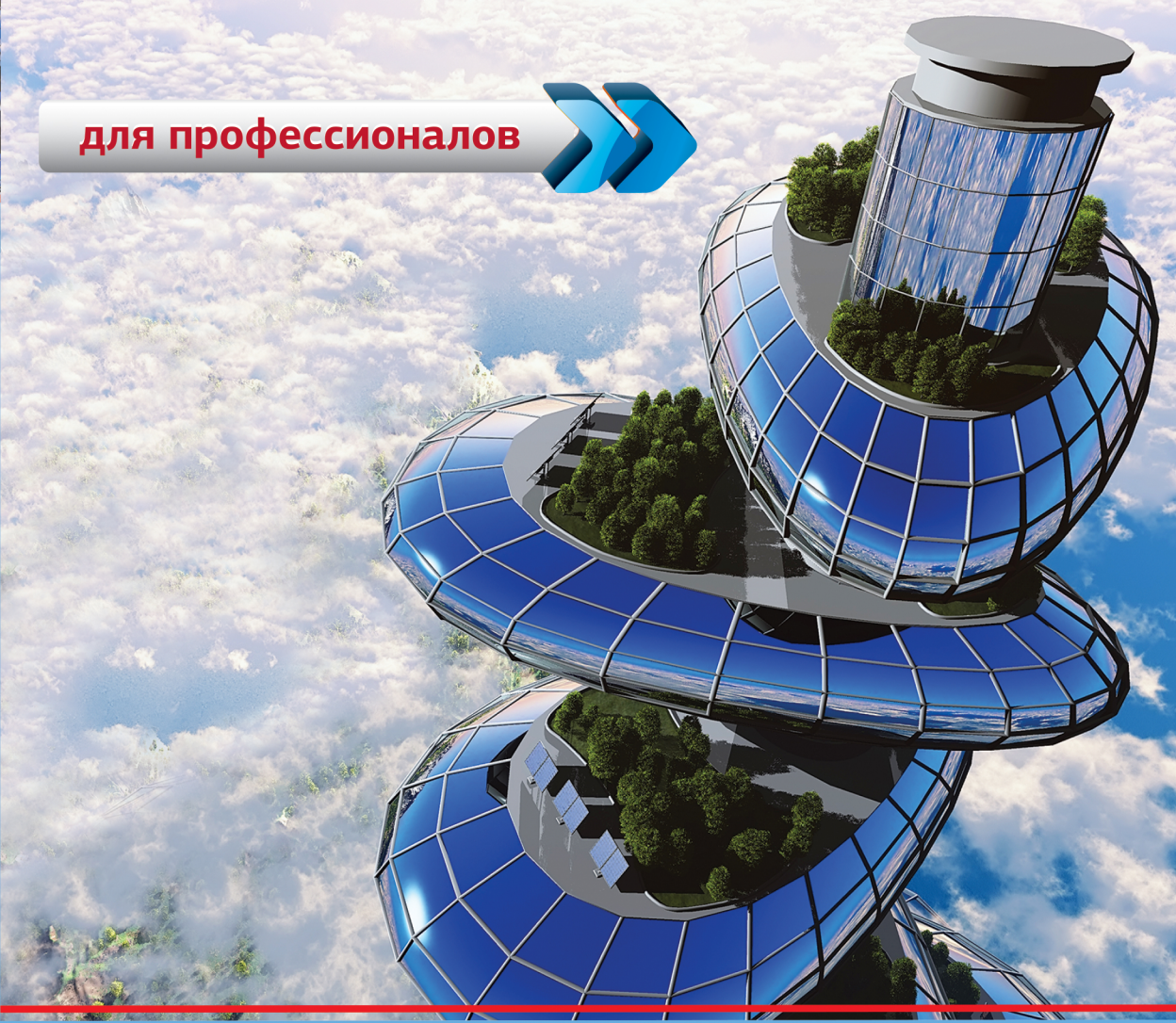
для профессионалов



Front-end

КЛИЕНТСКАЯ РАЗРАБОТКА

Node.js • ES6 • REST

Крис Аквино, Тодд Ганди

BIG NERD RANCH

Chris Aquino, Todd Gandee

Front-End Web Development:

The Big Nerd Ranch Guide

Крис Аквино, Тодд Ганди

Front-end

КЛИЕНТСКАЯ РАЗРАБОТКА для ПРОФЕССИОНАЛОВ

Node.js, ES6, REST

ПИТЕР®

Санкт-Петербург • Москва • Екатеринбург • Воронеж
Нижний Новгород • Ростов-на-Дону
Самара • Минск

2017

Крис Аквино, Тодд Ганди

Front-end. Клиентская разработка для профессионалов. Node.js, ES6, REST

Серия «Для профессионалов»

Перевел на русский *И. Пальти*

Заведующая редакцией	*Ю. Сергиенко*
Руководитель проекта	*О. Сивченко*
Ведущий редактор	*Н. Гринчик*
Литературный редактор	*О. Андросик*
Художники	*С. Заматевская*
Корректоры	*О. Андриевич, Е. Павлович*
Верстка	*А. Барцевич*

ББК 32.988-02-018
УДК 004.738.5

Аквино К., Ганди Т.

А38 Front-end. Клиентская разработка для профессионалов. Node.js, ES6, REST. — СПб.: Питер, 2017. — 512 с.: ил. — (Серия «Для профессионалов»).

ISBN 978-5-496-02930-8

В книге «Front-end. Клиентская разработка для профессионалов» рассмотрены все важнейшие навыки работы с JavaScript, HTML5 и CSS3, требующиеся серьезному разработчику, чтобы преуспеть в создании современного клиентского кода. Читатель быстро освоится с новыми инструментами и технологиями, с проверенными практиками, которые актуальны уже сегодня. В каждой главе рассматриваются важнейшие концепции и API, неотделимые от качественной веб-разработки, тщательно проверенные и отточенные в процессе решения реальных практических задач.

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

ISBN 978-0134433943 англ.
ISBN 978-5-496-02930-8

Изготовлено в России. Изготовитель: ООО «Питер Пресс». Место нахождения и фактический адрес: 192102, Россия, город Санкт-Петербург, улица Андреевская, дом 3, литер А, помещение 7Н. Тел.: +78127037373.

Дата изготовления: 05.2017. Наименование: книжная продукция. Срок годности: не ограничен.

Налоговая льгота — общероссийский классификатор продукции ОК 034-2014, 58.11.12 — Книги печатные профессиональные, технические и научные.

Подписано в печать 28.04.17. Формат 70×100/16. Бумага офсетная. Усл. п. л. 41,280. Тираж 1000. Заказ 0000.

Отпечатано в ОАО «Первая Образцовая типография». Филиал «Чеховский Печатный Двор».
142300, Московская область, г. Чехов, ул. Полиграфистов, 1.
Сайт: www.chpk.ru. E-mail: marketing@chpk.ru. Факс: 8(496) 726-54-10, телефон: (495) 988-63-87

Маме и папе, купившим нам первый компьютер. Дэйву и Гленн, позволившим младшему брату полностью его оккупировать. И Анжеле, благодаря которой у меня есть жизнь за пределами монитора.

К. А.

Моим маме и папе в благодарность за предоставленную возможность самому искать свой собственный путь. И спасибо моей жене за то, что она полюбила фаната компьютеров.

Т. Г.

Оглавление

Часть IV. Архитектура приложения

Благодарности

Как авторам нам принадлежит заслуга написания текста и создания иллюстраций (да, нам!). Но правда в том, что мы до сих пор сидели бы перед чистым листом бумаги, если бы не усилия целой армии помощников, соавторов и советчиков.

Мы искренне благодарим:

- Аарона Хиллегаса — за веру в нашу способность создать труд, достойный имени Big Nerd Ranch. Спасибо тебе за твою безмерную веру и поддержку;
- Матта Матиаса — за руководство процессом создания этой книги, особенно на последнем, самом ответственном этапе. Ты сделал все, чтобы время, которое могло быть потрачено на просмотр видеороликов про котов или повторных показов серий «Аббатства Даунтон», было посвящено написанию книги;
- Брэнди Поттер — за заботу и за то, что подкармливала авторов. Ты незаметно так организовала нашу работу, что написание этой книги не растянулось на долгие годы. Спасибо тебе;
- Джонатана Мартину — одного из наших учителей и специалиста по языкам программирования. Спасибо за ваш энтузиазм в преподавании курса, на котором основана эта книга, и за содержательную критику на этапе редактирования;
- наших корректоров, технических рецензентов и подопытных кроликов (Майк Зорнек, Джереми Шерман, Джош Джастис, Джейсон Риис, Гарри Смит, Эндрю Джонс, Стивен Кристофер и Билл Филипс). Спасибо вам за вашу работу на общественных началах;
- Элизабет Холидей — нашего мегатерпеливого редактора. Спасибо вам за то, что помогли нам избежать эффекта эхо-камеры и всегда напоминали о наших читателях;
- Элли Волькхаузен, разработавшую обложку;
- Симону Пэймент — нашего корректора, обеспечившего единообразие стиля книги;
- Криса Луперу из компании IntelligentEnglish.com, спроектировавшего и создавшего печатную и электронную версии данной книги. Его набор программных средств DocBook очень облегчил нам жизнь.

Наконец, спасибо всем студентам, посещавшим наши недельные курсы. Без вашей любознательности и ваших вопросов не было бы и этой книги. Надеемся, что выдры сделали прохождение курсов немного более интересным.

Введение

Разработка веб-приложений клиентской части

Разработка веб-приложений клиентской части сильно отличается от разработки для других платформ. Вот несколько моментов, которые желательно помнить при изучении этой темы.

Браузер — это платформа.

Возможно, у вас есть опыт разработки нативных приложений для мобильных операционных систем iOS или Android, написания серверного кода на языках программирования Rubi или PHP либо создания традиционных приложений для операционных систем OS X или Windows. Ваш код как разработчика клиентской части будет ориентирован на браузер — платформу, доступную практически на любом телефоне, планшете и ПК.

Разработка приложений клиентской части очень многогранна.

С одной стороны, это вид веб-страницы и пользовательские ощущения от нее: скругленные углы, тени, цвета, шрифты, пробелы и т. д. С другой — логика, управляющая замысловатым поведением веб-страницы: сменой изображений в интерактивной фотогалерее, проверкой вводимых в форму данных, отправкой сообщений в чате и т. д. Вам придется в совершенстве изучить основные технологии, причем часто потребуется использовать их в комплексе, чтобы создать хорошее веб-приложение.

Веб-технологии открыты.

Не существует какой-то компании, контролирующей работу браузеров. Это значит, что разработчики клиентской части не получают ежегодной версии SDK, содержащей все изменения, с которыми им придется иметь дело на протяжении следующего года. Нативные платформы можно сравнить с замерзшим прудом, а Интернет — с рекой: изгибающейся, быстро текущей и местами каменистой, но в этом отчасти и заключается его притягательность. Поэтому разработчик клиентской части должен уметь приспосабливаться к изменениям.

Цель данной книги — научить вас разрабатывать приложения для браузера. По мере чтения этого руководства вы создадите несколько проектов. Каждый из них будет требовать различных технологий клиентской части. Инструментов,

библиотек и фреймворков для разработки клиентской части огромное количество, поэтому в книге мы сконцентрируемся только на самых важных и переносимых паттернах и методах.

Необходимые знания

Эта книга не введение в программирование. Мы предполагаем, что у вас есть представление об основах написания программного кода и что вы знакомы с основными типами данных, функциями и объектами.

Несмотря на это, знать JavaScript не обязательно. В книге мы по мере необходимости познакомим вас с соответствующими концепциями JavaScript.

Структура книги

Изучая эту книгу, вы напишете четыре веб-приложения. Каждому приложению посвящена отдельная часть книги, а каждая глава добавляет в создаваемое приложение новые функциональные возможности.

Создание этих четырех веб-приложений даст вам возможность изучить все технологии, требуемые для создания клиентской части.

- *Ottergram*. Наш первый проект посвящен веб-фотогалерее. Создание Ottergram научит вас основам программирования для браузеров с помощью языка разметки HTML, таблиц стилей CSS и языка программирования JavaScript. Вы вручную создадите пользовательский интерфейс и узнаете, как браузер загружает и визуализирует контент.
- *CoffeeRun*. Частично форма заказа кофе, частично — список заказов. CoffeeRun познакомит вас с множеством методов языка программирования JavaScript, включая написание модульного кода, использование преимуществ замыканий и взаимодействие с удаленным сервером с помощью технологии Ajax.
- *Chattrbox*. Часть, описывающая приложение Chattrbox, — самая короткая, и это приложение больше всего отличается от остальных. В нем будет использоваться язык программирования JavaScript для создания системы общения в Интернете, включая написание сервера чата с помощью платформы Node.js, а также браузерного клиента для чата.
- *Tracker*. Последний проект использует Ember.js — один из самых функциональных фреймворков для разработки клиентской части. Мы напишем приложение для каталогизации случаев наблюдения редких, экзотических и мифических существ. По ходу дела вы узнаете, как использовать возможности богатейшей экосистемы, лежащей в основе фреймворка Ember.js.

По мере создания этих приложений вы познакомитесь с множеством инструментов, включая:

- текстовый редактор Atom и некоторые полезные плагины для работы с кодом;
- источники документации, например Mozilla Developer Network;
- командную строку с использованием приложения терминала OS X или командной строки Windows;
- утилиту browser-sync;
- инструменты разработчика браузера Google Chrome (Google Chrome's Developer Tools);
- файл `normalize.css`;
- фреймворк Bootstrap;
- библиотеки jQuery, crypto-js и moment;
- платформу Node.js, систему управления пакетами Node (npm) и модуль nodemon;
- протокол WebSockets и модуль wscat;
- компилятор Babel и модули Babelify, Browserify и Watchify;
- фреймворк Ember.js и такие дополнения к нему, как интерфейс командной строки Ember CLI, плагин для Chrome Ember Inspector, дополнение Ember CLI Mirage и шаблонизатор Handlebars;
- систему управления пакетами Bower;
- систему управления пакетами Homebrew;
- утилиту Watchman.

Как пользоваться книгой

Эта книга не справочник. Ее цель — помочь вам преодолеть первоначальные трудности и достичь уровня, когда вы сможете сами извлекать всю нужную вам информацию из доступных справочников. Она основана на пятидневном курсе лекций, который мы читали в компании Big Nerd Ranch, и предназначена для проработки на практике от начала и до конца. Главы зависят одна от другой, и не стоит пропускать какие-либо из них.

На наших занятиях студенты прорабатывали изложенный в книге материал и, смеем надеяться, получали удовольствие от посещения лекций, ведь к их услугам специально оборудованный класс, чай и печенье, удобная классная доска, группа заинтересованных в предмете однокурсников и преподаватель, который может ответить на все возникшие вопросы.

Вам не помешает похожая обстановка. Она гарантирует крепкий ночной сон и спокойное место для работы. Может помочь и следующее:

- создайте группу читателей из ваших друзей или сотрудников;
- выделите время для сосредоточенной работы над главами;
- примите участие в форуме этой книги на сайте forums.bignerdranch.com, где вы сможете обсудить прочитанное, найти ошибки и решения проблем;
- найдите кого-то, кто хорошо разбирается в разработке веб-приложений клиентской части, чтобы было к кому обратиться за помощью.

Упражнения

Большинство глав этой книги содержат упражнения. Это возможность проанализировать изученное и продвинуться на шаг дальше в своей работе. Мы рекомендуем вам выполнить столько упражнений, сколько сумеете. Это позволит закрепить полученные знания и перейти от *изучения* разработки на JavaScript к самостоятельному *выполнению* разработки на JavaScript.

Задания имеют три уровня сложности:

- бронзовые упражнения обычно требуют от вас выполнения чего-то очень похожего на то, что рассматривалось в главе. Они закрепляют приведенную информацию и предлагают писать код, не заглядывая в справочник. Как говорится, повторение — мать учения;
- серебряные упражнения требуют большего углубления в тему. Иногда вам придется пользоваться функциями, событиями, разметкой и стилями, с которыми вы еще не знакомы, но задачи все равно похожи на те, что рассматривались в главе;
- золотые упражнения сложны и отнимут немало времени на выполнение. Они потребуют от вас понимания идей данной главы и последующего серьезного обдумывания и решения задач своими силами. Зато это подготовит вас к реальной работе по написанию кода на JavaScript.

Рекомендуем сохранить копию вашего кода, прежде чем приступать к работе над заданиями любой главы. В противном случае внесенные вами изменения могут оказаться несовместимыми с последующими упражнениями.

Если у вас возникнут проблемы, вы всегда можете обратиться за помощью на forums.bignerdranch.com.

Для самых любознательных

Во многих главах есть разделы «Для самых любознательных», в которых приводятся более подробные пояснения или дополнительная информация по изучаемой теме. Ее, конечно, нельзя назвать абсолютно необходимой, но мы надеемся, что она покажется вам интересной и полезной.

Условные обозначения

В книге используются следующие условные обозначения.

Курсив

Применяется для обозначения новых понятий.

`Моноширинный шрифт`

Используется для текста листингов. Кроме того, применяется внутри абзацев для обозначения имен файлов и каталогов, а также для выделения таких элементов программ, как имена переменных, функций, тегов, атрибутов, значений, типов данных и т. д.

`Моноширинный полужирный шрифт`

Показывает команды или другой текст, который нужно набрать в таком же виде, в каком он приводится в книге.

~~`Моноширинный зачеркнутый шрифт`~~

Обозначает листинг, который нужно удалить в вашем коде.

`Моноширинный шрифт на сером фоне`

Показывает листинг, на который нужно обратить особое внимание.

`Моноширинный курсивный шрифт`

Показывает команды или другой текст, который должен быть заменен значением, предоставляемым пользователем или определяемым контекстом.

Шрифт без засечек

Используется для обозначения URL-адресов, названий пунктов меню, кнопок и т. д.

Часть I. Основы программирования приложений для браузеров

1 Настройка среды разработки

Инструментов и ресурсов для разработки клиентской части существует немало, причем все время появляются новые. Выбор наилучших из них — непростая задача для разработчика с любым уровнем опытности. По мере работы с проектами в данной книге мы научим вас пользоваться некоторыми из наших любимых инструментов.

Для начала понадобятся три основных инструмента: браузер, текстовый редактор и хорошая справочная документация для используемых при разработке клиентской части технологий. Пригодятся также несколько дополнительных инструментов, которые не являются предметами первой необходимости, но все же сделают разработку более приятной и удобной.

При работе с книгой мы рекомендуем вам использовать то же программное обеспечение, что и мы, чтобы извлечь максимум пользы из наших инструкций и снимков экранов. Данная глава освещает процесс установки и настройки браузера Google Chrome, текстового редактора Atom, платформы Node.js и нескольких плагинов и дополнительных инструментов. Вы также узнаете, где найти хорошую документацию, и пройдете ускоренный курс по использованию командной строки в операционной системе Windows и на компьютерах Mac. В следующей главе воспользуемся всеми этими ресурсами в нашем первом проекте.

Установка Google Chrome

На вашей машине уже должен быть установленный по умолчанию браузер, но лучше всего использовать для разработки клиентской части браузер Google Chrome. Если у вас пока еще нет свежей версии Chrome, вы можете скачать ее на сайте www.google.com/chrome/browser/desktop (рис. 1.1).

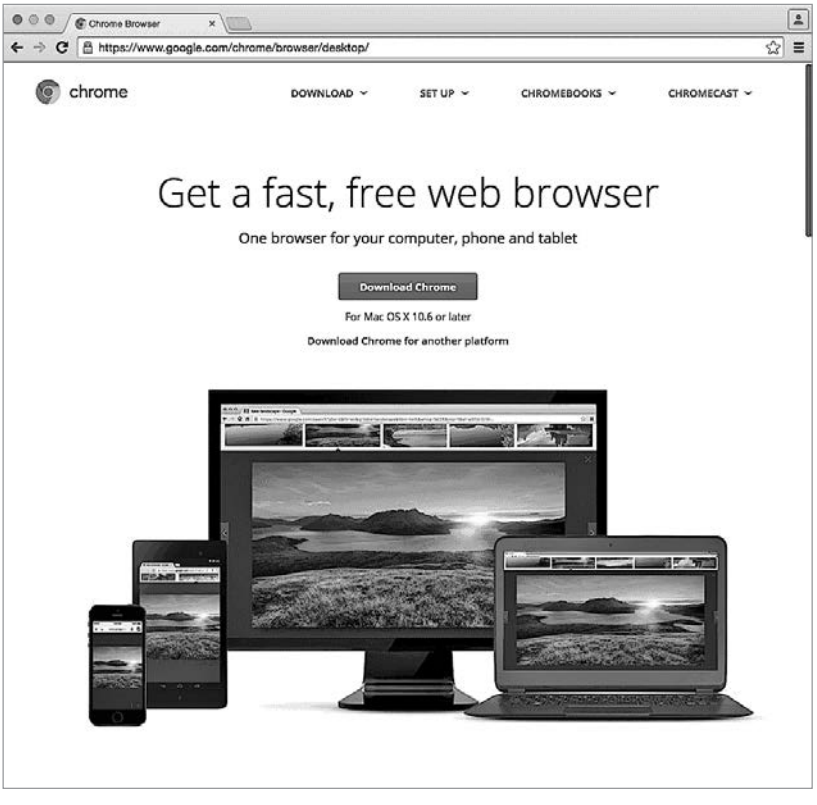


Рис. 1.1. Скачивание Google Chrome

Установка и настройка Atom

Один из лучших текстовых редакторов для разработки клиентской части — Atom от GitHub. Он предоставляет широкие возможности изменения конфигурации, для него существуют многочисленные плагины, облегчающие написание кода, к тому же скачать его можно бесплатно.

Скачать редактор Atom для Windows или Mac можно с сайта atom.io (рис. 1.2).

Следуйте инструкциям по установке для вашей платформы. После установки Atom вы, вероятно, захотите установить для него несколько плагинов.

Плагины для редактора Atom. Основное, для чего нам нужен текстовый редактор, — поиск по документации, автодополнение и линтинг кода (через минуту мы расскажем об этом подробнее). Atom предоставляет некоторые из этих возможностей по умолчанию, но установка парочки плагинов заметно улучшит положение дел.

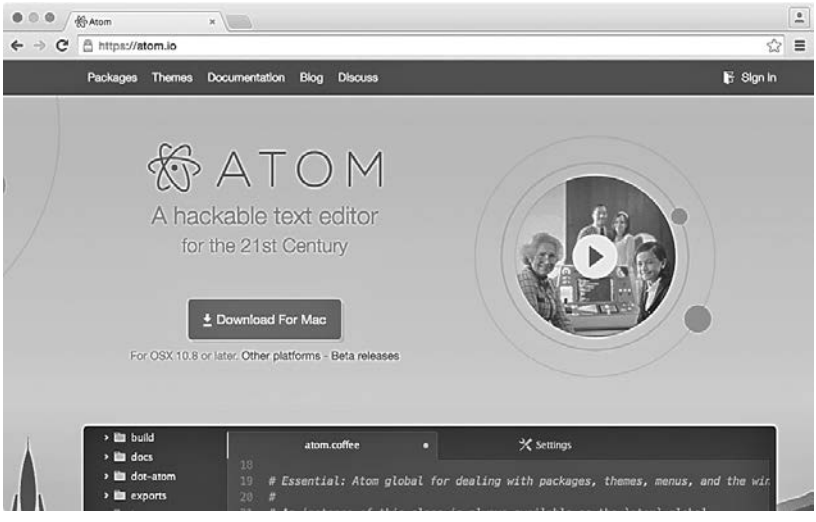


Рис. 1.2. Скачивание Atom

Откройте Atom и перейдите на экран Settings (Настройки). На Mac это можно сделать, выбрав Atom ▸ Preferences (Atom ▸ Установки) или нажав сочетание горячих клавиш Command+, (то есть клавиша Command плюс запятая). В Windows можно получить доступ через File ▸ Settings (Файл ▸ Настройки) или нажатием сочетания горячих клавиш Ctrl+,.

В левой части экрана Settings (Настройки) нажмите + Install (Установить) (рис. 1.3).

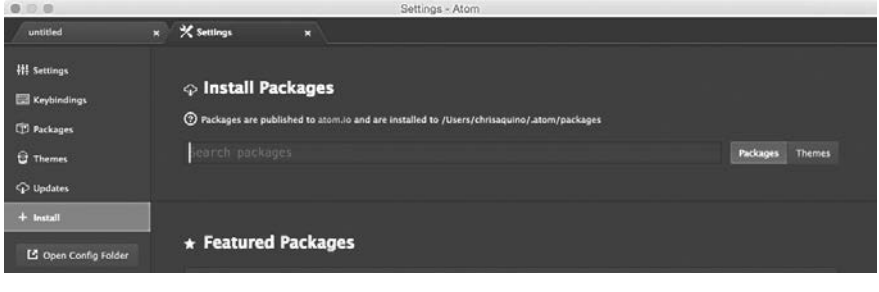


Рис. 1.3. Экран Install Packages (Установка пакетов) редактора Atom

Здесь можно выполнить поиск пакетов плагинов по названию. Начнем с поиска по запросу `emmet`.

Написание большого количества кода HTML может быть утомительным и чревато ошибками. Плагин emmet (рис. 1.4) позволяет писать хорошо форматированный HTML с помощью удобной сокращенной записи. Нажмите кнопку Install (Установить) для установки emmet.

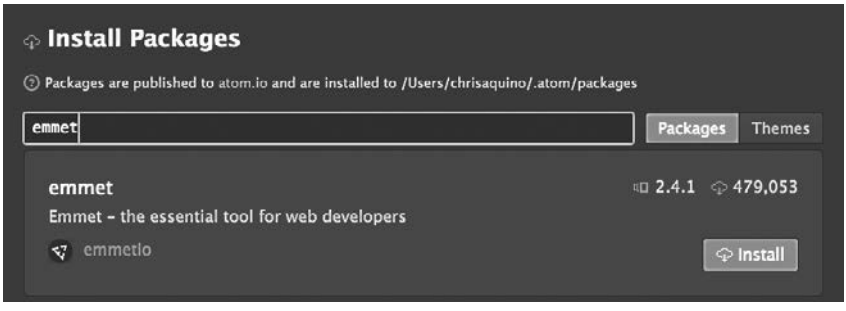


Рис. 1.4. Установка emmet

Далее выполните поиск по запросу `atom-beautify`. Плагин atom-beautify (рис. 1.5) помогает расставлять в коде отступы, которые повышают удобочитаемость. Нажмите кнопку Install (Установить) для установки плагина.

Рис. 1.5. Установка atom-beautify

Найдите и установите плагин autocomplete-paths (рис. 1.6). Очень часто приходится ссылаться в коде на другие файлы и каталоги проекта. Этот плагин предлагает имена файлов в меню автодополнения по мере набора текста.

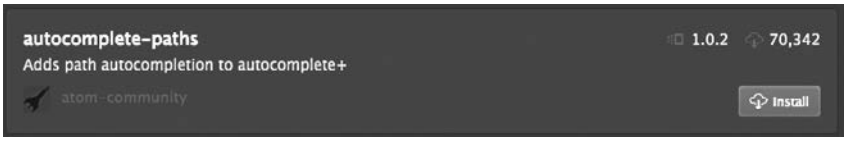


Рис. 1.6. Установка autocomplete-paths

Следующий плагин, который нужно установить, — пакет api-docs (рис. 1.7). Он даст возможность выполнять поиск в документации по ключевому слову. Плагин отображает документацию в отдельном окне в редакторе.

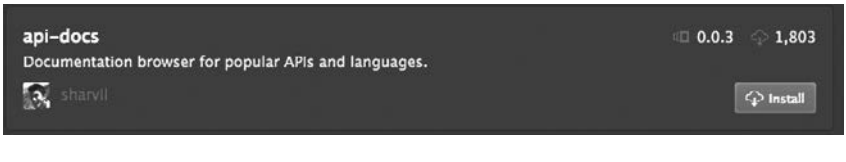


Рис. 1.7. Установка пакета api-docs

Далее найдите и установите пакет linter (рис. 1.8). *Линтер* — программа, проверяющая синтаксис и стиль кода. Убедитесь, что вы нашли и установили пакет, который называется просто linter. Это основной линтер, работающий с плагинами для конкретных языков. Он понадобится, чтобы использовать другие плагины-линтеры, указанные ниже.

Рис. 1.8. Установка пакета linter

Существует три дополнения для linter, которые пригодятся для проверки вашего кода, написанного на CSS, HTML и JavaScript. Начнем с дополнения linter-csslint. Оно обеспечивает синтаксическую корректность CSS и предлагает советы по написанию производительного CSS-кода.

Рис. 1.9. Установка linter-csslint

Следующий дополнительный плагин для linter — linter-htmlhint (рис. 1.10), который подтверждает, что код HTML правильно сформирован. В случае рассогласования тегов HTML он выведет предупреждение.

