия Pre-GA Offerings Terms, описанные в документе General Service Terms, и предоставляется «как есть» с ограниченной поддержкой. Подробности можно найти в описании стадий запуска.

Поскольку HPA является встроенным механизмом Kubernetes, вам не нужно ничего устанавливать. Достаточно подготовить тестовый объект Deployment и создать ресурс HorizontalPodAutoscaler.

Развертывание приложения для демонстрации принципа работы HPA

Для тестирования HPA мы будем использовать стандартную метрику загрузки ЦП. Это значит, что нам необходимо указать соответствующие запросы (`requests`) для подов объекта Deployment — без этого абсолютного значения расчет процентной метрики и автомасштабирование будут невозможны. В данном случае нам также требуется объект Deployment с предсказуемым уровнем потребления ресурсов ЦП. Разумеется, в реальной практике изменение загрузки ЦП будет вызвано фактическим спросом со стороны конечных пользователей приложения.

Первым делом активизируйте сервер метрик для вашего кластера Kubernetes. Вы можете развернуть в своем кластере стандартный сервер метрик (https://github.com/kubernetes-sigs/metrics-server). При использовании кластера `minikube` активизируйте сервер метрик следующим образом:

```
$ minikube addons enable metrics-server
```

Для реализации HPA выполните описанные ниже шаги.

1. Чтобы изолировать ресурсы, создайте новое пространство имен. Для этого подготовьте YAML-манифест со следующим содержимым:

```
# hpa/hpa-demo-ns.yaml
---
apiVersion: v1
kind: Namespace
metadata:
  name: hpa-demo
  labels:
    project: hpa-demo
```

2. Примените этот манифест, чтобы создать пространство имен:

```
$ kubectl apply -f hpa/hpa-demo-ns.yaml
namespace/hpa-demo created
```

3. Для демонстрации мы используем простой контейнер с веб-сервером на основе пользовательского образа `quay.io/iamgini/one-page-web:1.0`. Следующий YAML-манифест содержит определение объекта Deployment, создающего одну реплику пода:

```
---
# hpa/todo-deployment.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: todo-app
  namespace: hpa-demo
spec:
  replicas: 1  # Скорректируйте при необходимости
  selector:
    matchLabels:
      app: todo
  template:
    metadata:
      labels:
        app: todo
    spec:
      containers:
        - name: todoapp
          image: quay.io/ginigangadharan/todo-app:2.0
          ports:
            - containerPort: 3000
              resources:
                requests:
                  memory: "50Mi"   # Запрашиваем 50 МиБ памяти
                  cpu: "50m"       # Запрашиваем 0,05 ядра ЦП
                limits:
                  memory: "100Mi"  # Максимум 100 МиБ памяти
                  cpu: "100m"      # Максимум 0,1 ядра ЦП
```

4. Примените конфигурацию и убедитесь в том, что поды запущены:

```
$ kubectl apply -f hpa/todo-deployment.yaml
deployment.apps/todo-app created

$ kubectl get po -n hpa-demo
NAME                        READY   STATUS    RESTARTS   AGE
todo-app-5cfb496d77-l6r69   1/1     Running   0          8s
```

5. Чтобы открыть доступ к приложению, создайте объект Service с помощью следующего YAML-манифеста:

```
# hpa/todo-service.yaml
apiVersion: v1
kind: Service
```

```
metadata:
  name: todo-app
  namespace: hpa-demo
spec:
  type: ClusterIP
  selector:
    app: todo
  ports:
    - port: 8081       # Порт, доступный внутри кластера
      targetPort: 3000 # containerPort в подах
```

6. Примените конфигурацию и выведите сведения о созданном сервисе:

```
$ kubectl apply -f hpa/todo-service.yaml
service/todo-app created

$ kubectl get svc -n hpa-demo
NAME       TYPE        CLUSTER-IP     EXTERNAL-IP   PORT(S)     AGE
todo-app   ClusterIP   10.96.171.71   <none>        8081/TCP    15s
```

7. Теперь приложение работает и доступно через сервис типа `ClusterIP`. Давайте выполним команду `kubectl port-forward`, чтобы получить доступ к приложению извне кластера:

```
$ kubectl port-forward svc/todo-app -n hpa-demo 8081:8081
Forwarding from 127.0.0.1:8081 -> 3000
Forwarding from [::1]:8081 -> 3000
```

8. Откройте браузер и перейдите по адресу `http://localhost:8081`. Вы увидите, что приложение Todo запущено (рис. 20.4).
9. Нажмите Ctrl+C в консоли, чтобы остановить переадресацию портов.

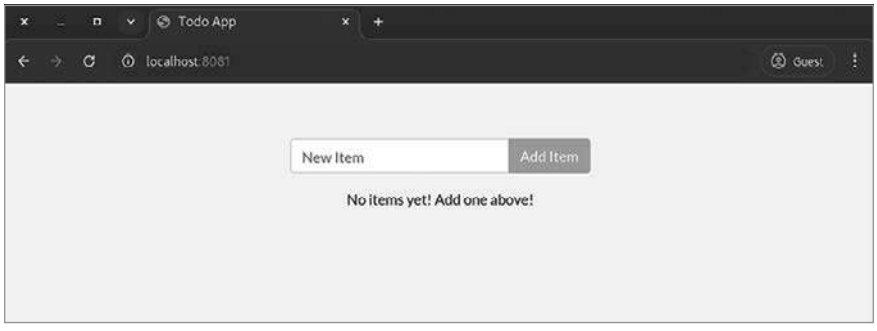

Рис. 20.4. Приложение Todo запущено в Kubernetes

Теперь, когда объект Deployment приложения Todo работает в кластере, мы можем приступить к изучению горизонтального автоскейлера. В следующем подразделе мы создадим объект HPA и нагрузим приложение, чтобы увидеть автомасштабирование в действии.

Реализация HPA

Вы уже знаете, что количество подов можно изменить вручную, используя команду `kubectl scale` (например: `kubectl scale deployment one-page-web -n hpa-demo --replicas 3`). Но сейчас нам важно понять, как HPA осуществляет автоматизированное масштабирование в зависимости от нагрузки.

Как было сказано выше, HPA инициирует масштабирование на основе метрик, поэтому нам необходимо создать нагрузку на поды. Существует несколько инструментов, позволяющих сгенерировать искусственную нагрузку на веб-приложение для целей нагрузочного и стресс-тестирования. В ходе этой демонстрации мы воспользуемся легковесным инструментом `hey`, написанным на языке Go. Он позволяет с легкостью генерировать трафик к веб-приложениям с целью измерения их производительности под нагрузкой и упрощает проведение сравнительного анализа, позволяя быстро отправлять множество запросов и оценивать такие показатели, как время отклика и пропускная способность.

Нагрузку можно увеличить и другими способами. Например, вы можете запустить дополнительный контейнер, который будет обращаться к подам приложения, выполнив команду:

```
$ kubectl run -n hpa-demo -i --tty load-generator --rm \
  --image=busybox:1.28 \
  --restart=Never -- \
  /bin/sh -c "while sleep 0.01; do wget -q -O- http://todo-app; done"
```

Однако такой метод менее эффективен в плане точного контроля нагрузки.

Приложение `hey` доступно для Linux, macOS и Windows (https://github.com/rakyll/hey), а его установка не требует особых усилий.

1. Скачайте пакет `hey` для вашей операционной системы.
2. Сделайте файл исполняемым и добавьте его в путь поиска исполняемых файлов (например, `ln -s ~/Downloads/ hey_linux_amd64 ~/.local/bin/`).

Теперь создадим ресурс HPA для масштабирования объекта Deployment в зависимости от уровня нагрузки.

1. Подготовьте YAML-манифест со следующим содержимым:

```
# hpa/todo-hpa.yaml
apiVersion: autoscaling/v2
kind: HorizontalPodAutoscaler
```

```
metadata:
  name: todo-hpa
  namespace: hpa-demo
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: todo-app
  minReplicas: 1
  maxReplicas: 5
  metrics:
    - type: Resource
      resource:
        name: cpu
        target:
          type: Utilization
          averageUtilization: 80
```

2. Примените конфигурацию, чтобы создать объект HPA:

```
$ kubectl apply -f hpa/todo-hpa.yaml
horizontalpodautoscaler.autoscaling/todo-hpa created

$ kubectl get hpa -n hpa-demo
NAME     REFERENCE             TARGETS             MINPODS MAXPODS REPLICAS AGE
todo-hpa Deployment/todo-app   cpu: <unknown>/80% 1       5       0        6s
```

3. Выполните команду `kubectl port-forward`, чтобы получить доступ к приложению извне:

```
$ kubectl port-forward svc/todo-app -n hpa-demo 8081:8081
Forwarding from 127.0.0.1:8081 -> 3000
Forwarding from [::1]:8081 -> 3000
```

4. Сейчас приложение Todo никто не использует, поэтому количество реплик остается равным 1. Давайте смоделируем нагрузку с помощью утилиты `hey`. В отдельном окне консоли выполните команду:

```
$ hey -z 4m -c 25 http://localhost:8081
```

Пояснение к параметрам команды:

- `-z 4m` — тест длится четыре минуты, что позволяет дольше поддерживать повышенный уровень нагрузки;
- `-c 25` — 25 конкурентных соединений призваны приблизить уровень загрузки ЦП к 80 %;
- `http://localhost:8081` — URL-адрес приложения Todo (доступного через `kubectl port-forward`).

По мере того как инструмент `hey` создает нагрузку на наше одностраничное веб-приложение, в окне `kubectl port-forward` будут появляться многочисленные записи о подключениях.

5. Откройте третье окно консоли (не дожидаясь завершения работы `hey`) и проверьте текущий уровень потребления ресурсов подами:

```
$ watch 'kubectl get po -n hpa-demo; kubectl top pods -n hpa-demo'
Every 2.0s: kubectl get po -n hpa-demo; kubectl top pods -n hpa-demo

NAME                         READY   STATUS    RESTARTS   AGE
todo-app-5cfb496d77-5kc27    1/1     Running   0          76s
todo-app-5cfb496d77-l6r69    1/1     Running   0          10m
todo-app-5cfb496d77-pb7tx    1/1     Running   0          76s
NAME                         CPU(cores)   MEMORY(bytes)
todo-app-5cfb496d77-5kc27    10m          14Mi
todo-app-5cfb496d77-l6r69    100m         48Mi
todo-app-5cfb496d77-pb7tx    7m           14Mi
```

Вы увидите возросшее количество подов, поскольку увеличенная нагрузка на приложение Todo заставила HPA создать дополнительные реплики.

6. Выведите сведения об объекте Deployment, чтобы просмотреть события масштабирования:

```
$ kubectl describe deployments.apps todo-app -n hpa-demo
Name:                   todo-app
... <фрагмент опущен для краткости> ...
Replicas:               3 desired | 3 updated | 3 total | 3 available | 0
unavailable
StrategyType:           RollingUpdate
... <фрагмент опущен для краткости> ...
Events:
  Type    Reason             Age    From                   Message
  ----    ------             ----   ----                   -------
  Normal  ScalingReplicaSet  16m    deployment-controller  Scaled up
replica set todo-app-749854577d to 1
  Normal  ScalingReplicaSet  13m    deployment-controller  Scaled up
replica set todo-app-5cfb496d77 to 1
  Normal  ScalingReplicaSet  13m    deployment-controller  Scaled down
replica set todo-app-749854577d to 0 from 1
  Normal  ScalingReplicaSet  4m9s   deployment-controller  Scaled up
replica set todo-app-5cfb496d77 to 3 from 1
```

Итак, вы успешно настроили горизонтальное автомасштабирование объекта Deployment с помощью HPA. Удалите созданные ресурсы, удалив пространство имен `hpa-demo` (например, с помощью команды `kubectl delete namespaces hpa-demo`). В следующем разделе мы обсудим автомасштабирование узлов Kubernetes с помощью кластерного автоскейлера, который обеспечивает еще большую гибкость при совместном использовании с HPA.

Автомасштабирование узлов Kubernetes с помощью кластерного автоскейлера

До этого мы говорили о масштабировании рабочих нагрузок в Kubernetes на уровне отдельных подов, однако это не единственный способ. Мы можем масштабировать и сам кластер, адаптируя его к изменяющемуся спросу на вычислительные ресурсы. В какой-то момент для запуска дополнительных подов нам потребуется увеличить число узлов. Вы можете задать фиксированное количество узлов и управлять емкостью кластера вручную. Такой подход остается применимым даже при автоматизации процессов создания, обслуживания и удаления узлов.

Задачу автомасштабирования кластера решает *кластерный автоскейлер* (Cluster Autoscaler, CA) из репозитория Kubernetes `autoscaler`: https://github.com/kubernetes/autoscaler/tree/master/cluster-autoscaler. Поскольку CA должен иметь возможность добавлять и удалять узлы в кластере Kubernetes, для его использования требуются плагины, специфичные для конкретного облачного провайдера. Перечень поддерживаемых облаков приведен здесь: https://github.com/kubernetes/autoscaler/tree/master/cluster-autoscaler#deployment.

Кластерный автоскейлер периодически проверяет состояние подов и узлов и в зависимости от ситуации выполняет те или иные действия.

- При наличии подов, которые не могут быть запланированы и находятся в состоянии ожидания (`Pending`) из-за нехватки ресурсов в кластере, CA добавляет новые узлы — вплоть до заданного максимального значения.
- Если узлы используются недостаточно эффективно и все поды можно разместить на меньшем количестве узлов, CA удаляет узлы, если только кластер не достиг заранее заданного минимального размера. Перед удалением узлы корректно освобождаются от подов.
- Некоторые облачные провайдеры позволяют кластерному автоскейлеру выбирать различные типы виртуальных машин (SKU), чтобы оптимизировать затраты на эксплуатацию кластера.

Чтобы CA работал корректно, для контейнеров подов должны быть указаны запросы (requests) на вычислительные ресурсы. Эти значения должны отражать реальные потребности приложения, в противном случае CA не сможет принимать верные решения относительно вашего типа рабочей нагрузки.

Кластерный автоскейлер дополняет возможности HPA. Если HPA определяет, что объекту Deployment или StatefulSet требуется больше подов, но новые

поды не могут быть запланированы, CA может вмешаться и увеличить размер кластера.

Прежде чем рассматривать принцип работы кластерного автоскейлера, давайте обсудим некоторые его ограничения.

Ограничения кластерного автоскейлера

У CA есть несколько ограничений, способных повлиять на его эффективность.

- Между моментом, когда CA запрашивает новый узел у облачного провайдера, и моментом, когда этот узел становится доступным, есть задержка. Она нередко длится несколько минут, что может отрицательно сказаться на производительности приложения в периоды высокой нагрузки.
- Решения CA о масштабировании основаны исключительно на значениях `requests` и `limits`, заданных для подов, а не на фактическом уровне потребления ресурсов ЦП или памяти. Если поды запрашивают слишком большие объемы ресурсов, это может привести к снижению эффективности их использования и недозагрузке узлов.
- CA разработан в первую очередь для облачных сред. Хотя его можно адаптировать для локальной инфраструктуры и других типов окружений, это требует дополнительных усилий, в том числе написания пользовательских сценариев или использования инструментов для управления добавлением и удалением узлов, а также настройки самого автоскейлера для взаимодействия с этими механизмами. Без облачных функций автомасштабирования управление размером кластера становится более сложным и требует тщательного мониторинга.

Процесс включения кластерного автоскейлера зависит от используемого облачного провайдера. Кроме того, некоторые параметры конфигурации специфичны для конкретных платформ. В следующем разделе мы рассмотрим процесс активизации CA в среде GKE.

Предупреждение, касающееся потребления ресурсов

Будьте крайне внимательны при изменении конфигурации кластерного автоскейлера, поскольку многие варианты настройки могут привести к чрезмерному потреблению ресурсов и вызвать нестабильность системы или ее непредсказуемое поведение при масштабировании. Всегда отслеживайте состояние кластера и корректируйте параметры конфигурации, чтобы избежать истощения ресурсов или ухудшения производительности.

Включение кластерного автоскейлера в GKE

В GKE проще всего создать новый кластер с уже включенным CA. Для этого выполните следующую команду:

```
$ gcloud container clusters create k8sbible \
    --enable-autoscaling \
    --num-nodes 3 \
    --min-nodes 2 \
    --max-nodes 10 \
    --region=us-central1-a
...<фрагмент опущен для краткости>...
Creating cluster k8sbible in us-central1-a... Cluster is being health-checked
(master is healthy)...done.
Created [https://container.googleapis.com/v1/projects/k8sbible-project/zones/
us-central1-a/clusters/k8sbible].

To inspect the contents of your cluster, go to:
https://console.cloud.google.com/kubernetes/workload_/gcloud/us-central1-a/
k8sbible?project=k8sbible-project

kubeconfig entry generated for k8sbible.
NAME      LOCATION       MASTER_VERSION           MASTER_IP     MACHINE_TYPE
NODE_VERSION           NUM_NODES   STATUS
k8sbible  us-central1-a  1.29.7-gke.1008000       <removed>     e2-medium
1.29.7-gke.1008000     3           RUNNING
```

Пояснения к параметрам команды:

- `cloud container clusters create k8sbible` — создает новый кластер Kubernetes с именем `k8sbible`;
- `--enable-autoscaling` — активизирует функцию автомасштабирования для пулов узлов кластера;
- `--num-nodes 3` — задает начальное количество узлов равным `3`;
- `--min-nodes 2` — задает минимальное число узлов равным `2`;
- `--max-nodes 10` — задает максимальное число узлов равным `10`;
- `--region=us-central1-a` — указывает регион `us-central1-a`.

Перед началом работы убедитесь, что ваша учетная запись GCP правильно настроена и имеет необходимые разрешения, включая доступ к Virtual Private Cloud (VPC), сетям, средствам безопасности и другим нужным компонентам.

Если кластер уже существует, необходимо активизировать функцию автомасштабирования для существующего пула узлов. Например, для кластера `k8sforbeginners` с пулом узлов `nodepool1` выполните команду:

```
$ gcloud container clusters update k8sforbeginners \
    --enable-autoscaling \
    --min-nodes=2 \
    --max-nodes=10 \
    --zone=us-central1-a \
    --node-pool=nodepool1
```

Обновление займет несколько минут.