Рис. 1.10. Установка linter-htmlhint

Последний дополнительный плагин для linter — linter-eslint (рис. 1.11). Он проверяет синтаксис кода JavaScript и может быть настроен для проверки стиля и форматирования кода (например, количества пробелов в отступах строк или количества пустых строк до и после комментариев).

Теперь браузер Chrome и редактор Atom готовы для разработки клиентской части. Осталось организовать доступ к документации, разобраться с основами командной строки и скачать еще два инструмента.

Рис. 1.11. Установка linter-eslint

Где найти документацию и справочную информацию

Разработка клиентской части отличается от программирования для таких платформ, как iOS и Android. Помимо описания очевидных различий для технологий клиентской части, не существует официальной документации разработчика, не считая технических спецификаций. Следовательно, вам нужно будет где-то искать инструкции. Мы советуем ознакомиться с приведенными ниже ресурсами и обращаться к ним регулярно по мере работы с данной книгой и в дальнейшем при разработке клиентской части.

Сеть разработчиков Mozilla (Mozilla Developer Network, MDN) — наилучший справочник по всему, что относится к языкам HTML, CSS и JavaScript. Можно получить к ней доступ через devdocs.io (замечательный интерфейс для работы с документацией) (рис. 1.12). Он извлекает документацию из MDN для основных технологий клиентской части, причем может работать в режиме офлайн, так что вы сможете обратиться к нему даже при отсутствии подключения к Интернету.

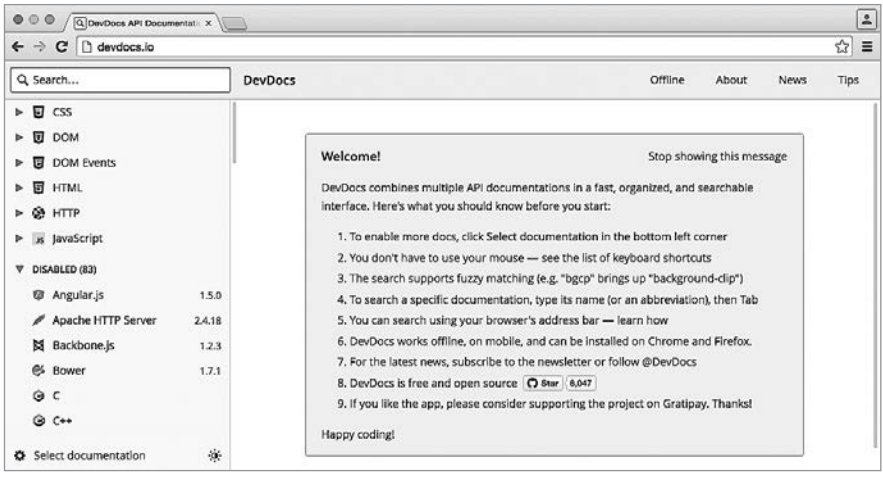


Рис. 1.12. Доступ к документации через devdocs.io

Обратите внимание, что браузер Safari пока не поддерживает механизм кэширования, используемый devdocs.io. Для доступа к нему вам понадобится другой браузер, например Chrome.

Можно также использовать сайт MDN, developer.mozilla.org/en-US (рис. 1.13), или добавить MDN в запрос поисковой системы для нахождения нужной информации.

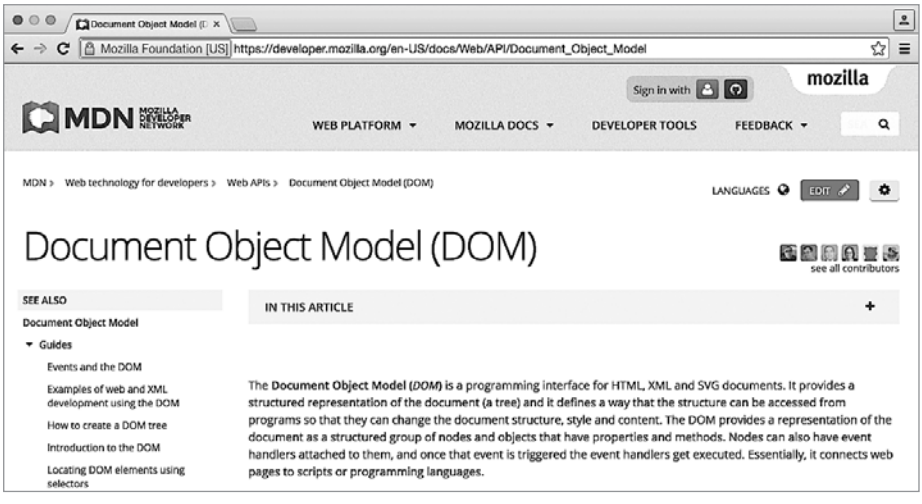


Рис. 1.13. Сайт Mozilla Developer Network

Другой полезный сайт — stackoverflow.com (рис. 1.14). Официально он не является источником документации. Это место, где разработчики могут задавать друг другу вопросы насчет кода. Качество ответов разное, но зачастую они вполне исчерпывающие и весьма полезные. Так что, когда будете пользоваться этим ресурсом, не забывайте, что написанное там вовсе не истина в последней инстанции (из-за краудсорсинговой природы).

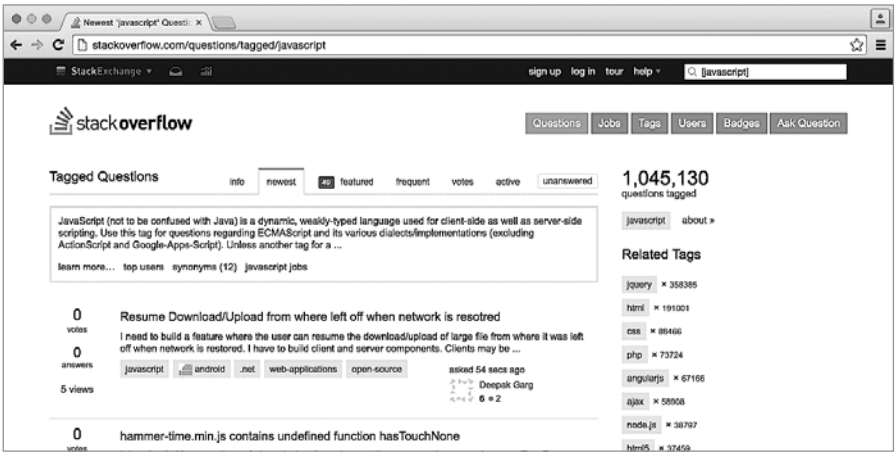


Рис. 1.14. Сайт Stack Overflow

Веб-технологии постоянно обновляются. Поддержка различных возможностей и API варьируется от браузера к браузеру и меняется с течением времени. Выяснить, какие браузеры (или версии конкретных браузеров) поддерживают какие возможности, вам помогут два сайта: html5please.com и caniuse.com. Если вам необходима информация о поддержке конкретной возможности, советуем начать с html5please.com, чтобы узнать, рекомендуется ли вообще использовать данную возможность. Для получения более подробных сведений о том, какие версии браузеров поддерживают конкретную возможность, обратитесь к caniuse.com.

Ускоренный курс по использованию командной строки

В этой книге мы будем предлагать вам использовать *командную строку*, или *терминал*, потому что многие инструменты запускаются исключительно через командную строку.

Чтобы получить доступ к командной строке на компьютерах Mac, откройте Finder и перейдите в каталог Applications (Приложения), а затем в папку Utilities (Утилиты). Найдите и запустите программу Terminal (рис. 1.15).

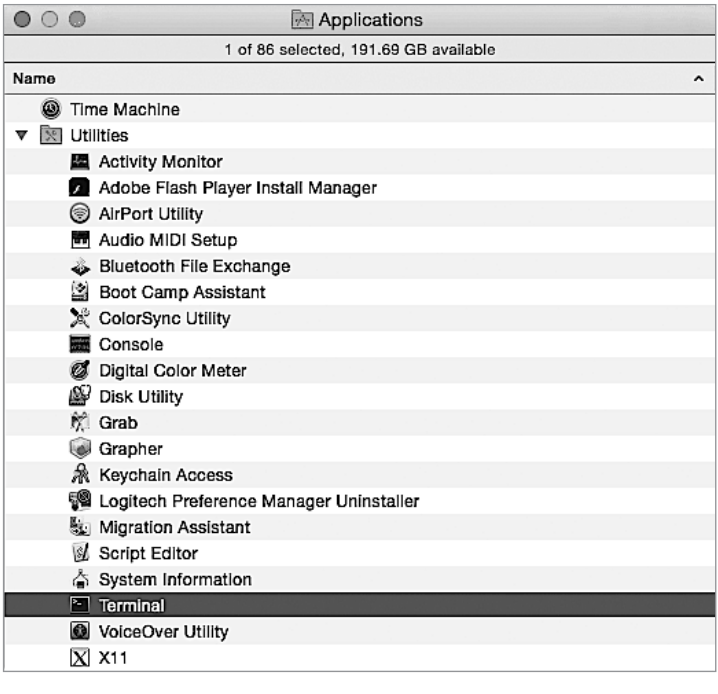


Рис. 1.15. Нахождение приложения Terminal на Mac

Вы увидите такое же окно, как на рис. 1.16.

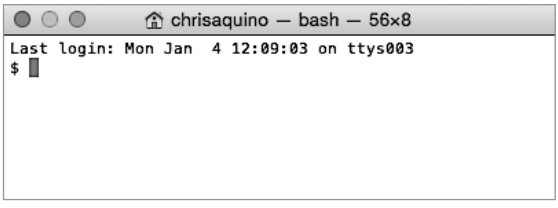


Рис. 1.16. Командная строка Mac

Для доступа к командной строке в операционной системе Windows перейдите в меню Start (Пуск) и выполните поиск по запросу `cmd`. Найдите и запустите программу Command Prompt (Командная строка) (рис. 1.17).

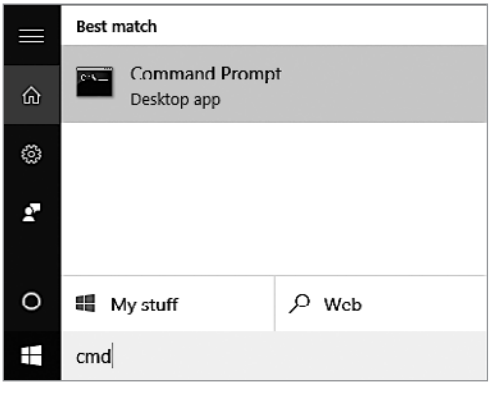


Рис. 1.17. Нахождение программы Command Prompt (Командная строка) в Windows

Щелкните на ней для запуска стандартного интерфейса командной строки Windows (выглядит как на рис. 1.18).

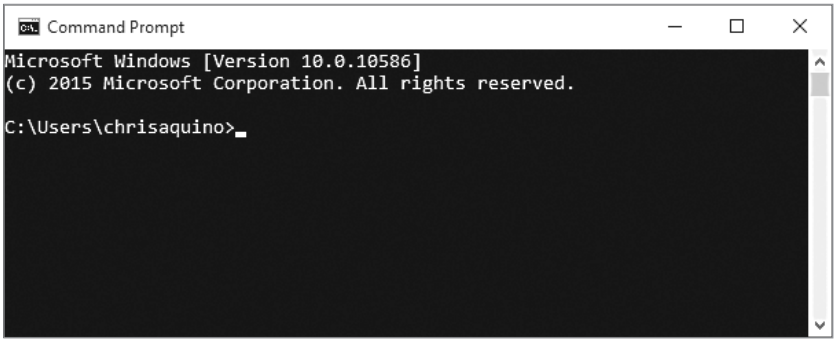


Рис. 1.18. Командная строка Windows

С этого момента мы будем называть терминалом или командной строкой и терминал Mac, и командную строку Windows. Если вы не знаете, как работать с командной строкой, далее приведен краткий обзор наиболее распространенных задач. Для ввода команды наберите ее в строке приглашения и нажмите клавишу Enter.

Выясняем, в каком каталоге мы находимся

Командная строка зависит от местоположения. Это значит, что в каждый конкретный момент времени она находится в конкретном каталоге файловой структуры и любые вводимые команды будут выполняться внутри этого каталога. Приглашение командной строки отображает сокращенную версию каталога, в котором она находится. Чтобы увидеть полный путь на Mac, введите команду `pwd` (сокращение от print working directory — «вывести текущий каталог»), как показано на рис. 1.19.

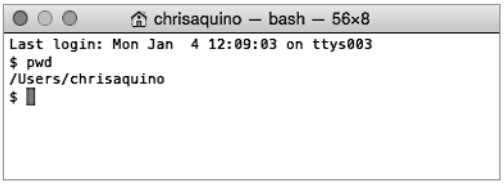


Рис. 1.19. Отображение текущего пути на Mac с помощью команды pwd

В Windows, чтобы увидеть путь, воспользуйтесь командой `echo %cd%` (рис. 1.20).

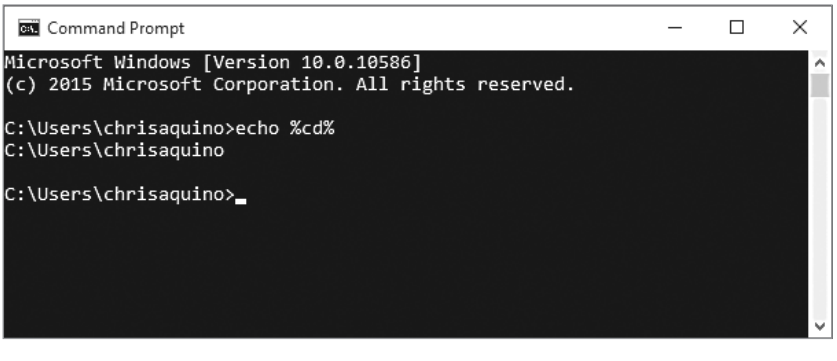


Рис. 1.20. Отображение текущего пути в Windows с помощью команды echo %cd%

Создание каталога

Структура каталогов проектов клиентской части очень важна. Ваши проекты могут быстро увеличиваться в размерах, и лучше с самого начала правильно их организовывать. Во время разработки вы регулярно будете создавать новые каталоги.

Это выполняется с помощью команды `mkdir` (make directory — «создать каталог»), за которой следует название нового каталога.

Чтобы увидеть эту команду в действии, сделаем каталог для проектов, которые будем создавать при работе с книгой. Введите следующую команду:

```
mkdir front-end-dev-book
```

Далее создадим новый каталог для нашего первого проекта Ottergram, к которому мы приступим в следующей главе. Необходимо, чтобы данный каталог был подкаталогом только что созданного front-end-dev-book. Это можно выполнить из нашей домашней папки, предварив название нового каталога названием папки с проектами и слешем (косой чертой) (на Mac):

```
mkdir front-end-dev-book/ottergram
```

В Windows вместо прямого нужно использовать обратный слеш:

```
mkdir front-end-dev-book\ottergram
```

Смена каталога

Для перемещения по файловой структуре предназначена команда `cd` (change directory — «сменить каталог») с указанием пути к каталогу, в который необходимо перейти.

Вам не обязательно использовать в команде `cd` полный путь к каталогу. Например, для перехода в любой подкаталог того каталога, в котором вы находитесь, достаточно указать имя этого подкаталога. То есть, если вы находитесь в каталоге `front-end-dev-book`, путь к каталогу `ottergram` — просто `ottergram`.

Перейдите в наш новый каталог проекта:

```
cd front-end-dev-book
```

Теперь можно перейти в папку `ottergram`:

```
cd ottergram
```

Чтобы вернуться в родительский каталог, используйте команду `cd ..` (за `cd` следуют пробел и две точки, означающие путь родительского каталога):

```
cd ..
```

Не забудьте, что узнать текущий каталог можно с помощью команды `pwd` (или `echo %cd%` в Windows). Рисунок 1.21 демонстрирует, как автор создает каталоги, перемещается по ним и узнает текущий каталог.

Вы можете за один раз переходить более чем на один каталог вверх и вниз. Пусть наша структура каталогов будет более сложная, например такая, как на рис. 1.22.

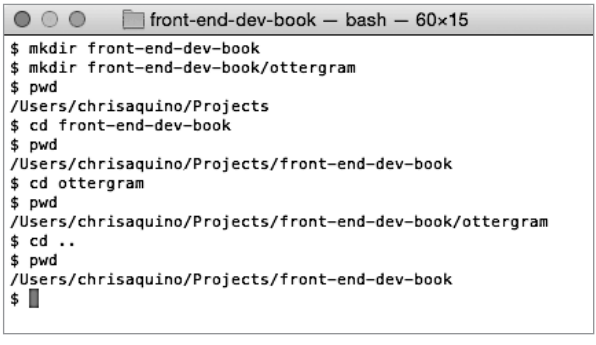


Рис. 1.21. Смена и проверка каталогов

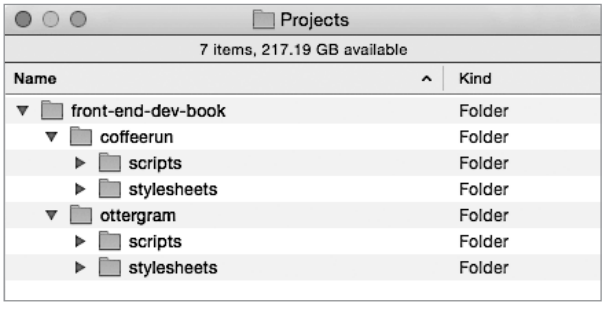


Рис. 1.22. Пример файловой структуры

Допустим, мы находимся в каталоге `ottergram` и хотим перейти в подкаталог `stylesheets` каталога `coffeerun`. Это можно сделать с помощью команды `cd` с последующим указанием пути, означающим, что «каталог `stylesheets` расположен в папке `coffeerun`, находящейся в каталоге, родительском для того, в котором я сейчас нахожусь»:

```
cd ../coffeerun/stylesheets
```

В Windows нужно использовать ту же команду, но с обратными слешами:

```
cd ..\coffeerun\stylesheets
```

Вывод списка файлов в каталоге

Вам может понадобиться посмотреть список файлов в текущем каталоге. На Mac для этого используется команда `ls` (рис. 1.23). Если нужно вывести список файлов в другом каталоге, можно указать его путь:

```
ls
ls ottergram
```

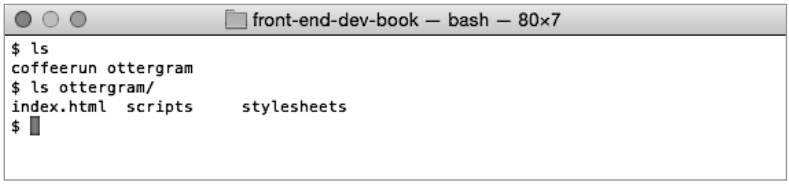


Рис. 1.23. Использование команды ls для вывода списка файлов в каталоге

По умолчанию `ls` ничего не выводит, если каталог пуст.

В Windows соответствующая команда называется `dir` (рис. 1.24), в которой тоже можно при необходимости задать путь:

```
dir
dir ottergram
```

По умолчанию команда `dir` выводит информацию о датах, времени и размерах файлов.

Получение прав администратора

В некоторых версиях операционных систем OS X и Windows для выполнения определенных команд, таких как команды установки программного обеспечения, или внесения изменений в защищенные файлы, необходимы права суперпользователя (superuser), они же права администратора.

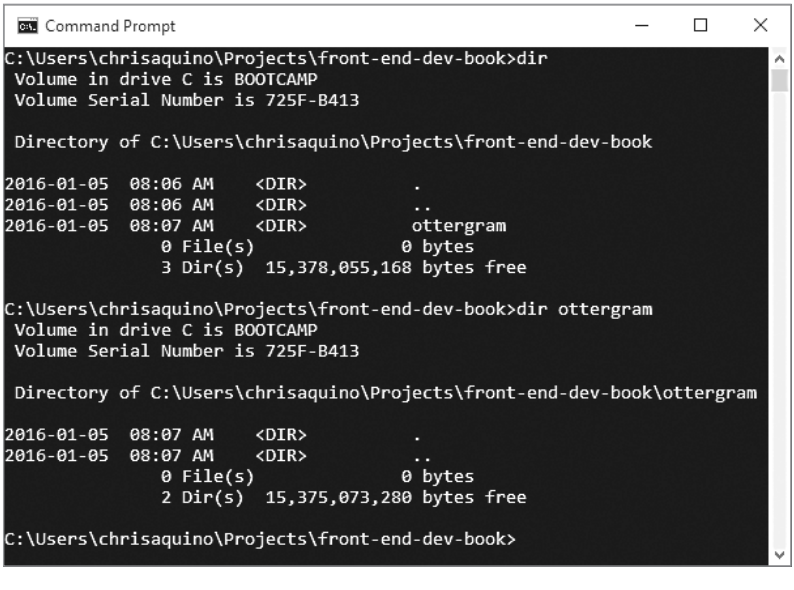


Рис. 1.24. Использование команды dir для вывода списка файлов в каталоге

На Mac предоставить себе такие права можно, указав перед командой sudo. При первом использовании sudo будет выдано строгое предупреждение, показанное на рис. 1.25.

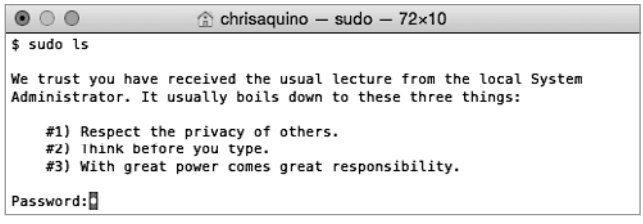


Рис. 1.25. Предупреждение sudo

Перед выполнением команды от имени суперпользователя sudo запросит пароль. При его наборе символы не будут отображаться на экране, так что будьте внимательны.

В Windows предоставлять себе права необходимо в процессе открытия интерфейса командной строки. Найдите командную строку в меню Start (Пуск), щелкните на ней правой кнопкой мыши, выберите Run as Administrator (Запуск от имени администратора) (рис. 1.26). Все команды, которые будут выполняться в этой командной строке, запустятся от имени суперпользователя, так что будьте внимательны.

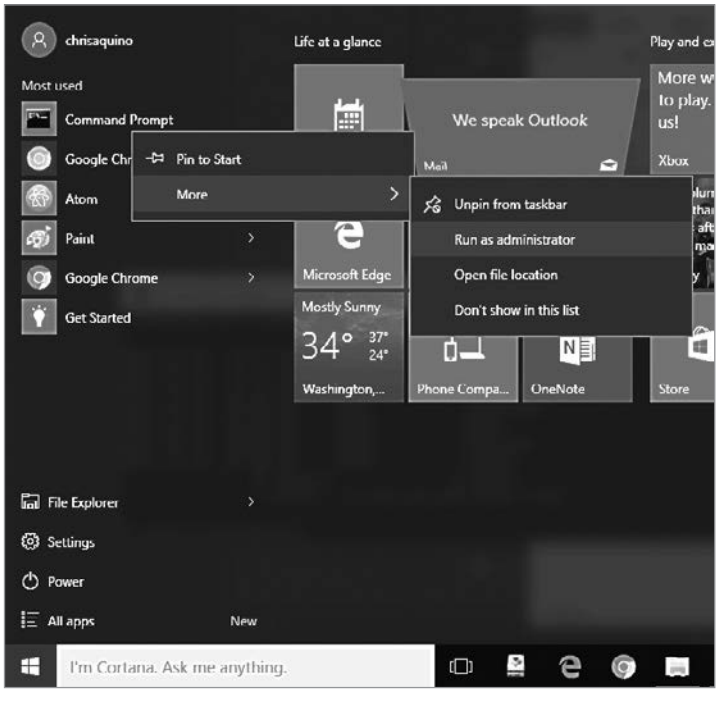


Рис. 1.26. Запуск командной строки от имени администратора

Выход из программ

По мере чтения книги вы будете запускать из командной строки разные приложения. Некоторые из них будут завершаться автоматически, а другие будут работать до тех пор, пока вы их не остановите. Чтобы выйти из программы командной строки, нажмите Control+C[1].

Установка Node.js и browser-sync

Это последнее, что нужно сделать, прежде чем начать наш первый проект.

Платформа Node.js (или просто Node) позволяет использовать написанные на JavaScript программы из командной строки. Большинство инструментов разработчика клиентской части рассчитано на применение Node.js. Мы познакомимся с Node.js подробнее в главе 15, но один основанный на нем инструмент (browser-sync) начнем использовать прямо сейчас.

Установите Node, скачав программу установки с сайта nodejs.org (рис. 1.27). В этой книге описывается версия Node.js 5.1.11, но вы, вероятно, увидите другие доступные для скачивания версии.

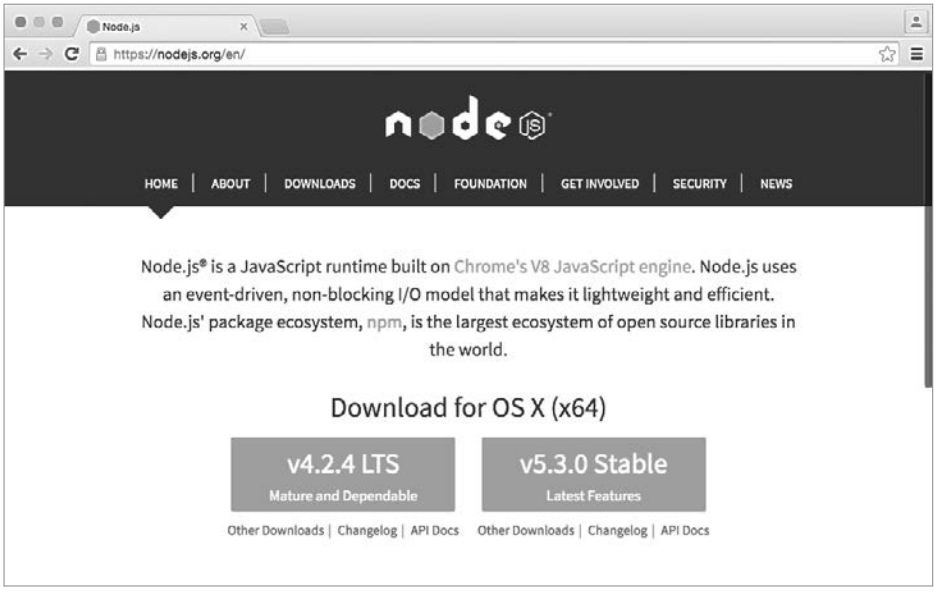


Рис. 1.27. Скачивание Node.js

[1] Автор допускает неточность: в большинстве версий Windows для этого необходимо выполнить команду exit или закрыть окно терминала. — *Примеч. пер.*

Выполните двойной щелчок на программе установки и следуйте инструкциям.

Установленный Node предоставляет две программы командной строки: node и npm. Программа node запускает написанные на JavaScript программы. Она не понадобится нам вплоть до главы 15. Программа npm — это система управления пакетами Node, необходимая для установки свободно распространяемых инструментов разработчика из Интернета.

Один из таких инструментов — browser-sync. Он будет чрезвычайно полезен нам на протяжении всей работы: облегчит запуск кода примеров в браузере и станет автоматически обновлять страницу в браузере при сохранении изменений в коде.

Установите browser-sync с помощью следующей команды в командной строке:

```
npm install -g browser-sync
```

(`-g` означает global — «глобальный». Глобальная установка пакета означает, что можно выполнить browser-sync из любого каталога.)

Не имеет значения, из какого каталога вы будете выполнять эту команду, но вам, вероятно, понадобятся права суперпользователя. В этом случае на Mac выполните команду с помощью `sudo`:

```
sudo npm install -g browser-sync
```

Если же вы работаете в Windows, сначала откройте интерфейс командной строки от имени администратора, как описано выше. После запуска, который вы выполните в следующей главе, browser-sync будет работать до тех пор, пока вы не нажмете Control+C. Разумно выходить из browser-sync при перерыве в работе над проектом. Это значит, что вам придется запускать browser-sync каждый раз, начиная работу над первыми двумя проектами этой книги (Ottergram и CoffeeRun).

Теперь у нас есть все инструменты, необходимые для начала работы над проектом Ottergram!

Для самых любознательных: альтернативы редактору Atom

Текстовых редакторов очень много, так что есть из чего выбирать. Если вы не особо заинтересовались Atom, то, работая над проектами из этой книги, можете попробовать два альтернативных варианта. Оба доступны бесплатно для ОС Mac и Windows, и у обоих имеется множество плагинов для подгонки под ваши требования как разработчика. Аналогично Atom они оба созданы на основе HTML, CSS и JavaScript, но представляют собой традиционные приложения.

Visual Studio Code — текстовый редактор с открытым исходным кодом, созданный компанией Microsoft специально для разработки веб-приложений. Его можно скачать с сайта code.visualstudio.com (рис. 1.28).