Проверьте состояние функции автомасштабирования с помощью интерфейса командной строки `gcloud`:

```
$ gcloud container node-pools describe default-pool --cluster=k8sdemo | grep
autoscaling -A 1
autoscaling:
  enabled: true
```

Подробнее об автомасштабировании в GKE можно узнать в официальной документации: https://cloud.google.com/kubernetes-engine/docs/concepts/cluster-autoscaler.

После завершения настройки вы можете перейти к подразделу «Использование кластерного автоскейлера».

Включение кластерного автоскейлера в Amazon EKS

Настройка кластерного автоскейлера (CA) в Amazon EKS в настоящее время не может быть выполнена с помощью одной команды или «в один клик». Необходимо создать соответствующую IAM-политику и роль, развернуть ресурсы CA в Kubernetes-кластере и задать несколько параметров вручную. По этой причине мы не будем рассматривать процесс настройки CA в EKS в этой книге и предлагаем вам обратиться к официальной инструкции: https://docs.aws.amazon.com/eks/latest/userguide/cluster-autoscaler.html.

После завершения настройки вы можете перейти к подразделу «Использование кластерного автоскейлера».

Включение кластерного автоскейлера в AKS

В AKS кластерный автоскейлер настраивается примерно так же, как в GKE. Вы можете либо создать новый кластер с уже включенным CA с помощью одной команды, либо активизировать этот механизм в существующем кластере. Чтобы создать новый кластер `k8sforbeginners-aks` в группе ресурсов `k8sforbeginners-rg`, выполните следующую команду:

```
$ az aks create --resource-group k8sforbeginners-rg \
    --name k8sforbeginners-aks \
    --node-count 1 \
    --enable-cluster-autoscaler \
    --min-count 1 \
    --max-count 10 \
    --vm-set-type VirtualMachineScaleSets \
    --load-balancer-sku standard \
    --generate-ssh-keys
```

При создании нового кластера можно задать минимальное число узлов через параметр `--min-count`, а максимальное — через `--max-count`.

Чтобы активизировать кластерный автоскейлер в уже существующем кластере AKS `k8sforbeginners-aks`, выполните команду:

```
$ az aks update \
    --resource-group k8sforbeginners-rg \
    --name k8sforbeginners-aks \
    --enable-cluster-autoscaler \
    --min-count 2 \
    --max-count 10
```

Обновление займет несколько минут.

Более подробную информацию можно найти в официальной документации: https://docs.microsoft.com/en-us/azure/aks/cluster-autoscaler. Кроме того, CA в AKS имеет дополнительные параметры, настраиваемые через *профиль автоскейлера*. Их описание приведено здесь: https://docs.microsoft.com/en-us/azure/aks/cluster-autoscaler#using-the-autoscaler-profile.

Далее мы поговорим об использовании кластерного автоскейлера в Kubernetes.

Использование кластерного автоскейлера

Между завершением настройки кластерного автоскейлера (CA) и началом его работы может пройти некоторое время. Продолжительность этой задержки зависит от конфигурации CA, которая может различаться у разных провайдеров. Например, CA в AKS проверяет состояние кластера каждые 10 секунд (параметр `scan-interval`), чтобы определить целесообразность изменения числа узлов. Если после увеличения этого числа возникает необходимость его уменьшить, действует задержка 10 минут (параметр `scale-down-delay-after-add`). Уменьшение числа узлов происходит в том случае, если отношение общего объема запрошенных ресурсов к общей емкости узлов становится меньше 0,5 (параметр `scale-down-utilization-threshold`).

Таким образом, после включения CA число узлов в кластере может автоматически увеличиться, уменьшиться или остаться без изменений.

В качестве демонстрации мы используем кластер GKE, состоящий из двух узлов:

```
$ kubectl get nodes -o custom-columns=NAME:.metadata.name,CPU_ALLOCATABLE:.
status.allocatable.cpu,MEMORY_ALLOCATABLE:.status.allocatable.memory
NAME                                       CPU_ALLOCATABLE   MEMORY_ALLOCATABLE
gke-k8sdemo-default-pool-1bf4f185-6422  940m              2873304Ki
gke-k8sdemo-default-pool-1bf4f185-csv0  940m              2873312Ki
```

- Этому кластеру доступно примерно 1,88 ядра ЦП и 5611,34 МиБ памяти.
- Помните, что часть ресурсов ЦП потребляется подами из пространства имен `kube-system`.
- Точный объем доступных ресурсов ЦП и памяти можно определить с помощью команды `kubectl top nodes`.

К сожалению, добиться предсказуемой и изменяющейся нагрузки на ЦП внутри контейнера встроенными средствами невозможно. Поэтому для демонстрации нам необходимо подготовить объект Deployment с шаблоном пода, который позволит смоделировать такую ситуацию. Мы создадим объект Deployment `elastic-hamster` (см. каталог `Chapter20/ca` в репозитории GitHub). Shell-сценарий `hamster.sh`, непрерывно выполняющийся в контейнере, будет увеличивать нагрузку на основе значения `TOTAL_HAMSTER_USAGE`. С помощью этого значения мы зададим общий объем работы, который должны выполнить все генераторы нагрузки во всех подах. Каждый под будет обращаться к API Kubernetes, чтобы определить текущее число запущенных реплик объекта Deployment. Затем общий объем необходимой работы будет поделен на количество реплик для вычисления нагрузки, приходящейся на каждый генератор.

Например, если задать общий объем работы для всех генераторов нагрузки равным 1,0 (что соответствует суммарному потреблению ресурсов ЦП в кластере) и развернуть пять реплик, то каждый генератор будет выполнять объем работы равный 1,0 / 5 = 0,2 (0,2 секунды активной работы и 0,8 секунды ожидания). Если изменить число реплик в объекте Deployment до десяти, то каждый генератор нагрузки будет работать в течение 0,1 секунды и 0,9 секунды находиться в состоянии ожидания. Таким образом, совокупный объем работы всегда остается равным 1,0, независимо от количества реплик. Такое поведение моделирует реальную ситуацию, когда конечные пользователи генерируют трафик, который необходимо обрабатывать, а нагрузка распределяется между репликами подов. Чем больше реплик подов запущено, тем меньшую долю нагрузки приходится обрабатывать каждой из них, что приводит к снижению среднего уровня потребления ресурсов ЦП одним подом.

Для увеличения нагрузки вы можете применять любые знакомые вам инструменты и методы. Тем не менее, чтобы не усложнять пример, мы используем упрощенный подход для демонстрации роста нагрузки и процесса масштабирования.

Чтобы реализовать автомасштабирование кластера, выполните следующие шаги.

1. Для изоляции эксперимента создайте пространство имен `ca-demo`:

```
# ca/ca-demo-ns.yaml
---
apiVersion: v1
kind: Namespace
metadata:
  name: ca-demo
  labels:
    project: ca-demo
```

2. Чтобы поды могли обращаться к объектам Deployment через API Kubernetes, необходимо назначить дополнительные разрешения RBAC. Подробные сведения о механизме RBAC приведены в главе 18. Подготовьте определение объекта `Role`:

```
# ca/deployment-reader-role.yaml
---
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  namespace: ca-demo
  name: deployment-reader
rules:
  - apiGroups: ["apps"]
    resources: ["deployments"]
    verbs: ["get", "watch", "list"]
```

3. Создайте объект ServiceAccount, который будут использовать поды генераторов нагрузки:

```
# ca/elastic-hamster-serviceaccount.yaml
apiVersion: v1
kind: ServiceAccount
metadata:
  name: elastic-hamster
  namespace: ca-demo
```

4. Подготовьте YAML-манифест для объекта RoleBinding:

```
# ca/read-deployments-rolebinding.yaml
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: read-deployments
  namespace: ca-demo
subjects:
  - kind: ServiceAccount
    name: elastic-hamster
    namespace: default
roleRef:
  kind: Role
  name: deployment-reader
  apiGroup: rbac.authorization.k8s.io
```

5. Создайте объект Deployment. Он достаточно прост, но использует специальный образ контейнера (см. файл `ca/elastic-hamster-deployment.yaml`):

```
...
    spec:
      serviceAccountName: elastic-hamster
      containers:
        - name: hamster
          image: quay.io/iamgini/elastic-hamster:1.0
          resources:
            requests:
              cpu: 500m
              memory: 50Mi
          env:
            - name: TOTAL_HAMSTER_USAGE
              value: "1.0"
```

   Мы подготовили пользовательский контейнерный образ `elastic-hamster`, внутри которого находится сценарий `hamster.sh` (см. файлы `ca/Dockerfile` и `ca/hamster.sh` в каталоге `Chapter20`).

6. Создайте объект HPA, который будет выполнять автомасштабирование подов:

```
# elastic-hamster-hpa.yaml
apiVersion: autoscaling/v1
kind: HorizontalPodAutoscaler
metadata:
  name: elastic-hamster-hpa
  namespace: ca-demo
spec:
  minReplicas: 1
  maxReplicas: 25
  metrics:
    - resource:
        name: cpu
        target:
          averageUtilization: 75
          type: Utilization
      type: Resource
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: elastic-hamster
```

7. Чтобы не использовать YAML-файлы по отдельности, примените сразу все манифесты из каталога `ca` следующим образом:

```
$ kubectl apply -f ca/
namespace/ca-demo created
role.rbac.authorization.k8s.io/deployment-reader created
deployment.apps/elastic-hamster created
```

```
horizontalpodautoscaler.autoscaling/elastic-hamster-hpa created
serviceaccount/elastic-hamster created
rolebinding.rbac.authorization.k8s.io/read-deployments created
```

В YAML-манифесте HPA указан параметр `maxReplicas: 25`. Согласно логике сценария HPA попытается запланировать 25 подов с запросом `cpu: 500m`. Поскольку в кластере недостаточно ресурсов, кластерный автоскейлер начнет увеличивать число узлов.

8. Если вы проверите список подов, то увидите, что часть из них находится в состоянии ожидания (`Pending`) из-за нехватки ресурсов:

```
$ kubectl get po -n ca-demo
NAME                                READY   STATUS    RESTARTS   AGE
elastic-hamster-87d4db7fd-4tmxn     0/1     Pending   0          7m20s
elastic-hamster-87d4db7fd-59lcd     1/1     Running   0          8m4s
elastic-hamster-87d4db7fd-5d2gf     0/1     Pending   0          7m20s
elastic-hamster-87d4db7fd-5m27q     0/1     Pending   0          8m4s
elastic-hamster-87d4db7fd-7nc48     0/1     Pending   0          7m19s
...<фрагмент опущен для краткости>...
elastic-hamster-87d4db7fd-st7r5     0/1     Pending   0          7m34s
elastic-hamster-87d4db7fd-twb86     1/1     Running   0          8m48s
elastic-hamster-87d4db7fd-xrppp     0/1     Pending   0          7m34s
```

9. Проверив список узлов, вы увидите, что в кластере их всего десять (это максимальное количество, заданное параметром `--max-nodes 10`):

```
$ kubectl top pod -n ca-demo
NAME                                      CPU(cores)   CPU%   MEMORY(bytes)
MEMORY%
gke-k8sdemo-default-pool-1bf4f185-6422    196m         20%    1220Mi
43%
gke-k8sdemo-default-pool-1bf4f185-csv0    199m         21%    1139Mi
40%
gke-k8sdemo-default-pool-1bf4f185-fcsd    751m         79%    935Mi
33%
gke-k8sdemo-default-pool-1bf4f185-frq6    731m         77%    879Mi
31%
gke-k8sdemo-default-pool-1bf4f185-h8hw    742m         78%    846Mi
30%
gke-k8sdemo-default-pool-1bf4f185-j99r    733m         77%    923Mi
32%
gke-k8sdemo-default-pool-1bf4f185-k6xq    741m         78%    986Mi
35%
...<фрагмент опущен для краткости>...
```

Этот пример наглядно показал, как кластерный автоскейлер (CA) и горизонтальный автоскейлер подов (HPA) совместно обеспечили плавное масштабирование как объекта Deployment, так и самого кластера, чтобы справиться с возросшей нагрузкой (которая в нашем случае была ограничена максимальным

размером кластера). Чтобы понять, как происходит автоматическое уменьшение числа узлов, выполните следующие шаги.

1. Чтобы снизить нагрузку в кластере, уменьшите максимальное число реплик в объекте HPA. Для этого можно было бы отредактировать YAML-манифест и повторно применить его, но мы воспользуемся командой `kubectl patch`:

```
$ kubectl patch hpa elastic-hamster-hpa -n ca-demo -p '{"spec":
{"maxReplicas": 2}}'
horizontalpodautoscaler.autoscaling/elastic-hamster-hpa patched
```

2. Количество подов будет скорректировано в соответствии с измененной конфигурацией HPA:

```
$ kubectl get pod -n ca-demo
NAME                                READY   STATUS    RESTARTS   AGE
elastic-hamster-87d4db7fd-2qghf     1/1     Running   0          20m
elastic-hamster-87d4db7fd-mdvpx     1/1     Running   0          19m
```

3. Поскольку потребность в вычислительных ресурсах сократилась, кластерный автоскейлер начнет уменьшать число узлов. Однако процесс уменьшения масштаба предусматривает десятиминутную задержку, в течение которой поды переназначаются другим узлам. Проверив список узлов спустя 10 минут, вы увидите, что узлы, которые больше не нужны, были удалены.

```
$ kubectl get nodes
NAME                                  STATUS   ROLES   AGE    VERSION
gke-k8sdemo-default-pool-1bf4f185-6422  Ready    <none>  145m   v1.29.7-
gke.1008000
gke-k8sdemo-default-pool-1bf4f185-csv0  Ready    <none>  145m   v1.29.7-
gke.1008000
```

Этот пример наглядно показывает, как эффективно кластерный автоскейлер среагировал на снижение нагрузки в кластере после того, как HPA уменьшил количество подов в объекте Deployment. Ранее без нашего вмешательства количество узлов в кластере ненадолго увеличилось до десяти, а затем сократилось до двух. Только представьте разницу в стоимости между постоянной эксплуатацией восьмиузлового кластера и использованием кластера, масштаб которого автоматически корректируется в зависимости от уровня нагрузки!

Чтобы избежать лишних затрат на облачные ресурсы, очистите кластер или отключите кластерный автоскейлер.

Итак, мы подошли к концу главы, посвященной автомасштабированию подов и узлов. Но прежде, чем ее завершить, давайте кратко обсудим альтернативные инструменты автомасштабирования для Kubernetes.

Альтернативные автоскейлеры для Kubernetes

По сравнению со стандартным автоскейлером Kubernetes такие решения, как *Kubernetes Event-Driven Autoscaling* (KEDA) и *Karpenter*, предлагают большую гибкость и эффективность, поскольку осуществляют масштабирование ресурсов на основе метрик и нагрузок, характерных для конкретного приложения. KEDA допускает автомасштабирование на основе событий, происходящих вне кластера, а также на основе пользовательских метрик. Такой подход особенно актуален для событийно-ориентированных приложений.

С другой стороны, Karpenter упрощает процесс подготовки и масштабирования узлов, автоматически адаптируя их количество в зависимости от требований рабочей нагрузки. Это позволяет более эффективно и экономично использовать ресурсы кластера.

Вместе эти инструменты обеспечивают высокоточное управление процессом масштабирования, позволяя приложениям стабильно работать в условиях изменяющейся нагрузки.

В следующих разделах мы подробнее рассмотрим эти два популярных автоскейлера для Kubernetes.

Автоскейлер KEDA

Инструмент KEDA (https://keda.sh) реализует событийно-ориентированное автомасштабирование в Kubernetes, позволяя изменять число реплик подов на основе пользовательских метрик или внешних событий. В отличие от традиционных автоскейлеров, которые опираются на уровень потребления ресурсов ЦП или памяти, KEDA может инициировать масштабирование на основе данных из различных источников событий, таких как очереди сообщений, а также на основе частоты HTTP-запросов и пользовательских метрик приложения. Благодаря этому данный инструмент особенно полезен для рабочих нагрузок, зависящих от специфических событий или метрик, а не от общего уровня потребления ресурсов.

KEDA органично интегрируется с горизонтальным автоскейлером (HPA) Kubernetes и может увеличивать или уменьшать количество реплик приложения в зависимости от динамики рабочей нагрузки. Поддержка широкого набора источников событий обеспечивает гибкость и высокую точность принимаемых решений. KEDA помогает эффективно распределять ресурсы в соответствии с реальным уровнем нагрузки, оптимизируя затраты и повышая производительность приложения.

Архитектура и основные компоненты KEDA представлены на следующей схеме (рис. 20.5).

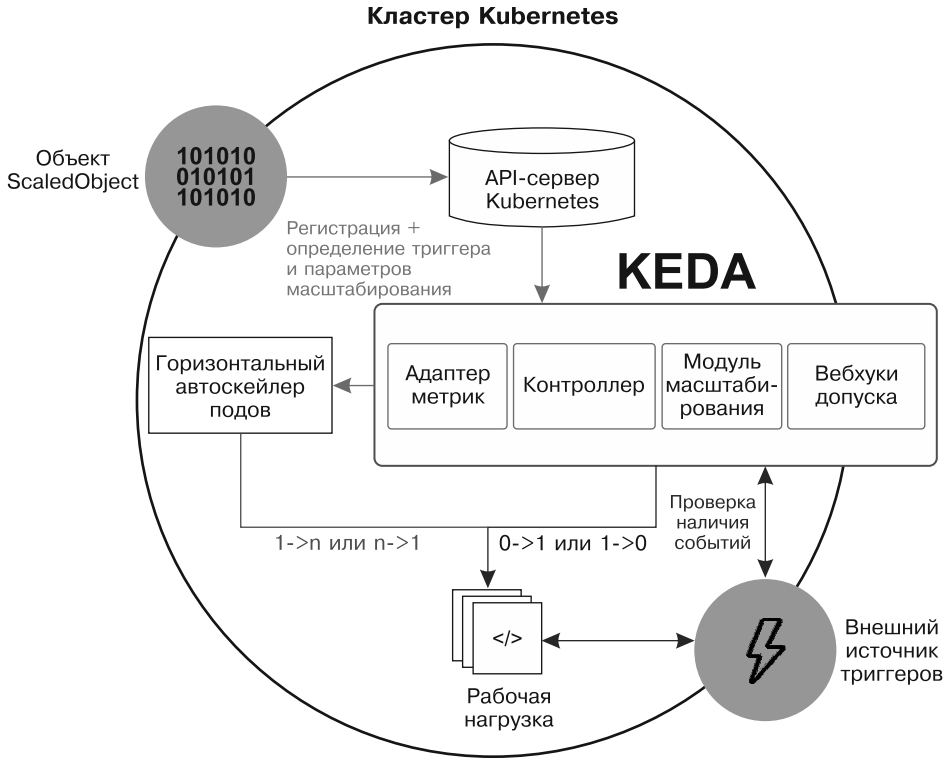

Рис. 20.5. Архитектура KEDA (источник изображения: https://keda.sh/docs/2.15/concepts/)

KEDA — это проект с открытым исходным кодом, развивающийся под эгидой CNCF. Его поддержка осуществляется по принципу *best-effort* (без гарантий качества) через GitHub, где можно оставлять отчеты об ошибках и запросы на улучшения. Некоторые поставщики услуг интегрируют KEDA в свои решения, среди которых можно выделить Azure Container Apps, Red Hat OpenShift Autoscaler с пользовательскими метриками и KEDA Add-On для Azure Kubernetes Service.

Автоскейлер Karpenter

Karpenter (https://karpenter.sh) — это продвинутый автоскейлер Kubernetes, ориентированный на оптимизацию подготовки и масштабирования узлов внутри кластера. Он автоматически регулирует количество узлов, динамически подстраивая их под потребности рабочих нагрузок. Karpenter разработан таким образом, чтобы быстро реагировать на изменения уровня нагрузки и оптимизировать емкость кластера, повышая как производительность, так и экономическую эффективность.

Принцип работы Karpenter в Kubernetes-кластере проиллюстрирован на следующей схеме (рис. 20.6).

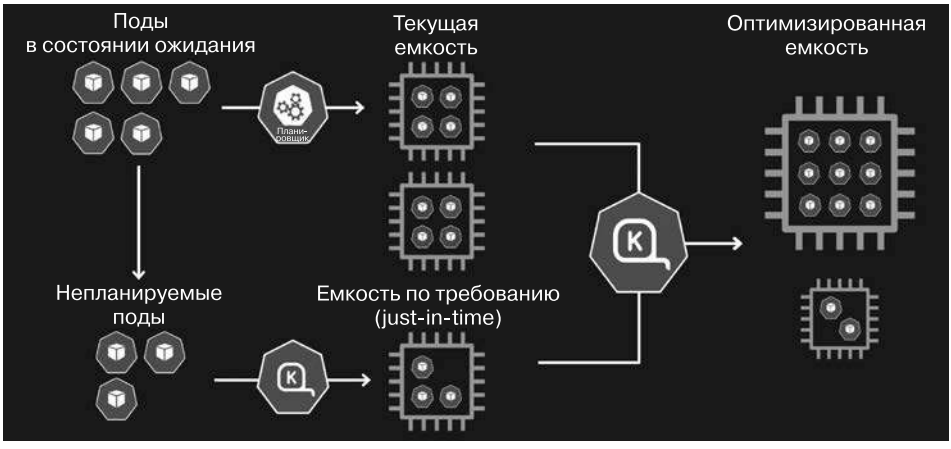

Рис. 20.6. Принцип работы Karpenter (источник: https://karpenter.sh)

Инструмент Karpenter обеспечивает быстрое и эффективное масштабирование узлов, предлагая такие возможности, как оптимизация емкости и интеллектуальное выделение ресурсов. Он следит за тем, чтобы в кластере были доступны узлы нужных типов и в необходимом количестве, что позволяет удовлетворять требования рабочей нагрузки и предотвратить неэффективное использование ресурсов. Благодаря расширенным возможностям масштабирования и подготовки узлов Karpenter помогает поддерживать стабильный уровень производительности кластера, удерживая эксплуатационные расходы под контролем.

Реализация автомасштабирования с использованием KEDA или Karpenter выходит за рамки данной книги. Дополнительную информацию можно найти в документации: https://keda.sh/docs/latest.

Резюме

В этой главе мы рассмотрели техники автомасштабирования, применяемые в кластерах Kubernetes. Сначала мы обсудили запросы и лимиты ресурсов подов и выяснили, почему они критически важны для автомасштабирования и планирования подов.

Затем вы познакомились с вертикальным автоскейлером (VPA), который автоматически изменяет значения запросов и лимитов подов на основе текущих и исторических метрик. После этого мы поговорили о горизонтальном автоскейлере (HPA), позволяющем автоматически изменять количество реплик объектов Deployment или StatefulSet на основе уровня потребления ресурсов ЦП, памяти

или пользовательских метрик. Наконец, мы рассмотрели роль кластерного автоскейлера (CA) в облачных средах, вы научились эффективно сочетать HPA и CA для одновременного масштабирования рабочих нагрузок и самого кластера.

VPA, HPA и CA предусматривают множество конфигурационных параметров — в этой главе мы лишь поверхностно коснулись их возможностей. Мы также упомянули альтернативные автоскейлеры Kubernetes, такие как KEDA и Karpenter.

В следующей главе мы обсудим такие продвинутые темы, как управление сетевым трафиком с помощью объектов Ingress, стратегии работы с несколькими кластерами, а также ряд развивающихся технологий.

Дополнительные источники информации

- Горизонтальное автомасштабирование подов: https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/.
- Автомасштабирование рабочих нагрузок: https://kubernetes.io/docs/concepts/workloads/autoscaling/.
- Пошаговое руководство по использованию объекта HorizontalPodAutoscaler: https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale-walkthrough/.
- Автомасштабирование кластера: https://kubernetes.io/docs/concepts/cluster-administration/cluster-autoscaling/.

Для получения дополнительной информации об автомасштабировании в Kubernetes вы можете обратиться к следующим книгам издательства Packt Publishing.

- *Baier J., Sayfan G., White J.* The Complete Kubernetes Guide (https://www.packtpub.com/en-in/product/the-complete-kubernetes-guide-9781838647346).
- *Baier J., White J.* Getting Started with Kubernetes. 3 ed. (https://www.packtpub.com/en-in/product/getting-started-with-kubernetes-9781788997263).
- *Heck J.* Kubernetes for Developers (https://www.packtpub.com/en-in/product/kubernetes-for-developers-9781788830607).
- *Tylenda P.* Hands-On Kubernetes on Windows (https://www.packtpub.com/product/hands-on-kubernetes-on-windows/9781838821562).

Кроме того, вы можете обратиться к официальной документации.

- Документация по Kubernetes (https://kubernetes.io/docs/home/) является наиболее актуальным источником информации об этой системе.
- Инструкция по установке VPA: https://github.com/kubernetes/autoscaler/tree/master/vertical-pod-autoscaler#installation.
- Документация по использованию VPA в EKS: https://docs.aws.amazon.com/eks/latest/userguide/vertical-pod-autoscaler.html.

21 Продвинутые возможности Kubernetes

В заключительной главе мы обсудим продвинутые темы, связанные с Kubernetes. Мы начнем с использования объектов Ingress для реализации сложных схем маршрутизации трафика к подам. Затем мы обсудим эффективные методы устранения неполадок и повышения безопасности, а также лучшие практики оптимизации работы кластера.

Первая часть этой главы будет посвящена продвинутым методам маршрутизации трафика в Kubernetes с использованием объектов Ingress, которые позволяют публиковать поды, работающие за сервисом, во внешней среде с помощью протоколов HTTP и HTTPS. До этого мы рассматривали способы публикации приложений с использованием самих объектов Service, в частности с помощью сервиса типа `LoadBalancer`. Однако этот метод подходит только для облачных сред, в которых работает компонент `cloud-controller-manager`. Принцип его действия состоит в настройке внешних балансировщиков нагрузки для использования с такими сервисами. Кроме того, каждый сервис типа `LoadBalancer` требует отдельного экземпляра облачного балансировщика, что приводит к дополнительным расходам и увеличивает операционные издержки.

Далее вы познакомитесь с объектом Ingress и Ingress-контроллером, которые могут использоваться в любой среде для маршрутизации трафика и балансировки нагрузки на приложения. Вы также узнаете, как использовать веб-сервер Nginx в качестве Ingress-контроллера и как настроить специализированный контроллер Azure *AGIC* (Application Gateway Ingress Controller) для кластера AKS.

Затем мы рассмотрим некоторые современные проекты экосистемы Kubernetes, среди которых KubeVirt, обеспечивающий виртуализацию, и бессерверные решения вроде Knative и OpenFaaS. Вы также узнаете об эфемерных контейнерах и их применении при устранении неполадок, а также о роли различных плагинов Kubernetes и способах управления несколькими кластерами. Большую часть этих тем мы затронем лишь вскользь, поскольку их подробное рассмотрение выходит за рамки данной книги.

Технические требования

Для работы с этой главой вам потребуется следующее.

- Развернутый кластер Kubernetes. Мы рекомендуем отдать предпочтение многоузловому кластеру, размещенному в облаке. В среде `minikube` также можно использовать объект Ingress после включения соответствующих надстроек.
- Для освоения материала раздела, посвященного Azure AGIC, необходим кластер AKS.
- Интерфейс командной строки `kubectl`, установленный на локальной машине и настроенный для управления кластером Kubernetes.

Инструкции по созданию кластера Kubernetes и настройке клиента `kubectl` приведены в главах 2 и 3.

В главах 15, 16 и 17 рассматривался процесс развертывания полнофункциональных кластеров Kubernetes в различных облачных средах и установки соответствующих CLI-инструментов для управления ими.

Примеры кода для этой главы можно загрузить из официального репозитория GitHub по адресу https://github.com/PacktPublishing/The-Kubernetes-Bible-Second-Edition/tree/main/Chapter21.

Продвинутая маршрутизация трафика с помощью объекта Ingress

В этом разделе вы узнаете о том, как объект Ingress используется для организации сложных сетевых взаимодействий и маршрутизации трафика в Kubernetes. По сути, Ingress — это ресурс Kubernetes, выполняющий роль обратного прокси. Он направляет входящие внешние запросы к сервисам внутри кластера в соответствии с правилами, заданными в его конфигурации. Один такой объект может служить единой точкой входа, обеспечивая доступ внешних пользователей к приложениям, развернутым в кластере.

Прежде чем перейти к рассмотрению объекта Ingress и связанных с ним ресурсов, давайте вспомним типы сервисов Kubernetes, которые мы использовали для доступа к приложениям.

Типы сервисов Kubernetes

В главе 8 вы познакомились с объектами Service, которые позволяют направлять внутренний и внешний трафик к подам. Внутри кластера они реализуются как виртуальные IP-адреса, управляемые компонентом `kube-proxy` на каждом узле. Перечислим основные типы сервисов.

- `ClusterIP` — публикует поды через внутренний виртуальный IP-адрес, управляемый компонентом `kube-proxy` на каждом узле. Это означает, что такой сервис доступен только в пределах кластера.
- `NodePort` — включает в себя функциональность `ClusterIP` и дополнительно доступен через IP-адрес любого узла и заданный порт. Компонент `kube-proxy` открывает порт на каждом узле в диапазоне `30000`–`32767` (диапазон можно изменить) и задает правила перенаправления трафика с этого порта к соответствующему сервису `ClusterIP`.
- `LoadBalancer` — обычно используется в облачных средах, где доступна *программно определяемая сеть* (SDN, software-defined network) и можно по запросу создавать балансировщики нагрузки, направляющие трафик в кластер. В `cloud-controller-manager` автоматическое создание облачных балансировщиков нагрузки выполняется с помощью специфичных для поставщика плагинов. Такой сервис сочетает подход, используемый в сервисе типа `NodePort`, с дополнительным внешним балансировщиком нагрузки, который перенаправляет трафик на соответствующие порты узлов. При этом сервис остается доступным и внутри кластера через его кластерный IP-адрес.

Несмотря на то что использование сервисов Kubernetes кажется очевидным способом организации внешнего доступа, существуют ситуации, когда такой подход не является оптимальным. В следующем подразделе вы познакомитесь с объектом Ingress и с тем, зачем он нужен и в каких случаях его следует использовать вместо сервисов для обработки внешнего трафика.

Обзор объекта Ingress

В предыдущем подразделе мы кратко рассмотрели сервисы Kubernetes и их роль в маршрутизации трафика. В плане обработки входящего трафика наиболее важными являются сервисы типа `NodePort` и `LoadBalancer`. Как правило, сервис `NodePort` используется совместно с другими средствами маршрутизации и балансировки нагрузки, поскольку публикация множества внешних конечных точек на всех узлах Kubernetes не является безопасной. Таким образом, основным вариантом остается сервис `LoadBalancer`, который внутренне опирается на `NodePort`. Однако у этого типа сервиса есть ограничения, которые не позволяют использовать его в некоторых случаях.

- Сервис `LoadBalancer` уровня L4 работает на транспортном уровне модели OSI и маршрутизирует трафик с использованием протоколов TCP/UDP. Большинство HTTP/HTTPS-приложений требует использования L7-балансировщика, относящегося к прикладному уровню модели OSI.
- L4-балансировщик не поддерживает функции завершения (termination) и разгрузки (offloading) для HTTPS-трафика.

- Один и тот же L4-балансировщик нельзя использовать для обслуживания нескольких доменов по технологии виртуального хостинга, основанного на именах.
- Маршрутизация на основе пути возможна только при использовании L7-балансировщика. Балансировщик уровня L4 в принципе нельзя сконфигурировать так, чтобы он проксировал запросы вида `https://<loadBalancerIp>/service1` к сервису Kubernetes `service1`, а `https://<loadBalancerIp>/service2` — к сервису `service2`, поскольку L4-балансировщик не понимает протокол HTTP(S).
- Реализация некоторых функций, таких как «липкие» сессии (sticky sessions) или привязка к куки (cookie affinity), требует использования балансировщика нагрузки уровня L7.

В Kubernetes все эти задачи могут быть решены с помощью объекта Ingress, который позволяет реализовать функциональность L7-балансировщика нагрузки. Объект Ingress используется только для определения правил маршрутизации и балансировки, например для сопоставления путей с сервисами Kubernetes.

Вот пример YAML-манифеста для объекта Ingress:

```
# ingress/portal-ingress.yaml
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: portal-ingress
  namespace: ingress-demo
  annotations:
    nginx.ingress.kubernetes.io/rewrite-target: /
spec:
  rules:
    - host: k8sbible.local
      http:
        paths:
          - path: /video
            pathType: Prefix
            backend:
              service:
                name: video-service
                port:
                  number: 8080
          - path: /shopping
            pathType: Prefix
            backend:
              service:
                name: shopping-service
                port:
                  number: 8080
```

Принцип работы Ingress-контроллера представлен на рис. 21.1.

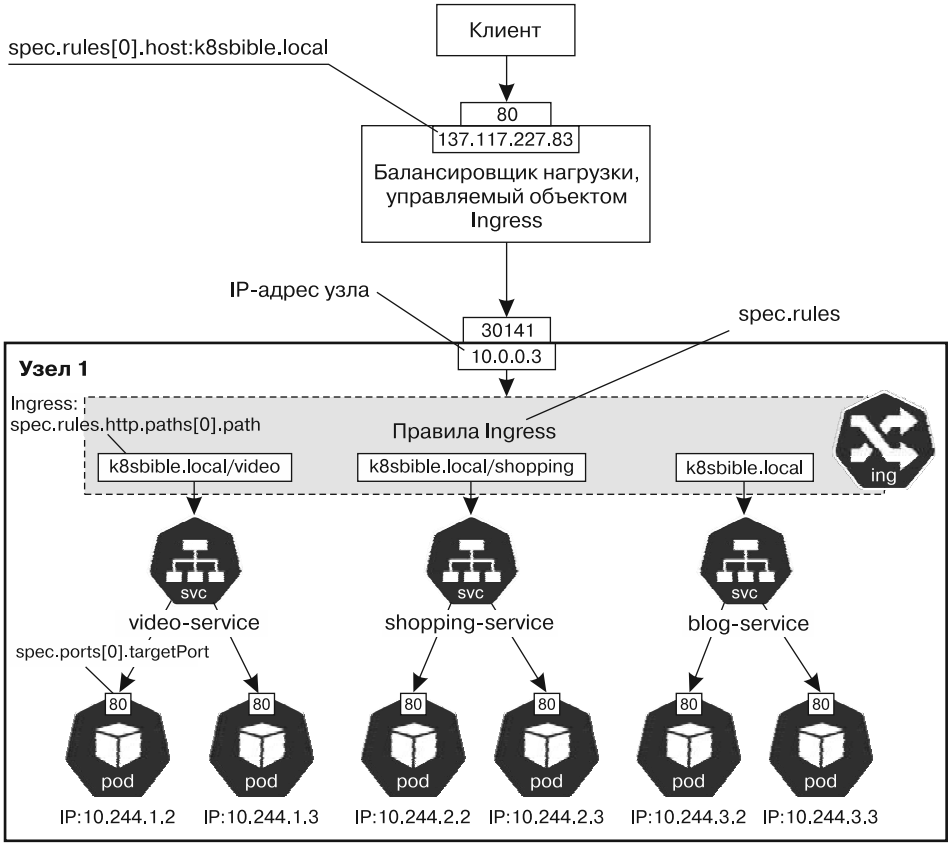

Рис. 21.1. Использование Nginx в качестве Ingress-контроллера в облачной среде

Проще говоря, объект Ingress представляет собой абстрактное определение правил маршрутизации для ваших сервисов. Сам по себе он ничего не делает. Для обработки и реализации этих правил необходим Ingress-контроллер. Вы можете применить манифест, но без контроллера это не окажет никакого эффекта. Прежде чем продолжить, давайте разберемся, как формируются правила HTTP-маршрутизации в объекте Ingress. Каждое правило в спецификации включает следующее.

- *Необязательное поле host.* В нашем примере оно не используется, поэтому определенное правило применяется ко всему входящему трафику. При заполнении этого поля правило применяется только к запросам, адресованным указанному хосту. При этом несколько доменных имен могут разрешаться

в один и тот же IP-адрес. Поле `host` допускает использование подстановочных знаков.