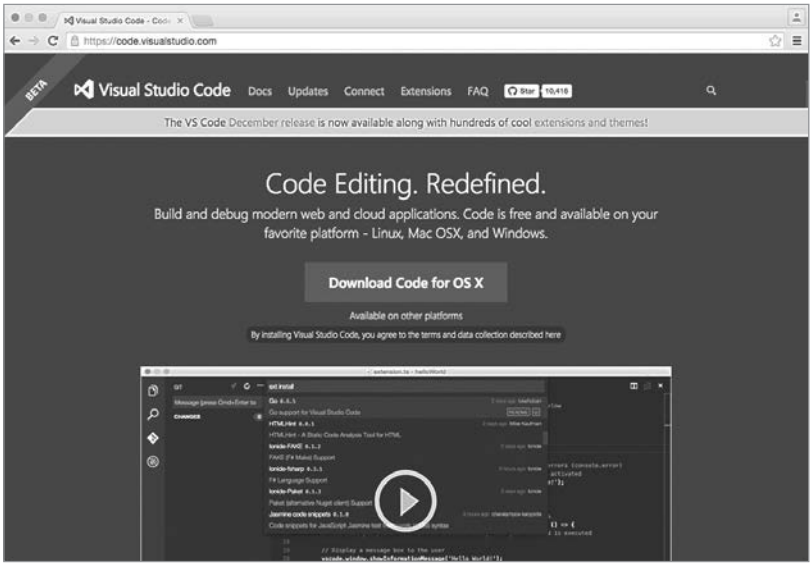


Рис. 1.28. Сайт Visual Studio Code

Текстовый редактор Brackets от компании Adobe особенно хорошо подходит для построения пользовательских интерфейсов с помощью языка разметки HTML и стилей CSS. По сути, он является расширением, облегчающим работу с многослойными файлами изображений PSD от Adobe. Brackets можно скачать с brackets.io (рис. 1.29).

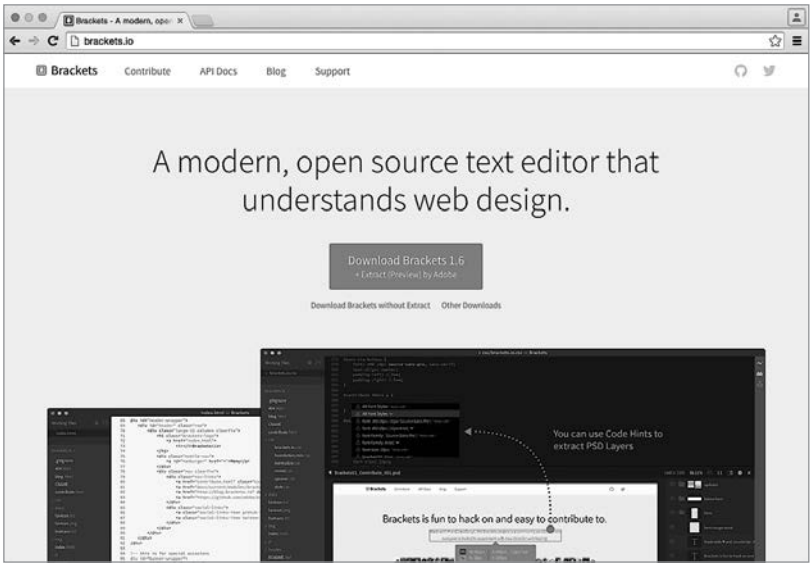


Рис. 1.29. Сайт Brackets

2 Настраиваем наш первый проект

Когда вы заходите на сайт, у вашего браузера происходит примерно следующий диалог с сервером — другим компьютером в Интернете.

Браузер: Привет! Можно мне, пожалуйста, получить содержимое файла `cat-videos.html`?

Сервер: Конечно. Дай-ка оглядеться... Вот он!

Браузер: А он говорит, что мне нужен еще один файл — `styles.css`.

Сервер: Еще бы. Дай-ка оглядеться... Вот он!

Браузер: Хорошо, но этот файл говорит мне, что нужен еще один файл — `animated-background.gif`.

Сервер: Нет проблем. Дай-ка оглядеться... Вот он!

Этот диалог продолжается некоторое время — вплоть до тысяч миллисекунд (рис. 2.1).

Задача браузера — отправлять запросы серверу, интерпретировать полученный в ответ код HTML, CSS и JavaScript и отображать результат пользователю. Каждая из этих трех технологий играет свою роль во взаимодействии пользователя с сайтом. Если бы ваше приложение было живым существом, разметка HTML была бы его скелетом и органами (механикой), стили CSS — кожей (видимым ее слоем), а код JavaScript — его личностью (речь о том, как приложение себя ведет).

В данной главе мы создадим простую разметку HTML для нашего первого проекта Ottergram. В следующей главе — стили CSS (с дальнейшим их усовершенствованием в главе 4). В главе 6 мы приступим к добавлению кода на языке программирования JavaScript.

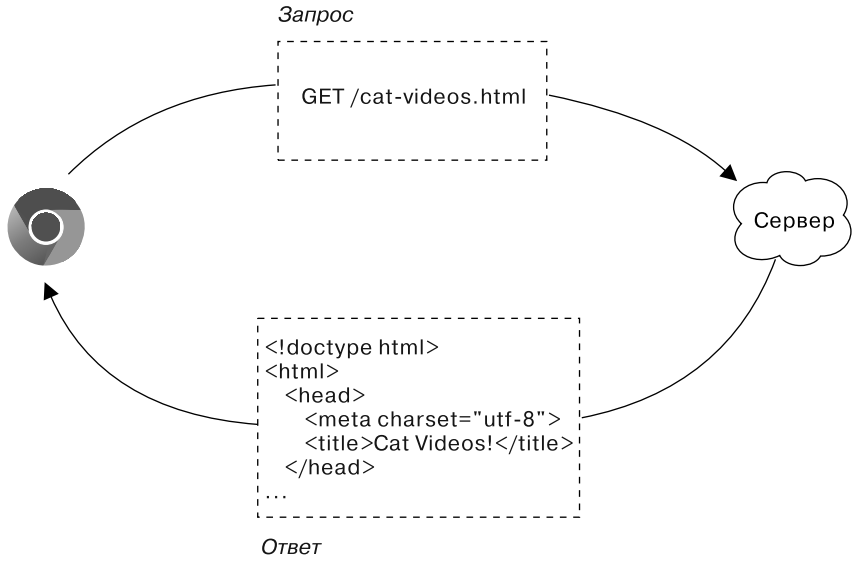


Рис. 2.1. Браузер отправляет запрос, сервер отвечает

Настройка Ottergram

В главе 1 мы создали каталог для проектов этой книги, а также папку для Ottergram. Запустите текстовый редактор Atom и откройте папку `ottergram`, нажав File ▸ Open (Файл ▸ Открыть) (в Windows: File ▸ Open Folder (Файл ▸ Открыть каталог)). В диалоговом окне перейдите в каталог `front-end-dev-book` и выберите каталог `ottergram`. Нажмите Open, чтобы указать редактору Atom, что нужно использовать этот каталог (рис. 2.2).

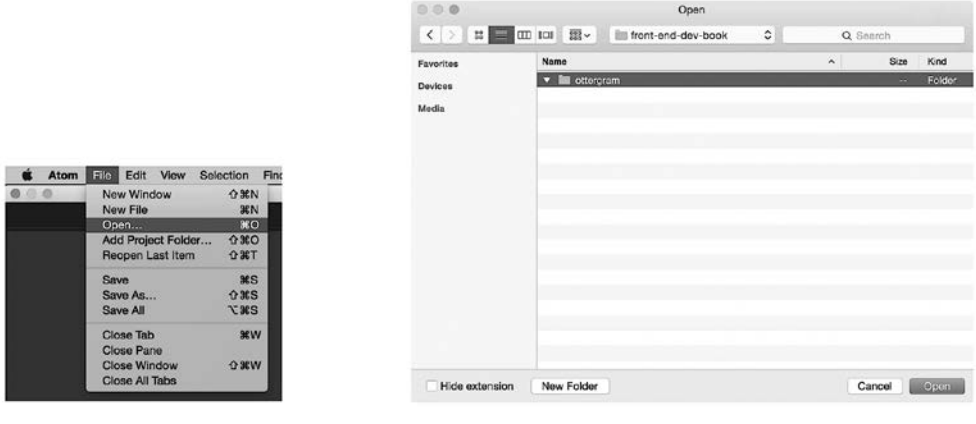


Рис. 2.2. Открываем каталог проекта в Atom

Вы увидите каталог `ottergram` на левой панели редактора Atom. Эта панель предназначена для навигации по файлам и каталогам проекта.

Нам нужно создать несколько файлов и каталогов внутри каталога `ottergram` с помощью Atom. Щелкните кнопкой мыши, нажав клавишу Control (щелкните правой кнопкой мыши в Windows), на каталоге `ottergram` на панели слева, затем щелкните на пункте New File (Новый файл) во всплывающем меню. Atom запросит у вас имя для этого нового файла. Введите `index.html` и нажмите клавишу Enter (рис. 2.3).

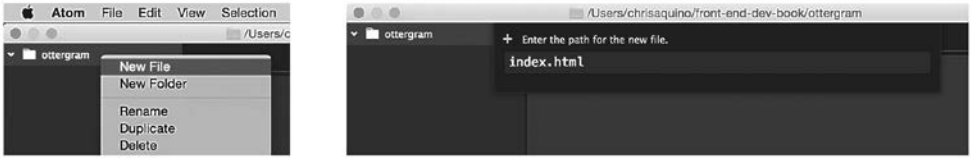


Рис. 2.3. Создание нового файла в Atom

Для создания каталогов с помощью редактора Atom можно использовать тот же способ. Щелкните кнопкой мыши, нажав клавишу Control (щелкните правой кнопкой мыши в Windows), на каталоге `ottergram` на панели слева, но теперь во всплывающем окне щелкните на пункте New Folder (Новый каталог). Укажите название `stylesheets` в появившемся приглашении ввести данные (рис. 2.4).

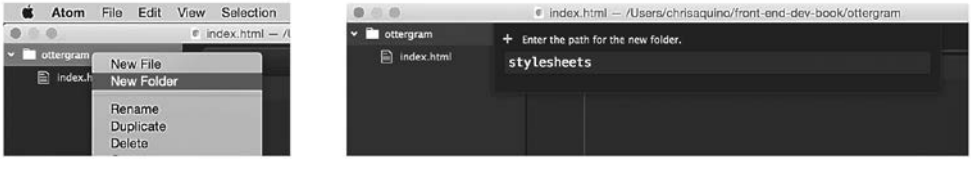


Рис. 2.4. Создание нового каталога в Atom

Наконец, создайте файл `styles.css` в каталоге `stylesheets`: щелкните кнопкой мыши, нажав клавишу Control (щелкните правой кнопкой мыши в Windows), на каталоге `ottergram` панели слева и выберите New File (Новый файл). В приглашении на ввод будет уже указано `stylesheets/`. Введите `styles.css` и нажмите клавишу Enter (рис. 2.5).

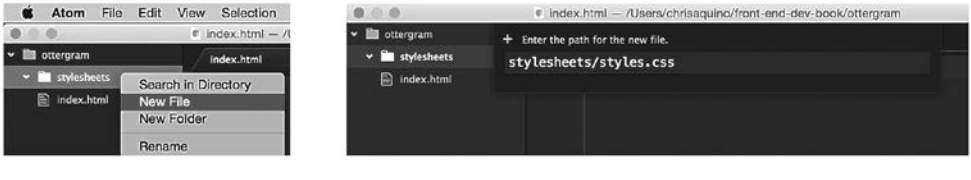


Рис. 2.5. Создание нового файла стилей CSS в Atom

После этого каталог проекта должен выглядеть так, как показано на рис. 2.6.

Правил относительно структурирования файлов и каталогов или их наименования нет. Однако Ottergram (как и остальные проекты в этой книге) придерживается соглашений, используемых многими разработчиками клиентской части. В файле `index.html` будет находиться код HTML. Основной файл HTML принято называть `index.html` еще с момента появления Интернета; это соглашение работает и сейчас.

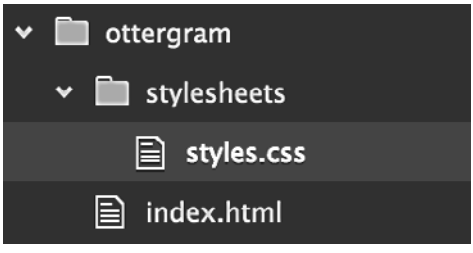


Рис. 2.6. Исходные файлы и каталоги для Ottergram

В каталоге `stylesheets` (таблицы стилей), как и подразумевает его название, будет находиться один или несколько файлов с информацией о стилях для Ottergram (файлы CSS (cascading style sheets — каскадные таблицы стилей)). Иногда разработчики дают файлам CSS имена, описывающие, к какой части страницы или сайта они относятся, например `header.css` или `blog.css`. Ottergram — простой проект, которому достаточно одного файла CSS, так что назовите его `styles.css`, чтобы отразить его глобальную роль.

Исходный HTML

Пора начинать писать код. Откройте `index.html` в Atom и вставьте туда для начала простейший HTML.

Начните вводить `html`. Atom предложит вам автодополнение, как показано на рис. 2.7 (если этого не произошло, проверьте, установили ли вы плагин emmet, описанный в главе 1).

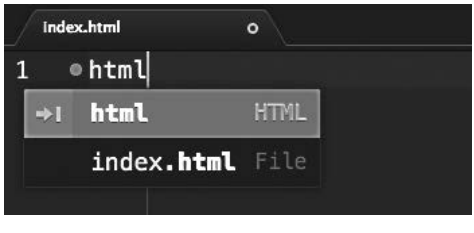


Рис. 2.7. Меню автодополнения Atom

Нажмите клавишу Enter — и Atom создаст базовые элементы HTML, чтобы можно было приступить к работе (рис. 2.8).

Курсор находится между `<title>` и `</title>` — открывающим и закрывающим тегами `title`. Наберите `ottergram`, чтобы указать название проекта. Теперь щелкните кнопкой мыши, чтобы установить курсор на пустую строку между открывающим и закрывающим тегами `body`. Введите `header` и нажмите клавишу Enter. Atom преобразует текст `header` («заголовок») в открывающий и закрывающий теги `header` с пустой строкой между ними (рис. 2.9).

Далее наберите `h1` и нажмите Enter. Ваш текст будет преобразован в теги, но на этот раз без пустой строки. Снова введите `ottergram`. Это будет заголовок, отображаемый на вашей веб-странице.

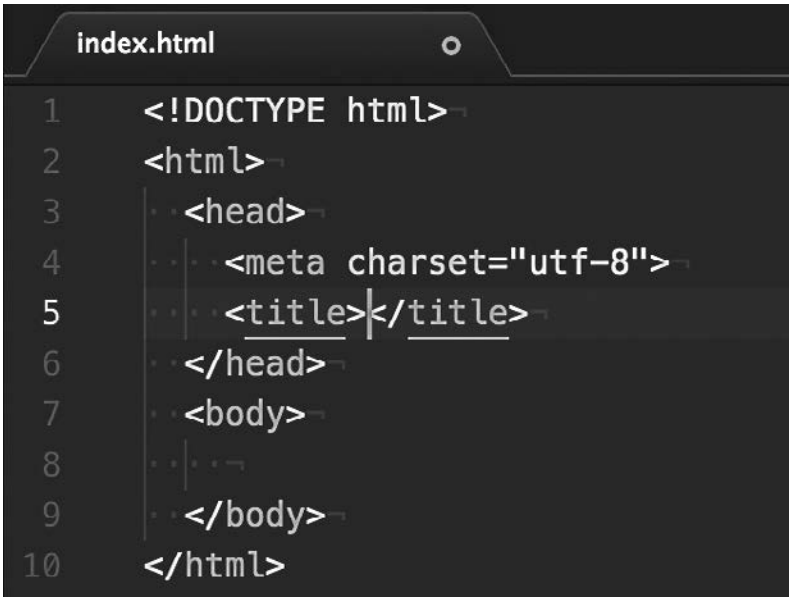


Рис. 2.8. Созданный с помощью автодополнения HTML-код

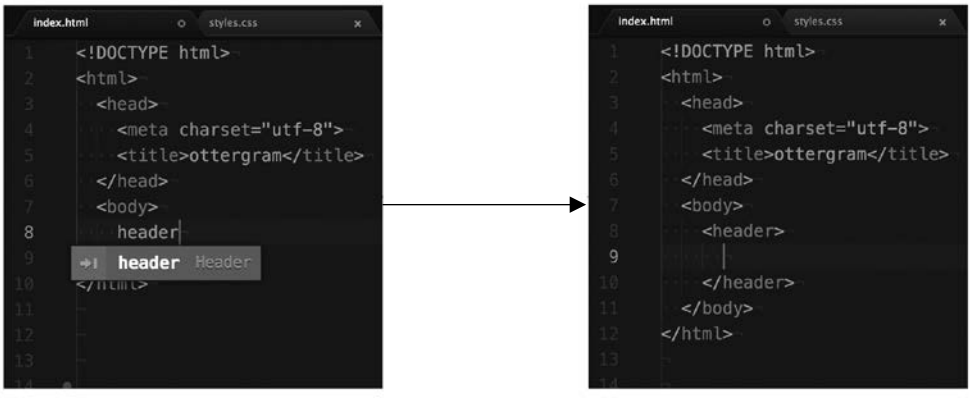


Рис. 2.9. Созданный с помощью автодополнения тег header

Файл должен выглядеть так:

```
<!doctype html>
<html>
  <head>
    <meta charset="utf-8">
    <title>ottergram</title>
  </head>
```

```
  <body>
    <header>
      <h1>ottergram</h1>
    </header>
  </body>
</html>
```

Редактор Atom и плагин emmet сэкономили вам время на набор текста и помогли создать хорошо отформатированный исходный HTML.

Проанализируем этот код. Первая строка `<!doctype html>` описывает *doctype* (тип документа), указывающий браузеру, в какой версии HTML создан документ. Браузер может визуализировать (отрисовывать) документ немного по-разному в зависимости от его типа.

В предыдущих версиях языка разметки HTML типы документов часто были длинными, запутанными и сложными для запоминания, например:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
```

Поэтому приходилось искать doctype в справочнике каждый раз при создании нового документа.

В HTML5 тип документа короткий и понятный. Именно он будет использоваться во всех проектах этой книги (и именно его вам следует применять для своих приложений).

После типа документа идет простая разметка HTML, состоящая из `head` и `body`.

В теге `head` содержится информация о документе и о том, как браузеру следует его обрабатывать. Например, в `head` указано название документа, какие файлы CSS и JavaScript использует страница и когда документ в последний раз изменялся.

В данном случае `head` содержит тег `<meta>`. Теги `<meta>` предоставляют браузеру информацию о самом документе: имя автора или ключевые слова для поисковых систем. Тег `<meta>` в Ottergram (`<meta charset="utf-8">`) указывает, что документ закодирован с помощью набора символов UTF-8, включающего все символы Unicode. Используйте этот тег в ваших документах, чтобы как можно больше браузеров могли корректно их интерпретировать, особенно если вы ожидаете международного трафика.

В теге `body` содержится весь код HTML, представляющий содержимое страницы: изображения, ссылки, текст, кнопки и видео.

Большинство тегов содержат внутри какой-нибудь другой контент. Рассмотрим вставленный нами заголовок `h1`. Его структура показана на рис. 2.10.

Аббревиатура HTML расшифровывается как Hypertext Markup Language (язык разметки гипертекста). Теги используются для разметки контента, объявления

его назначения (заголовки, элементы списков и ссылки).

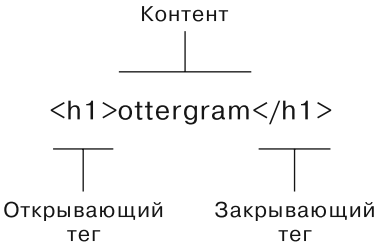


Рис. 2.10. Устройство простого HTML-тега

Контент, заключенный в набор тегов, может, в свою очередь, включать другой HTML. Например, тег `<header>` заключает в себя показанный выше тег `<h1>` (а теги `<body>` заключают в себя `<header>`!).

Тегов насчитывается более 140. Чтобы посмотреть их список, откройте справочник элементов HTML сети MDN, расположенный на developer.mozilla.org/en-US/docs/Web/HTML/Element. Справочник содержит краткое описание каждого элемента и группирует элементы по назначению (например, текстовый контент, разбиение контента или мультимедийные элементы).

Привязка таблицы стилей

В главе 3 мы опишем правила оформления документа в таблице стилей `styles.css`. Но помните диалог между браузером и сервером в начале этой главы? Браузер может запросить файл с сервера, только если ему сообщили о существовании такого файла. Нам необходимо *привязать (link)* нашу таблицу стилей, чтобы браузер знал, что ее нужно запросить. Изменим `head` файла `index.html`, добавив ссылку на файл `styles.css`.

```
<!doctype html>
<html>
  <head>
    <meta charset="utf-8">
    <title>ottergram</title>
    <link rel="stylesheet" href="stylesheets/styles.css">
  </head>
  <body>
...
```

Тег `<link>` позволяет подключить внешнюю таблицу стилей к документу HTML. У него есть два *атрибута*, предоставляющие браузеру дополнительную информацию о назначении тега (рис. 2.11). (Порядок атрибутов HTML не имеет значения.)

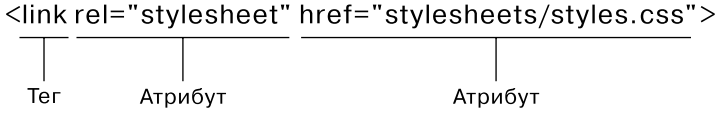


Рис. 2.11. Устройство тега с атрибутами

Мы задаем атрибут `rel` (relationship — «отношение») тега `stylesheet`, указывающий браузеру, что связанный документ содержит информацию о стилях. Атрибут `href` сообщает браузеру о необходимости отправить серверу запрос на файл `styles.css`, расположенный в каталоге `stylesheets`. Обратите внимание, что данный путь файла указывается *относительно* текущего документа.

Сохраните файл `index.html`, прежде чем работать дальше.

Добавление контента

Веб-страница без контента как утро без чашки кофе. Добавим после тега `header` список, чтобы у нашего проекта появился смысл жизни.

Нам нужно добавить *неупорядоченный* (маркированный) *список* с помощью тега `<ul>`. В этот список мы вставим пять элементов с помощью тегов `<li>`, а в каждый элемент добавим текст, заключенный в теги `<span>`.

Измененный файл `index.html` показан ниже. Обратите внимание, что во всей книге мы выделяем добавляемый новый код полужирным шрифтом. Код, который необходимо будет удалить, зачеркнут. Существующий код набран обычным шрифтом, чтобы вам было проще вносить изменения в файл.

Мы настоятельно советуем использовать возможности автодополнения и автоформатирования редактора Atom. Поставив курсор в соответствующее место, наберите `ul` и нажмите Enter. Далее наберите `li` и дважды нажмите Enter, затем наберите `span` и нажмите Enter один раз. Введите имя выдры, после чего создайте еще четыре элемента списка и тега `<span>` аналогичным образом.

```
<!doctype html>
<html>
  <head>
    <meta charset="utf-8">
    <title>ottergram</title>
    <link rel="stylesheet" href="stylesheets/styles.css">
  </head>
  <body>
    <header>
      <h1>ottergram</h1>
    </header>
    <ul>
      <li>
        <span>Barry</span>
      </li>
      <li>
        <span>Robin</span>
      </li>
      <li>
        <span>Maurice</span>
      </li>
```

```
      <li>
        <span>Lesley</span>
      </li>
      <li>
        <span>Barbara</span>
      </li>
    </ul>
  </body>
</html>
```

Теги `<span>`, вложенные в каждый из тегов `<li>`, не несут какого-либо особого смысла — это просто контейнеры для контента. Они будут использоваться в Ottergram для оформления. На страницах этой книги мы познакомим вас и с другими элементами-контейнерами.

Далее добавим изображения выдр в соответствии с введенными именами.

Добавление изображений

Файлы ресурсов для всех проектов этой книги находятся по адресу www.bignerdranch.com/downloads/front-end-dev-resources.zip. Они содержат пять фотографий выдр (взяты нами с `commons.wikimedia.org`, имеют лицензию Creative Commons), сделанных Майклом Л. Бейрдом, Джо Робертсоном и Agunther.

Скачайте и разархивируйте ресурсы. В каталоге `ottergram-resources` найдите каталог `img`. Скопируйте `img` в папку проекта `ottergram/` (архив содержит и другие ресурсы, но пока что нам нужен только каталог `img`).

Хотелось бы, чтобы список, помимо названий, содержал миниатюры изображений, на которых можно было бы щелкать. Для этого добавьте тег-якорь и тег изображения к каждому элементу в `ul`. Мы поясним эти изменения более подробно после того, как вы их выполните (если вы используете автодополнение, обратите внимание, что вам понадобится переставить теги `</a>`, чтобы они следовали за тегами `<span>`).