- *Перечень правил маршрутизации*. Каждый путь связан с backend-компонентом Ingress, который задается с помощью параметров `serviceName` и `servicePort`. В приведенном выше примере все запросы, поступающие по пути с префиксом `/video`, будут перенаправлены к подам сервиса `video-service`, а запросы, поступающие по пути с префиксом `/shopping`, — к подам сервиса `shopping-service`. Поле `path` поддерживает сопоставление на основе как префиксного, так и точного совпадения, а также специальные режимы сопоставления, зависящие от реализации конкретного Ingress-контроллера.

Такой подход позволяет настраивать сложные правила маршрутизации, охватывающие несколько сервисов внутри кластера, при этом внешнему пользователю они будут видны как единая конечная точка с несколькими доступными путями.

Чтобы такие объекты Ingress начали работать, в кластере должен быть установлен Ingress-контроллер, о котором мы поговорим в следующем подразделе.

Использование Nginx в качестве Ingress-контроллера

Ingress-контроллер — это контроллер Kubernetes, который необходимо вручную развернуть в кластере. Обычно он представляет собой объект DaemonSet или Deployment, запускающий специальные поды, отвечающие за обработку входящего трафика, балансировку нагрузки и интеллектуальную маршрутизацию. Именно он обрабатывает объекты Ingress, которые ссылаются на него, и отвечает за динамическое применение соответствующих правил маршрутизации.

В отличие от других типов контроллеров, работающих в составе компонента kube-controller-manager, Ingress-контроллеры не запускаются автоматически вместе с кластером. Проект Kubernetes поддерживает несколько Ingress-контроллеров, включая решения AWS, GCE и Nginx. Со списком сторонних решений можно ознакомиться здесь: https://kubernetes.io/docs/concepts/services-networking/ingress-controllers/#additional-controllers.

Одним из самых популярных Ingress-контроллеров для Kubernetes является Nginx или *Nginx Ingress Controller*[1] (https://www.f5.com/products/nginx/nginx-ingress-controller). Он устанавливается в кластер в виде объекта Deployment и содержит набор правил для обработки API-объектов Ingress. Этот контроллер публикуется в виде сервиса, тип которого зависит от способа установки; в облачных средах это обычно сервис типа `LoadBalancer`.

[1] В 2026 году планируется закрытие проекта и переход на Gateway API, подробности здесь: https://kubernetes.io/blog/2025/11/11/ingress-nginx-retirement/.

В облачных средах часто используются специализированные Ingress-контроллеры, задействующие особые возможности конкретного поставщика облачных услуг, позволяющие внешнему балансировщику нагрузки напрямую взаимодействовать с подами. Данный подход не требует создания дополнительных служебных подов, а иногда даже позволяет обойтись без сервисов типа NodePort. Вся маршрутизация осуществляется на уровне программно определяемой сети (SDN) и сетевого интерфейса контейнера (CNI), причем балансировщик нагрузки может напрямую использовать частные IP-адреса подов. Пример применения такого подхода мы рассмотрим в разделе, посвященном контроллеру Azure AGIC.

Инструкции по установке `ingress-nginx` для различных сред можно найти в официальной документации: https://kubernetes.github.io/ingress-nginx/deploy/.

Хотя применение инструмента Helm является предпочтительным способом развертывания, некоторые среды могут предусматривать специальные инструкции. В облачных средах установка `ingress-nginx` обычно крайне проста и сводится к применению одного YAML-манифеста (или к включению `ingress-nginx` при создании управляемого облачного кластера Kubernetes), который создает множество объектов Kubernetes. Например, в AKS или GKE развернуть необходимые компоненты Ingress-контроллера можно с помощью единственной команды:

```
$ kubectl apply -f https://raw.githubusercontent.com/kubernetes/ingress-nginx/
controller-v1.11.2/deploy/static/provider/cloud/deploy.yaml
```

Эта информация была актуальна на момент написания данной главы и тестирования соответствующего развертывания. Для получения сведений о текущей стабильной версии обратитесь к документации Ingress-контроллера. Также учтите, что разные дистрибутивы Kubernetes могут предусматривать различные условия для реализации подобных функций; например, для включения ingress-nginx в GKE вам потребуются права администратора кластера. Подробности вы можете найти в документации: https://kubernetes.github.io/ingress-nginx/.

В AWS для публикации Nginx Ingress Controller используется *сетевой балансировщик нагрузки* (Network Load Balancer, NLB), который настраивается с помощью сервиса типа `LoadBalancer`:

```
$ kubectl apply -f https://raw.githubusercontent.com/kubernetes/ingress-nginx/
controller-v1.11.2/deploy/static/provider/aws/deploy.yaml
```

YAML-манифест включает несколько ресурсов, необходимых для настройки объекта Ingress в кластере, в том числе `Role`, `RoleBinding`, `Namespace` и `ConfigMap`.

Если у вас нет доступа к облачной среде или облачному Kubernetes-развертыванию, перейдите к следующему подразделу, чтобы развернуть Ingress-контроллер в кластере `minikube`.

Развертывание Ingress-контроллера Nginx в кластере minikube

Многоузловой кластер Kubernetes на базе `minikube` можно развернуть с помощью следующей команды:

```
$ minikube start --cni calico --nodes 3 --kubernetes-version=v1.31.0
```

После запуска кластера Kubernetes активизируйте функциональность Ingress в `minikube`:

```
$ minikube addons enable ingress
💡  ingress is an addon maintained by Kubernetes. For any concerns contact
minikube on GitHub.
You can view the list of minikube maintainers at: https://github.com/kubernetes/
minikube/blob/master/OWNERS
💡  After the addon is enabled, please run "minikube tunnel" and your ingress
resources would be available at "127.0.0.1"

...<фрагмент опущен для краткости>...
🔎  Verifying ingress addon...
🌟  The 'ingress' addon is enabled
```

Проверьте статус подов в пространстве имен `ingress-nginx`, выполнив команду:

```
$ kubectl get pods -n ingress-nginx
NAME                                        READY   STATUS      RESTARTS   AGE
ingress-nginx-admission-create-rsznt        0/1     Completed   0          78s
ingress-nginx-admission-patch-4c7xh         0/1     Completed   0          78s
ingress-nginx-controller-6fc95558f4-zdhp7 1/1     Running     0          78s
```

Теперь Ingress-контроллер готов отслеживать ресурсы Ingress. В следующем разделе мы обсудим способ их развертывания в Kubernetes-кластере.

Развертывание ресурсов Ingress в Kubernetes

Теперь можно переходить к развертыванию приложения. В каталоге `Chapter21/ingress` репозитория вы найдете следующие YAML-файлы.

- `00-ingress-demo-ns.yaml` — для создания пространства имен `ingress-demo`.
- `video-portal.yaml` — для создания видеопортала (ConfigMap, Deployment и Service).
- `blog-portal.yaml` — для создания блог-портала (ConfigMap, Deployment и Service).

- `shopping-portal.yaml` — для создания торгового портала (ConfigMap, Deployment и Service).
- `portal-ingress.yaml` — для создания ресурса Ingress, реализующего маршрутизацию по путям сайта `k8sbible.local`.

Правило в файле `portal-ingress.yaml` предписывает Ingress-контроллеру направлять трафик с адреса `k8sbible.local/video` на сервис `video-service`:

```
# ingress/portal-ingress.yaml
...
spec:
  rules:
    - host: k8sbible.local
      http:
        paths:
          - path: /video
            pathType: Prefix
            backend:
              service:
                name: video-service
                port:
                  number: 8080
...
```

Следующее правило предписывает Ingress-контроллеру направлять трафик с адреса `k8sbible.local/shopping` на сервис `shopping-service`:

```
...
          - path: /shopping
            pathType: Prefix
            backend:
              service:
                name: shopping-service
                port:
                  number: 8080
...
```

Еще одно правило предписывает Ingress-контроллеру направлять трафик с адреса `k8sbible.local/` или `k8sbible.local` на сервис `blog-service`:

```
...
          - path: /
            pathType: Prefix
            backend:
              service:
                name: blog-service
                port:
                number: 8080
```

Поскольку мы уже подробно изучили объекты Deployment, ConfigMap и Service, мы не будем их снова обсуждать. При необходимости вы можете обратиться к YAML-файлам в репозитории.

Примените YAML-файлы из каталога `ingress`:

```
$ kubectl apply -f ingress/
namespace/ingress-demo created
configmap/blog-configmap created
deployment.apps/blog created
service/blog-service created
ingress.networking.k8s.io/portal-ingress created
configmap/shopping-configmap created
deployment.apps/shopping created
service/shopping-service created
configmap/video-configmap created
deployment.apps/video created
service/video-service created
```

Проверьте состояние подов, сервисов и ресурсов Ingress:

```
$ kubectl get po,svc,ingress -n ingress-demo
NAME                            READY   STATUS    RESTARTS   AGE
pod/blog-675df44d5-5s8sg        1/1     Running   0          88s
pod/shopping-6f88c5f485-lw6ts   1/1     Running   0          88s
pod/video-7d945d8c9f-wkxc5      1/1     Running   0          88s

NAME                      TYPE        CLUSTER-IP     EXTERNAL-IP   PORT(S)    AGE
service/blog-service      ClusterIP   10.111.70.32   <none>        8080/TCP   88s
service/shopping-service  ClusterIP   10.99.103.137  <none>        8080/TCP   88s
service/video-service     ClusterIP   10.109.3.177   <none>        8080/TCP   88s

NAME                                      CLASS   HOSTS           ADDRESS
PORTS   AGE
ingress.networking.k8s.io/portal-ingress  nginx   k8sbible.local  192.168.39.18
80      88s
```

Если вы используете облачный кластер Kubernetes, то доменное имя `k8sbible.local` (или любое другое, указанное в конфигурации Ingress) должно быть привязано к IP-адресу облачного балансировщика нагрузки. Если у вас нет зарегистрированного доменного имени, вы можете смоделировать его с помощью локального файла `/etc/hosts` (в Windows он находится по адресу `C:\Windows\System32\drivers\etc\hosts`).

Например, предположим, что мы развернули кластер `minikube` и добавили в файл `/etc/hosts` следующие IP-адреса виртуальной машины, полученные с помощью команды `minikube ip`:

```
# k8sbible minikube
192.168.39.18 k8sbible.local
```

Теперь вы можете получить доступ к своему порталу по адресу `http://k8sbible.local`. Откройте браузер (или используйте команду `curl`) и протестируйте различные сервисы, как показано на рис. 21.2.

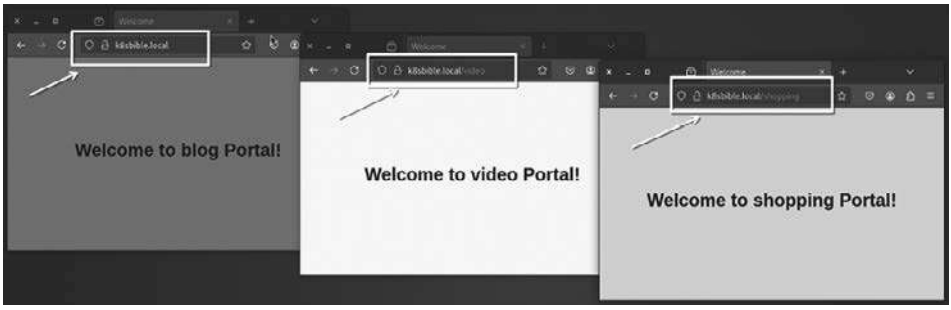

Рис. 21.2. Ingress-контроллер направляет трафик на различные сервисы

При обращении по адресу `http://k8sbible.local/video` HTTP-запрос будет перенаправлен через Nginx на сервис `video-service`[1]. Аналогичным образом запросы, направляемые по пути `/shopping`, будут маршрутизироваться на сервис `shopping-service`. Обратите внимание, что при выполнении этой операции используется только один облачный балансировщик нагрузки (или публичный IP-адрес/доменное имя), а фактическая маршрутизация к сервисам Kubernetes выполняется подами Ingress-контроллера на основе путей запросов.

На практике для обеспечения надлежащего уровня безопасности следует настраивать SSL-сертификаты для конечных точек HTTP. Их можно настроить и для объекта Ingress, однако для этого потребуется доменное имя или его локальные альтернативы. В наших примерах мы сознательно не настраиваем локальные доменные имена для сохранения простоты и наглядности. Более подробную информацию можно найти в документации по cert-manager: https://cert-manager.io/docs/tutorials/acme/nginx-ingress/.

Итак, вы успешно создали объект Ingress и настроили Ingress-контроллер в своем кластере.

Как мы упоминали в начале этого раздела, существует множество вариантов реализации Ingress-контроллеров и способов их использования. Прежде чем перейти к изучению следующего способа, познакомьтесь с ресурсом IngressClass.

---

[1] Обратите внимание, что ingress обращается к порту и имени сервиса, а не пода.

Использование ресурса IngressClass и нескольких Ingress-контроллеров

В некоторых ситуациях вам может потребоваться использовать разные конфигурации Ingress-контроллера. При наличии только одного такого контроллера реализовать это бывает невозможно, поскольку его специфичная настройка может повлиять на другие объекты Ingress в кластере Kubernetes. В таких случаях можно развернуть несколько Ingress-контроллеров в одном кластере, используя механизм IngressClass. Ниже описаны типичные сценарии, в которых такой подход может быть актуальным.

- *Разные Ingress-классы для удовлетворения различных требований.* Ingress-контроллеры Kubernetes можно снабдить аннотациями, обозначающими специфические Ingress-классы, например `nginx-public` и `nginx-private`. Это позволяет разделять типы трафика: например, публичный трафик может обслуживаться контроллером, оптимизированным для высокой производительности, а внутренние сервисы — оставаться под более строгим контролем доступа.
- *Поддержка разных протоколов.* Разным приложениям могут требоваться разные протоколы, включая HTTP/HTTPS и TCP/UDP. Их потребности можно обеспечить, задействуя отдельные Ingress-контроллеры для каждого протокола. Таким образом, приложения, использующие различные протоколы, смогут работать в одном кластере Kubernetes, не опираясь на универсальный контроллер. Данный подход повышает производительность и упрощает конфигурацию.

Важно обратить внимание на значение поля `.metadata.name` ресурса IngressClass, поскольку оно используется при заполнении поля `ingressClassName` в объекте Ingress. Поле `ingressClassName` пришло на смену устаревшему способу привязки Ingress к определенному контроллеру с помощью аннотаций, как прописано в спецификации *IngressSpec* `v1`.

Если вы не укажете IngressClass при создании объекта Ingress и в кластере существует только один IngressClass, помеченный как класс по умолчанию, Kubernetes автоматически применит его к вашему объекту Ingress. Чтобы пометить IngressClass как используемый по умолчанию, необходимо добавить к нему аннотацию `ingressclass.kubernetes.io/is-default-class` со значением `true`. Хотя именно так определено в спецификации, важно учитывать, что разные Ingress-контроллеры могут незначительно различаться в плане реализации этих возможностей.

Теперь проверим Ingress-контроллер Nginx, использованный в предыдущей практической части главы, чтобы определить его IngressClass:

```
$ kubectl get IngressClass -o yaml
apiVersion: v1
items:
  - apiVersion: networking.k8s.io/v1
    kind: IngressClass
    metadata:
      name: nginx
      annotations:
        ingressclass.kubernetes.io/is-default-class: "true"
...<фрагмент опущен для краткости>...
    spec:
      controller: k8s.io/ingress-nginx
```

В приведенном выше фрагменте можно отметить следующее:

- именем IngressClass является `nginx` (`.metadata.name`);
- присутствует аннотация `ingressclass.kubernetes.io/is-default-class: "true"`.

В следующем подразделе мы рассмотрим специализированный контроллер Azure AGIC для кластера AKS.

Контроллер Azure AGIC для кластера AKS

Как было сказано в предыдущем подразделе, использование Ingress-контроллера Nginx является достаточно гибким способом маршрутизации трафика внутри кластера Kubernetes. Однако при переключении на облачные сервисы, такие как *AKS* (Azure Kubernetes Service), этот подход может усложниться из-за нескольких уровней балансировки нагрузки, увеличивающих число потенциальных точек отказа.

Для решения этих проблем AKS предлагает собственный сервис балансировки нагрузки уровня L7 — *Azure AGIC* (Application Gateway Ingress Controller, контроллер входящего трафика для шлюза прикладного уровня). AGIC работает в связке с сетевыми службами Azure и обеспечивает значительно более эффективную и надежную маршрутизацию трафика, позволяя балансировщику нагрузки напрямую взаимодействовать с подами через их приватные IP-адреса. Это является возможным благодаря таким функциям Azure SDN, как *VNet Peering* (пиринг виртуальных сетей).

Причины для использования AGIC в AKS

AGIC стоит использовать в AKS по следующим причинам.

- *Упрощение процесса балансировки нагрузки.* AGIC устраняет необходимость в использовании отдельного балансировщика нагрузки Azure, который должен был бы проксировать запросы к подам Ingress-контроллера Nginx

с помощью сервиса типа `NodePort`. Вместо этого AGIC перенаправляет трафик напрямую к подам. Это уменьшает количество уровней балансировки нагрузки и снижает вероятность появления дополнительных точек отказа.

- *Организация прямого взаимодействия с подами.* AGIC использует возможности Azure SDN, позволяя балансировщику нагрузки напрямую взаимодействовать с подами, исключая участие компонента `kube-proxy` в маршрутизации трафика сервисов.

Архитектура AGIC представлена на рис. 21.3.