```
...
    <ul>
      <li>
        <a href="#">
          <img src="img/otter1.jpg" alt="Barry the Otter">
          <span>Barry</span>
        </a>
      </li>
      <li>
        <a href="#">
          <img src="img/otter2.jpg" alt="Robin the Otter">
          <span>Robin</span>
        </a>
      </li>
      <li>
```

```
        <a href="#">
          <img src="img/otter3.jpg" alt="Maurice the Otter">
          <span>Maurice</span>
        </a>
      </li>
      <li>
        <a href="#">
          <img src="img/otter4.jpg" alt="Lesley the Otter">
          <span>Lesley</span>
        </a>
      </li>
      <li>
        <a href="#">
          <img src="img/otter5.jpg" alt="Barbara the Otter">
          <span>Barbara</span>
        </a>
      </li>
    </ul>
...
```

Если отступы в строках выглядят недостаточно красиво, можно воспользоваться установленным плагином `atom-beautify`. Щелкните на Packages ▸ Atom Beautify ▸ Beautify — и код будет выровнен и структурирован.

Разберемся, что же мы добавили.

Тег `<a>` — это *тег-якорь (anchor)*. Теги-якоря делают элементы на странице такими, чтобы на них можно было щелкнуть кнопкой мыши — в результате пользователь будет перенаправлен на другую страницу. Обычно их называют ссылками, но обратите внимание: это не то же, что использовавшийся нами ранее тег `<link>`.

У тегов-якорей имеется атрибут `href`, задающий ресурс, на который указывает якорь. Обычно его значение представляет собой URL. Иногда, однако, необходимо, чтобы ссылка вела в никуда. Наш случай именно таков, так что мы присвоили атрибутам `href` фиктивное значение `#`. Это заставит браузер прокручивать страницу до верха при нажатии на изображение. Далее мы воспользуемся тегами-якорями, чтобы открывать увеличенные копии изображений при щелчке на миниатюре.

Внутри тегов-якорей мы вставили теги `<img>` (image — «изображение») с атрибутами `src`, указывающими на названия ранее добавленных файлов из каталога `img`. Мы добавили в теги изображений описательные атрибуты `alt`. Атрибуты `alt` содержат текст, замещающий изображение, которое не удается загрузить. Текст `alt` также используется программами чтения экрана для описания изображений пользователям с нарушениями зрения.

Теги изображений отличаются от большинства других элементов тем, что они не служат оберткой для других элементов, а ссылаются на ресурсы. Когда браузер обнаруживает тег `<img>`, он визуализирует на странице изображение (носит название *замещаемого элемента (replaced element)*). Среди других замещаемых элементов можно назвать встраиваемые документы и апплеты.

Теги `<img>` не обертывают контент или другие элементы, поэтому у них отсутствует соответствующий закрывающий тег. Такие теги называются *одинарными*. Порой вам будут встречаться одинарные теги, написанные с косой чертой перед правой угловой скобкой, например `<img src="otter.jpg"/>`. Ставить ли косую черту — ваше личное дело, для браузера никакой разницы нет. В данной книге одинарные теги везде написаны без косой черты.

Сохраните `index.html`. Через минуту мы увидим результат написанного кода.

Просмотр веб-страницы в браузере

Для просмотра веб-страницы необходимо, чтобы была запущена утилита browser-sync, установленная в главе 1.

Откройте терминал и перейдите в каталог `ottergram`. Как вы помните из главы 1, папку можно сменить с помощью команды `cd` с указанием после нее пути каталога, в который вы хотели бы перейти. Удобный способ получить путь `ottergram` — щелкнуть кнопкой мыши с нажатой клавишей Control (правой кнопкой мыши в Windows) на каталоге `ottergram` на левой панели Atom и выбрать пункт Copy Full Path (Скопировать полный путь) (рис. 2.12). Затем в командной строке наберите `cd`, вставьте путь и нажмите Enter.

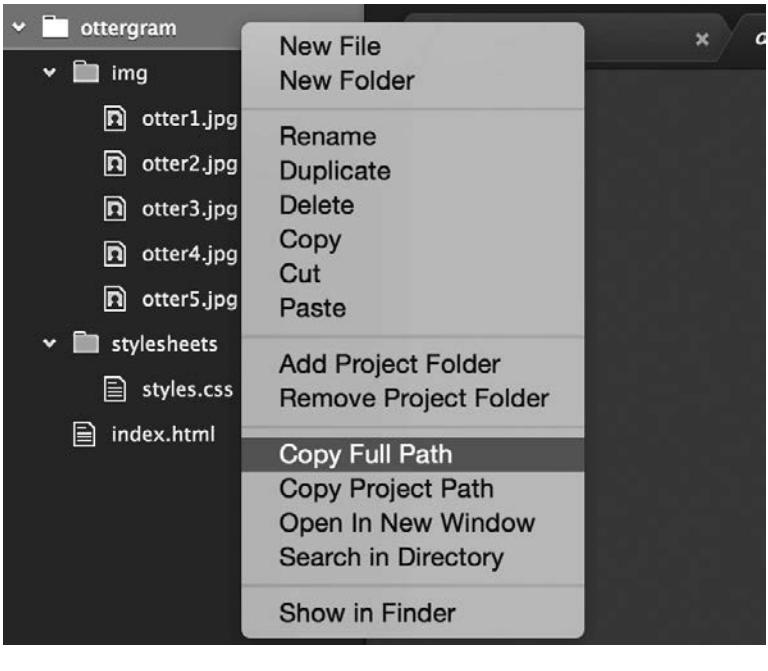


Рис. 2.12. Копирование пути каталога ottergram из Atom

Введенный путь может выглядеть примерно вот так:

```
cd /Users/chrisaquino/Projects/front-end-dev-book/ottergram
```

Оказавшись в каталоге `ottergram`, выполните следующую команду для открытия Ottergram в Chrome (мы разбили команду на две строки, чтобы она поместилась на странице, вам необходимо ввести ее одной строкой):

```
browser-sync start --server --browser "Google Chrome"
                   --files "stylesheets/*.css, *.html"
```

Если Chrome — ваш браузер по умолчанию, можете опустить часть команды `--browser "Google Chrome"`:

```
browser-sync start --server --files "stylesheets/*.css, *.html"
```

Эта команда запустит утилиту browser-sync в серверном режиме, позволяя ей отправлять ответы при передаче браузером запроса на получение файла, например созданного вами файла `index.html`.

Введенная вами команда также указывает browser-sync автоматически обновлять страницу в браузере при изменении каких-либо файлов HTML или CSS. Это значительно облегчает разработку. До появления таких утилит, как browser-sync, приходилось вручную обновлять страницу в браузере после каждого изменения.

Рисунок 2.13 демонстрирует результат выполнения этой команды на Mac.

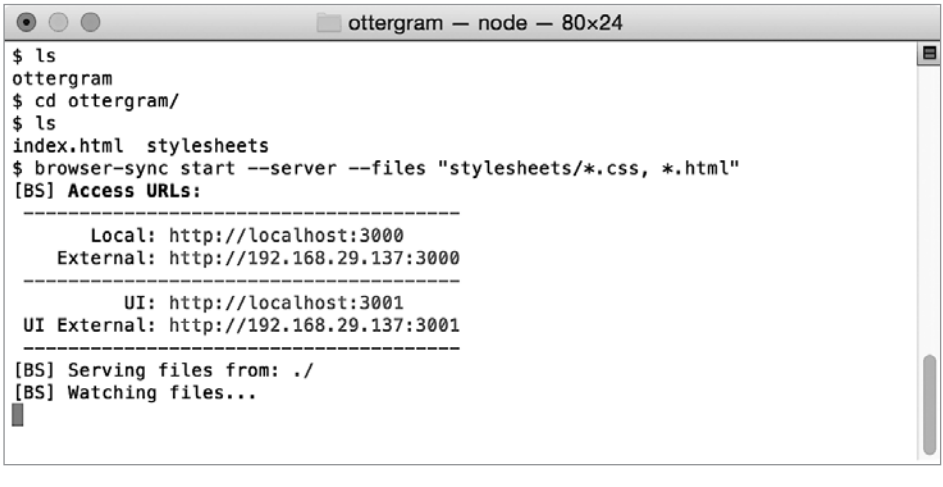


Рис. 2.13. Запуск browser-sync в терминале OS X

В Windows вы должны увидеть аналогичную картину (рис. 2.14).

После загрузки страницы Ottergram в Chrome вы должны увидеть страницу с заголовком ottergram, с ottergram в качестве названия вкладки браузера и с набором фотографий и имен выдр (рис. 2.15).

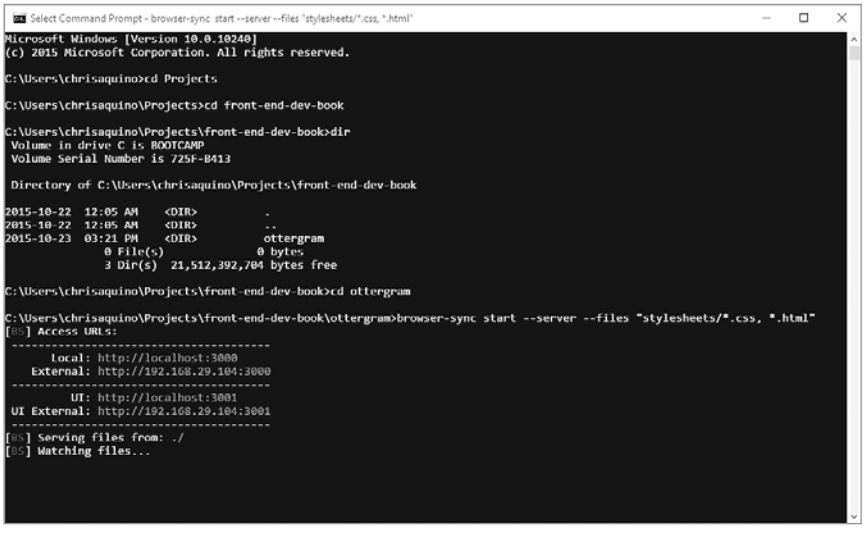


Рис. 2.14. Запуск browser-sync в командной строке Windows

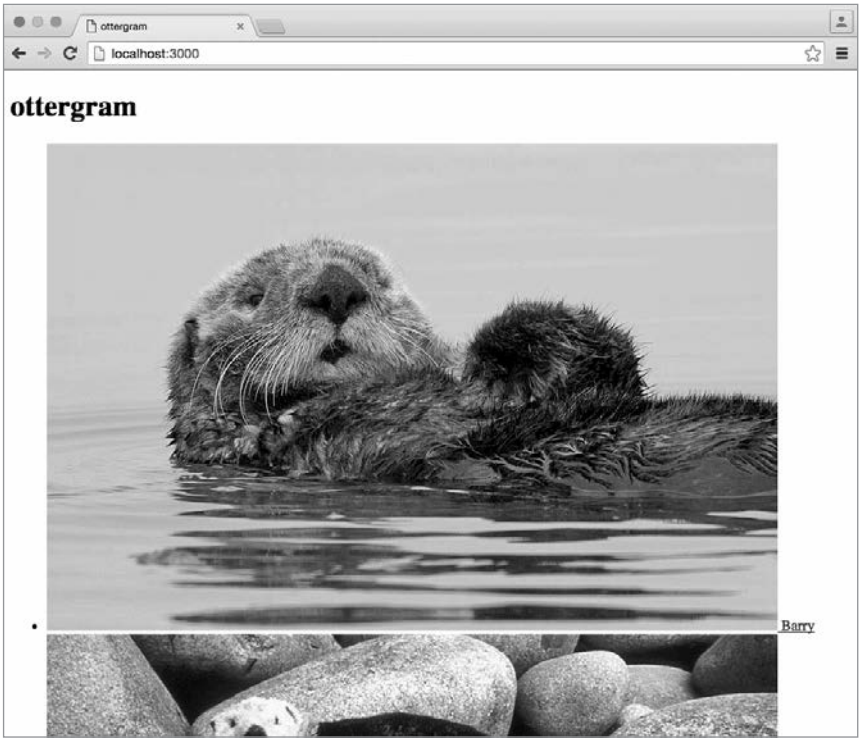


Рис. 2.15. Просмотр Ottergram в браузере

Инструменты разработчика Chrome

У браузера Chrome имеются встроенные инструменты разработчика (Developer Tools, часто сокращаемые до DevTools) — одни из лучших, которые только существуют для тестирования стилей, макетов (и не только во время работы). Использовать DevTools намного эффективнее, чем проверять что-то в коде. DevTools — очень мощные инструменты, они станут вашим постоянным помощником при разработке клиентской части.

Мы начнем использовать DevTools в следующей главе. А пока что откройте окно и познакомьтесь с их внешним видом.

Чтобы открыть DevTools, щелкните на значке ☰ справа от адресной строки в Chrome. Далее щелкните на More Tools ▸ Developer Tools (рис. 2.16).

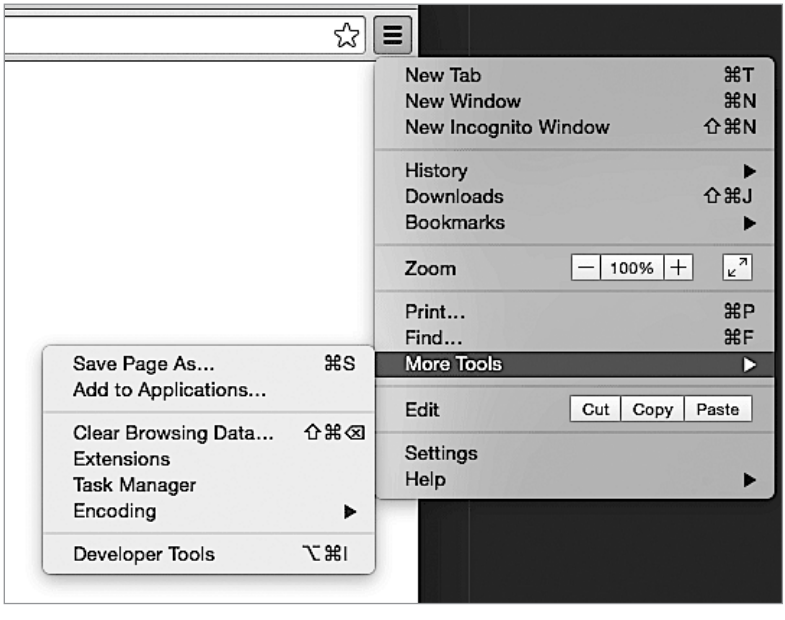


Рис. 2.16. Открываем Developer Tools

По умолчанию Chrome отображает DevTools справа. Ваш экран будет выглядеть примерно так, как показано на рис. 2.17.

DevTools показывают соотношение кода и получающихся в результате элементов страницы. Они предоставляют возможность исследовать атрибуты отдельных элементов и стилей и сразу же видеть, как браузер интерпретирует код. Возможность видеть это соотношение критически важна как для разработки, так и для отладки.

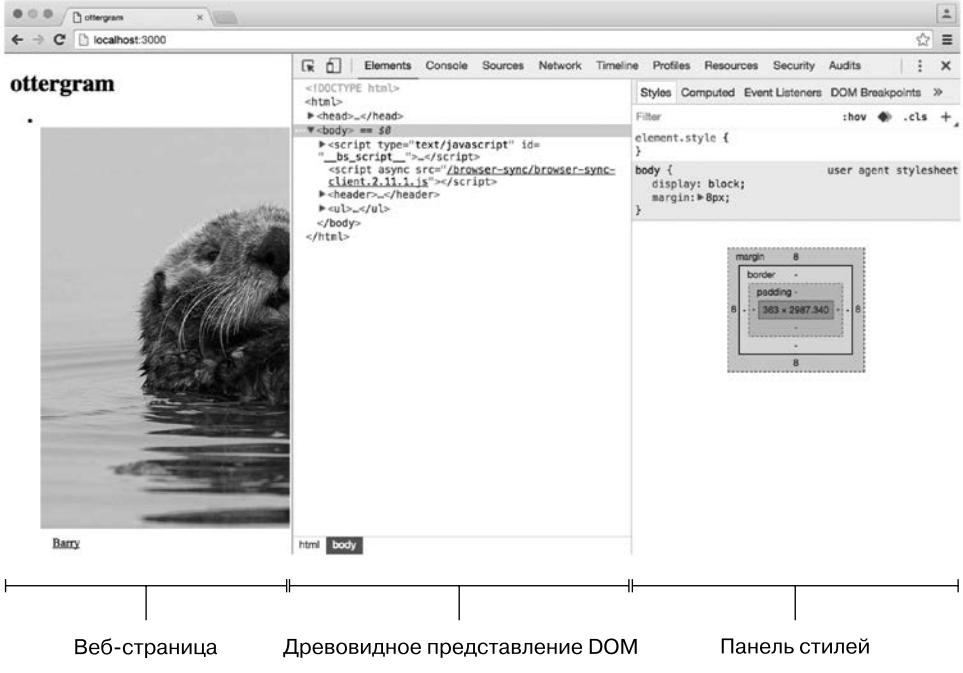


Рис. 2.17. DevTools с открытой панелью элементов

На рис. 2.17 можно видеть DevTools с открытой панелью элементов рядом с веб-страницей. Панель элементов разделена на две части. Слева находится древовидное представление DOM (DOM tree view). Это представление HTML в терминах элементов DOM (в последующих главах вы узнаете больше о DOM — document object model — объектной модели документов). В правой части панели элементов находится панель стилей. Она отображает все визуальные стили, применяемые к отдельным элементам.

На время работы удобно закрепить DevTools в правой части экрана. Если вам нужно поменять местоположение DevTools, нажмите кнопку ⋮ в верхнем правом углу. Это действие отобразит меню возможностей, включая кнопки панели Dock side (Закрепить на), меняющие место закрепления DevTools (рис. 2.18).

Теперь, когда выдры и разметка готовы и DevTools открыты, в следующей главе мы сможем приступить к работе со стилями проекта.

Для самых любознательных: версии CSS

История версий CSS включает стандартные версии 1, 2 и 2.1. После версии 2.1 было принято решение разбивать стандарт на отдельные части из-за слишком

большого размера. Версии 3 не существует. Вместо этого CSS3 представляет собой общий термин для множества модулей, у каждого из которых свой номер версии (табл. 2.1).

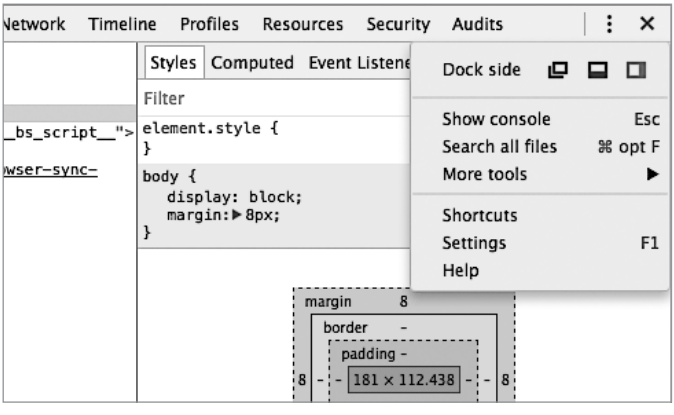


Рис. 2.18. Меняем сторону экрана, на которой закреплены DevTools

Таблица 2.1. Версии CSS

Номер версии	Год выпуска	Важные возможности
1	1996	Основные свойства шрифтов (font-family, font-style), цвета фона и переднего плана, выравнивание текста, отступы, граница и поля
2	1998	Абсолютное, относительное и фиксированное позиционирование; новые свойства шрифтов
2.1	2011	Исключены возможности, недостаточно поддерживаемые браузерами
3	Разные	Набор различных спецификаций, таких как медиазапросы, новые селекторы, полупрозрачные цвета, правило @font-face

Для самых любознательных: favicon.ico

Замечали ли вы когда-нибудь маленький значок, появляющийся в левом конце адресной строки браузера при посещении большинства сайтов? Иногда он также появляется на вкладке браузера (рис. 2.19).

Рис. 2.19. favicon.ico сайта bignerdranch.com

Это файл изображения `favicon.ico`. Он есть у многих сайтов, и браузеры запрашивают его по умолчанию. Поскольку у Ottergram его нет, вы увидите в DevTools ошибку, показанную на рис. 2.20.

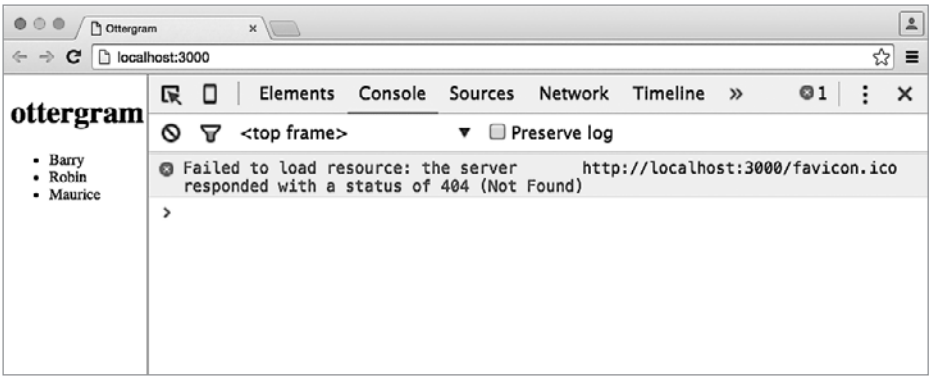


Рис. 2.20. Ошибка при отсутствующем favicon.ico

Не беспокойтесь при появлении этой ошибки. Она никак не повлияет на наш проект. Однако можно добавить в проект изображение `favicon.ico` — это и есть ваше первое упражнение.

Серебряное упражнение: добавить favicon.ico

Вы решили, что смотреть на выдр вам нравится больше, чем на сообщение об ошибке `favicon.ico`. Вы хотели бы создать файл `favicon.ico` на основе одной из фотографий выдр.

Задайте поиск в Интернете по ключевым словам `favicon generator` (генератор favicon) — и вы получите список сайтов, которые могут выполнить для вас преобразование файла. Большинство из них предоставит возможность загрузить изображение, вернув вам после этого вариант `favicon.ico`.

Выберите один из сайтов и загрузите туда любую фотографию выдры.

Сохраните получившийся файл `favicon.ico` в том же каталоге, где находится файл `index.html`. Наконец, обновите страницу в браузере. Вкладка в браузере после этого будет выглядеть примерно так, как показано на рис. 2.21.

Рис. 2.21. Ottergram с favicon.ico

3 Стили

В этой главе нам предстоит создать статическую версию Ottergram. В следующих главах мы сделаем Ottergram интерактивным.

В конце этой главы наш сайт будет выглядеть так, как показано на рис. 3.1.

Эта глава познакомит вас с множеством понятий. Кроме того, в ней немало примеров. Не беспокойтесь, если, дочитав ее до конца, почувствуете, что не полностью во всем разобрались. По мере изучения книги вы будете сталкиваться с этими понятиями снова и снова, а ваша работа на протяжении данной главы обеспечит прочный фундамент для дальнейшего их понимания.

Конечно, в книге мы можем познакомить вас только с малой толикой всех доступных в CSS стилей. Но вы можете обратиться к MDN за информацией обо всех свойствах и их значениях.

Разработчикам клиентской части приходится выбирать между двумя подходами к созданию стилей сайта: начинать с общего макета и прорабатывать все до мельчайших деталей или начинать с мельчайших деталей и доводить до общего макета.

Второй подход не только позволяет создать более чистый и легче используемый повторно код, но и носит красивое название *«атомарная стилизация» (atomic styling)*. Сначала мы воспользуемся этим подходом при стилизации миниатюр выдр, а затем — в макете списка миниатюр. А в следующей главе займемся макетом сайта в целом.

Создание фундамента для стилей

Начнем с добавления в проект файла `normalize.css`. Это обеспечит единообразное отображение создаваемого нами CSS в разных браузерах. У каждого браузера имеется набор стилей по умолчанию, но у разных браузеров они отличаются. `normalize.css` — хороший отправной пункт для разработки своего собственного пользовательского CSS для сайта или веб-приложения.

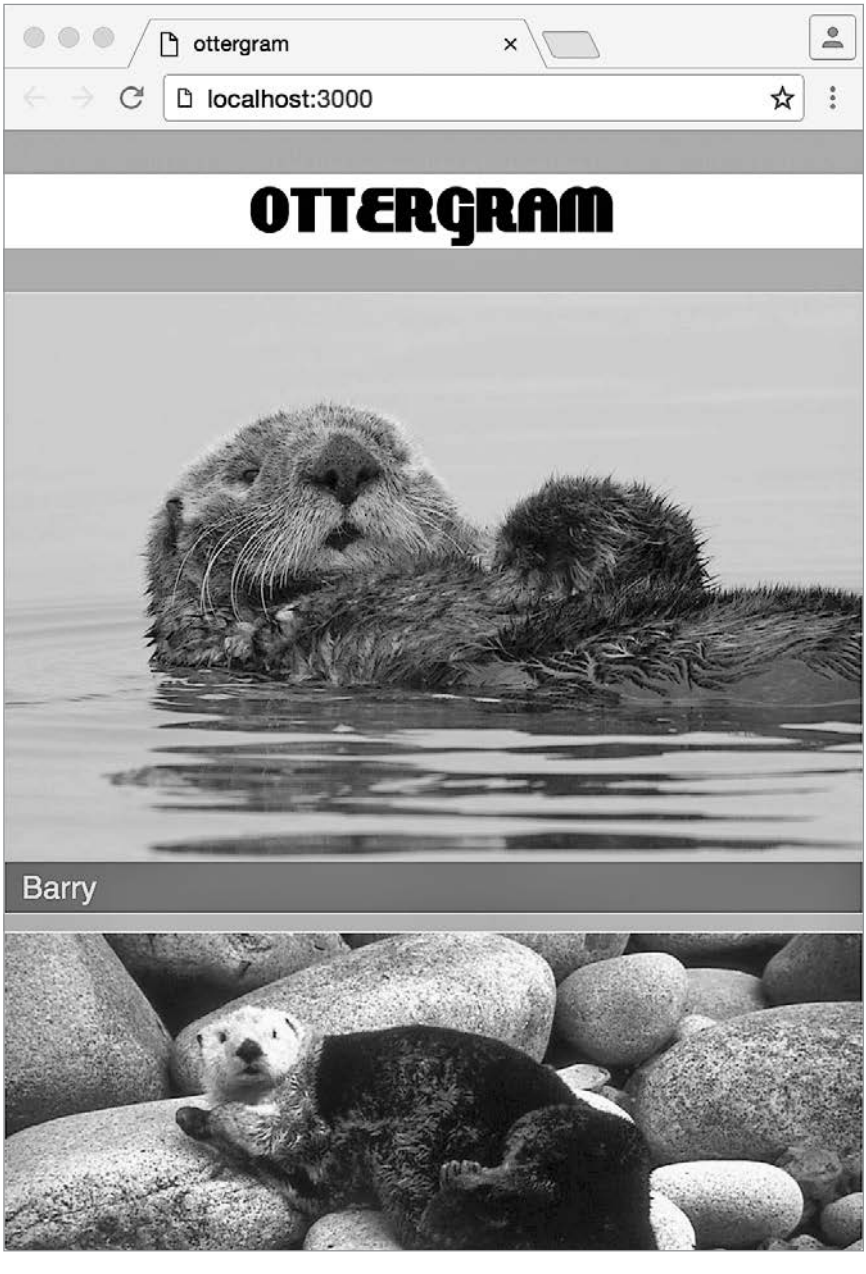


Рис. 3.1. Стилизованная версия Ottergram

Файл `normalize.css` доступен в Интернете бесплатно. Вам не нужно его скачивать, достаточно сослаться на него в `index.html`.

Чтобы удостовериться, что мы используем наиболее свежую версию `normalize.css`, необходимо получить его адрес с сайта, обеспечивающего совместное использование контента. Перейдите по адресу `cdnjs.com/libraries/normalize` и найдите версию данного файла, заканчивающуюся на `.min.css` (эта версия меньше по размеру, чем остальные, за счет удаления пробельных символов). Нажмите кнопку `Copy`, чтобы скопировать его адрес (рис. 3.2).

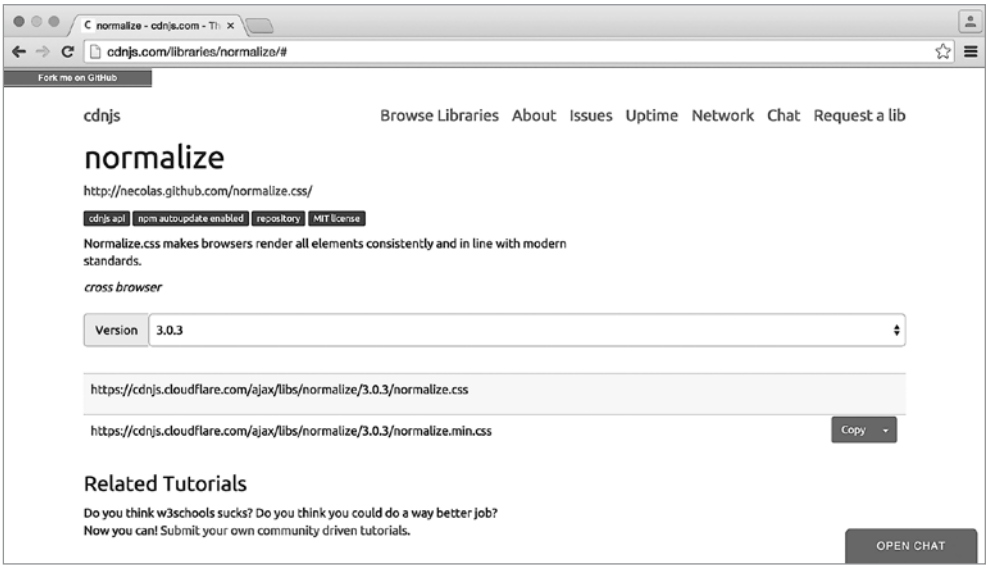


Рис. 3.2. Получение ссылки на normalize.css с cdnjs.com

На момент написания книги актуальна версия 3.0.3, но у вас может оказаться более свежая версия. Откройте в редакторе Atom каталог Ottergram, затем откройте файл `index.html`. Добавьте новый тег `<link>` и вставьте туда вышеупомянутый адрес. (В приведенном ниже коде `<link>` разбит на несколько строк, чтобы поместиться на странице. Вы можете оставить его однострочным.)

```
<!doctype html>
<html>
  <head>
    <meta charset="utf-8">
    <title>ottergram</title>
    <link rel="stylesheet"
      href="https://cdnjs.cloudflare.com/ajax/libs/normalize/3.0.3/
            normalize.min.css">
    <link rel="stylesheet" href="stylesheets/styles.css">
  </head>
...
```

Убедитесь, что вы добавили тег `<link>` для `normalize.css` *до* тега `<link>` для `styles.css`. Браузеру необходимо прочитать стили из `normalize.css` до того, как он прочитает ваши стили.

Вот и все: наш проект может пользоваться этим полезным инструментом. Больше не требуется никакой настройки.

Может быть непонятно, зачем мы привязали здесь адрес, относящийся совершенно к другому серверу. На самом деле стандартная практика — указывать для файла HTML ресурсы, расположенные на различных серверах (рис. 3.3).

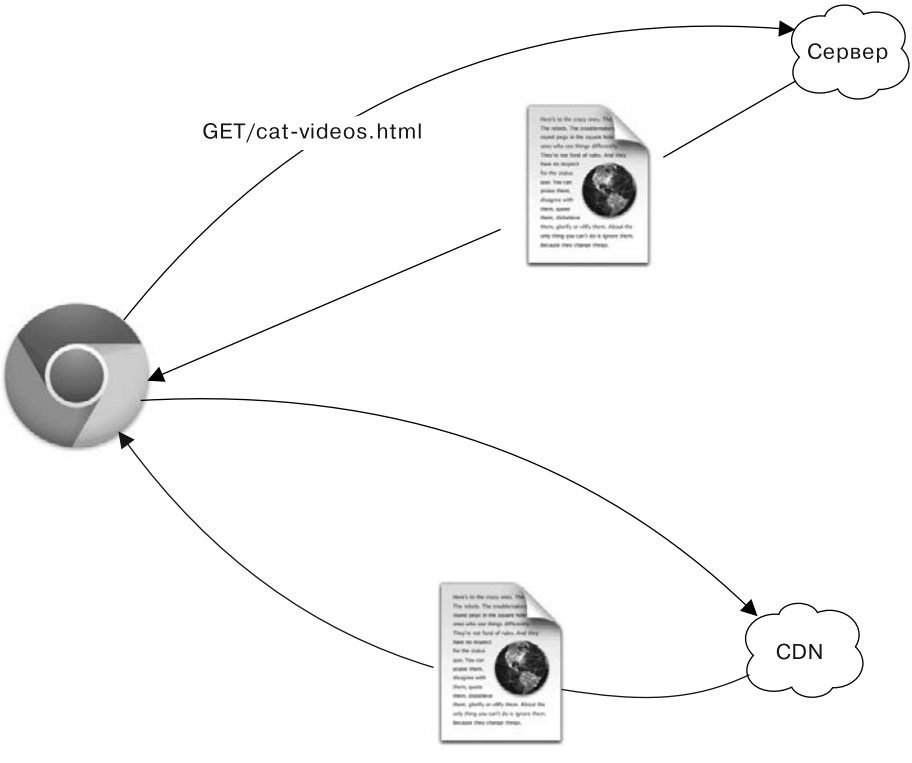


Рис. 3.3. Запрос ресурсов с различных серверов

В данном случае `normalize.css` расположен на `cdnjs.com` — общедоступном сервере, являющемся частью *сети доставки контента* (content delivery network, CDN). У CDN имеются серверы по всему миру, каждый — с копией одних и тех же файлов. Когда пользователи запрашивают файл, то получают его с ближайшего сервера, что сокращает время загрузки. На `cdnjs.com` есть множество версий популярных библиотек и фреймворков клиентской части.

Подготовка HTML для стилизации

В предыдущей главе мы создали таблицу стилей `styles.css`, а в текущей главе нам предстоит добавить в нее *правила оформления*. Но прежде, чем начинать добавлять стили, необходимо задать в HTML целевые объекты, на которые будут ссылаться правила оформления.

Нам нужно добавить атрибуты `class` для идентификации элементов `span` с именами выдр в качестве названий миниатюр. Атрибуты `class` — это способ идентификации группы элементов HTML (обычно для целей стилизации). Атрибут `class` со значением `thumbnail-title` позволит легко присвоить стиль сразу всем названиям.

В файле `index.html` добавьте имя класса `thumbnail-title` в качестве атрибута элементов `span` внутри элементов `li` следующим образом:

```
...
    <ul>
      <li>
        <a href="#">
          <img src="img/otter1.jpg" alt="Barry the Otter">
          <span>Barry</span>
          <span class="thumbnail-title">Barry</span>
        </a>
      </li>
      <li>
        <a href="#">
          <img src="img/otter2.jpg" alt="Robin the Otter">
          <span>Robin</span>
          <span class="thumbnail-title">Robin</span>
        </a>
      </li>
      <li>
        <a href="#">
          <img src="img/otter1.jpg" alt="Maurice the Otter">
          <span>Maurice</span>
          <span class="thumbnail-title">Maurice</span>
        </a>
      </li>
      <li>
        <a href="#">
          <img src="img/otter4.jpg" alt="Lesley the Otter">
          <span>Lesley</span>
          <span class="thumbnail-title">Lesley</span>
        </a>
      </li>
      <li>
        <a href="#">
          <img src="img/otter5.jpg" alt="Barbara the Otter">
          <span>Barbara</span>
          <span class="thumbnail-title">Barbara</span>
```

```
        </a>
    </ul>
...
```

Через минуту мы воспользуемся этим именем класса для стилизации всех названий изображений.

Внутреннее устройство стиля

Отдельные стили создаются при написании правил оформления, состоящих из двух основных частей: *селекторов* и *объявлений* (рис. 3.4).

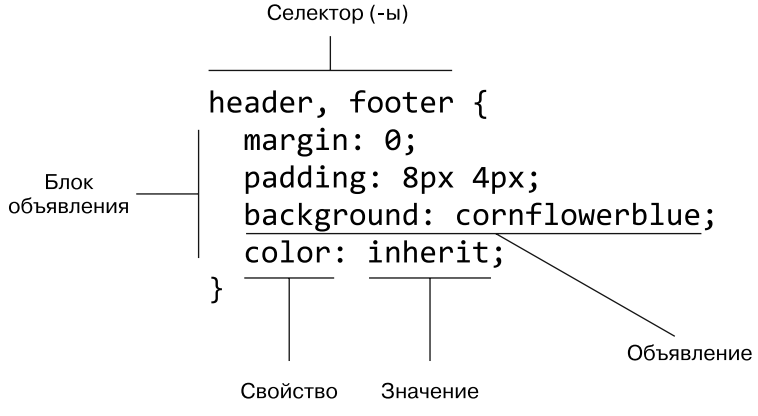


Рис. 3.4. Устройство правила оформления

Первая часть правила оформления — один или несколько селекторов. Селекторы описывают элементы, к которым применяется правило оформления, например `h1`, `span` или `img`. Но использование селекторов не ограничивается только именами тегов. Можно создавать селекторы, применяемые к более узкому набору элементов, путем повышения их *приоритетности (специфичности)*.

Например, можно создавать селекторы, основанные на атрибутах, такие как только что добавленный нами в теги `<span>` атрибут класса `thumbnail-title`. Основанные на атрибутах селекторы более приоритетны, чем селекторы, основанные на именах элементов.

Помимо гарантии, что стили применяются только к ограниченному набору элементов (например, элементов с именем класса `thumbnail-title`, а не всех элементов `<span>`), приоритетность также определяет относительный приоритет селектора. Если таблица стилей содержит несколько стилей, которые могут быть применены к одному элементу, стили с селектором большей приоритетности будут использованы вместо стилей, приоритетность селектора которых меньше.

Узнать больше о приоритетности вы можете в разделе «Для самых любознательных» данной главы.

В этой главе вы познакомитесь с множеством различных видов селекторов, отличающихся по приоритетности. Хотя часто имеется несколько способов задать один и тот же объект для стилизации, понимание приоритетности — ключ к выбору наилучшего селектора для удобства сопровождения стилей.

Вторая часть правила оформления — блок объявления, заключенный в фигурные скобки, определяющий применяемые стили. Каждое отдельное объявление в этом блоке включает имя свойства и его значение.

В нашем первом правиле оформления будет использоваться атрибут `class`, который мы только что добавили в качестве селектора, чтобы применить стили к именам выдр.

Наше первое правило оформления

Чтобы использовать атрибут `class` в качестве селектора в правиле оформления, необходимо поставить перед именем класса точку — вот так: `.thumbnail-title`. Первые стили, которые мы сейчас добавим, зададут цвета фона и переднего плана для класса `.thumbnail-title`.

Откройте файл `styles.css` и вставьте в него наше правило оформления:

```
.thumbnail-title {
  background: rgb(96, 125, 139);
  color: rgb(202, 238, 255);
}
```

Мы расскажем вам о цветах чуть позже в данной главе. Пока что просто посмотрите, что поменялось. Сохраните файл `styles.css` и убедитесь, что утилита browser-sync запущена. Если ее необходимо перезапустить, воспользуйтесь командой, которая выглядит вот так:

```
browser-sync start --server --browser "Google Chrome"
                   --files "stylesheets/*.css, *.html"
```

В результате откроется наша веб-страница в браузере Chrome (рис. 3.5).

Как вы можете видеть, цвет фона для названий миниатюр — насыщенный серо-синий, а цвет шрифта — более светлый синий. Получилось элегантно.

Продолжаем стилизацию названий миниатюр: вернитесь к `styles.css` и добавьте следующий код в уже существующее правило оформления для класса `.thumbnail-title`, как показано тут:

```
.thumbnail-title {
  display: block;
```

```
  margin: 0;
  padding: 4px 10px;

  background: rgb(96, 125, 139);
  color: rgb(202, 238, 255);
}
```

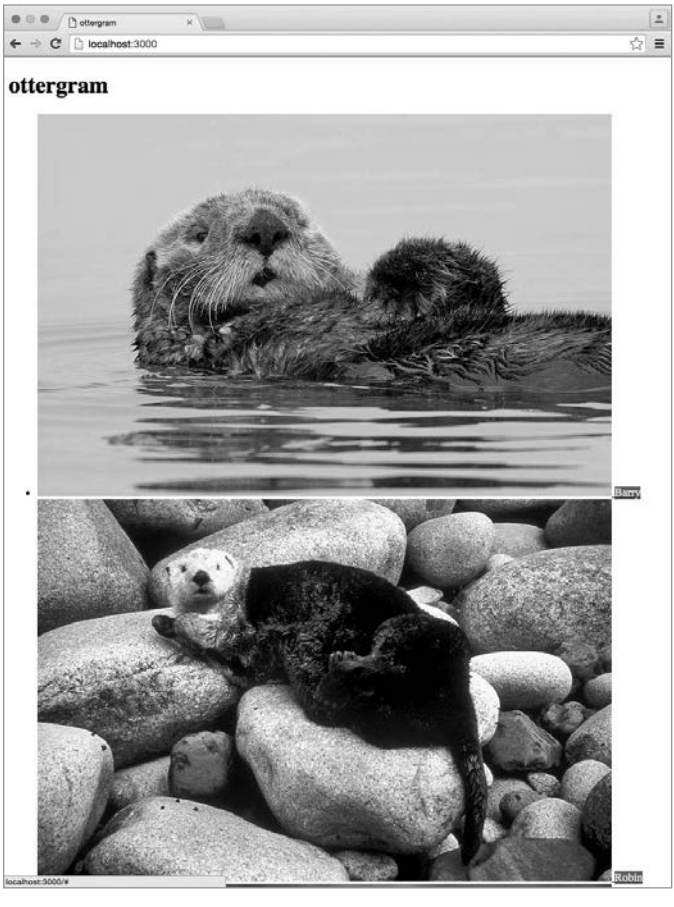


Рис. 3.5. Чуть более красочный Ottergram

Эти три добавленных объявления влияют на *блок* элемента. Для каждого тега HTML, у которого существует визуальное представление, браузер рисует на странице прямоугольник. Чтобы определить размеры этого прямоугольника, браузер использует схему, называемую *стандартной блочной моделью* или просто блочной моделью.

Блочная модель. Чтобы понять, что такое блочная модель, посмотрим на ее представление в DevTools. Сохраните `styles.css`, перейдите в браузер Chrome и убедитесь, что инструменты DevTools открыты (рис. 3.6).

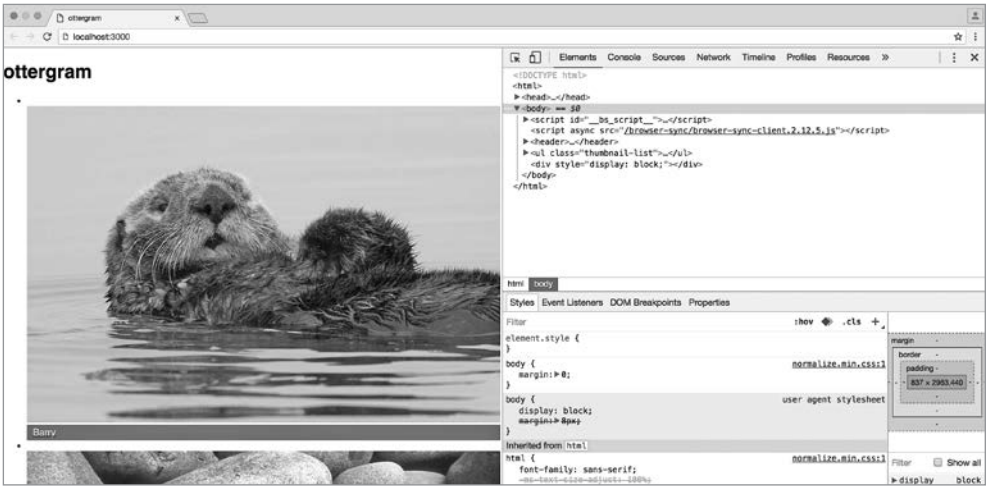


Рис. 3.6. Исследуем блочную модель

Нажмите кнопку слева вверху на панели элементов (это кнопка Inspect Element (Исследовать элемент)). Теперь проведите указателем мыши над словом ottergram на веб-странице. Когда вы остановитесь над ним, DevTools окружит заголовок синим и желто-оранжевым прямоугольником (рис. 3.7).

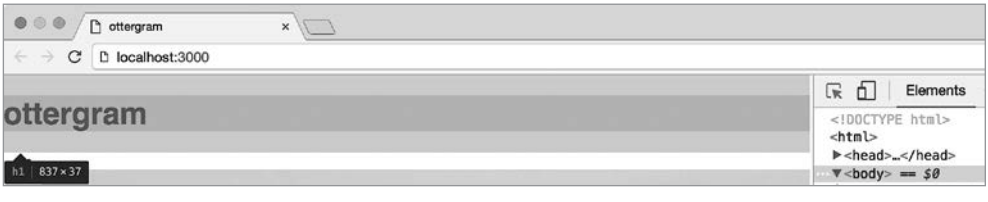


Рис. 3.7. Вид заголовка при нахождении над ним указателя мыши

Щелкните на слове ottergram на веб-странице. Хотя многоцветное перекрытие больше не видно, элемент выбран — и древовидное представление DOM на панели элементов будет развернуто для отображения и подсветки соответствующего тега `<h1>`.

Прямоугольная схема в нижнем правом углу панели элементов представляет блочную модель для элемента `h1`. Можно видеть, что цвета некоторых областей схемы такие же, как и прямоугольник, который перекрывал заголовок, когда вы его исследовали (рис. 3.8).

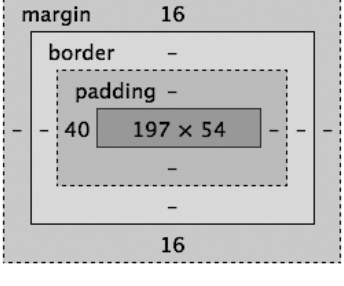


Рис. 3.8. Просмотр блочной модели для элемента

Блочная модель включает четыре аспекта отрисовываемого для элемента прямоугольника (которые DevTools визуализирует на схеме четырьмя различными цветами).

- Контент (в центре) — визуальный контент, в данном случае текст.
- Поле (вокруг контента) — прозрачная область вокруг контента.
- Граница (вокруг поля) — граница, которую можно сделать видимой, вокруг контента и поля.
- Отступ (вокруг границы) — прозрачная область вокруг границы.

Числа на рис. 3.8 — величины в *пикселах* (пиксел — единица измерения, равная наименьшей прямоугольной области экрана компьютера, которая может отображать отдельный цвет). В случае элемента `h1` области контента было выделено 197 × 54 пиксела (ваши значения могут отличаться в зависимости от размера окна браузера). Слева имеется поле размером 40 пикселов. Граница установлена равной 0 пикселам, и есть отступ 16 пикселов сверху и снизу от элемента.

Откуда берется величина отступа? У каждого браузера есть таблица стилей по умолчанию, называемая *таблицей стилей агента пользователя*, на тот случай, если HTML-файл ее не предоставляет. Задаваемые нами стили перекрывают стили по умолчанию. Поскольку мы не указали значения для блока элемента `h1`, были использованы стили по умолчанию.

Теперь вы готовы уяснить смысл добавленных нами объявлений стилей:

```
.thumbnail-title {
  display: block;
  margin: 0;
  padding: 4px 10px;

  background: rgb(96, 125, 139);
  color: rgb(202, 238, 255);
}
```

Объявление `display: block` меняет блок для всех элементов `.thumbnail-title` таким образом, что они теперь занимают все пространство по ширине, отводимое им содержащими их элементами (обратите внимание, что на рис. 3.8 фоновый цвет для названий теперь охватывает более широкую область). Другие значения `display`, такие как `display: inline`, с которым вы познакомитесь позднее, устанавливают ширину элемента в зависимости от его содержимого.

Мы также установили значение отступа для названий миниатюр равным 0 и поле, равное двум различным значениям — 4px и 10px (px — сокращение от pixels (пикселы)). Эта настройка задает размер поля равным конкретным значениям в пикселах, перекрывая размеры по умолчанию, задаваемые таблицами стилей агента пользователя.

Поля, отступы и некоторые другие стили могут быть записаны в сокращенном виде, когда одно значение применяется для нескольких свойств. Мы воспользовались

этим приемом и здесь: при указании двух значений для полей первое используется для обоих горизонтальных полей (сверху и снизу), а второе — для обоих вертикальных (слева и справа). Можно также указать одно значение для всех четырех сторон или задать отдельное значение для каждой стороны.

Подытоживая, отметим: новые объявления гласят, что блок для каждого из элементов класса `.thumbnail-title` будет заполнять всю ширину своего контейнера без отступов и с полями 4px сверху и снизу и 10px с правой и левой сторон.

Наследование стилей

Теперь мы добавим стили, чтобы поменять размер и внешний вид текста.

Добавим новое правило оформления в файл `styles.css`, чтобы задать размер шрифта для элемента `body`. Для этого мы воспользуемся другой разновидностью селектора (селектором элемента), просто указав имя элемента.

```
body {
  font-size: 10px;
}

.thumbnail-title {
  display: block;
  margin: 0;
  padding: 4px 10px;

  background: rgb(96, 125, 139);
  color: rgb(202, 238, 255);
}
```

Это правило оформления устанавливает значение свойства `font-size` элемента `body` равным `10px`.

Вам нечасто придется задействовать селекторы элементов в ваших таблицах стилей, поскольку редко приходится применять один и тот же стиль для каждого случая использования какого-либо тега. Помимо этого, селекторы элементов ограничивают возможности повторного применения стилей (иначе может возникнуть необходимость набирать одно и то же объявление во всех ваших таблицах стилей). Это снижает удобство сопровождения, когда нужно внести изменения в эти стили.

Но в данном случае использование в качестве целевого объекта элемента `body` — как раз нужный нам уровень приоритетности. Здесь может быть только один элемент `body`, и повторно использовать его стили не придется.

Сохраните `styles.css` и взгляните на веб-страницу в Chrome (рис. 3.9).

Заголовок и названия миниатюр стали меньше. Возможно, вы этого ожидали, а может, и нет. В то время как заголовок находится непосредственно внутри элемента `body`

(там, где вы объявили свойство `font-size`), названия миниатюр вложены на несколько уровней глубже. Однако многие стили, включая размер шрифта, применяются и к определяемым правилом оформления элементам, и к *потомкам* этих элементов.

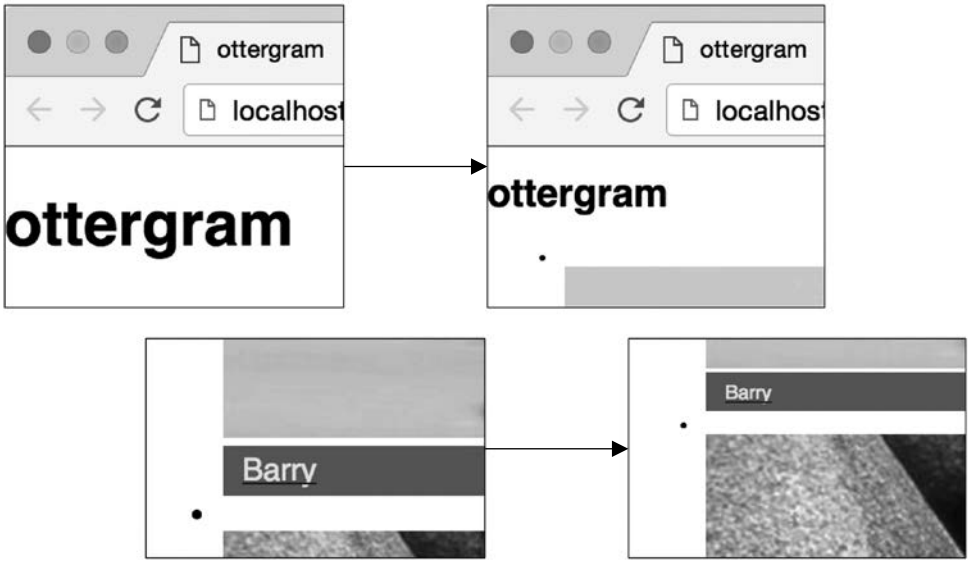


Рис. 3.9. Вид фрагмента страницы после задания размера шрифта для body

Структуру вашего документа можно описать с помощью древовидной схемы, показанной на рис. 3.10. Представить элементы в виде дерева — отличный способ визуализации DOM.

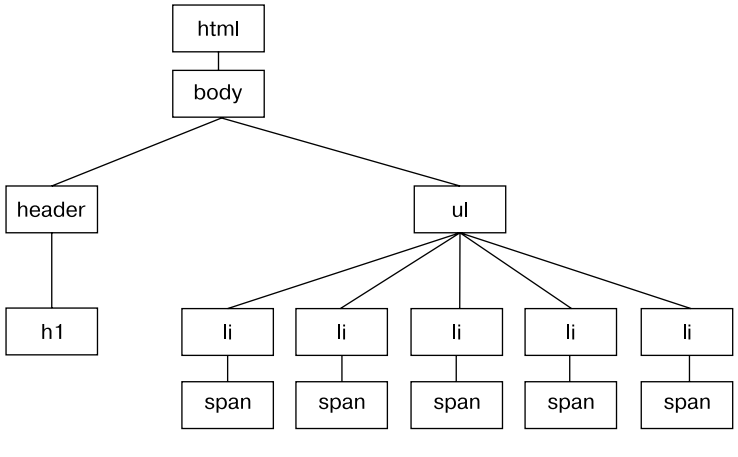


Рис. 3.10. Упрощенная структура Ottergram

Элемент, содержащийся внутри другого элемента, называют его *потомком*. В данном случае элементы `span` — потомки элемента `body` (как и элементы `ul` и соответствующие им `li`), так что они наследуют стиль свойства `font-size` элемента `body`.

В древовидном представлении DOM DevTools найдите и выберите один из элементов `span`. На панели стилей взгляните на блоки, помеченные как Inherited from a (Унаследовано от a), Inherited from li (Унаследовано от li) и Inherited from ul (Унаследовано от ul). Эти три области отображают стили, унаследованные на каждом уровне из таблицы стилей агента пользователя. В области Inherited from body (Унаследовано от body) можно видеть, что свойство `font-size` было унаследовано из набора стилей элемента `body` в `styles.css` (рис. 3.11).

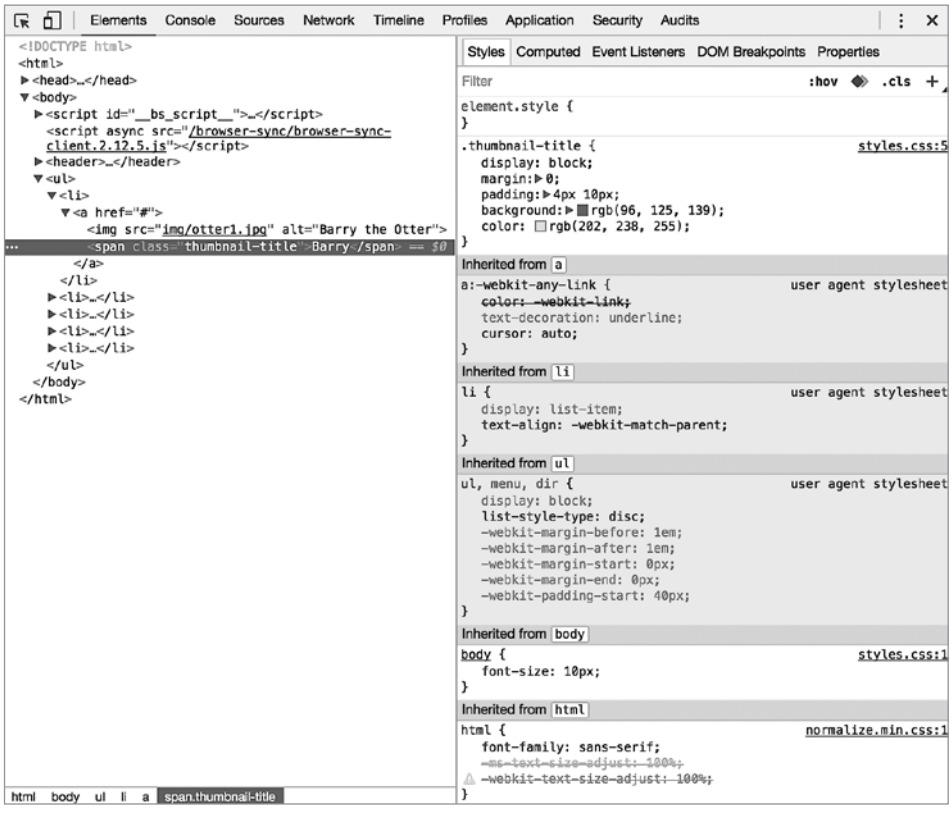


Рис. 3.11. Унаследованные от элементов-предков стили

Но что, если на каком-то уровне, например на уровне элемента `ul`, будет задан другой размер шрифта? Стили от более близкого предка имеют приоритет, так что размер шрифта, заданный в файле `styles.css` для элемента `ul`, будет перекрывать значение, заданное для элемента `body`, а значение размера шрифта, заданное для самого элемента `span`, перекроет их оба.

Чтобы увидеть это, щелкните на элементе `ul` в древовидном представлении DOM. Это позволит вам опробовать стили на лету. Добавленные вами здесь стили будут немедленно отражаться на внешнем виде веб-страницы, но не будут добавляться в файлы проекта.

Вверху панели стилей (на панели элементов) можно увидеть раздел, помеченный как `elements.style`. Щелкните в любом месте между фигурными скобками `elements.style` — и DevTools выведет приглашение на ввод (рис. 3.12).

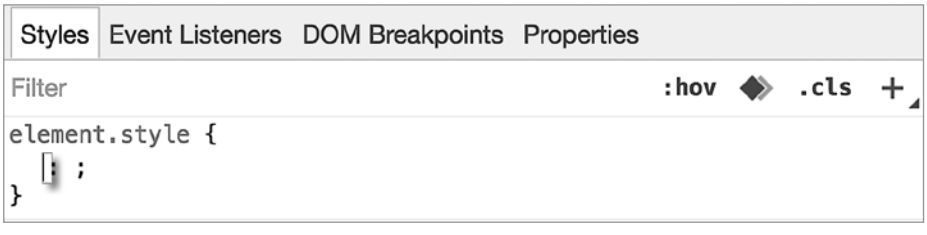


Рис. 3.12. Приглашение на ввод правила оформления

Начните вводить `font-size` — и DevTools предложит вам возможные завершения ввода (рис. 3.13).

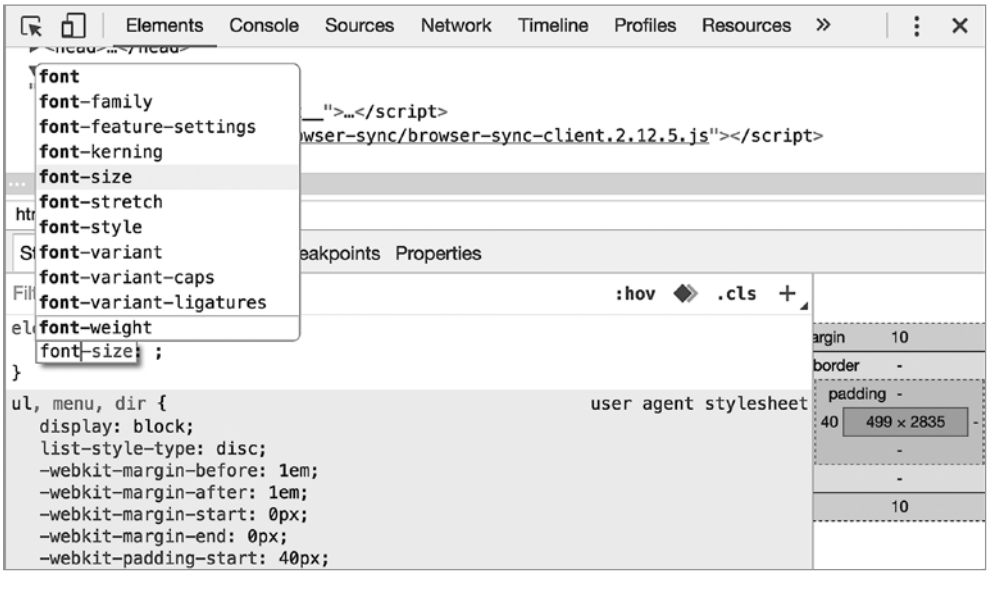


Рис. 3.13. Возможности автодополнения на панели стилей

Выберите свойство `font-size` и нажмите клавишу Tab. Введите значение побольше, например `50px`, и нажмите Enter. Может понадобиться прокрутить страницу, но вы

увидите, что значение `font-size` элемента `ul` перекрыло `font-size` элемента `body` (рис. 3.14).

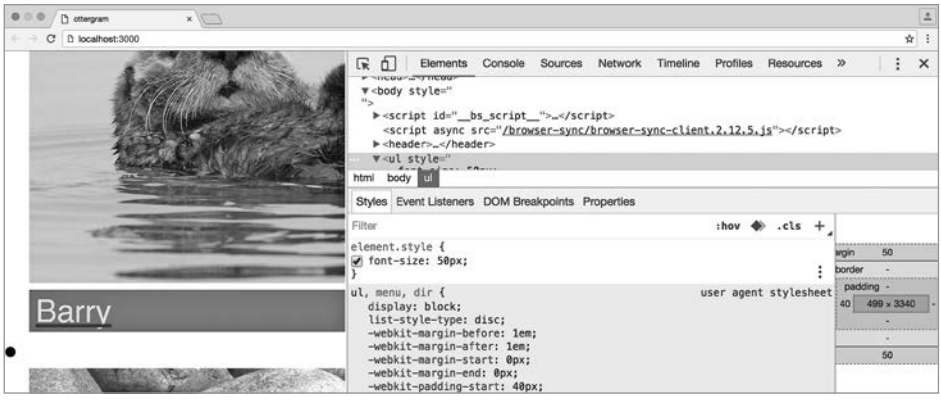


Рис. 3.14. Установка значения 50px для font-size элемента ul

Не все свойства стилей наследуются — `border`, например, нет. Чтобы узнать, наследуется ли свойство, обратитесь к соответствующей странице справочника MDN.

Вернитесь в `styles.css` и поменяйте блок объявления для класса `.thumbnail-title`, чтобы он перекрывал свойство `font-size` элемента `body` и использовал более крупный размер шрифта.