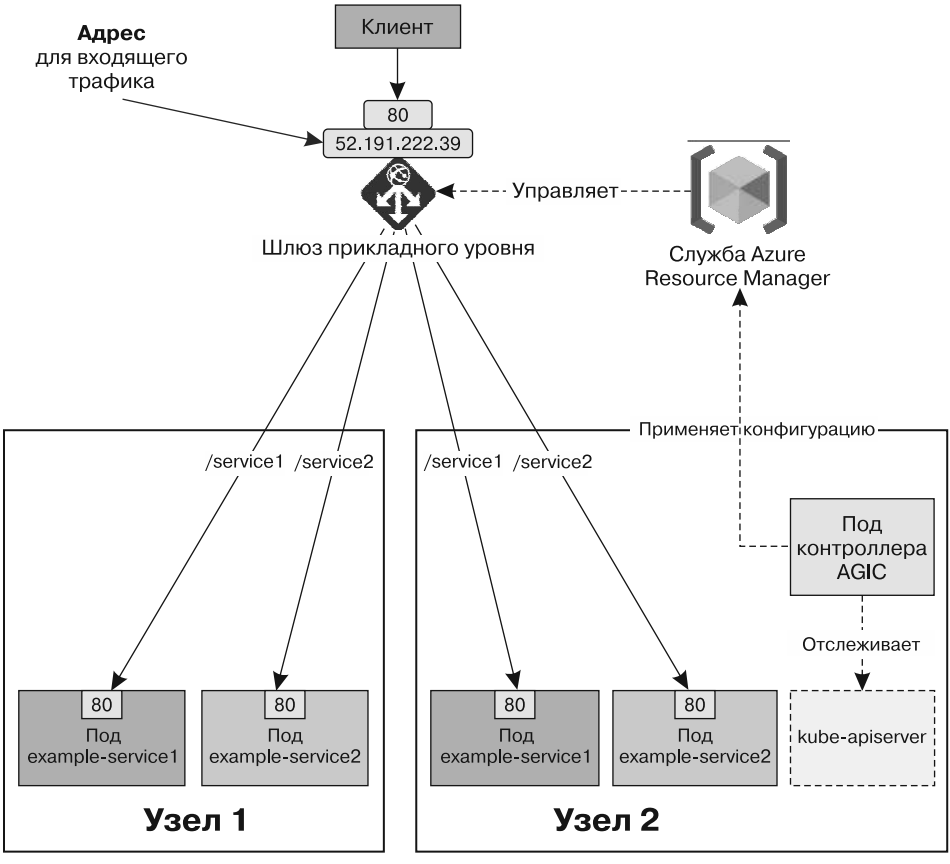

Рис. 21.3. Контроллер AGIC в кластере AKS

Контроллер AGIC можно настроить и в уже существующем кластере AKS; соответствующая инструкция приведена в официальной документации: https://docs.microsoft.com/en-us/azure/application-gateway/tutorial-ingress-controller-add-on-existing.

Для простоты и наглядности мы создадим новый кластер AKS с активизированным контроллером AGIC, используя единственную команду. Чтобы развернуть двухузловой кластер с именем `k8sforbeginners-aks-agic` в группе ресурсов `k8sforbeginners-rg`, выполните команду:

```
$ az aks create \
    --resource-group myResourceGroup \
    --name myAKSCluster \
    --node-count 2 \
    --network-plugin azure \
    --enable-managed-identity \
    -a ingress-appgw \
    --appgw-name MyAppGateway \
    --appgw-subnet-cidr "10.2.0.0/16" \
    --generate-ssh-keys
```

В результате будет создан компонент Azure Application Gateway с именем `AksApplicationGateway` и подсетью CIDR `10.2.0.0/16`.

После развертывания кластера необходимо сгенерировать файл `kubeconfig` для использования с утилитой `kubectl`. Выполните следующую команду (она переключит контекст на новый кластер, при этом предыдущий контекст останется доступен):

```
$ az aks get-credentials \
    --resource-group k8sforbeginners-rg \
    --name k8sforbeginners-aks-agic
Merged "k8sforbeginners-aks-agic" as current context in .kube/config
```

Теперь мы можем использовать YAML-манифесты для объектов Deployment и Service из каталога `ingress` репозитория книги, которые использовали в предыдущем разделе. Однако для работы AGIC в YAML-файлах необходимо внести некоторые изменения. Для большей ясности мы скопируем содержимое каталога `ingress` в новый каталог `aks_agic` и внесем изменения там. Скорректируйте определение ресурса Ingress следующим образом:

```
# aks-agic/portal-ingress.yaml
...
spec:
  ingressClassName: azure-application-gateway
...
```

Чтобы изолировать тестовый пример, мы также переименовали пространство имен в `agic-demo`. Примените YAML-манифесты из каталога `aks_agic` следующим образом:

```
$ kubectl apply -f aks_agic/
```

Подождите несколько минут, пока компонент Application Gateway обновит свою конфигурацию. Чтобы получить внешний IP-адрес Ingress-контроллера, выполните следующую команду:

```
$ kubectl get ingress
NAME              CLASS   HOSTS   ADDRESS         PORTS   AGE
example-ingress   <none>  *       52.191.222.39   80      36m
```

В нашем случае используется IP-адрес `52.191.222.39`.

Проверьте конфигурацию, перейдя по путям `/video` и `/shopping` с использованием полученного IP-адреса.

- *Сервис 1:* `http://<`*`внешний-IP`*`>/video` будет обслуживаться подами сервиса `video-service`.
- *Сервис 2:* `http://<`*`внешний-IP`*`>/shopping` будет обслуживаться подами сервиса `shopping-service`.
- *Сервис по умолчанию:* `http://<`*`внешний-IP`*`>/` будет обслуживаться подами сервиса `blog-service`.

Итак, вы успешно настроили и протестировали контроллер AGIC в AKS.

В следующем разделе вы познакомитесь с Gateway API — относительно новым и мощным механизмом управления маршрутизацией трафика в кластере Kubernetes.

Gateway API

Gateway API — это развивающийся набор ресурсов Kubernetes, предлагающий более выразительный и расширяемый способ описания правил маршрутизации сетевого трафика в кластере.

Он призван со временем заменить Ingress API, предоставив более мощный и гибкий механизм для настройки балансировки нагрузки, HTTP-маршрутизации и других сетевых функций.

Gateway API включает в себя следующие основные ресурсы.

- `GatewayClass` — определяет класс шлюзов (Gateways), которые разделяют общий набор конфигураций и управляются одним и тем же контроллером.
- `Gateway` — экземпляр среды, трафик в которой управляется контроллером, например облачным балансировщиком нагрузки.
- `HTTPRoute` — задает правила маршрутизации HTTP-трафика от слушателя шлюза к сетевым конечным точкам, обычно представленным объектами Service.

Высокоуровневая схема взаимодействия ресурсов Gateway API представлена на рис. 21.4.

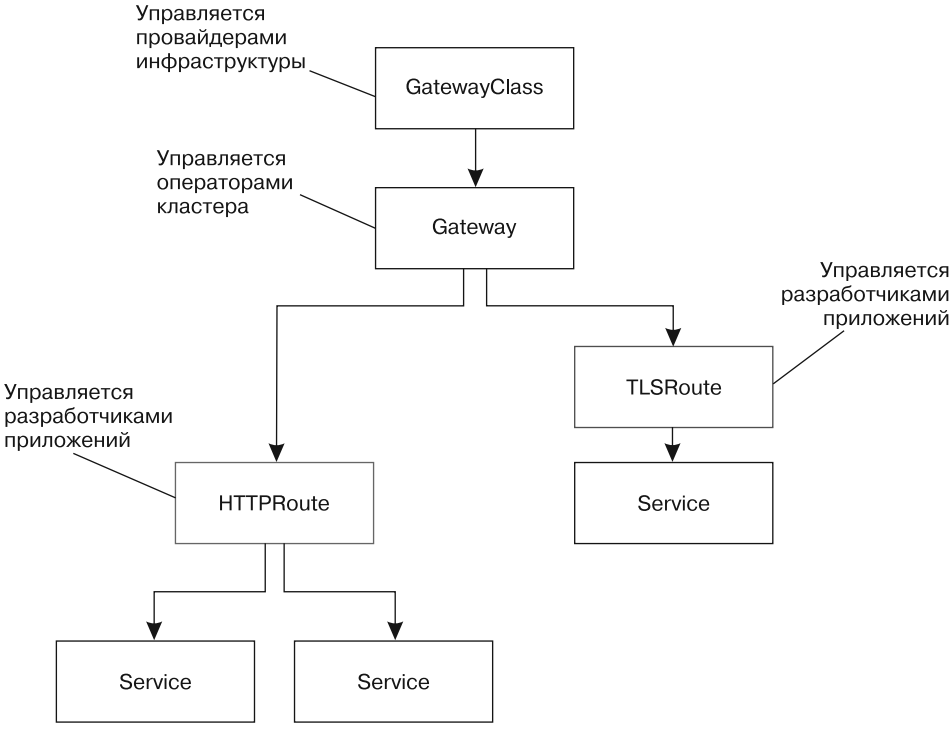

Рис. 21.4. Модель ресурсов Gateway API

К ключевым преимуществам Gateway API, по сравнению с Ingress API, относятся гибкость при реализации сложных сценариев маршрутизации, например многоуровневой маршрутизации трафика между разными пространствами имен, расширяемая архитектура, позволяющая сторонним разработчикам создавать собственные контроллеры шлюзов, бесшовно интегрируемые с Kubernetes, и детализированный контроль над правилами маршрутизации, политиками обработки трафика и настройкой балансировки нагрузки.

Вот пример типичного ресурса `GatewayClass`:

```
# gateway_api/gatewayclass.yaml
apiVersion: gateway.networking.k8s.io/v1
kind: GatewayClass
metadata:
  name: dev-cluster-gateway
spec:
  controllerName: "example.net/gateway-controller"
```

Файл `gateway-api/gateway_api/gateway.yaml` содержит пример ресурса `Gateway`, ссылающегося на `dev-cluster-gateway` в поле `gatewayClassName`:

```
apiVersion: gateway.networking.k8s.io/v1
kind: Gateway
metadata:
  name: dev-gateway
  namespace: gateway-api-demo
spec:
  gatewayClassName: dev-cluster-gateway
  listeners:
    - name: http
      protocol: HTTP
      port: 80
```

Наконец, ресурс `HTTPRoute` (по назначению схожий с объектом Ingress) содержит правила перенаправления трафика к различным сервисам:

```
# gateway-api/httproute.yaml
apiVersion: gateway.networking.k8s.io/v1
kind: HTTPRoute
metadata:
  name: dev-httproute
  namespace: gateway-api-demo
spec:
  parentRefs:
    - name: dev-gateway
      kind: Gateway
      namespace: gateway-api-demo
  hostnames:
    - "k8sbible.local"
  rules:
    - matches:
        - path:
            type: PathPrefix
            value: /video
      backendRefs:
        - name: video-service
          port: 80
```

Компоненты, участвующие в работе Gateway API, представлены на рис. 21.5.

Если вы хотите глубже разобраться в этой теме, обратитесь к документации и попробуйте реализовать Gateway API в своем кластере. Gateway API создавался в качестве замены Ingress API, однако он не поддерживает ресурс Ingress напрямую. Поэтому вам придется единоразово преобразовать существующие ресурсы Ingress в ресурсы Gateway API. Подробные рекомендации приведены в руководстве по миграции с Ingress (https://gateway-api.sigs.k8s.io/guides/migrating-from-ingress).

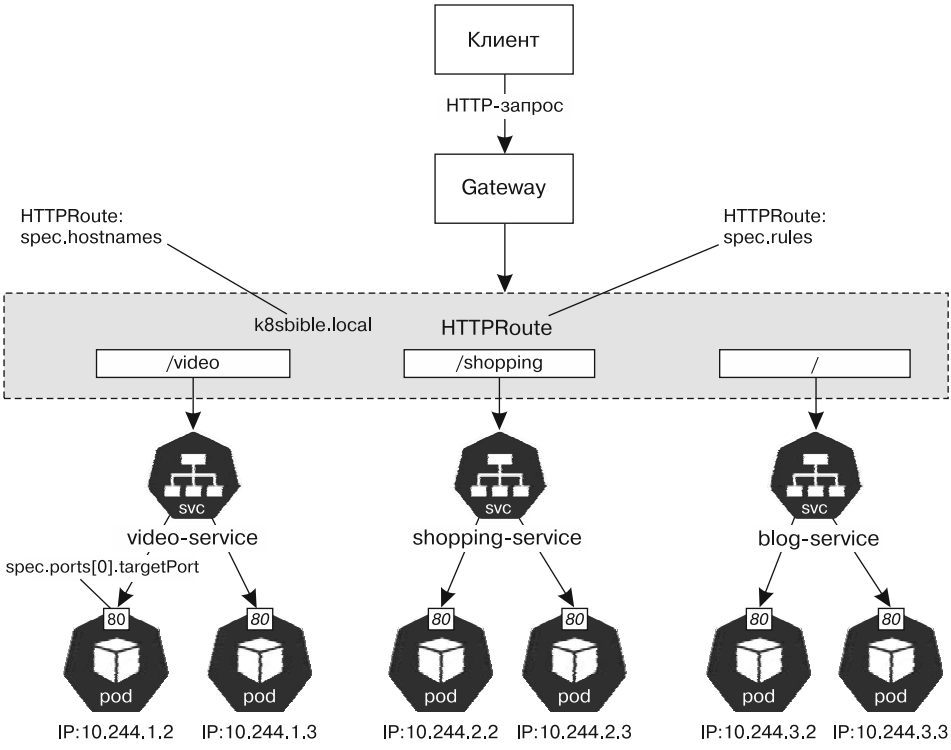

Рис. 21.5. Компоненты Gateway API

Прежде чем завершить обсуждение продвинутой маршрутизации трафика, Ingress и Gateway API, познакомьтесь с ресурсом EndpointSlice.

Ресурсы Endpoints и EndpointSlice

Традиционно система Kubernetes управляла развертыванием приложений с помощью подов, а объекты Service выступали в роли надежных сетевых посредников. В этой модели сервис служит своего рода единой точкой доступа к подам и обновляет соответствующий объект Endpoints, содержащий список активных и исправных подов, удовлетворяющих селектору сервиса. Однако данный подход плохо масштабируется по мере роста нагрузки.

Допустим, сервис представляет несколько подов. Соответствующий ему объект Endpoints содержит IP-адрес и порт каждого пода; эта информация распространяется по всему кластеру и используется в сетевых настройках. При любом, даже минимальном обновлении этого объекта изменения затрагивают

все узлы кластера, что приводит к значительному росту объема сетевого трафика и высокой нагрузке на узлы.

Для решения этой проблемы в Kubernetes был добавлен ресурс *EndpointSlice*, который «разрезает» монолитный объект Endpoints на более мелкие фрагменты. Каждый EndpointSlice по умолчанию вмещает до 100 конечных точек, содержащих сетевые данные о подах.

Благодаря механизму EndpointSlice обновления становятся гораздо более точечными: вместо пересылки всего объекта Endpoints происходит обновление только того среза, который содержит нужный под. Это уменьшает объем сетевого трафика и нагрузку на узлы, а главное — обеспечивает повышенную масштабируемость и производительность, открывая многообещающие возможности для эволюции Kubernetes.

Чтобы узнать больше о ресурсе EndpointSlices, обратитесь к документации: (https://kubernetes.io/docs/concepts/services-networking/endpoint-slices/).

В следующем разделе мы обсудим такие современные технологии, как бессерверные вычисления, машинное обучение, виртуализация, и покажем, как они интегрируются с Kubernetes.

Современные технологии в контексте Kubernetes

Система Kubernetes находится на переднем крае в плане интеграции и поддержки современных технологий, трансформирующих ИТ-отрасль. Она предлагает гибкую и масштабируемую платформу, способную легко интегрировать передовые решения для бессерверных вычислений, такие как *Knative*; платформы типа FaaS (Function as a Service, функция как услуга), например *OpenFaaS*; средства для управления виртуальными машинами, такие как *KubeVirt*; а также инструменты машинного обучения наподобие *Kubeflow*. Эти решения расширяют функциональность Kubernetes и помогают организациям быстрее внедрять инновации и переключаться на новые парадигмы.

Далее мы рассмотрим два наиболее мощных фреймворка — Knative и OpenFaaS, а также основные сценарии их использования.

Бессерверные вычисления с Knative и OpenFaaS

Бессерверные вычисления меняют парадигму создания и развертывания приложений, позволяя разработчикам сосредоточиться на написании кода, делегировав управление инфраструктурой автоматизированным платформам.

Использование *Knative* и *OpenFaaS* в среде Kubernetes обеспечивает полноценную реализацию бессерверного подхода, охватывающего развертывание, масштабирование и управление FaaS-решениями.

Knative — это платформа, построенная поверх Kubernetes и абстрагирующая большую часть сложности, связанной с управлением контейнеризованными приложениями. Она автоматизирует такие задачи, как масштабирование, управление трафиком и выполнение функций в ответ на некие события. Благодаря этому инструмент Knative особенно эффективен в сценариях, предполагающих переменный уровень нагрузки или решение задач, инициируемых внешними событиями. С его помощью можно масштабировать микросервисы в периоды пиковой нагрузки, выполнять фоновые операции (например, обработку загружаемых пользователями файлов) или создавать высокоотзывчивые событийно-ориентированные системы.

Еще один гибкий фреймворк *OpenFaaS* значительно упрощает развертывание функций в Kubernetes. Он позволяет запускать легковесные бессерверные функции в контейнерах, обеспечивая простоту их масштабирования и управления. Это особенно полезно в рамках микросервисной архитектуры, где каждая функция может масштабироваться отдельно в зависимости от спроса. OpenFaaS идеально подходит для обработки данных в реальном времени, выполнения функций в ответ на внешние события и создания специализированных API (например, для изменения размера изображений или преобразования данных) без издержек, сопряженных с развертыванием полноценного приложения. Использование Knative совместно с OpenFaaS позволяет организациям более полно задействовать возможности Kubernetes для уменьшения сложности инфраструктуры и масштабирования своих приложений.

Kubeflow — машинное обучение в Kubernetes

Kubeflow — это платформа с открытым исходным кодом, упрощающая развертывание, масштабирование и управление рабочими процессами машинного обучения (machine learning, ML) в Kubernetes. Она объединяет разнообразные инструменты и фреймворки в одну систему, позволяя дата-сайентистам и разработчикам ПО сосредоточиться на создании и тестировании ML-моделей, не заботясь об управлении инфраструктурой.

Система Kubeflow (https://www.kubeflow.org) способна автоматизировать весь цикл машинного обучения: от подготовки данных и обучения моделей до их развертывания и мониторинга. Она поддерживает большинство популярных ML-фреймворков, таких как *TensorFlow*, *PyTorch* и *XGBoost*, позволяя легко интегрировать их в существующие рабочие процессы. Kubeflow работает поверх

Kubernetes, что обеспечивает автоматическое масштабирование ML-нагрузок и их восстановление после сбоев.

Инструмент Kubeflow особенно эффективен при реализации масштабных ML-проектов, предполагающих обучение моделей на распределенных наборах данных, развертывание моделей в рабочей среде и их регулярное переобучение. Таким образом, Kubeflow представляет собой мощную и гибкую платформу, ускоряющую разработку и эксплуатацию моделей машинного обучения в Kubernetes.

В следующем подразделе мы поговорим о проекте KubeVirt.

KubeVirt — виртуальные машины в Kubernetes

KubeVirt (https://kubevirt.io) — это проект с открытым исходным кодом, расширяющий функционал Kubernetes за счет средств управления виртуальными машинами (ВМ). Благодаря такой интеграции организации могут запускать ВМ внутри кластера Kubernetes, размещая традиционные приложения, работающие на виртуальных машинах, и современные контейнеризованные приложения на одной и той же управляемой платформе.

Система KubeVirt обеспечивает органичное сосуществование виртуальных машин и контейнеров, позволяя применять мощные механизмы оркестрации и масштабирования Kubernetes ко всем типам рабочих нагрузок. Это особенно актуально для организаций, переключающихся на использование облачной архитектуры, но нуждающихся в поддержке унаследованных приложений, работающих на ВМ. В таких случаях KubeVirt позволяет управлять, масштабировать и оркестровать виртуальные машины так же, как контейнеризованные приложения в единой среде Kubernetes.

Для пользователей Red Hat OpenShift существует продуктовая версия KubeVirt — OpenShift Virtualization. Она предоставляет все перечисленные возможности, позволяя запускать виртуальные машины параллельно с контейнерными рабочими нагрузками и управлять ими непосредственно в среде OpenShift. Это снижает операционные затраты и сложность, обеспечивает более гибкое и эффективное использование ресурсов, а также упрощает модернизацию ИТ-инфраструктуры с сохранением поддержки существующих приложений, работающих на ВМ.

Ранее при обсуждении развертывания новых кластеров речь в основном шла о таких средах разработки, как `minikube`. Однако в реальных условиях кластеры Kubernetes обычно используются для выполнения критически важных приложений. В связи с этим особую значимость приобретает грамотное выполнение задач технического обслуживания кластеров в рамках операци-

онной деятельности, также называемых задачами второго дня эксплуатации (day-2 operations).

Мы рассмотрим некоторые из таких задач в следующих разделах.

Обслуживание кластеров Kubernetes: задачи второго дня эксплуатации

В следующих подразделах мы рассмотрим стандартные задачи обслуживания Kubernetes-кластеров, такие как резервное копирование, обновление, управление несколькими кластерами и не только.

Резервное копирование и восстановление Kubernetes-кластера

Резервное копирование и восстановление Kubernetes-кластера имеет ключевое значение для обеспечения целостности данных и непрерывности бизнес-процессов в любой рабочей среде. Среди всех элементов, которые должны быть охвачены процессом резервного копирования, наиболее важным является `etcd` — хранилище данных типа «ключ — значение», содержащее критически важные конфигурации и состояние кластера. В случае локальных или самостоятельно управляемых кластеров резервное копирование содержимого `etcd` выполняется путем создания снимков состояния (snapshots) и их безопасного хранения.

Создание резервной копии etcd

Создание резервной копии содержимого `etcd` имеет решающее значение для сохранности всех объектов Kubernetes, поскольку именно в `etcd` хранится состояние кластера. Регулярное резервное копирование позволяет восстановить кластер даже в случае потери всех узлов управляющего слоя. В процессе резервного копирования создается файл снимка, содержащий все состояние Kubernetes-кластера и другие критически важные данные. Поскольку эти данные могут включать конфиденциальную информацию, файлы снимков рекомендуется шифровать.

Механизм резервного копирования через утилиту etcdctl реализован на уровне самого проекта etcd. Многие дистрибутивы Kubernetes предусматривают собственные инструменты и механизмы для резервного копирования etcd. Например, в OpenShift сценарий cluster-backup.sh, входящий в состав etcd Cluster Operator, инкапсулирует выполнение команды etcdctl snapshot save, упрощая создание снимков состояния для кластеров etcd.

Инструмент `etcdctl` позволяет создать снимок состояния кластера `etcd` непосредственно с его активного узла. Этот процесс не влияет на производительность экземпляра `etcd`.

Инструменты `etcdctl` и `etcdutl` можно установить со страницы релизов `etcd` (https://github.com/etcd-io/etcd/releases/):

```
$ ETCDCTL_API=3 etcdctl \
    --endpoints=https://127.0.0.1:2379 \
    --cacert=/etc/kubernetes/pki/etcd/ca.crt \
    --cert=/etc/kubernetes/pki/etcd/server.crt \
    --key=/etc/kubernetes/pki/etcd/server.key \
    snapshot save /tmp/snapshot-pre-patch.db
```

Файлы `trusted-ca-file`, `cert-file` и `key-file` обычно можно найти в описании пода `etcd` (например, в файле `/etc/kubernetes/manifests/etcd.yaml`).

После создания снимка следует проверить его целостность с помощью инструмента `etcdutl`:

```
$ etcdutl --write-out=table snapshot status snapshot.db
```

Эта команда выводит такие сведения, как хеш-значение, номер ревизии, общее количество ключей и размер снимка.

Если данные etcd хранятся в томе, поддерживающем создание снимков (например, в Amazon Elastic Block Store), их резервное копирование можно выполнить, создав снимок самого тома. Такой подход часто используется в облачных средах, позволяющих автоматизировать процесс создания снимков хранилища.

В облачных кластерах такие управляемые сервисы, как Google Kubernetes Engine (GKE), Amazon EKS и Azure AKS, упрощают процесс резервного копирования. Эти платформы обычно предоставляют встроенные инструменты для автоматизированного создания резервных копий и восстановления. Например, для регулярного резервного копирования состояния и конфигурации кластера в EKS можно использовать сервис AWS Backup, а в AKS — Azure Backup, не требующие ручного вмешательства.

Восстановление кластера etcd из снимка с помощью etcdutl

Восстановление кластера `etcd` из снимка — это сложная и критически важная задача, особенно если речь идет о многоузловой конфигурации, требующей обеспечения согласованности данных на всех узлах. Этот процесс следует выполнять с осторожностью, чтобы избежать ошибок, особенно если API-серверы

в это время продолжают работать. Перед началом восстановления необходимо остановить все экземпляры API-сервера во избежание рассогласования данных. По окончании этого процесса следует перезапустить API-серверы, а также ключевые компоненты Kubernetes, такие как `kube-scheduler`, `kube-controller-manager` и `kubelet`, чтобы они не опирались на устаревшие данные.

Для восстановления кластера используйте инструмент `etcdutl`, указав каталог для сохранения восстановленных данных:

```
$ etcdutl --data-dir <каталог_с_данными> snapshot restore snapshot.db
```

Указанный *<каталог_с_данными>* будет создан в процессе восстановления.

Перенастройка API-сервера Kubernetes

Если в результате восстановления изменились URL-адреса доступа к кластеру `etcd`, необходимо перенастроить и перезапустить API-серверы Kubernetes с указанием новых адресов `etcd` (замените `$NEW_ETCD_CLUSTER` на соответствующий IP-адрес):

```
...
--etcd-servers=$NEW_ETCD_CLUSTER
...
```

Если перед кластером `etcd` используется балансировщик нагрузки, не забудьте обновить и его конфигурацию.

Использование практик IaC и CaC при управлении кластерами

Сложность резервного копирования и восстановления `etcd` обусловлена многообразием вариантов выполнения этих процедур, каждый из которых требует обеспечения целостности данных и стабильности системы. Наиболее эффективным способом устранения соответствующих трудностей является использование практик IaC (Infrastructure as Code — инфраструктура как код) и CaC (Configuration as Code — конфигурация как код) при управлении кластерами Kubernetes и приложениями. Такой подход позволяет с легкостью воссоздать все компоненты с нуля благодаря тому, что все конфигурации являются версионируемыми, воспроизводимыми и единообразными.

При внедрении практик IaC и CaC в рамках Git-процессов важно соблюдать принцип четырех глаз (four-eyes principle). Он предполагает, что любые изменения должны проходить проверку как минимум двумя членами команды перед включением в основную ветку. Это повышает качество кода, обеспечивает

соответствие нормативным требованиям и снижает вероятность возникновения ошибок при резервном копировании и восстановлении.

Для надежной реализации такого подхода кластеры следует считать не сохраняющими состояние и неизменяемыми. Все конфигурации, включая пространства имен, операторы, настройки RBAC, сетевые политики и т. д., должны храниться в YAML-файлах. Эти файлы, в свою очередь, следует версионировать, хранить в репозитории и автоматически применять при создании нового кластера. Это гарантирует соответствие нового кластера прежнему, минимизирует время простоя и снижает вероятность возникновения ошибок, связанных с человеческим фактором.

Данный подход следует распространить и на приложения. Все, что связано с их развертыванием, начиная с ConfigMap и Service и заканчивая PVC, должно быть выражено в виде кода. В приложениях с сохранением состояния данные хранятся в постоянных томах, находящихся вне кластера. Благодаря отделению данных от конфигурации восстановление приложения сводится к повторному применению YAML-файлов и подключению к имеющимся данным.

Для дополнительной оптимизации процесса можно использовать Helm-шаблонизацию и механизмы непрерывного развертывания GitOps. Такая автоматизация обеспечивает единообразие конфигураций, поскольку изменения автоматически применяются к соответствующим средам при минимальном участии человека. Комплексный подход к управлению кластерами и приложениями существенно упрощает аварийное восстановление, повышая при этом масштабируемость, безопасность и операционную эффективность.

В следующем разделе мы поговорим об обновлении кластеров Kubernetes.

Обновление Kubernetes-кластера

Обновление Kubernetes-кластера — это одна из важнейших операций, обеспечивающих безопасность, стабильность и актуальность среды. Большинство управляемых дистрибутивов Kubernetes в облачных средах обновляются относительно просто, поскольку вся внутренняя сложность абстрагируется механизмами управляемого сервиса. К таким решениям относятся Amazon EKS, GKE и Azure AKS. Они предусматривают функцию обновления «в один клик», позволяющую переключиться на новую версию Kubernetes с минимальным простоем или вовсе без него.

В локальных или особым образом настроенных кластерах ситуация обстоит иначе. Например, для кластеров, развернутых с помощью `kubeadm`, предусмотрена документированная процедура обновления (https://kubernetes.io/docs/tasks/

administer-cluster/kubeadm/kubeadm-upgrade) с подробным описанием шагов, необходимых для актуализации управляющего слоя и рабочих узлов.

Независимо от того, работаете вы с облачными кластерами или управляете локальной инфраструктурой, ключевым фактором является соблюдение структурированного процесса обновления, подробный обзор которого приведен далее.

Подготовка к обновлению

Перед обновлением кластер необходимо подготовить, выполнив следующие шаги.

- *Проверка совместимости*. Убедитесь, что новая версия Kubernetes совместима со всеми существующими компонентами и надстройками. Для изучения матриц совместимости обратитесь к официальной документации Kubernetes.
- *Резервное копирование etcd*. Хранилище `etcd` — это центральный компонент Kubernetes-кластера. Чтобы защитить конфигурацию кластера, всегда создавайте его резервную копию перед обновлением.
- *Отключение подкачки памяти (swap)*. Kubernetes требует отключения подкачки памяти на всех узлах. Убедитесь в корректной настройке соответствующего параметра, чтобы избежать возможных проблем.

Процесс обновления

Процесс обновления обычно включает несколько этапов.

- *Освобождение узлов*. Выполните безопасное выселение всех подов с узлов, которые планируете обновить, с помощью команды `kubectl drain <освобождаемые_узлы> --ignore-daemonsets`. Это гарантирует, что узел не получит новые задания во время обновления.
- *Обновление управляющего слоя*. Сначала обновите компоненты управляющего слоя, такие как API-сервер, `etcd` и `kube-controller-manager`. Для установки последних версий используйте соответствующие команды вашего пакетного менеджера (например, apt-get или yum).
- *Обновление kubeadm*. Обновите инструмент `kubeadm` до нужной версии. Это обеспечит совместимость с новой версией Kubernetes.
- *Обновление kubelet и kubectl*. После обновления управляющего слоя обновите `kubelet` и `kubectl` на каждом узле. Эти компоненты взаимодействуют с управляющим слоем и управляют подами.
- *Разблокировка узлов*. После обновления узла разрешите назначение ему новых подов с помощью команды `kubectl uncordon <имя-узла>`.

- *Обновление вычислительных узлов*. Выполните постепенное обновление всех рабочих узлов, повторив шаги, предпринятые при обновлении управляющего слоя.
- *Обновление CNI-плагина*. Убедитесь, что ваш *сетевой плагин* (CNI) совместим с новой версией Kubernetes, и при необходимости обновите его.

Действия после обновления

После завершения обновления необходимо выполнить следующие действия.

- *Проверка состояния кластера*. Убедитесь, что все узлы находятся в состоянии `Ready`, выполнив команду `kubectl get nodes`.
- *Мониторинг etcd*. Отслеживайте состояние и производительность компонента `etcd` во время и после обновления.
- *Переключение репозиториев пакетов*. Если это еще не сделано, обновите используемые репозитории пакетов так, чтобы они указывали на источники новой версии Kubernetes.

План отката

Важно заранее разработать план отката на случай непредвиденных ошибок, которые могут возникнуть в процессе обновления. Этот план должен включать шаги, необходимые для возврата к предыдущим конфигурациям и восстановления данных из резервных копий. Внутренние изменения в API и структуре данных `etcd` затрудняют выполнение отката, однако наличие продуманного плана позволяет сократить время простоя и минимизировать операционные сбои. Четкое распределение ролей и ответственности внутри команды позволяет организовать своевременное и скоординированное реагирование, даже если необходимость в откате возникает крайне редко.

Дополнительные рекомендации

Ниже перечислены дополнительные рекомендации, которых стоит придерживаться.

- *Тестируйте обновления в промежуточной среде*. Перед обновлением рабочего кластера имеет смысл протестировать процесс обновления в промежуточной среде или в среде разработки.
- *Используйте инструмент обновления кластеров*. Существуют инструменты, автоматизирующие часть операций, связанных с обновлением, что уменьшает объем ручной работы и снижает вероятность возникновения ошибок.
- *Отслеживайте возникающие проблемы*. В процессе обновления и после его завершения отслеживайте состояние кластера, обращая внимание на любые признаки некорректной работы.

Эффективность процесса обновления можно дополнительно повысить за счет его автоматизации с помощью таких инструментов, как Ansible, Terraform, AWS CloudFormation и ARM-шаблоны. Эти решения позволяют управлять всем циклом обновления — от подготовки узлов и установки пакетов до реализации поэтапного обновления компонентов.

На практике такие подходы дают возможность автоматизировать обновление кластеров в мультиоблачных средах. Управлять мультикластерными развертываниями, гарантируя единообразное обновление кластеров в разных окружениях, можно с помощью инструментов вроде ArgoCD или Fleet. Это особенно актуально для организаций, использующих более одного кластера, так как это сокращает объем ручного вмешательства и обеспечивает согласованность конфигураций во всех средах.

В следующем подразделе мы рассмотрим некоторые из наиболее известных инструментов для управления несколькими кластерами.

Управление несколькими кластерами

Стремительный рост масштабов организаций усложняет процесс управления множеством Kubernetes-кластеров, развернутых в различных средах. Решить эту проблему призваны инструменты для управления несколькими кластерами, позволяющие централизованно развертывать, отслеживать и обновлять их. Многие из этих инструментов поддерживают автоматизированное создание кластеров и постепенные обновления, обеспечивая единообразие и безопасность всех управляемых кластеров.

Например, в мультиоблачной среде можно развертывать и управлять Kubernetes-кластерами в AWS, Azure и Google Cloud с помощью Terraform и ArgoCD. Такой подход позволяет автоматизировать процессы развертывания и обновления с минимальным риском возникновения человеческих ошибок и при использовании одинаковой версии Kubernetes во всех кластерах. Это особенно ценно для крупных организаций со множеством команд и региональных представительств, операционная эффективность которых напрямую зависит от поддержания единообразной и актуальной Kubernetes-среды.

Ниже приведен список наиболее известных инструментов и сервисов для управления несколькими Kubernetes-кластерами.

- *Rancher*. Платформа с открытым исходным кодом, упрощающая управление системой Kubernetes. Она обеспечивает централизованное управление кластерами в различных средах — как локальных, так и облачных. Rancher позволяет развертывать приложения в нескольких кластерах, использовать встроенные средства мониторинга и механизм RBAC для управления правами пользователей в масштабах всей инфраструктуры.

- *Lens*. Интегрированная среда разработки (IDE) для Kubernetes, позволяющая управлять несколькими кластерами через единый интерфейс. Она предоставляет ценные сведения в режиме реального времени, предусматривает встроенный терминал и удобные представления ресурсов, облегчая визуализацию и контроль Kubernetes-среды для разработчиков и операторов.
- *Kops (Kubernetes Operations)*. Инструмент для управления жизненным циклом Kubernetes-кластеров, в частности развернутых на платформе AWS. Он автоматизирует процессы создания, обновления и удаления кластеров и высоко ценится за упрощение операционной деятельности на различных облачных платформах.
- *Red Hat Advanced Cluster Management for Kubernetes*. Комплексное решение для управления Kubernetes-кластерами в гибридных и мультиоблачных средах. Этот инструмент предусматривает такие функции, как управление на основе политик, контроль жизненного цикла приложений и мониторинг состояния кластеров, обеспечивая их соответствие требованиям и оптимальную производительность.
- *Anthos (Google Cloud)*. Платформа для унифицированного управления Kubernetes-кластерами в гибридных и мультиоблачных средах. Данное решение предоставляет функции централизованного контроля, обеспечения безопасности и единообразного развертывания приложений, гарантируя согласованный операционный опыт во всех управляемых кластерах.
- *Azure Arc*. Сервис, позволяющий применять функции управления и контроля Azure к Kubernetes-кластерам, работающим в разных средах, включая локальные дата-центры, облака и периферийные устройства. С помощью Azure Arc вы можете управлять Kubernetes-кластерами через единый интерфейс, что позволяет применять согласованные политики, единые механизмы защиты и мониторинга ко всей вашей инфраструктуре.