```
body {
  font-size: 10px;
}

.thumbnail-title {
  display: block;
  margin: 0;
  padding: 4px 10px;

  background: rgb(96, 125, 139);
  color: rgb(202, 238, 255);

  font-size: 18px;
}
```

Вы поменяли размер шрифта на `18px` для элементов класса `.thumbnail-title`.

Сохраните файл `styles.css` и можете насладиться названиями миниатюр в браузере Chrome (рис. 3.15).

Ввыглядят они неплохо, но таблица стилей агента пользователя добавляет к элементам `.thumbnail-title` подчеркивания, потому что вы обернули их (вместе с элементами `.thumbnail-image`) в тег-якорь, из-за чего они унаследовали стиль подчеркивания.

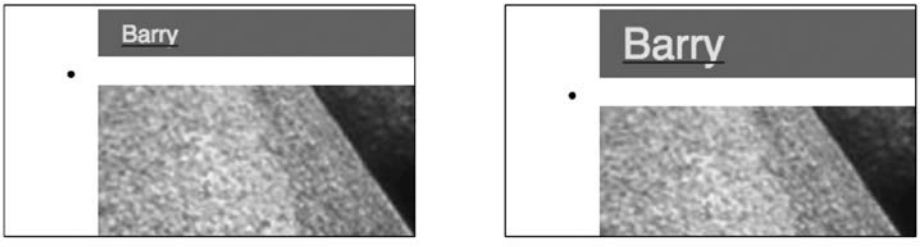


Рис. 3.15. Названия миниатюр с примененными стилями

Подчеркивания нам не нужны, так что уберем их, изменив свойства `text-decoration` тегов-якорей в новом стиле оформления в `styles.css`. Какой же селектор необходимо использовать для этого правила?

Если вы уверены, что хотели бы убрать подчеркивания из названий миниатюр, *а также из всех остальных элементов-якорей в Ottergram*, можно воспользоваться селектором элемента:

```
a {
  /* Объявление стиля */
}
```

(Текст между `/* */` — комментарий CSS. Комментарии в коде игнорируются браузером и предоставляют разработчику возможность делать заметки в коде в качестве справочной информации.)

Если же вы считаете, что могли бы использовать якоря для других целей (и захотите стилизовать их иначе), то можно скомбинировать селектор элемента с *селектором атрибута*:

```
a[href]{
  /* Объявление стиля */
}
```

Этот селектор будет применен ко всем элементам-якорям с атрибутом `href`. Конечно, у элементов-якорей обычно есть атрибуты `href`, так что это не настолько специализированный селектор, чтобы влиять только на названия и изображения миниатюр. Чтобы уточнить селектор атрибута, можно указать также значение атрибута следующим образом:

```
a[href="#"]{
  /* Объявление стиля */
}
```

Этот селектор будет применен только к тем элементам-якорям, значение атрибута `href` которых равно `#`.

Между прочим, можно использовать селекторы атрибутов сами по себе:

```
[href]{
  /* Объявление стиля */
}
```

Ottergram — довольно простой проект, и теги-якоря фактически не будут использоваться ни для чего, кроме миниатюр и их названий. Следовательно, можно спокойно использовать здесь селектор элемента (лучше так и поступить, поскольку это наиболее простое решение с нужным уровнем приоритетности).

Добавим объявление нового стиля в файл `styles.css`:

```
body {
  font-size: 10px;
}

a {
  text-decoration: none;
}

.thumbnail-title {
  ...
}
```

Сохраните файл и загляните в браузер. Подчеркиваний больше нет, названия миниатюр изящно стилизованы (рис. 3.16).

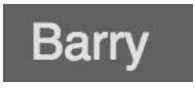


Рис. 3.16. Результат после установки значения text-decoration в none

Обратите внимание, что не следует убирать подчеркивания из ссылок в обычном тексте (который не является явным заголовком, названием или заглавием). Подчеркивания текста ссылок — важный визуальный признак, знакомый пользователям. Мы же это сделали потому, что миниатюрам не нужны подобные визуальные признаки. Пользователи и так будут ожидать, что на них можно щелкнуть кнопкой мыши.

В оставшейся части этой главы мы воспользуемся селекторами классов для стилизации изображений миниатюр, неупорядоченного списка изображений, списка элементов (включающего изображения миниатюр и их названия) и заголовка. Добавьте имена классов к элементам `h1`, `ul`, `li` и `img` в `index.html`, чтобы они были под рукой, когда понадобятся.

```
...
  </head>
  <body>
    <header>
      ~~<h1>ottergram</h1>~~
      <h1 class="logo-text">ottergram</h1>
    </header>
    ~~<ul>~~
```

```
    <ul class="thumbnail-list">
      <li>
      <li class="thumbnail-item">
        <a href="#">
          <img src="img/otter1.jpg" alt="Barry the Otter">
          <img class="thumbnail-image" src="img/otter1.jpg"
               alt="Barry the Otter">
          <span class="thumbnail-title">Barry</span>
        </a>
      </li>
      <li>
      <li class="thumbnail-item">
        <a href="#">
          <img src="img/otter2.jpg" alt="Robin the Otter">
          <img class="thumbnail-image" src="img/otter2.jpg"
               alt="Robin the Otter">
          <span class="thumbnail-title">Robin</span>
        </a>
      </li>
      <li>
      <li class="thumbnail-item">
        <a href="#">
          <img src="img/otter3.jpg" alt="Maurice the Otter">
          <img class="thumbnail-image" src="img/otter3.jpg"
               alt="Maurice the Otter">
          <span class="thumbnail-title">Maurice</span>
        </a>
      </li>
      <li>
      <li class="thumbnail-item">
        <a href="#">
          <img src="img/otter4.jpg" alt="Lesley the Otter">
          <img class="thumbnail-image" src="img/otter4.jpg"
               alt="Lesley the Otter">
          <span class="thumbnail-title">Lesley</span>
        </a>
      </li>
      <li>
      <li class="thumbnail-item">
        <a href="#">
          <img src="img/otter5.jpg" alt="Barbara the Otter">
          <img class="thumbnail-image" src="img/otter5.jpg"
               alt="Barbara the Otter">
          <span class="thumbnail-title">Barbara</span>
        </a>
      </li>
    </ul>
...
```

Указывая в этих элементах имена классов, мы получаем целевые объекты для добавляемых стилей.

Мы предпочитаем селекторы классов другим видам селекторов и рекомендуем вам делать то же самое. Это позволит создавать наглядные имена классов, что сделает код легким в разработке и сопровождении. Вы также сможете добавлять в элемент несколько имен классов, делая их гибким и мощным инструментом стилизации.

Не забудьте сохранить `index.html`, прежде чем продолжить.

Подгоняем изображения под размер окна

Согласно атомарному паттерну стилизации, изображения — следующие в очереди на стилизацию. Если окно браузера будет меньше размера изображений, они будут обрезаться. Добавим в файл `styles.css` правило оформления для `.thumbnail-image`, чтобы размер миниатюр приспосабливался к размеру окна:

```
...
a {
  text-decoration: none;
}

.thumbnail-image {
  width: 100%;
}

.thumbnail-title {
  ...
}
```

Мы задали свойство `width` (ширину) равным `100%`, что ограничивает миниатюру шириной контейнера. Это значит, что изображения будут пропорционально увеличиваться при расширении окна браузера. Попробуйте сохраните `styles.css`, перейдите в браузер и увеличивайте/уменьшайте окно браузера. Изображения будут увеличиваться/сжиматься вместе с окном браузера, всегда сохраняя свои пропорции. Рисунок 3.17 демонстрирует Ottergram в узком и более широком окнах браузера.

Если вы посмотрите более внимательно, то увидите, что поля для `.thumbnail-titles` не определены должным образом, поэтому кажется, что названия относятся к изображениям, находящимся под ними. Исправьте это в файле `styles.css`, задав свойство `display` элемента `.thumbnail-image` равным `block`.

```
...
.thumbnail-image {
  display: block;
  width: 100%;
}
...
```

Теперь пустого пространства между изображением и его названием больше нет (рис. 3.18).

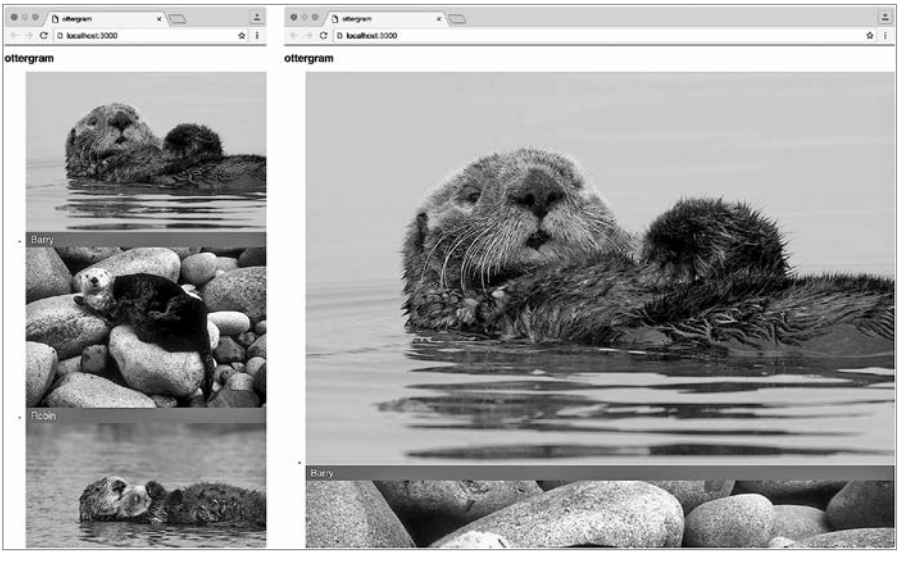


Рис. 3.17. Подгонка изображения по ширине

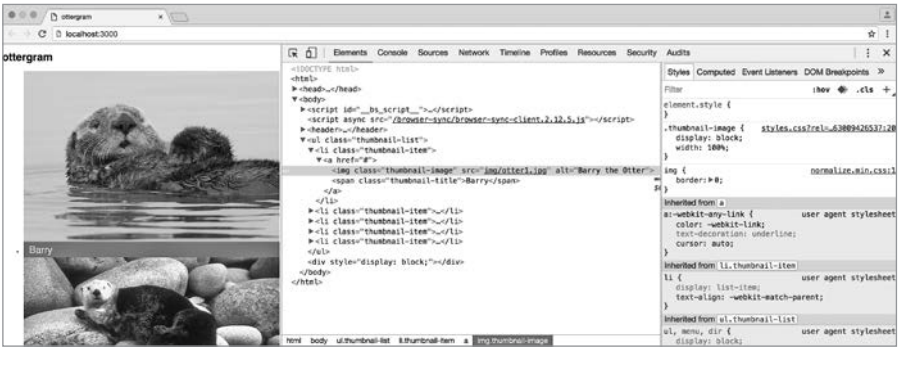


Рис. 3.18. Вид страницы после задания свойства display элемента .thumbnail-image

Почему это работает? Дело в том, что по умолчанию для изображений применяется свойство `display: inline`. Они подчиняются тем же правилам визуализации, что и текст. При визуализации текста буквы отрисовываются вдоль общей базовой линии. У некоторых букв, таких как *«ц»*, *«р»* и *«щ»*, есть *подстрочный элемент* (часть буквы, опускающаяся ниже базовой линии). Для них ниже базовой линии отведено небольшое пустое место.

Задание свойства `display` элемента `.thumbnail-image` равным `block` убирает это пустое место, поскольку здесь нет необходимости размещать какой-либо текст (или какие-либо другие элементы `display: inline`, которые могли бы визуализироваться возле изображения).

Цвет

Пришло время подробнее поговорить про цвет. Добавим следующие цветовые стили для элемента `body` и класса `.thumbnail-item` в файле `styles.css`:

```
body {
  font-size: 10px;
  background: rgb(149, 194, 215);
}
a {
  text-decoration: none;
}
.thumbnail-item {
  border: 1px solid rgb(100%, 100%, 100%);
  border: 1px solid rgba(100%, 100%, 100%, 0.8);
}
...
```

Мы объявили значения для границы класса `.thumbnail-item` дважды. Почему? Обратите внимание, что в этих двух объявлениях используются различные цветовые функции: `rgb` и `rgba`. Цветовая функция `rgba` принимает на входе еще один, четвертый, параметр — *непрозрачность*. Однако некоторые браузеры не поддерживают `rgba`, поэтому два объявления — метод указания *резервного* значения.

Все браузеры увидят первое объявление (`rgb`) и запишут его значение для свойства `border`. Когда не поддерживающие `rgba` браузеры увидят второе объявление, они его не поймут, проигнорируют и будут использовать значение из первого объявления. Браузеры же, которые поддерживают `rgba`, используют значение из второго объявления.

(Недоумеваете, почему цвет фона `body` объявлен с помощью целочисленного значения, а цвет границы класса `.thumbnail-item` — в виде процентов? Мы очень скоро вернемся к этому вопросу.)

Сохраните файл `styles.css` и перейдите в браузер (рис. 3.19).

Вы можете увидеть в DevTools, что Chrome поддерживает `rgba`. То, что цвет `rgb` не используется, отмечено зачеркиванием соответствующего стиля (рис. 3.20).

Теперь в DevTools выберите тег `body`. На панели стилей посмотрите на объявление только что добавленного цвета фона. Слева от значения RGB находится маленький квадратик, демонстрирующий, как выглядит этот цвет.

Щелкните на квадратике — откроется инструмент выбора цвета (рис. 3.21). Он позволяет выбрать цвет и отображает значение цвета CSS в множестве различных форматов.

Чтобы увидеть значение цвета фона в различных форматах цветов, нажмите на стрелки «вверх-вниз» справа от значений RGBA. Вы можете переходить по кругу между форматами HSLA, HEX и RGBA.

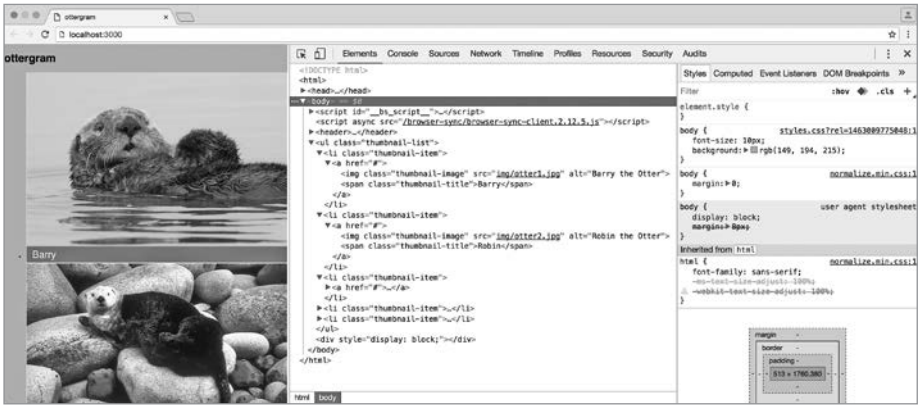


Рис. 3.19. Цвет фона и границы

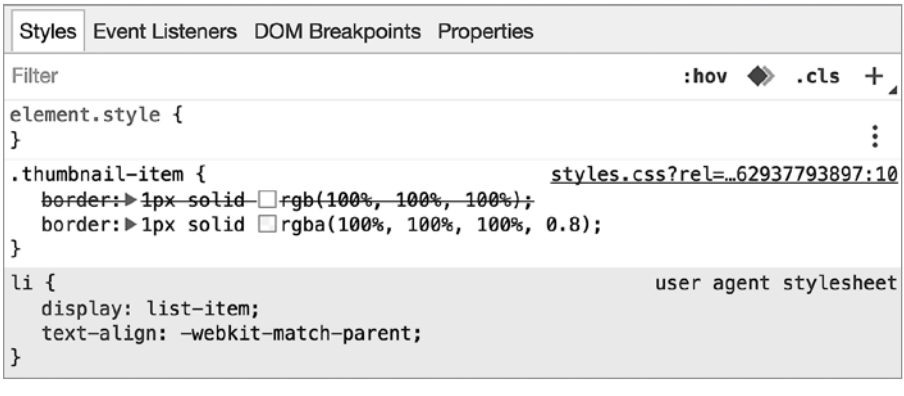


Рис. 3.20. Если браузер поддерживает функцию rgba, используется именно она

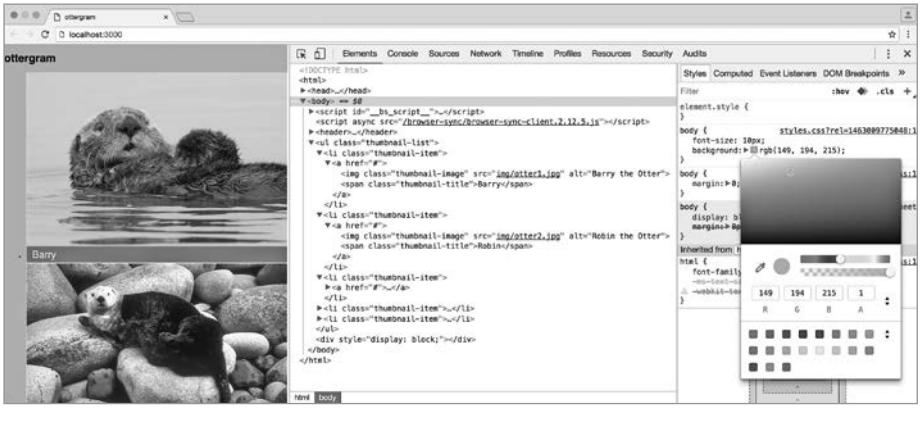


Рис. 3.21. Инструмент выбора цвета на панели стилей

Формат HSLA (расшифровывается как Hue, Saturation, Lightness, Alpha — «оттенок», «насыщенность», «яркость», «альфа») используется реже других в том числе из-за того, что некоторые наиболее популярные средства проектирования не предоставляют подходящих для CSS точных значений HSLA. Если HSLA вас заинтересовал, зайдите в HSLA Explorer по адресу `css-tricks.com/examples/HSLaExplorer`.

Рассмотрим подробнее HEX-значение для цвета фона: `#95C2D7`. HEX (hexadecimal — «шестнадцатеричный») — старейший формат задания цвета. Каждая цифра представляет собой значение от 0 до 15 (если вы не знакомы с шестнадцатеричным счислением, кратко отметим, что оно использует латинские буквы от A до F наряду с цифрами). Каждая пара цифр может представлять значение от 0 до 255. Слева направо пары цифр соответствуют интенсивности красного, зеленого и синего каналов в задаваемом цвете (рис. 3.22).

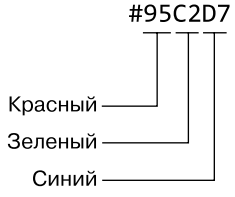


Рис. 3.22. Шестнадцатеричные значения, соответствующие красному, зеленому и синему значениям

Многие считают шестнадцатеричные значения непонятными. Современная альтернатива — использование значений RGB (красный, зеленый и синий). В этой модели каждому цвету также присваивается значение от 0 до 255, но значения обозначаются более привычными десятичными числами и разделены по цветам. Как уже упоминалось, для обладающих бо́льшими возможностями браузеров с помощью четвертого значения можно задавать непрозрачность (opacity) или прозрачность (transparency) цвета: от 0.0 (полностью прозрачный) до 1.0 (полностью непрозрачный). Непрозрачность официально называется значением *альфа* (буква A в аббревиатуре RGBA). Значение RGBA цвета фона элемента `body` равно `(149, 194, 215, 1)`.

В качестве альтернативы заданию целочисленных значений для красного, зеленого и синего каналов можно также использовать процентные соотношения, как мы сделали для границ класса `.thumbnail-item`. Функциональной разницы между этими двумя вариантами нет. Просто не мешайте в кучу проценты и целочисленные значения в одном объявлении.

Кстати, чтобы помочь в выборе приятной цветовой гаммы, Adobe предоставляет бесплатный онлайн-инструмент по адресу color.adobe.com.

Выравнивание расстояний между элементами

У Ottergram теперь есть приятные цвета, напоминающие о родном для выдр океане. Но добавление цветов обнаружило нежелательное пустое пространство внутри границ элементов с классом `.thumbnail-item`. Кроме того, надоедливые маркеры отвлекают внимание от созерцания выдр.

Чтобы избавиться от маркеров, установим в `styles.css` значение свойства `list-style` элементов с классом `.thumbnail-list` равным `none`:

```
...
.thumbnail-item {
  border: 1px solid rgb(100%, 100%, 100%);
  border: 1px solid rgba(100%, 100%, 100%, 0.8);
}

.thumbnail-list {
  list-style: none;
}

.thumbnail-image {
  ...
```

А чтобы избавиться от пустого пространства, воспользуемся тем же методом, что и для класса `.thumbnail-image`. У каждого элемента с классом `.thumbnail-item` имеется по умолчанию пустое пространство для размещения элементов списка — аналогично тому, как у элементов с классом `.thumbnail-image` имеется пустое пространство для размещения текста. Добавьте объявление `display: block` в класс `.thumbnail-item`, чтобы убрать его:

```
...
.thumbnail-item {
  display: block;
  border: 1px solid rgb(100%, 100%, 100%);
  border: 1px solid rgba(100%, 100%, 100%, 0.8);
}
...
```

С этими дополнениями маркеры и лишнее пустое место над фотографиями исчезают, в итоге макет выглядит более изящно (рис. 3.23).

Но зачем использовать маркированный список, если маркеры нам не нужны? Лучше выбрать теги HTML, основываясь на том, что нам нужно, а не на том, как браузер стилизует их по умолчанию. В данном случае нам нужен неупорядоченный список изображений, так что тег `ul` нам как раз подойдет. Контейнер `ul` для изображений позволит представить их в виде прокручиваемого списка, когда мы будем в главе 4 добавлять в проект изображение по умолчанию. Тот факт, что браузер по умолчанию отображает теги `ul` с маркерами, несущественен, ведь их легко можно убрать.

Далее мы собираемся выровнять расположение элементов списка. Между отдельными элементами с классом `.thumbnail-item` в настоящий момент нет пустого места. Добавим отступы между смежными миниатюрами.

Однако нам не хотелось бы добавлять отступ для *всех* элементов списка. Почему нет? Потому что у заголовка уже имеется отступ, так что для первого элемента списка он не требуется. Это значит, что мы не можем использовать селектор класса `.thumbnail-item`, по крайней мере не сам по себе. Вместо этого мы воспользуемся синтаксисом селектора, выбирающего элементы на основе их отношения к другим элементам.

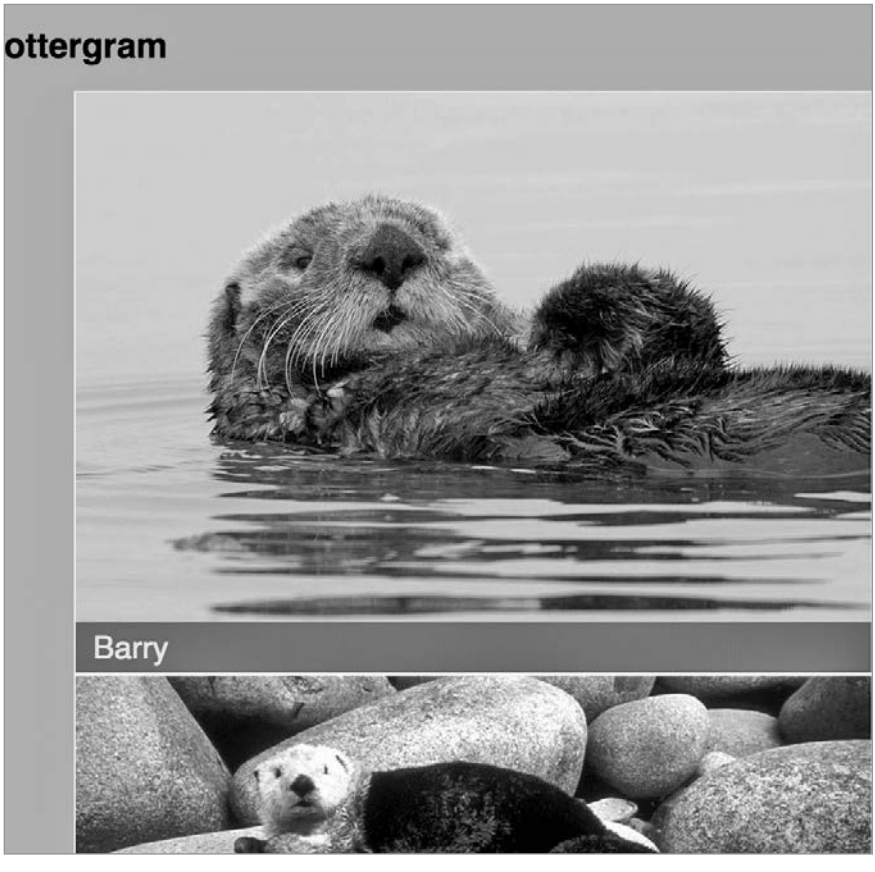


Рис. 3.23. Усовершенствованный макет

Селекторы отношений. Взгляните снова на схему проекта на рис. 3.10. Она очень напоминает генеалогическое дерево, не правда ли? Это сходство послужило основанием для названий селекторов отношений: *селекторы потомков (descendent selectors), дочерние селекторы (child selectors), родственные селекторы (sibling selectors) и смежные родственные селекторы (adjacent sibling selectors)*.

Синтаксис селекторов отношений включает два селектора (например, селекторы классов или селекторы элементов), объединенные символом, называемым *комбинатором (combinator)*, определяющим целевое отношение между ними. Чтобы понимать, как работают селекторы отношений, важно помнить, что браузер читает синтаксис селекторов *справа налево*. Рассмотрим несколько примеров.

Целевой объект селектора потомков — любой элемент заданного типа, являющийся потомком другого заданного элемента. Например, синтаксис для выбора любого элемента `span`, являющегося потомком элемента `body`, будет таким:

```
body span {
  /* Объявления стиля */
}
```

В этом синтаксисе комбинатор не используется. Поскольку синтаксис читается справа налево, его целевым объектом будет любой элемент `span` — потомок элемента `body`, что в нашем коде означает названия миниатюр. Он также повлияет на все `span`, которые будут добавляться внутри элемента `header` или еще где-то внутри `body`.

Обратите внимание, что можно также использовать селектор класса (или селектор атрибута, или какой-либо еще тип селектора) внутри селектора отношений, так что можно переписать вышеприведенный селектор:

```
body .thumbnail-title {
  /* Объявления стиля */
}
```

Целевыми объектами дочерних селекторов выступают элементы заданного типа, являющиеся непосредственными потомками другого заданного элемента. В синтаксисе дочерних селекторов используется комбинатор `>`. Синтаксис для выбора всех имеющихся в настоящий момент в Ottergram элементов `span` будет следующим:

```
li > span {
  /* Объявления стиля */
}
```

Читая справа налево, этот селектор выбирает все `span`, являющиеся непосредственными потомками элемента `li` (названия миниатюр).

Синтаксис селектора родственных элементов поддерживает комбинатор `~`. Как вы могли предположить, он дает возможность выбирать элементы, у которых один родитель. Однако из-за некоторых особенностей родственных селекторов результаты могут оказаться не совсем теми, которых вы ожидали. Рассмотрим следующий пример:

```
header ~ ul {
  /* Объявления стиля */
}
```

Этот селектор выберет все элементы `ul`, которым *предшествует* элемент `header` с тем же родительским элементом. Он успешно выберет `ul` Ottergram, поскольку у него есть имеющий того же родителя `header`, предшествующий ему в коде. Однако перевернутый вариант этого синтаксиса (`ul ~ header`) приведет к тому, что не будет выбрано никаких элементов, потому что не существует `header`, которому бы предшествовал `ul`.

Последний тип селектора отношений — смежный родственный селектор, предназначенный для выбора элементов, которым *непосредственно* предшествует имеющий того же родителя элемент заданного типа. Комбинатор смежных родственных селекторов обозначается как `+`:

```
li + li {
  /* Объявления стиля */
}
```

При использовании такого синтаксиса будут выбраны все элементы `li`, которым непосредственно предшествует имеющий того же родителя элемент `li`. В результате объявленные стили будут применены к `li` со второго по пятый, но не к первому, поскольку ему не предшествует другой `li`. Обратите внимание, что общий родственный и смежный родственный селекторы дадут одинаковый результат благодаря довольно простой структуре Ottergram.

Возвращаемся к текущей задаче: добавить отступ вверху каждого элемента списка, кроме первого. Если воспользоваться селектором потомков или дочерним селектором для выбора класса `.thumbnail-item` либо элементов `span` или `li`, отступ будет применен ко всем пяти миниатюрам. Но раз мы хотим стилизовать все, кроме первого элемента, то станем использовать в файле `styles.css` синтаксис смежного родственного селектора для добавления верхнего отступа только к тем миниатюрам, которым непосредственно предшествует другая миниатюра.