В следующем разделе мы рассмотрим лучшие практики обеспечения безопасности Kubernetes-кластеров.

Лучшие практики защиты кластера Kubernetes

Обеспечение безопасности Kubernetes-кластера необходимо для предотвращения несанкционированного доступа, утечек данных и сбоев в работе. Внедряя надежные меры защиты, вы можете обезопасить конфиденциальную информацию и обеспечить стабильную работу всей системы. В этом разделе приведены рекомендации и лучшие практики, позволяющие защитить кластер от случайных ошибок и целенаправленных атак.

Некоторые концепции, обсуждаемые в этом разделе, уже были затронуты в главе 18. Здесь мы вновь обратимся к ним, чтобы акцентировать их значение в контексте лучших практик эксплуатации Kubernetes.

Контроль доступа к API Kubernetes

Поскольку Kubernetes в значительной степени опирается на свой API, для обеспечения безопасности кластера в первую очередь нужно позаботиться об ограничении доступа к нему.

- *Использование TLS для защиты API-трафика*. Kubernetes по умолчанию шифрует коммуникацию с API с помощью протокола TLS. Большинство методов установки предполагают автоматическое создание и настройку соответствующих сертификатов. Однако администраторам важно учитывать наличие незащищенных локальных портов и своевременно закрывать их.
- *Аутентификация в API*. Выберите метод аутентификации, соответствующий вашим потребностям. Для небольших однопользовательских кластеров может быть достаточно сертификатов или статических bearer-токенов. Крупные кластеры обычно требуют интеграции с существующими системами аутентификации, такими как OIDC или LDAP.
- *Авторизация в API*. После аутентификации каждый запрос к API проходит проверку в рамках процедуры авторизации. Kubernetes использует механизм RBAC, сопоставляя пользователей или их группы с набором прав, определенных в ролях. Эти права описывают разрешенные действия над ресурсами и могут действовать как в пределах отдельных пространств имен, так и на уровне всего кластера. Для повышения безопасности рекомендуется сочетать авторизаторы Node и RBAC.

Контроль доступа к kubelet

Агенты `kubelet`, управляющие узлами и контейнерами, публикуют конечные точки HTTPS, предоставляющие значительный уровень контроля над узлом. В рабочих средах необходимо убедиться, что для `kubelet` включены механизмы аутентификации и авторизации.

Чтобы эффективно контролировать доступ к `kubelet` в рабочих кластерах, следует обеспечить корректную работу механизмов аутентификации и авторизации в API Kubelet. По умолчанию разрешены только запросы, проходящие через API-сервер Kubernetes, что предотвращает несанкционированные прямые обращения к `kubelet`. Дополнительно можно усилить защиту, настроив RBAC-разрешения для пользователей и сервисов и ограничив сетевую видимость конечных точек `kubelet` с помощью сетевых политик или правил межсетевого экрана.

Контроль возможностей рабочих нагрузок и пользователей во время выполнения

Механизмы авторизации Kubernetes работают на высоком уровне, однако при необходимости можно применять более детализированные политики для ограничения использования ресурсов и управления привилегиями контейнеров.

- *Ограничение использования ресурсов.* Задайте квоты ресурсов и диапазоны ограничений, чтобы контролировать объемы ресурсов ЦП, памяти или дискового пространства, доступные пространству имен. Эти механизмы не позволяют пользователям запрашивать чрезмерно большие или, напротив, слишком малые объемы ресурсов.
- *Контроль привилегий контейнеров.* Поды могут запрашивать разрешение на запуск от имени определенных пользователей или с повышенными привилегиями. Поскольку большинству приложений не требуются права суперпользователя, рекомендуется настраивать контейнеры так, чтобы они выполнялись от имени непривилегированных пользователей.
- *Предотвращение загрузки нежелательных модулей ядра.* Чтобы уменьшить риск эксплуатации уязвимостей, удалите или заблокируйте ненужные модули ядра на узле. Дополнительно можно использовать модули безопасности Linux (Linux Security Modules), например *SELinux*, чтобы запретить загрузку таких модулей в контейнерах.

Ограничение сетевого доступа

Kubernetes позволяет управлять сетевым доступом на разных уровнях.

- *Сетевые политики.* Применяйте сетевые политики, чтобы ограничить доступ к ресурсам вашего пространства имен со стороны подов из других пространств имен. Для управления запросами на использование портов узлов и сервисов с балансировкой нагрузки также можно применять квоты и диапазоны ограничений.
- *Ограничение доступа к облачному Metadata API.* Облачные платформы часто предоставляют сервисы метаданных, которые могут содержать конфиденциальную информацию. Для ограничения доступа к этим API следует использовать сетевые политики, а также избегать хранения секретов в облачных метаданных.

Защита компонентов кластера

Чтобы обезопасить кластер, необходимо защитить ключевые компоненты, такие как `etcd`, и настроить контроль доступа.

- *Ограничение доступа к etcd.* Получив доступ к хранилищу `etcd`, злоумышленник может фактически завладеть всем кластером. Используйте надежные

учетные данные и изолируйте серверы `etcd` за брандмауэром. Например, в частных Kubernetes-кластерах AWS можно создать группу безопасности, разрешающую доступ к `etcd` на порте `2379` только с IP-адресов управляющего слоя. Кроме того, можно использовать флаги `--client-cert-auth` и `--trusted-ca-file`, чтобы разрешить управляющему слою подключаться к `etcd` только по защищенным каналам.

- *Ведение журналов аудита*. Ведение журналов аудита (или аудит-логирование) предполагает фиксацию действий, выполняемых через API, для последующего анализа. Данная практика помогает своевременно обнаруживать и реагировать на возможные компрометации. Команде, управляющей кластером Kubernetes, необходимо определить собственную политику аудита для событий создания, удаления и изменения ресурсов, а также настроить безопасную отправку журналов в доверенное хранилище, например Elasticsearch. Ниже приведен пример настройки аудит-логирования в манифесте пода `kube-apiserver`:

```
...
--audit-log-path=/var/log/audit.log
--audit-policy-file=/etc/kubernetes/audit-policy.yaml
...
```

- *Регулярная ротация инфраструктурных учетных данных*. Использование учетных данных с коротким сроком действия снижает риск несанкционированного доступа. Регулярно обновляйте сертификаты, токены и другие чувствительные данные. Например, можно настроить `cert-manager` (https://cert-manager.io/) на автоматическое обновление TLS-сертификатов, а `kubelet` — на периодическое обновление собственных сертификатов с помощью флагов `RotateKubeletClientCertificate` и `RotateKubeletServerCertificate`.
- *Проверка разрешений, предоставленных сторонним инструментам*. При подключении сторонних инструментов тщательно проверяйте их разрешения. По возможности ограничивайте их доступ конкретными пространствами имен, чтобы минимизировать риски. Например, при установке инструментов Prometheus или Grafana достаточно предоставить им права только на чтение данных, создав роль с разрешением read-only и привязав ее к нужным пространствам имен с помощью объектов RoleBinding. Такой подход снижает потенциальный ущерб от утечки информации.
- *Шифрование секретов в состоянии покоя*. Kubernetes поддерживает шифрование секретов, хранящихся в `etcd`. Это гарантирует то, что даже при получении доступа к данным `etcd` злоумышленник не сможет с легкостью прочитать конфиденциальную информацию. Настройте параметр `EncryptionConfig` в конфигурации `kube-apiserver` для AES-шифрования секретов. Это обеспечит дополнительный уровень защиты в случае утечки содержимого `etcd`.

В табл. 21.1 приведены ключевые рекомендации по повышению безопасности Kubernetes-кластеров.

Таблица 21.1. Лучшие практики защиты Kubernetes-кластера

Раздел	Лучшие практики
Безопасная настройка кластера	Активизируйте механизм RBAC и используйте выделенные учетные записи сервисов. Поддерживайте компоненты Kubernetes в актуальном состоянии. Защитите доступ к API-серверу с помощью TLS
Контроль доступа к кластеру	Используйте надежные методы аутентификации. Реализуйте строгий контроль доступа и принцип наименьших привилегий. Регулярно проводите аудит и пересмотр прав доступа
Защита сетевых коммуникаций	Шифруйте внутренние коммуникации. Реализуйте сетевую сегментацию. Используйте безопасные сетевые плагины и применяйте сетевые политики
Безопасность контейнерных образов	Используйте доверенные реестры контейнеров. Сканируйте образы на наличие уязвимостей. Применяйте политики безопасности подов для ограничения привилегий контейнеров
Мониторинг и логирование активности в кластере	Внедрите решения для логирования и мониторинга. Активизируйте функцию аудита. Регулярно проверяйте логи на предмет подозрительной активности
Регулярные обновления и патчи	Своевременно применяйте обновления и патчи для устранения уязвимостей. Строго соблюдайте процедуру управления обновлениями
Непрерывное обучение и подготовка команды	Обучайте свою команду лучшим практикам обеспечения безопасности. Следите за последними тенденциями и новостями в области безопасности. Формируйте культуру обеспечения безопасности внутри организации

Для получения более подробных рекомендаций по усилению безопасности Kubernetes обратитесь к официальной документации и ресурсам сообщества. Имеет также смысл изучить комплексные рекомендации по защите Kubernetes, в частности руководство *Kubernetes Hardening Guidance*, подготовленное *Агентством оборонных информационных систем США* (DISA, Defense Information Systems Agency): https://media.defense.gov/2022/Aug/29/2003066362/-1/-1/0/CTR_KUBERNETES_HARDENING_GUIDANCE_1.2_20220829.PDF.

В следующем разделе мы рассмотрим распространенные методы устранения неполадок в Kubernetes.

Устранение неполадок в Kubernetes

Устранение неполадок в Kubernetes сводится к диагностике и решению проблем, влияющих на функциональность и стабильность кластера и приложений. Типичные проблемы могут быть связаны с планированием подов, сбоями в работе контейнеров, ошибками при скачивании образов, сетевыми неполадками или ограничениями ресурсов. Эффективное выявление и исправление таких проблем крайне важно для поддержания корректно функционирующей Kubernetes-среды.

В следующих подразделах мы рассмотрим ключевые навыки, необходимые для устранения неполадок в Kubernetes.

Получение сведений о ресурсах

При устранении неполадок в Kubernetes команды `kubectl get` и `kubectl describe` являются незаменимыми инструментами для диагностики и анализа состояния ресурсов в кластере. Вы уже неоднократно использовали эти команды в предыдущих главах, и здесь мы вновь обратимся к ним.

Команда `kubectl get` предоставляет обзорные сведения о различных ресурсах кластера, таких как поды, сервисы, развертывания и узлы. Например, если вы подозреваете, что под работает некорректно, вы можете выполнить команду `kubectl get pods`, чтобы вывести список всех подов с их текущими статусами. Эта команда покажет, находятся ли поды в состоянии `Running`, `Pending` или работают с ошибками, что поможет вам быстро выявить возможные проблемы.

Команда `kubectl describe` позволяет детально изучить конкретный ресурс. Она выводит подробную информацию о его конфигурации, связанных событиях и недавних изменениях. Например, если под `todo-app` работает с ошибками, команда `kubectl describe pod todo-app` предоставит подробные сведения о возможных причинах этого сбоя.

В выводе этой команды указаны события, связанные со сбойным подом, например неудачные попытки запуска контейнера или ошибки при загрузке образов. Кроме того, в нем содержатся подробные данные о конфигурации, такие как лимиты ресурсов и переменные окружения, что позволяет выявить некорректные настройки и другие источники проблем.

В качестве иллюстрации рассмотрим проблему с развертыванием. Выполнив команду `kubectl get deployments`, вы сможете узнать статус объекта Deployment и количество его реплик. Если этот объект завис или не обновляется должным образом, команда `kubectl describe deployment webapp` предоставит подробные сведения об истории выкатывания обновлений, условиях и ошибках, возникших в ходе этого процесса.

В следующем подразделе мы рассмотрим способы использования логов и событий для устранения неполадок в Kubernetes.

Использование логов и событий для устранения неполадок в Kubernetes

Kubernetes предоставляет такие мощные инструменты, как *события* и *аудит-логи* (журналы аудита), для эффективного мониторинга и защиты кластера. События — это ресурсы типа Event, доступные на уровне всего кластера. Они дают представление о ключевых действиях в реальном времени, в том числе о планировании подов, перезапуске контейнеров и возникающих ошибках. Это позволяет быстро диагностировать проблемы и определять текущее состояние кластера. Для просмотра событий используется команда:

```
$ kubectl get events
```

Ее вывод содержит хронологию событий, что облегчает выявление и устранение проблем. Чтобы сосредоточиться на определенных событиях, можно отфильтровать их по типу ресурса, пространству имен или периоду времени. Например, для просмотра событий, связанных с конкретным подом, используйте команду:

```
$ kubectl get events --field-selector involvedObject.name=todo-pod
```

Аудит-логи, представленные ресурсом типа Policy, играют ключевую роль в обеспечении безопасности среды Kubernetes и ее соответствия нормативным требованиям. Они фиксируют детальную историю всех запросов к API-серверу Kubernetes, включая сведения о пользователе, выполненном действии и его результате. Эта информация незаменима для отслеживания таких операций, как попытки входа в систему или повышения привилегий. Чтобы включить аудит-логирование, необходимо настроить соответствующую политику для API-сервера. Подробнее об этом можно узнать в официальной документации: https://kubernetes.io/docs/tasks/debug/debug-cluster/audit/.

При отладке Kubernetes-приложений команда `kubectl logs` является важнейшим инструментом для получения и анализа логов отдельных контейнеров внутри пода, помогающим эффективно выявлять и устранять проблемы.

Чтобы получить логи пода, используйте команду:

```
$ kubectl logs todo-app
```

Эта команда выводит логи первого контейнера в поде. Если под содержит несколько контейнеров, необходимо указать имя конкретного контейнера:

```
$ kubectl logs todo-app -c app-container
```

Для потоковой передачи логов в реальном времени (как при использовании команды `tail -f` в Linux) используйте флаг `-f`:

```
$ kubectl logs -f todo-app
```

Это весьма удобно для наблюдения за работающими процессами. Если под был перезапущен, логи его предыдущего экземпляра можно получить с помощью следующей команды:

```
$ kubectl logs todo-app --previous
```

Чтобы отфильтровать логи по меткам, комбинируйте `kubectl` с такими инструментами, как `jq`:

```
$ kubectl get pods -l todo -o json \
    | jq -r '.items[] | .metadata.name' \
    | xargs -I {} kubectl logs {}
```

Для эффективного управления логами и предотвращения переполнения диска важно обеспечить их ротацию, архивируя или удаляя старые записи по мере появления новых. Использование структурированных логов, например, в формате JSON упрощает их анализ и разбор с помощью инструментов вроде `jq`. Кроме того, внедрение централизованной системы логирования, например стека *EFK (Elasticsearch, Fluentd, Kibana)*, позволяет агрегировать и с легкостью находить логи любых компонентов кластера, получая целостное понимание поведения приложений.

Таким образом, события и аудит-логи предоставляют всесторонние возможности для мониторинга и обеспечения безопасности. События дают представление о состоянии и поведении приложений, а журналы аудита фиксируют все действия в кластере, помогая поддерживать безопасную и соответствующую требованиям среду.

Встроенный помощник kubectl explain

Команда `kubectl explain` — это мощный инструмент, который помогает разобраться в структуре и полях ресурсов Kubernetes. Она предоставляет подробную информацию о конкретных типах ресурсов, позволяя исследовать схему API прямо из командной строки. Это особенно полезно при написании или отладке YAML-манифестов, так как гарантирует использование корректных полей и структуры.

Например, чтобы изучить ресурс Pod, используйте команду:

```
$ kubectl explain pod
```

Эта команда выведет общее описание ресурса Pod с краткой характеристикой. Для детального изучения отдельных полей, например `spec`, можно расширить эту команду следующим образом:

```
$ kubectl explain pod.spec
```

Вывод будет содержать подробное описание поля `spec`, включая вложенные поля и ожидаемые типы данных, что поможет правильно сконфигурировать ресурсы Kubernetes.

Интерактивное устранение неполадок с помощью команды kubectl exec

Команда `kubectl exec` — это мощный инструмент для взаимодействия с работающими контейнерами в Kubernetes и устранения неполадок. Она позволяет выполнять команды непосредственно внутри контейнера, что делает ее незаменимой при отладке, проверке окружения контейнера и выполнении быстрых исправлений. Эта команда пригодится вам для просмотра логов, анализа конфигурационных файлов, диагностики сетевых проблем, то есть в любой ситуации, требующей взаимодействия с приложением в реальном времени.

Чтобы воспользоваться этим инструментом, начните с выполнения простой команды внутри контейнера (для тестирования можно использовать манифест `kubectl apply -f trouble/blog-portal.yaml`):

```
$ kubectl get po -n trouble-ns
NAME                    READY   STATUS    RESTARTS   AGE
blog-675df44d5-gkrt2    1/1     Running   0          29m
```

Для вывода переменных окружения контейнера выполните команду:

```
$ kubectl exec blog-675df44d5-gkrt2 -- env
```

Если в поде несколько контейнеров, укажите нужный с помощью флага `-c`:

```
$ kubectl exec blog-675df44d5-gkrt2 -c blog -- env
```

Одним из наиболее распространенных вариантов использования команды `kubectl exec` является запуск сеанса интерактивной оболочки внутри контейнера. Это позволяет выполнять диагностические команды, например просматривать журналы или изменять конфигурационные файлы, на лету. Чтобы запустить интерактивную оболочку (например, `/bin/sh` или `/bin/bash`), используйте следующую команду:

```
$ kubectl exec -it blog-675df44d5-gkrt2 -n trouble-ns -- /bin/bash
root@blog-675df44d5-gkrt2:/app# whoami; hostname; uptime
root
blog-675df44d5-gkrt2
14:36:03 up 10:19,  0 user,  load average: 0.17, 0.07, 0.69
root@blog-675df44d5-gkrt2:/app#
```

Здесь:

- `-i` означает интерактивный режим;
- `-t` выделяет псевдо-TTY.

Такой интерактивный сеанс особенно полезен, когда вам нужно исследовать окружение контейнера или выполнить серию команд для устранения неполадок.

Помимо выполнения команд, `kubectl exec` позволяет копировать файлы в контейнер и из контейнера с помощью команды `kubectl cp`. Это особенно удобно, когда требуется загрузить в контейнер сценарий или извлечь из него журнал для дальнейшего анализа. Например, чтобы скопировать файл с локальной машины в контейнер, используйте команду:

```
$ kubectl cp troubles/test.txt blog-675df44d5-gkrt2:/app/test.txt -n trouble-ns
$ kubectl exec -it blog-675df44d5-gkrt2 -n trouble-ns -- ls -l /app
total 8
-rw-r--r-- 1 root root  902 Aug 20 16:52 app.py
-rw-r--r-- 1 1000 1000   20 Aug 31 14:42 test.txt
```

А чтобы скопировать файл из контейнера в локальную систему, выполните команду:

```
$ kubectl cp blog-675df44d5-gkrt2:/app/app.py /tmp/app.py -n trouble-ns
```

Эта возможность значительно упрощает перенос файлов между вашей средой и контейнерами в Kubernetes-кластере, делая процесс диагностики и отладки более эффективным.

В следующем подразделе мы обсудим эфемерные контейнеры, которые играют важную роль в процессе устранения неполадок в Kubernetes.

Эфемерные контейнеры в Kubernetes

Эфемерные контейнеры — это особый тип контейнеров в Kubernetes, предназначенный для таких временных, выполняемых на лету задач, как отладка. В отличие от обычных контейнеров, которые рассчитаны на длительную работу внутри подов, эфемерные контейнеры применяются для инспекции и устранения неполадок. Они не перезапускаются автоматически и не имеют гарантированного набора ресурсов.

Такие контейнеры можно добавить в уже существующий под, чтобы упростить диагностику, и они являются особенно полезными в ситуациях, когда традиционные методы (например, `kubectl exec`) оказываются недоступными. К примеру, если под использует образ `distroless` без отладочных инструментов, вы можете добавить эфемерный контейнер, который предоставит оболочку и необходимые диагностические утилиты (например, `nslookup`, `curl`, `mysql client` и т. д.). Эфемерные контейнеры управляются отдельным API-обработчиком, поэтому их нельзя добавить через `kubectl edit` или изменить после создания.

Например, в главе 8 мы использовали образ `k8sutils (quay.io/iamgini/k8sutils:debian12)` в качестве отдельного пода для тестирования сервисов и выполнения других задач. С помощью эфемерных контейнеров мы можем использовать тот же образ, но внедрить контейнер прямо в рабочий под приложения для целей диагностики.

Допустим, у нас есть под и сервис `video-service`, развернутые в пространстве имен `ingress-demo` (см. файл `ingress/video-portal.yaml`). В этом случае можно начать отладку, запустив эфемерный контейнер на основе образа `k8sutils` с помощью команды:

```
$ kubectl debug -it pod/video-7d945d8c9f-wkxc5 \
    --image=quay.io/iamgini/k8sutils:debian12 \
    -c k8sutils \
    -n ingress-demo

root@video-7d945d8c9f-wkxc5:/# nslookup video-service
Server:         10.96.0.10
Address:        10.96.0.10#53

Name:   video-service.ingress-demo.svc.cluster.local
Address: 10.109.3.177

root@video-7d945d8c9f-wkxc5:/# curl http://video-service:8080

    <!DOCTYPE html>
    <html>
```

```
    <head>
      <title>Welcome</title>
      <style>
        body {
          background-color: yellow;
          text-align: center;
...<фрагмент опущен для краткости>...
```

Таким образом, эфемерные контейнеры позволяют исследовать состояние работающих подов, не изменяя существующую конфигурацию и не завися от ограничений базового контейнера.

В следующем подразделе мы рассмотрим распространенные проблемы и методы их решения.

Распространенные проблемы и методы их решения

Устранение неполадок в Kubernetes может оказаться весьма сложной задачей и сильно зависеть от конфигурации кластера и особенностей его работы, поскольку перечень возможных проблем весьма обширен. В этом подразделе мы сосредоточимся на наиболее распространенных проблемах, возникающих в Kubernetes, и способах их устранения.

- *Поды находятся в состоянии Pending.* Статус `Pending` означает, что под ожидает назначения конкретному узлу. Это может быть обусловлено недостатком ресурсов или ошибками в конфигурации. Для решения данной проблемы используйте команду `kubectl describe pod <имя_пода>`, чтобы просмотреть события, объясняющие причину ожидания, например ограничения ресурсов или состояние узлов. Если ресурсов в кластере недостаточно, под так и останется в состоянии ожидания. В этом случае можно скорректировать запросы ресурсов или добавить новые узлы. (Для тестирования этого сценария используйте файл `troubles/app-with-high-resource.yaml`.)
- *Ошибка CrashLoopBackOff или ошибки контейнера.* Ошибка `CrashLoopBackOff` возникает в тех случаях, когда контейнер не может успешно запуститься, например, из-за неправильных настроек, отсутствующих файлов или ошибок приложения. Для диагностики изучите логи с помощью команды `kubectl logs <имя_пода>` или выполните команду `kubectl describe pod <имя_пода>`, чтобы выявить причину проблемы. Обратите внимание на сообщения об ошибках или трассировки стека. Если в контейнере указана неверная команда запуска, он не сможет запуститься, что приведет к возникновению ошибки `CrashLoopBackOff`. Анализ кода выхода и логов контейнера позволит устранить проблему. (Для тестирования этого сценария используйте файл `troubles/failing-pod.yaml`.)
- *Сетевые проблемы.* Подобные ошибки обычно указывают на то, что сетевые политики блокируют трафик к поду или от него. Чтобы устранить эту

неполадку, изучите сетевые политики, влияющие на под, с помощью команды `kubectl describe pod <имя_пода>`, и проверьте конечные точки сервисов с помощью команды `kubectl get svc`. Слишком строгие сетевые политики могут блокировать необходимый трафик. Например, пустая Ingress-политика может полностью заблокировать входящий трафик; корректировка политики позволит сервисам взаимодействовать друг с другом. (Для тестирования этого сценария используйте файл `troubles/networkpolicy.yaml`.)

- *Узел не готов или недоступен.* Статус `NotReady` указывает на то, что узел не находится в рабочем состоянии, например, из-за сетевых проблем. Для выполнения диагностики проверьте статус узла с помощью команд `kubectl get nodes` и `kubectl describe node <имя_узла>`. Причиной этой проблемы может быть и ограничение (taint) узла, препятствующее назначению подов. Например, если для узла задано ограничение `NoSchedule`, он не будет принимать поды, пока проблема не будет устранена или ограничение не будет снято.
- *Проблемы с хранилищем.* Зависание объекта PersistentVolumeClaim в статусе `Pending` происходит в тех случаях, когда *PVC* (заявка на выделение постоянного тома) ожидает соответствующий *PV* (постоянный том) для привязки. Чтобы устранить проблему, проверьте статус PV и PVC с помощью команд `kubectl get pv` и `kubectl get pvc`. В случае с CSI убедитесь в том, что параметр `storageClass` настроен корректно и что в определении PVC запрошен именно этот класс хранилища. (Для тестирования этого сценария используйте файл `troubles/pvc.yaml`.)
- *Недоступность сервиса.* Ошибка `Service Unavailable` означает, что сервис недоступен, например, из-за некорректной настройки или сетевых проблем. Для устранения неполадки изучите сведения о сервисе с помощью команды `kubectl describe svc <имя_сервиса>`. Убедитесь, что сервис корректно сконфигурирован и ссылается на нужные поды с использованием правильных меток. Если сервис настроен неверно, он не сможет направлять трафик к ожидаемым конечным точкам, что приведет к его недоступности. Проверить конечные точки сервиса (поды) также можно с помощью команды `kubectl describe svc <имя_сервиса>`.
- *Проблемы с API-сервером или управляющим слоем.* Подобные ошибки обычно указывают на проблемы с подключением к API-серверу, вызванные неисправностями в управляющем слое или сетевыми неполадками. Поскольку команды `kubectl` не работают при недоступности API-сервера, необходимо подключиться напрямую к серверу управляющего слоя, где работают поды API-сервера. После входа в систему проверьте состояние компонентов управляющего слоя с помощью команды `crictl ps` (если используется среда `containerd`) или `docker ps` (если используется Docker) и удостоверьтесь в том, что под API-сервера запущен. Изучите логи и проверьте сетевые подключения, чтобы убедиться в корректном функционировании всех компонентов управляющего слоя.

- *Проблемы аутентификации и авторизации.* Ошибка `Unauthorized` указывает на неправильные учетные данные или недостаточные разрешения пользователя. Чтобы устранить эту неполадку, проверьте права пользователя с помощью команды `kubectl auth can-i <глагол> <ресурс>`. Например, если у пользователя отсутствует необходимая роль или объект RoleBinding, он столкнется с ошибками авторизации. Для решения проблемы скорректируйте роли и их привязки, предоставив необходимые разрешения.
- *Исчерпание ресурсов.* Ошибка `ResourceQuota Exceeded` возникает при превышении квоты ресурсов, что блокирует их дальнейшее выделение. Чтобы устранить проблему и контролировать потребление ресурсов, используйте команды `kubectl get quota`, `kubectl top nodes` и `kubectl top pods`. Если квота слишком мала, она будет препятствовать выделению новых ресурсов. Для решения этой проблемы увеличьте квоту или снизьте текущий уровень потребления ресурсов.
- *Проблемы с входящим трафиком или балансировщиком нагрузки.* Ошибка `IngressController Failed` указывает на некорректную работу Ingress-контроллера, что влияет на маршрутизацию входящего трафика. Для диагностики изучите сведения об объекте Ingress с помощью команды `kubectl describe ingress <имя_ingress>`. Убедитесь, что Ingress-контроллер корректно установлен и настроен, а правила Ingress правильно сопоставляют трафик с сервисами. Ошибки в этих правилах могут нарушить маршрутизацию трафика. Если в конфигурации Ingress используется необязательное поле `host`, убедитесь в корректном разрешении указанного в нем DNS-имени.

Резюме

В последней главе мы рассмотрели продвинутые методы маршрутизации трафика в Kubernetes с использованием объектов Ingress и Ingress-контроллеров. В самом начале мы освежили знания о сервисах типа `ClusterIP`, `NodePort` и `LoadBalancer`. Затем мы обсудили объекты Ingress и Ingress-контроллеры, а также их роль в маршрутизации трафика в Kubernetes. Теперь вы знаете, что обычные сервисы чаще всего применяются при балансировке нагрузки на уровне L4, но если ваше приложение использует конечные точки HTTP или HTTPS, предпочтительнее использовать L7-балансировку, реализуемую с помощью объектов Ingress и Ingress-контроллеров. Помимо этого, вы научились использовать веб-сервер Nginx в качестве Ingress-контроллера и протестировали его работу на конкретных примерах с объектами Deployment.

В завершение мы обсудили использование объектов Ingress и Ingress-контроллеров в облачных средах, предусматривающих нативную поддержку балансировки нагрузки на уровне L7. В качестве демонстрации мы развернули кластер

AKS с контроллером *AGIC* (Application Gateway Ingress Controller) для обработки объектов Ingress.

Вы также увидели, что система Kubernetes активно интегрирует современные технологии, такие как Knative и KubeVirt, которые расширяют ее функциональность в таких областях, как бессерверные вычисления, управление виртуальными машинами и машинное обучение. Кроме того, мы рассмотрели важнейшие операции второго дня эксплуатации, выполняемые администратором кластера, в том числе резервное копирование и обновление, а также базовые рекомендации по защите кластера и ключевые техники устранения распространенных неполадок. Опираясь на эти базовые принципы, инженеры могут безопасно и эффективно проектировать, обслуживать и защищать Kubernetes-среды, обеспечивая непрерывную работу своих инновационных решений.

Поздравляем! Вы проделали долгий путь по захватывающему миру Kubernetes и оркестрации контейнеров. Желаем вам удачи в дальнейшем изучении этой системы и благодарим за внимание.

Дополнительные источники информации

- Объекты Ingress: https://kubernetes.io/docs/concepts/services-networking/ingress/.
- Ingress-контроллеры: https://kubernetes.io/docs/concepts/services-networking/ingress-controllers/.
- Руководство по установке Ingress-контроллера: https://kubernetes.github.io/ingress-nginx/deploy.
- Настройка Ingress в среде Minikube с Ingress-контроллером Nginx: https://v1-32.docs.kubernetes.io/docs/tasks/access-application-cluster/ingress-minikube/.
- Эфемерные контейнеры: https://kubernetes.io/docs/concepts/workloads/pods/ephemeral-containers/.
- Обзор контроллера AGIC (Application Gateway Ingress Controller): https://learn.microsoft.com/en-us/azure/application-gateway/ingress-controller-overview.
- Работа с `etcd`-кластерами для Kubernetes: https://kubernetes.io/docs/tasks/administer-cluster/configure-upgrade-etcd/.
- Защита кластера: https://kubernetes.io/docs/tasks/administer-cluster/securing-a-cluster/.
- Аудит: https://kubernetes.io/docs/tasks/debug/debug-cluster/audit/.

Для получения дополнительной информации о Kubernetes вы можете обратиться к следующим книгам издательства Packt Publishing:

- *Baier J., Sayfan G., White J.* The Complete Kubernetes Guide (https://www.packtpub.com/en-in/product/the-complete-kubernetes-guide-9781838647346).
- *Baier J., White J.* Getting Started with Kubernetes. 3 ed. (https://www.packtpub.com/en-in/product/getting-started-with-kubernetes-9781788997263).

- *Heck J.* Kubernetes for Developers (https://www.packtpub.com/en-in/product/kubernetes-for-developers-9781788830607).
- *Tylenda P.* Hands-On Kubernetes on Windows (https://www.packtpub.com/product/hands-on-kubernetes-on-windows/9781838821562).

Кроме того, вы можете обратиться к официальной документации.

- Документация по Kubernetes (https://kubernetes.io/docs/home/) является наиболее актуальным источником информации об этой системе.
- Список доступных Ingress-контроллеров: https://kubernetes.io/docs/concepts/services-networking/ingress-controllers/.
- Встроенный управляемый Ingress-контроллер для GKE GKE Ingress: https://cloud.google.com/kubernetes-engine/docs/concepts/ingress (список реализованных функций Ingress в GKE: https://cloud.google.com/kubernetes-engine/docs/how-to/ingress-features) .
- AWS Load Balancer Controller для Amazon EKS: https://docs.aws.amazon.com/eks/latest/userguide/albingress.html.

Эту книгу мы выпустили вместе с Orion soft

Orion soft создает ПО для ИТ-инфраструктуры Enterprise-бизнеса. Главные принципы разработки, которым отвечает каждый продукт компании, — стабильность, простота использования и безопасность.

В продуктовую экосистему Orion soft входит zVirt — платформа виртуализации серверов, дисков и сетей с одной из крупнейших баз инсталляций в России.

Orion soft стремится постоянно развивать как свои компетенции, так и российское ИТ-сообщество, собирая и распространяя лучшие современные практики разработки.

Билджин Ибрам, Роланд Хасс

ПАТТЕРНЫ KUBERNETES: ШАБЛОНЫ РАЗРАБОТКИ ОБЛАЧНЫХ ПРИЛОЖЕНИЙ

2-е издание

«Паттерны Kubernetes» — это не очередное справочное руководство по настройке кластеров или описанию API. Это фундаментальный труд, который отвечает на самый главный вопрос: «Как эффективно проектировать и строить приложения, предназначенные для работы в Kubernetes?». Билджин Ибрам и Роланд Хасс систематизировали лучшие практики сообщества, преобразовав их в паттерны. Книга учит не просто использовать Kubernetes, а мыслить категориями платформы, правильно комбинируя примитивы для создания отказоустойчивых, масштабируемых и удобных в эксплуатации систем. Это must-read для любого разработчика, который хочет выйти за рамки простого деплоя контейнеров и овладеть всеми возможностями Kubernetes.

Каждый паттерн включает описание проблемы и решение с использованием Kubernetes. Все паттерны сопровождаются конкретными примерами кода. Это обновленное издание идеально подходит для разработчиков и архитекторов, которые знакомы с базовыми концепциями Kubernetes, но хотят узнать, как решать задачи проектирования с помощью проверенных паттернов.

SPRINT book

Микаэль Криф

TERRAFORM: ЛУЧШИЕ ПРАКТИКИ. ПОДГОТОВКА, ЗАПУСК И МАСШТАБИРОВАНИЕ ОБЛАЧНОЙ АРХИТЕКТУРЫ НА РЕАЛЬНЫХ ПРИМЕРАХ

С появлением Terraform, замечательного инструмента управления инфраструктурой как кодом (IaC), язык конфигурирования HashiCorp (HCL) изменил подход к определению и формированию инфраструктуры центров обработки данных. Узнайте, как Terraform упрощает управление сложной инфраструктурой.

Обновленное издание включает практические примеры развертывания инфраструктуры Azure, AWS и GCP с помощью Terraform. Углубитесь в изучение ручных и автоматизированных подходов к тестированию конфигураций Terraform и познакомьтесь с приемами создания сбалансированной, эффективной и универсальной инфраструктуры и управления ею с помощью модулей Terraform. Вы узнаете, как автоматизировать развертывание конфигурации Terraform с помощью непрерывной интеграции и непрерывной поставки (CI/CD).

Кроме того, в издание добавлено несколько новых глав, в которых описывается использование Terraform вместе с Docker и Kubernetes, рассматриваются сложные вопросы, связанные с практикой GitOps, и объясняется, как тестировать конфигурации Terraform с помощью различных инструментов и проверять их соответствие требованиям безопасности. В последней главе исследуются вопросы устранения распространенных проблем с Terraform и предлагаются решения для часто встречающихся ошибок.

Извлеките максимальную пользу из Terraform и организуйте эффективное управление инфраструктурой.

SPRINT book