```
...
a {
  text-decoration: none;
}

.thumbnail-item + .thumbnail-item {
  margin-top: 10px;
}

.thumbnail-item {
  ...
```

Сохраните файл и проверьте в браузере, что получилось (рис. 3.24).

Обратите внимание, что DevTools позволяют легко выяснить путь вложенности элемента (это может быть полезно при создании селекторов отношений). Если вы щелкнете на одном из элементов `span` внутри одного из элементов `li`, то увидите его путь внизу панели элементов (рис. 3.25).

Чтобы внести еще одну небольшую поправку во внешний вид списка миниатюр, вернитесь в файл `styles.css` и перекройте значение поля, унаследованное элементом `ul` от таблицы стилей агента пользователя, чтобы у изображений больше не было отступов.

```
...
.thumbnail-list {
  list-style: none;
  padding: 0;
}
...
```

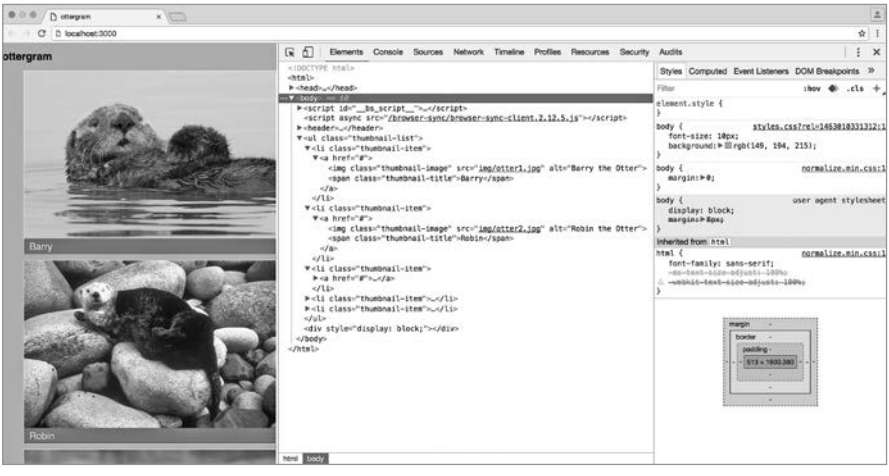


Рис. 3.24. Пространство между смежными элементами с классом .thumbnail-item

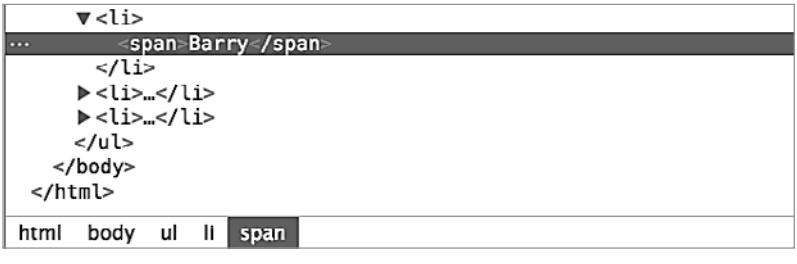


Рис. 3.25. Путь вложенности, отображаемый на панели элементов

Как обычно, сохраните ваш файл и перейдите в браузер, чтобы оценить результат (рис. 3.26).

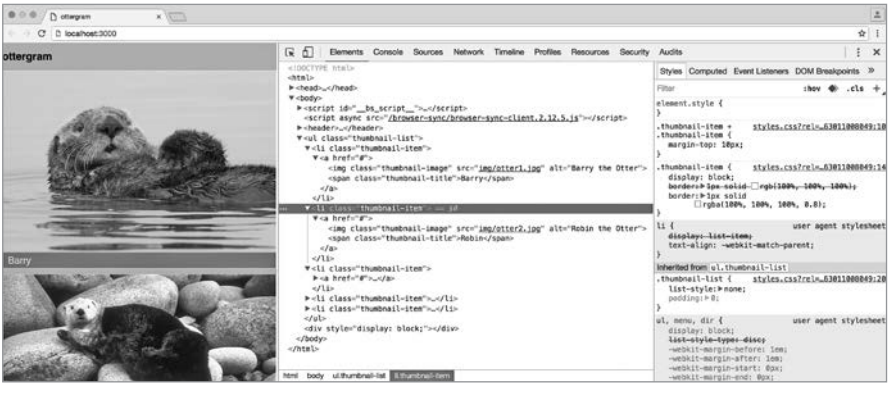


Рис. 3.26. Результат после установки ul без полей

Ottergram начинает выглядеть весьма элегантно. Еще порция стилизации заголовка — и у вас будет изящная статическая веб-страница.

Добавление шрифта

Ранее вы добавили в элемент `h1` класс `.logo-text`. Воспользуемся этим классом в качестве селектора для нового правила оформления в файле `styles.css`. Вставьте его после стилей для тега-якоря. (В целом порядок стилей имеет значение только при наличии нескольких наборов правил для одного селектора. В Ottergram стили расположены примерно в том же порядке, как они встречаются в коде. Это дело вкуса — вы можете организовывать свои стили так, как вам удобнее.)

```
...
a {
  text-decoration: none;
}

.logo-text {
  background: white;

  text-align: center;
  text-transform: uppercase;
  font-size: 37px;
}

.thumbnail-item + .thumbnail-item {
  ...
```

Во-первых, мы задали для заголовка белый цвет фона. Затем центрировали текст внутри элемента с классом `.logo-text` и использовали свойство `text-transform`, чтобы преобразовать его в верхний регистр. Наконец, задали размер шрифта. Результат представлен на рис. 3.27.

Ottergram выглядит отлично. Отлично... но немного уныло для сайта с *выдрами*. Чтобы чуть оживить его, можно для заголовка выбрать шрифт, отличающийся от используемого по умолчанию в таблице стилей агента пользователя.

Мы включили несколько шрифтов в файлы ресурсов, которые вы уже скачали и добавили в свой каталог проекта. Чтобы добраться до них, необходимо скопировать каталог `fonts` в ваш проект. Поместите его *внутрь* папки `stylesheets` (рис. 3.28).

Теперь нам только нужно, чтобы какие-нибудь стили указывали на эти шрифты.

Файлы ресурсов содержат множество форматов каждого шрифта. Как и всегда, различные производители браузеров поддерживают разные виды шрифтов. Чтобы обеспечить поддержку максимально широкого набора браузеров, вам придется включить их все в ваш проект. Да, их все.

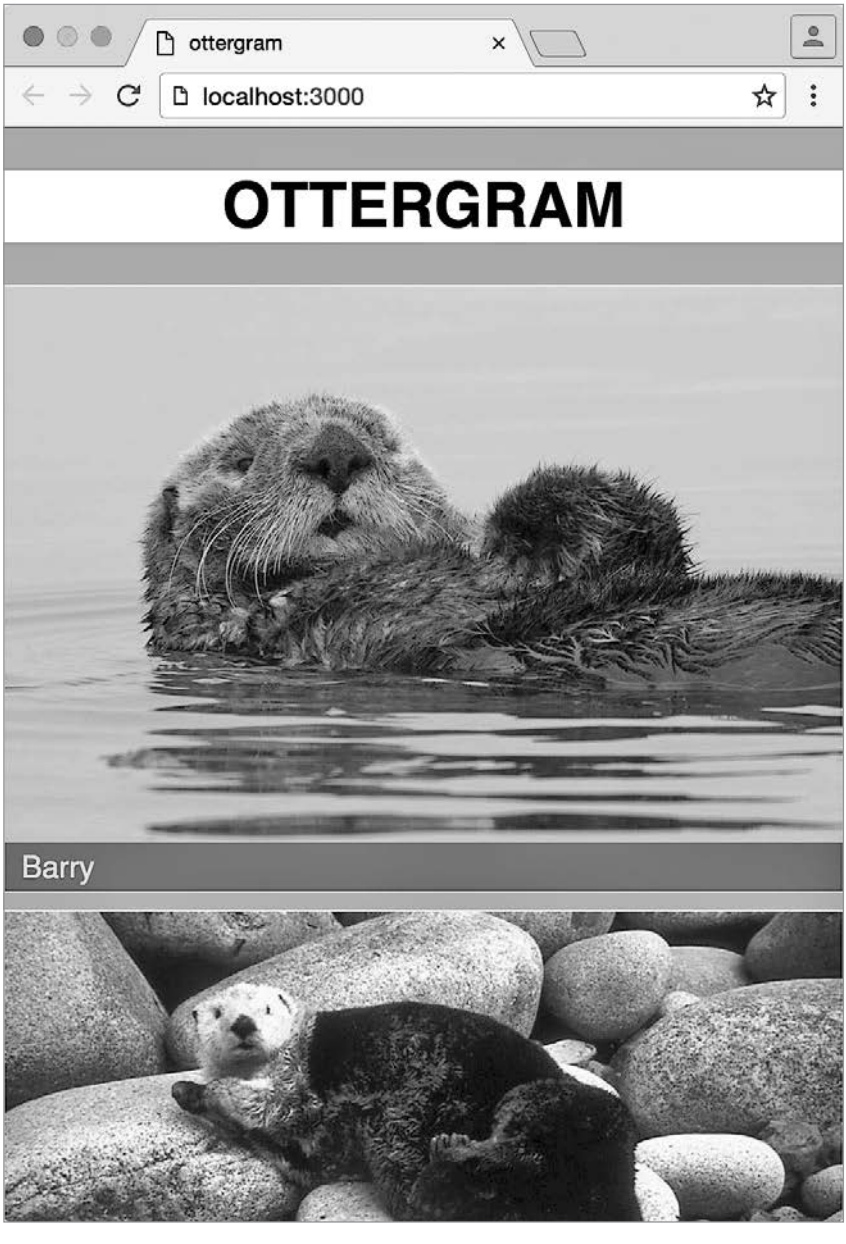


Рис. 3.27. Стилизация заголовка

Для упрощения задачи синтаксис правила `@font-face` позволяет задавать пользовательское название для семейства шрифтов, которое затем можно применять в других стилях.

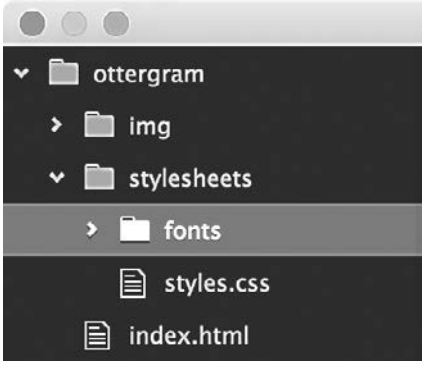


Рис. 3.28. Каталог fonts внутри каталога stylesheets

Блок `@font-face` немного отличается от уже используемых нами блоков объявлений. Он содержит три основные части:

- во-первых, свойство `font-family` (его значение представляет собой строку, определяющую пользовательское имя шрифта, которое можно использовать в своем файле CSS);
- во-вторых, несколько объявлений `src`, задающих различные файлы шрифтов (обратите внимание, что порядок важен!);
- и наконец, объявления, меняющие внешний вид шрифта, например `font-weight` и `font-style`.

Добавьте объявление `@font-face` для семейства шрифтов `lakeshore` вверху файла `styles.css` и объявление стиля, чтобы использовать для класса `.logo-text` новый шрифт.

```
@font-face {
    font-family: 'lakeshore';
    src: url('fonts/LAKESHOR-webfont.eot');
    src: url('fonts/LAKESHOR-webfont.eot?#iefix') format('embedded-opentype'),
         url('fonts/LAKESHOR-webfont.woff') format('woff'),
         url('fonts/LAKESHOR-webfont.ttf') format('truetype'),
         url('fonts/LAKESHOR-webfont.svg#lakeshore') format('svg');
    font-weight: normal;
    font-style: normal;
}

body {
  font-size: 10px;
  background: rgb(149, 194, 215);
}

a {
  text-decoration: none;
}
```

```
.logo-text {
 background: white;

 text-align: center;
 text-transform: uppercase;
 font-family: lakeshore;
 font-size: 37px;
}
...
```

Правда, надо отметить, что правильное объявление правила `@font-face` может оказаться непростой задачей, поскольку существенен порядок отдельных значений `url`. Нелишне хранить копию этого объявления для справочных целей. Не помешает также заглянуть в документацию по сниппетам редактора Atom на сайте flight-manual.atom.io/using-atom/sections/snippets, чтобы разобраться, как создать свой собственный сниппет (шаблон).

После объявления пользовательского правила `@font-face` у остальной части вашего CSS появляется доступ к новому значению `lakeshore` для свойства `font-family`. Чтобы применить новый шрифт, необходимо указать `font-family: lakeshore` в объявлении класса `.logo-text`.

Сохраните файл `styles.css`, переключитесь на Chrome и оцените, насколько это приятно, когда твоя веб-страница такая же симпатичная, как и выдры (рис. 3.29).

Мы проделали огромную работу по стилизации в этой главе, зато Ottergram выглядит потрясающе! В следующей главе сделаем его еще лучше, добавив интерактивную функциональность.

Бронзовое упражнение: изменение цвета

Поменяйте цвет фона для элемента `body`. Воспользуйтесь инструментом выбора цвета в DevTools (см. рис. 3.21).

Чтобы получить доступ к более усовершенствованной цветовой палитре, отправляйтесь на сайт color.adobe.com и создайте свою собственную схему для цветов фона `body` и `.thumbnail-title`.

Для самых любознательных: приоритетность! Конфликты селекторов

Вы уже научились перекрывать стили. Например, вставляли ссылку для файла `normalize.css` перед ссылкой на файл `styles.css`. Это заставило браузер использовать стили из `normalize.css` в качестве базовых — с приоритетом ваших стилей над базовыми.

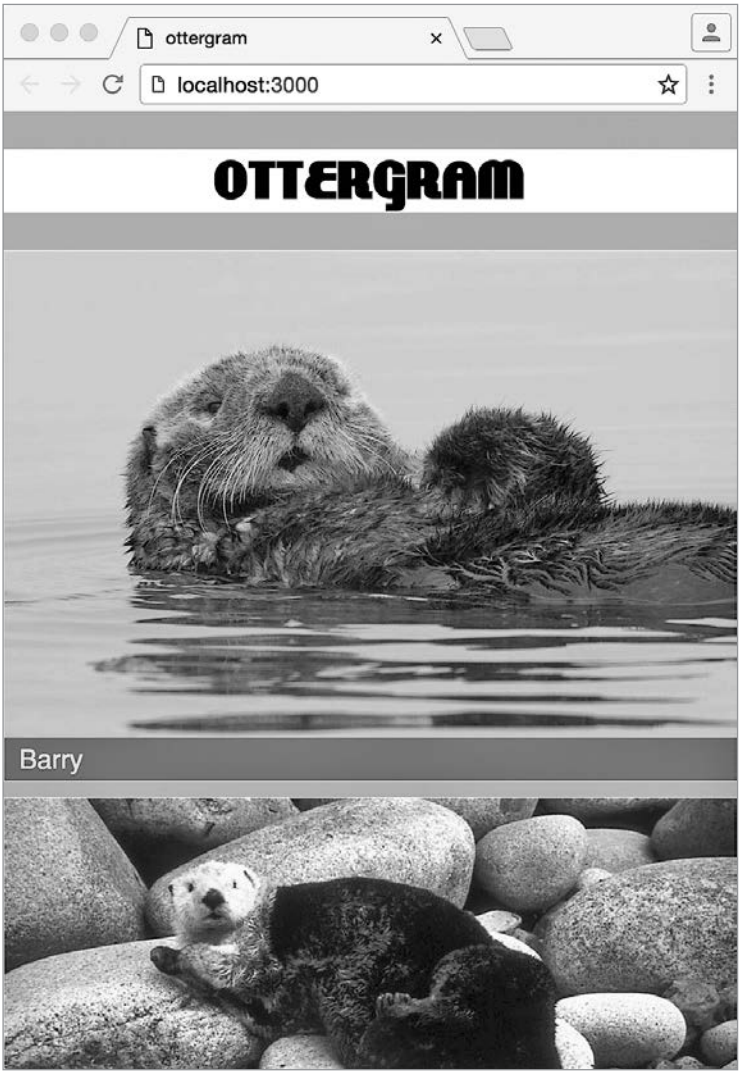


Рис. 3.29. Применяем пользовательский шрифт к заголовку

Это основной принцип того, как браузер выбирает, какие стили применять к элементам на странице, известный разработчикам клиентской части под названием *«новизна»*. При обработке правил CSS более новые могут перекрывать обработанные ранее правила. Контролировать порядок обработки браузером CSS можно путем смены порядка тегов `<link>`.

Это достаточно просто, если у правил один и тот же селектор (например, если ваши CSS-правила и правила из файла `normalize.css` должны были объявить различные

значения свойства `margin` для элемента `body`). В этом случае браузер выбирает более свежее объявление. Но что насчет элементов, которые соответствуют более чем одному селектору?

Допустим, у вас есть два следующих правила в CSS Ottergram:

```
.thumbnail-item {
  background: blue;
  }

li {
  background: red;
}
```

Им обоим соответствуют наши элементы `li`. Какой же цвет фона будет у наших элементов `<li>`? Хотя правило `li { background: red; }` более свежее, будет использовано правило `.thumbnail-item { background: blue; }`. Почему? Потому, что оно задействует селектор класса — более приоритетный (то есть ему присвоено более высокое значение приоритетности), чем селектор элемента.

У селекторов классов и селекторов атрибутов — одинаковая степень приоритетности (и у обоих более высокая приоритетность, чем у селекторов элементов). Самая высокая степень приоритетности — у *селекторов ID*, с которыми вы пока еще не сталкивались. Если мы присвоим элементу атрибут `id`, то сможем создать селектор ID — более приоритетный, чем любой другой.

Атрибуты ID выглядят так же, как и все остальные. Например:

```
<li class="thumbnail-item" id="barry-otter">
```

Для использования ID в качестве селектора необходимо поставить перед ним `#`:

```
.thumbnail-item {
  background: blue;
}

#barry-otter {
  background: green;
}

li {
  background: red;
}
```

В этом примере тег `<li>` выбирается всеми тремя селекторами, но у него окажется зеленый фон, поскольку приоритетность селектора ID наивысшая. Порядок правил здесь значения не имеет, потому что у всех различная приоритетность.

Небольшое примечание относительно селекторов ID: лучше их избегать. Значения ID должны быть уникальными для документа, поэтому мы не смогли бы

воспользоваться атрибутом `id="barry-otter"` ни для какого другого элемента в документе. Хотя у селекторов ID и наивысшая приоритетность, связанные с ними стили нельзя применять повторно, что делает их неподходящим для целей сопровождения решением.

Чтобы узнать больше о приоритетности, загляните на страницу MDN developer.mozilla.org/en-US/docs/Web/CSS/Specificity.

Калькулятор приоритетности по адресу specificity.keegan.st — отличный инструмент для сравнения приоритетности различных селекторов. Воспользуйтесь им, чтобы лучше разобраться, как вычисляется приоритетность.

4 Создание адаптивных макетов с помощью флекс-блоков

Одна из задач разработчика клиентской части — сделать работу пользователей с ней как можно более удобной независимо от применяемых устройств или браузеров.

Так было не всегда, и частично за это в ответе компании — производители браузеров. В первые годы развития Интернета производители браузеров вели настоящие войны. Каждый изобретал новые нестандартные возможности в попытке превзойти других. В ответ на это веб-разработчики создавали схемы для определения, какой именно браузер запросил документ и какой размер экрана используется. На основе этой информации выдавалась соответствующая версия документа.

К сожалению, это осложняло разработку клиентской части: приходилось создавать множество копий всех страниц сайта — каждая из них делалась со своей разметкой и стилями, которые бы подходили для конкретной версии браузера, работающей при определенном размере экрана. Поддержка всех этих копий не только отнимала много времени, но и раздражала разработчиков.

К счастью, время войн браузеров прошло: создатели браузеров теперь стремятся следовать одному и тому же набору стандартных возможностей — и нынешние разработчики клиентской части могут сосредоточить внимание на единой базе кода для всего сайта. Но это не значит, что разработчики не могут создавать специально приспособленные под различные размеры экрана и ориентацию страницы. Новые технологии — такие как *флекс-блоки*, с которыми мы познакомимся в данной главе, — позволяют макетам приспосабливаться к размеру экрана пользователя, не требуя для этого создания отдельных документов.

В этой главе мы собираемся превратить Ottergram из простого списка изображений в настоящий пользовательский интерфейс, готовый к работе с интерактивным

контентом. С помощью флекс-блоков и позиционирования CSS мы создадим набор интерфейсных компонентов, которые будут по мере необходимости приспосабливаться к размеру окна браузера (общий макет останется неизменным). К концу данной главы в Ottergram появятся прокручиваемый список миниатюр изображений и область для демонстрации большой версии одного из изображений (рис. 4.1).

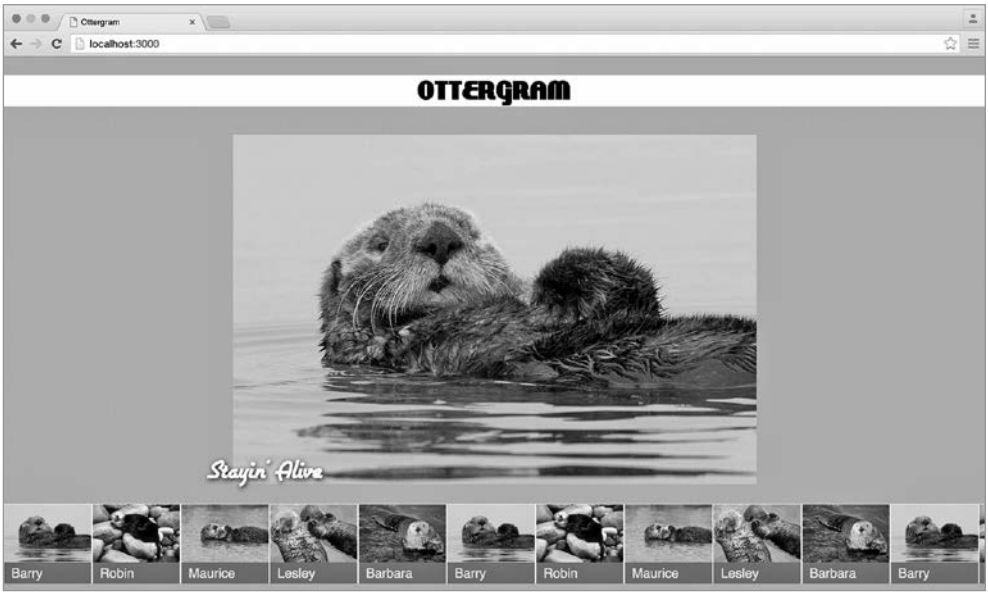


Рис. 4.1. Ottergram с гибким макетом

Мы сделаем это за два подхода. Сначала добавим минимальные разметку и стили, необходимые для отображения на странице большого изображения и для того, чтобы уменьшать миниатюры в размере и делать их прокручиваемыми. Затем добавим стили, которые позволят частям страницы растягиваться и сжиматься при изменении размера окна (для приспособления к экранам различного размера).

Расширяем интерфейс

С момента появления iPhone набирает обороты тенденция заходить в Интернет со смартфона, а не с настольного компьютера или ноутбука.

Для разработчиков клиентской части это означает, что лучший подход к разработке — использование *принципа «сначала мобильные»*, то есть сначала приложение проектируется в расчете на маленькие экраны, затем расширяется до размера экрана планшетов и, наконец, наращивается до размера экрана настольных компьютеров.

Простой макет Ottergram уже подходит для мобильных устройств. Он отображает текст и изображения в подходящем для маленьких экранов масштабе. Благодаря этому можно перейти сразу к добавлению в наш макет следующего уровня сложности.

Прокручиваемый вертикально список выдр неплох, но можно сделать его еще лучше, предоставив пользователю возможность видеть увеличенный вариант изображения. План для Ottergram состоит в том, чтобы список прокручивался горизонтально одновременно с отображением увеличенного изображения. Пока что увеличенное изображение будет находиться ниже списка. Этот план схематически показан на рис. 4.2.

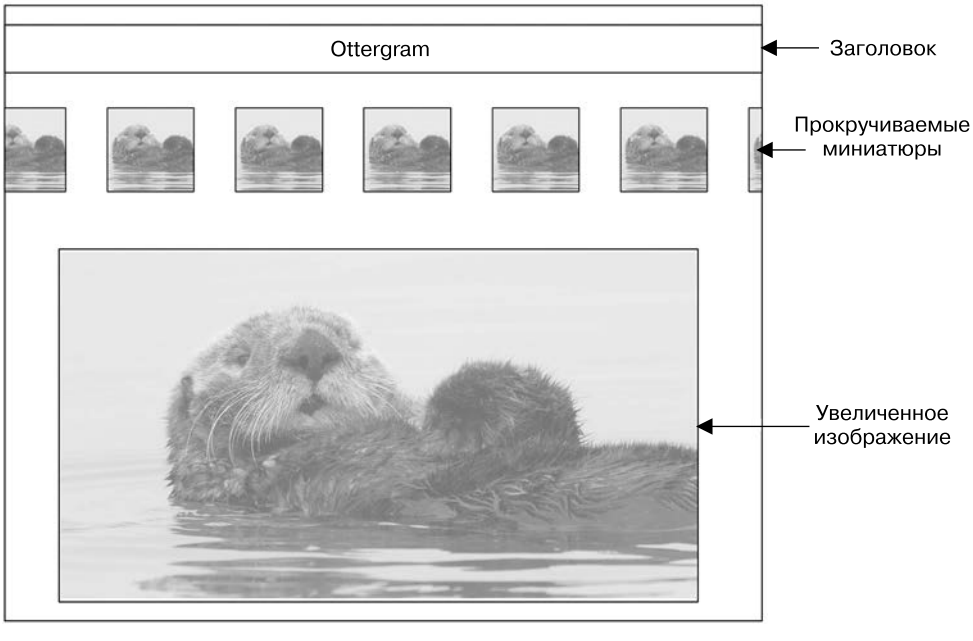


Рис. 4.2. Новый макет для Ottergram

Начнем с добавления увеличенного изображения.

Добавляем увеличенное изображение

Пока что наше увеличенное изображение будет одним и тем же. В главе 6 мы добавим функциональность, чтобы пользователь мог нажать на любую из миниатюр и увеличить ее.

Добавим в файл `index.html` новый раздел кода для создания увеличенного изображения:

```
...
      <li class="thumbnail-item">
        <a href="#">
          <img class="thumbnail-image" src="img/otter5.jpg"
               alt="Barbara the Otter">
          <span class="thumbnail-title">Barbara</span>
        </a>
      </li>
    </ul>

    <div class="detail-image-container">
      <img class="detail-image" src="img/otter1.jpg" alt="">
      <span class="detail-image-title">Stayin' Alive</span>
    </div>

  </body>
</html>
```

Мы добавили тег `<div>` с классом `detail-image-container`. `<div>` — обобщенный контейнер, обычно используемый для применения стилей к заключенному в нем контенту. Именно для этого он нам и нужен.

Внутри тега `<div>` мы вставили тег `<img>`, чтобы отобразить увеличенный вариант фотографии выдры. Мы также вставили там тег `<span>`, оборачивающий текст названия увеличенного изображения. Теги `<img>` и `<span>` получили названия `detail-image` и `detail-image-title` соответственно.

Сохраните файл `index.html`, перейдите в файл `styles.css` и в его конце ограничьте ширину нашего нового класса `.detail-image`.

```
...
.thumbnail-title {
  ...
}

.detail-image {
  width: 90%;
}
```

Сохраните файл `styles.css` и запустите утилиту browser-sync, чтобы открыть проект в браузере Chrome (рис. 4.3). (Команда для этого: `browser-sync start --server --browser "Google Chrome" --files "stylesheets/*.css,*.html"`.)

Изображение с классом `.detail-image` станет отображаться внизу страницы и будет чуть уже, чем миниатюры. Сделав ширину увеличенного изображения равной 90 % ширины его контейнера, мы оставили рядом с ним немного свободного пространства. Браузер поместит в это пространство текст из элемента с классом `.detail-image-title`. (Стиль для этого текста мы зададим чуть позднее в текущей главе.)

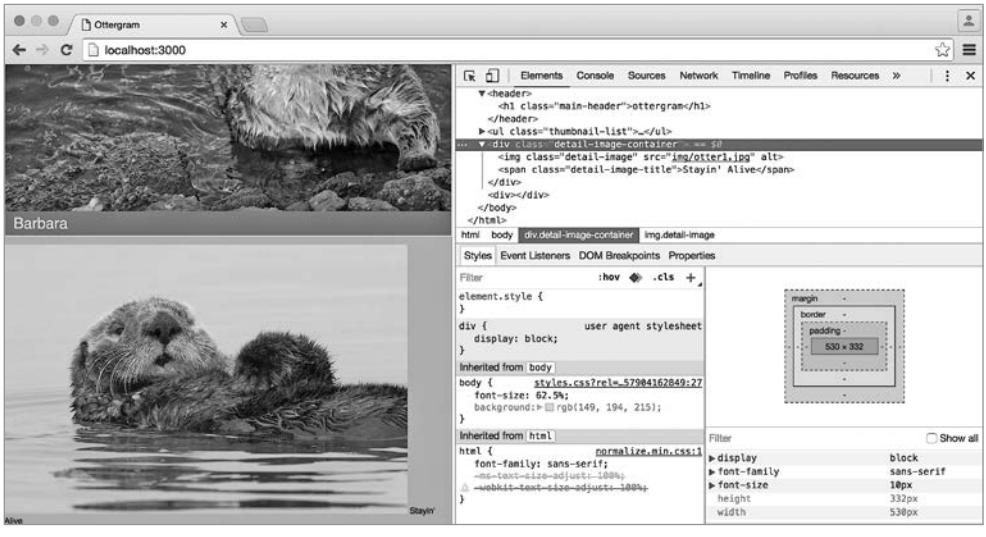


Рис. 4.3. Первоначальная стилизация для увеличенного изображения

Если вы измените размер страницы, то обнаружите ошибку: приспосабливаясь к новой ширине, миниатюры выталкивают увеличенное изображение из области видимости. (Мы разберемся с этой проблемой позднее в данной главе.)

Горизонтальный макет для миниатюр

Далее исправим классы `.thumbnail-list` и `.thumbnail-item` для горизонтальной прокрутки изображений.

Чтобы было удобнее тестировать прокрутку, скопируем все пять элементов `<li>` в файл `index.html` (это обеспечит нам достаточный объем контента для прокрутки). Выделите все строки между `<ul class="thumbnail-list">` и `</ul>`, скопируйте их и вставьте результат сразу над `</ul>`. У вас должно получиться десять элементов списка, содержащих изображения с `otter1.jpg` по `otter5.jpg` по два раза каждое.

Не забудьте сохранить файл `index.html` после завершения. Дублирование контента при разработке — хороший метод создания более устойчивого к ошибкам проекта. Оно позволяет увидеть, как создаваемый код обрабатывает возникающие при реальной эксплуатации ситуации.

Для горизонтально прокручиваемого списка миниатюр необходимо ограничить ширину каждой из них (миниатюры нужно разместить горизонтально в одну линию).

Свойство `display: block`, которое мы уже использовали несколько раз, не даст нам желаемого результата. Оно приводит к визуализации браузером разрывов строки

до и после элемента. Однако родственный ему стиль `display: inline-block` идеально подходит для решения данной задачи. При использовании значения `inline-block` блок элемента отрисовывается так же, как если бы вы объявили `display: block`, но без разрывов строки, что позволяет выстроить миниатюры в одну линию.

Добавим объявление свойства `width` и поменяем объявление свойства `display` для класса `.thumbnail-item` в файле `styles.css`:

```
...
.thumbnail-item {
  display: block;
  display: inline-block;
  width: 120px;
  border: 1px solid rgb(100%, 100%, 100%, 0.8);
  border: 1px solid rgba(100%, 100%, 100%, 0.8);
}
...
```

Обратите внимание, что линтер редактора Atom может выдать предупреждение Using width with border can sometimes make elements larger than you expect (Использование width совместно с border может иногда сделать элементы больше, чем вы ожидаете). Это происходит, потому что свойство `width` применяется только к относящейся к контенту — не к полю и не к границе — части блока элемента. Ничего предпринимать из-за этого предупреждения не нужно.

При ширине элемента с классом `.thumbnail-item`, равной абсолютному значению `120px`, проблема с `.thumbnail-item` тоже решена, поскольку `.thumbnail-item` приспосабливается к ширине своего контейнера.

Почему бы просто не задать `width: 120px` для класса `.thumbnail-item`? Дело в том, что нам хотелось бы, чтобы ширина элементов с классами `.thumbnail-image` и `.thumbnail-title` была одинаковой. Вместо того чтобы задавать свойство `width` для каждого из них, мы задаем его для их общего родительского элемента. Таким образом, если понадобится изменить ширину, то нужно будет поменять ее только в одном месте. Лучше сразу писать код так, чтобы вложенные элементы приспосабливались к своим контейнерам.

Сохраните файл `styles.css` и посмотрите на страницу в Chrome. Вы увидите, что элементы `.thumbnail-image` стоят бок о бок, но когда они заполняют ширину своего контейнера, то переходят на следующую строку (рис. 4.4).

Чтобы достичь желаемого поведения миниатюр при прокрутке, настроим класс `.thumbnail-list` в файле `styles.css` так, чтобы предотвратить перенос на следующую строку и разрешить прокрутку.

```
...
.thumbnail-list {
  list-style: none;
```

```
  padding: 0;

  white-space: nowrap;
  overflow-x: auto;
}
...
```

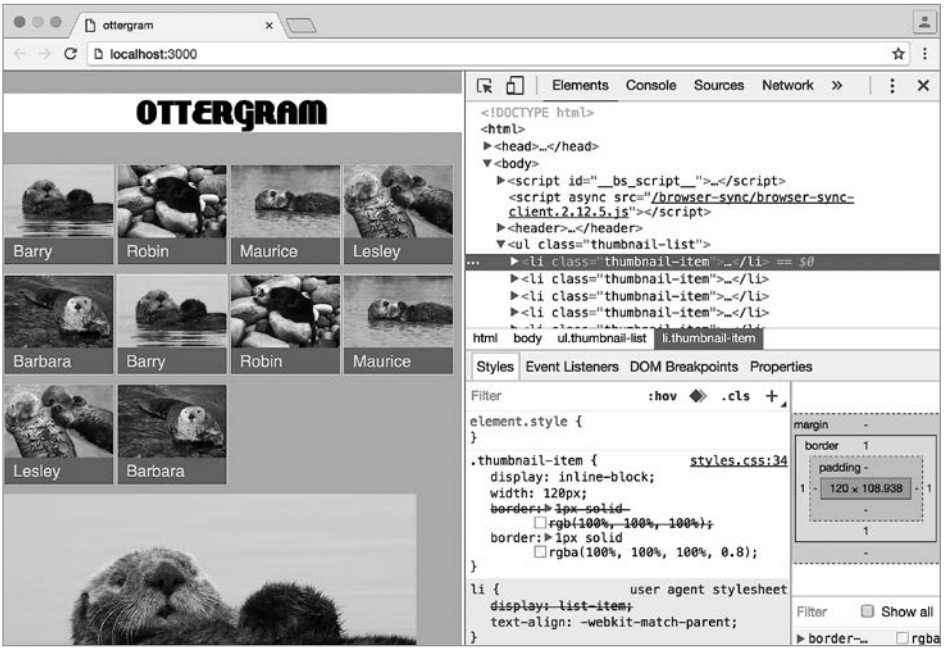


Рис. 4.4. Использование значения inline-block приводит к созданию строк, автоматически переносящихся на следующую строку

Объявление `white-space: nowrap` предотвращает перенос элементов с классом `.thumbnail-item` на новую строку. Объявление `overflow-x: auto` сообщает браузеру о необходимости добавить полосу прокрутки вдоль горизонтального пространства (ось *X*) элемента с классом `.thumbnail-list`, чтобы приспособиться к переполнению контента (то есть к контенту, не вмещающемуся в элемент с классом `.thumbnail-list`). Без этого объявления пришлось бы прокручивать всю веб-страницу, чтобы увидеть дополнительные миниатюры.

Сохраните файл и взгляните на результаты в браузере. Миниатюры выстроены в один ряд, и есть возможность прокручивать список горизонтально (рис. 4.5).

Это хорошее начало для создания улучшенного интерфейса Ottergram. Для некоторых размеров экрана все работает отлично. Однако страница неидеальна, поскольку не приспосабливается к широкому диапазону размеров, особенно намного меньших или бóльших, чем у монитора, за которым вы сейчас работаете.

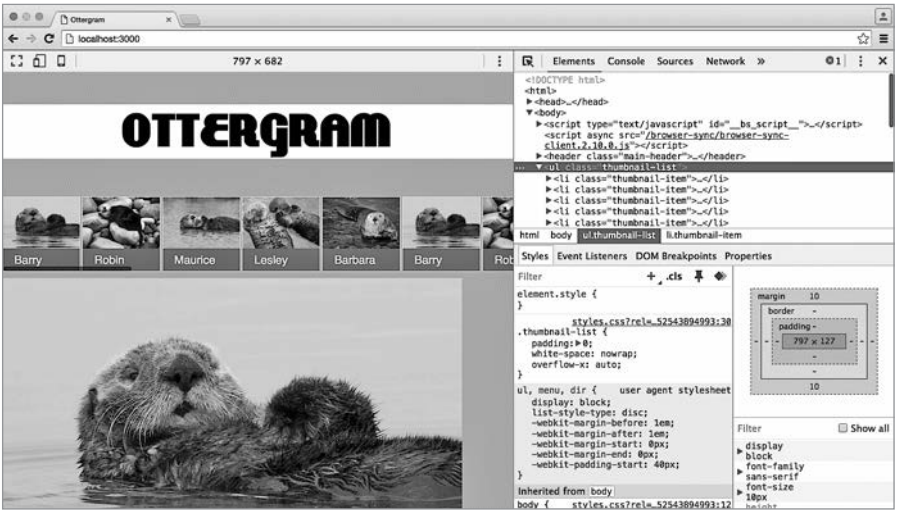


Рис. 4.5. Горизонтально прокручиваемые миниатюры

В следующих двух разделах мы добавим код, который сделает макет Ottergram более гибким и даст возможность его UI — *интерфейсу пользователя* — переключаться между различными макетами, чтобы подстраиваться под разные размеры экрана.

Флекс-блок

Мы уже задавали для свойства `display` значения `block` и `inline`. Встроенные элементы, такие как элементы миниатюр в нашем новом прокручиваемом списке, располагаются один рядом с другим, в то время как блочные элементы представлены отдельными горизонтальными рядами. Таким образом, блочные элементы располагаются сверху вниз, а встроенные элементы — слева направо (рис. 4.6).

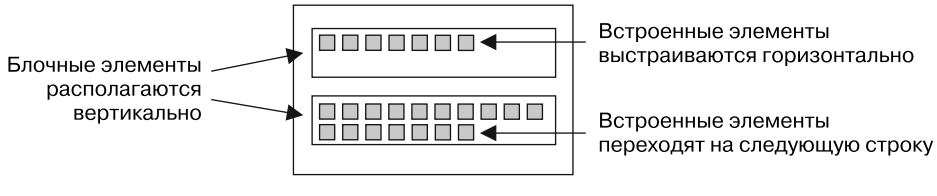


Рис. 4.6. Блочные элементы по сравнению со встроенными

Свойство `display` говорит браузеру, как элемент должен располагаться в макете. Для блогов и онлайн-энциклопедий вполне подходят значения `inline` и `block`. Но для макетов, выглядящих как приложения, скажем для веб-интерфейса электронной почты и сайтов социальных медиа, используется новая спецификация

CSS, позволяющая элементам располагаться динамически. Это *гибкая блочная модель*, иначе говоря — флекс-блоки.

Свойства флекс-блоков CSS гарантируют, что области для миниатюр и увеличенного изображения будут заполнять весь экран и сохранять пропорции относительно друг друга. Это как раз то, что нам требуется для Ottergram. Можно также использовать свойства флекс-блоков для центрирования содержимого области увеличенного изображения как по горизонтали, так и по вертикали (эту задачу сложно решить с помощью свойств стандартной блочной модели).

Создание флекс-контейнера

Прежде чем добавить первое свойство флекс-блока, в файле `styles.css` присвоим для элементов `<html>` и `<body>` значение `height: 100%`.

`<html>` — корневой элемент нашего дерева DOM, а `<body>` — отрисовываемый внутри него дочерний элемент. Задание для них обоих высоты, равной `100%`, позволяет контенту заполнять все окно браузера или устройства.

```
@font-face {
  ...
}

html, body {
  height: 100%;
}

body {
  font-size: 10px;
  background: rgb(149, 194, 215);
}
...
```

Обратите внимание, что мы сгруппировали в этом правиле оформления два селектора, разделив их запятой. Когда браузер обнаруживает дополнительные объявления стилей для селектора, он просто добавляет их к существующей информации о стилях этого селектора. В данном случае он сначала обнаружит, что элемент `<body>` должен иметь высоту `100%`, и сохранит эту информацию. При чтении следующего правила оформления для `<body>` он сохранит информацию о свойствах `background` и `font-size` вместе со стилем для свойства `height`.

Теперь мы готовы создать наш первый *флекс-контейнер*. Если элемент является флекс-контейнером, он может контролировать, как выстраиваются его дочерние элементы *(флекс-элементы)*.

Во флекс-контейнере размер и расположение флекс-элементов определяются вдоль *главной* и *поперечной осей* (рис. 4.7).

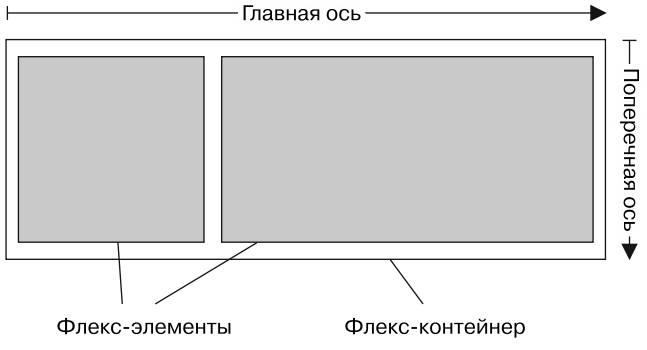


Рис. 4.7. Главная и поперечная оси флекс-контейнера

Сделаем наш элемент `<body>` флекс-контейнером, добавив к его правилу оформления в файле `styles.css` объявление `display: flex`.

```
...
body {
  display: flex;

  font-size: 10px;
  background: rgb(149, 194, 215);
}
...
```

Если вы сейчас сохраните внесенные изменения, браузер отобразит страницу Ottergram довольно удручающего вида (рис. 4.8). Так произошло потому, что главная ось идет слева направо, размещая флекс-элементы (все дочерние элементы `<body>`) в один ряд.

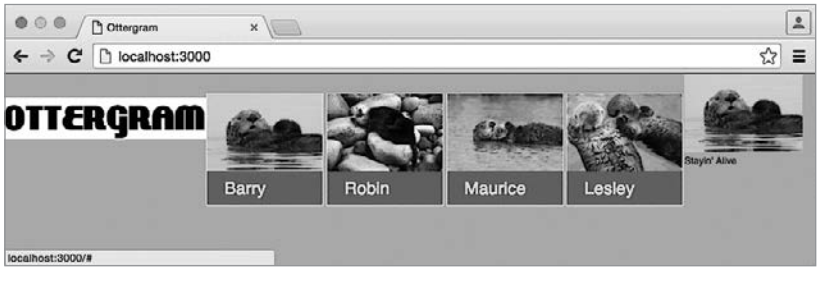


Рис. 4.8. Флекс-элементы размещаются вдоль главной оси

Однако видно, что отдельные элементы сжимаются, дабы уместиться в отведенном для них пространстве, вместо того чтобы переноситься в следующий ряд. Это первая хорошая новость. Вторая заключается в том, что весь макет можно исправить с помощью всего одного стиля (ну почти).

Меняем свойство flex-direction

Чтобы исправить макет, зададим в файле `styles.css` свойство `flex-direction` элемента `<body>` равным `column`:

```
...
body {
  display: flex;
  flex-direction: column;

  font-size: 10px;
  background: rgb(149, 194, 215);
}
...
```

Это действие поменяет местами главную и поперечную оси для флекс-контейнера, как показано на рис. 4.9.

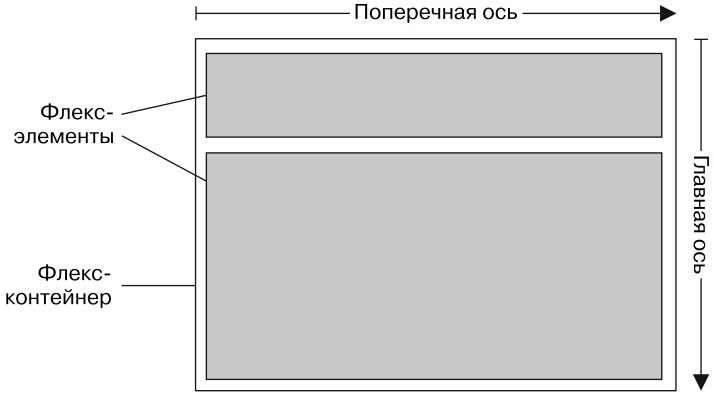


Рис. 4.9. Главная и поперечная оси при flex-direction: column

После изменения значения свойства `flex-direction` на `column` Ottergram вернется к (почти) нормальному виду. В макете есть показанная на рис. 4.10 визуальная погрешность, возникающая, когда ширина окна браузера намного превышает его высоту.

Исправить это можно, добавив оборачивающий элемент и применив новые свойства флекс-блока.

Группировка элементов во флекс-блоке

У элемента `<body>` имеется три флекс-элемента: `<header>`, `.thumbnail-list` и `.detail-image-container`. Что бы ни происходило во время разработки (и использования) Ottergram, макет элемента `<header>` (или его степень сложности) вряд ли сильно изменится. Он всегда будет вверху страницы, отображая текст. Это что касается `header`.

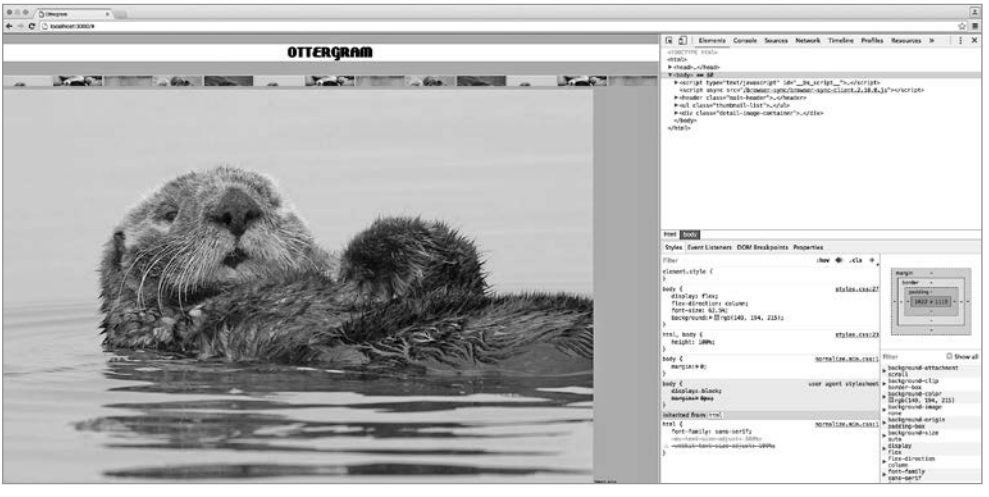


Рис. 4.10. При вытягивании страницы по ширине миниатюры пропадают

С другой стороны, при разработке Ottergram элементы с классами `.thumbnail-list` и `.detail-image-container`, а также их контент могут значительно поменять макет и степень сложности. Кроме того, изменение в одном из этих элементов вполне может повлиять на другой.

Именно поэтому мы собираемся сгруппировать элементы с классами `.thumbnail-list` и `.detail-image-container` в их собственный флекс-контейнер. Для этого обернем их в тег `<main>` с именем класса `.main-content` (рис. 4.11).

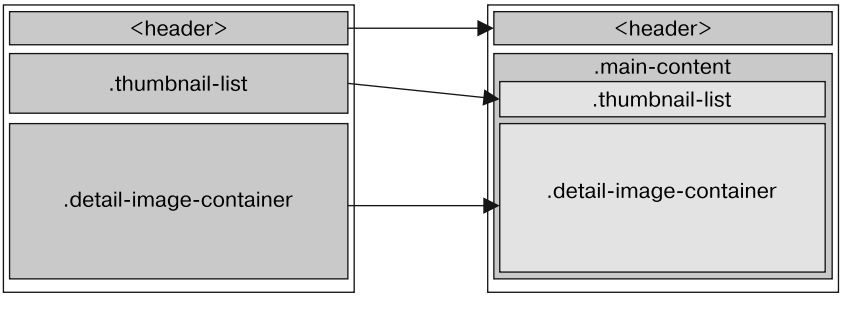


Рис. 4.11. Обертывание элементов с классами .thumbnail-list и .detail-image-container

Выполним в файле `index.html` следующие действия: присвоим элементу `<header>` класс `main-header`, затем обернем элементы с классами `.thumbnail-list` и `.detail-image-container` в элемент `<main>` с классом `main-content`.

```
...
  <body>
    <header>
```

```
    <header class="main-header">
      <h1 class="logo-text">ottergram</h1>
    </header>
    <main class="main-content">
      <ul class="thumbnail-list">
      ...
      </ul>

      <div class="detail-image-container">
        <img class="detail-image" src="img/otter1.jpg" alt="">
        <span class="detail-image-title">Stayin' Alive</span>
      </div>
    </main>
...
```

Теперь элементы с классами `main-header` и `main-content` стали двумя флекс-элементами в `<body>`.

Благодаря обертыванию `.thumbnail-list` и `.detail-image-container` в элемент с классом `main-content` мы можем объявить свойство `height` для `<header>`, оставив все остальное вертикальное пространство элемента `<body>` под элемент с классом `main-content`. Таким образом, пространство внутри элемента с классом `main-content` можно выделить для элементов с классами `.thumbnail-list` и `.detail-image-container`, не затрагивая заголовок.

Сохраните файл `index.html`. Теперь, когда у нас есть разметка для двух флекс-элементов в `<body>`, можно задать их размеры относительно друг друга с помощью свойства `flex`.

Сокращенная запись: свойство flex

Флекс-контейнер распределяет отведенное под него место между находящимися внутри него флекс-элементами. Если для флекс-элементов не задан размер по главной оси, то контейнер распределяет пространство поровну по количеству флекс-элементов (причем каждый получает одинаковый кусочек вдоль главной оси). Такое поведение по умолчанию проиллюстрировано на рис. 4.12.

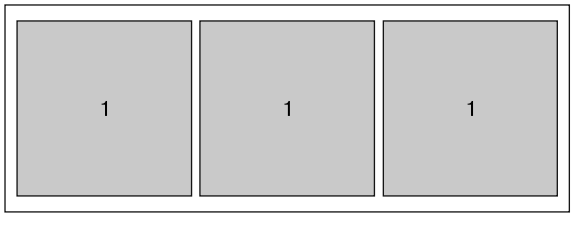


Рис. 4.12. Равномерное распределение места между тремя флекс-элементами

Однако допустим, что один из этих трех флекс-элементов (см. рис. 4.12) более «жадный», чем остальные, и требует две трети общего пространства. В данном случае флекс-контейнер делит пространство вдоль главной оси на четыре части. «Жадный» элемент занимает две из них (половину места), а остальные два — по одной части (рис. 4.13).

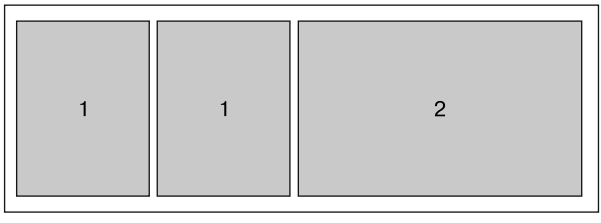


Рис. 4.13. Неравномерное распределение места между тремя флекс-элементами

Нам хотелось бы, чтобы в Ottergram элемент с классом `.main-content` был «жадным» и занял столько места вдоль главной оси, сколько возможно. Элемент с классом `.main-header`, с другой стороны, должен занимать как можно меньше места.

Свойство `flex` дает возможность указывать флекс-элементам, какую часть доступного места они будут занимать. Форма сокращенной записи показана на рис. 4.14.

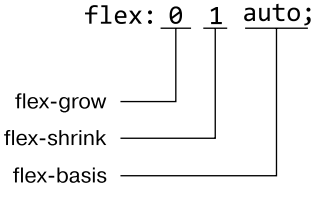


Рис. 4.14. Сокращенная запись: свойство flex и его возможные значения

Мы настоятельно рекомендуем использовать `flex` вместо отдельных свойств, которые оно представляет. Это обезопасит вас от нечаянного пропуска какого-нибудь свойства и получения нежелательных результатов.

Сейчас мы сосредоточим внимание на первом значении, так как оно определяет, насколько может вырасти в размерах флекс-элемент. По умолчанию флекс-элементы вообще не увеличиваются. Такое поведение подходит для нашего элемента с классом `.main-header`, но не для элемента с классом `.main-content`.

Добавьте в файл `styles.css` блок объявления для селектора класса `.main-header`, задав свойство `flex` со значениями по умолчанию: `0 1 auto`.

```
...
a {
  text-decoration: none;
}

.main-header {
  flex: 0 1 auto;
}
```

```
.logo-text {
  background: white;
  ...
```

Значения `01 auto` можно интерпретировать как «я не хочу больше расширяться; я буду сжиматься при необходимости; вычислите, пожалуйста, мой размер за меня». В итоге этот элемент с классом `.main-header` будет занимать ровно столько места, сколько ему требуется, и не больше.

Далее добавьте блок объявления для элемента с классом `.main-content`, задав значение `11 auto` для его свойства `flex`.

```
...
.logo-text {
  ...
}

.main-content {
  flex: 11 auto;
}

.thumbnail-item + .thumbnail-item {
  ...
```

Первое значение в объявлении свойства `flex` элемента с классом `.main-content` соответствует свойству `flex-grow`. Значение `1` сообщает контейнеру: «Я хотел бы увеличиваться настолько, насколько возможно». Поскольку единственный его родственный элемент объявил, что *не будет* увеличиваться, элемент с классом `.main-content` начнет увеличиваться, чтобы занять все пространство, которое не нужно элементу с классом `.main-header`.

Два флекс-элемента `<body>` — с классами `.main-header` и `.main-content` — теперь динамически занимают место в соответствии с их потребностями. Пришло время настроить макет элемента с классом `.main-content`.

Упорядочение, выравнивание флекс-элементов вдоль главной и поперечной осей

Флекс-блок также дает возможность распределять флекс-элементы по флекс-контейнерам. Этот метод позволяет сосредоточиться на слоях. Через минуту мы сделаем из нашего элемента с классом `.main-content` флекс-контейнер.

Вместо того чтобы стилизовать сначала самые маленькие внутренние элементы, а затем дойти до самых больших элементов, при работе над макетом с помощью флекс-блоков полезнее начинать с внешних элементов и продвигаться внутрь.

И вот чем мы займемся дальше: сделаем из элемента с классом `.main-content` флекс-контейнер с вертикальной главной осью и зададим свойства `flex` для флекс-элементов

с классом `.main-content` таким образом, чтобы элемент с классом `.thumbnail-list` занимал определенный по умолчанию объем пространства, а элемент с классом `.detail-image-container` увеличивался до тех пор, пока не займет все оставшееся место. Наконец, мы переместим элемент с классом `.thumbnail-list` под элемент с классом `.detail-image-container` (рис. 4.15).

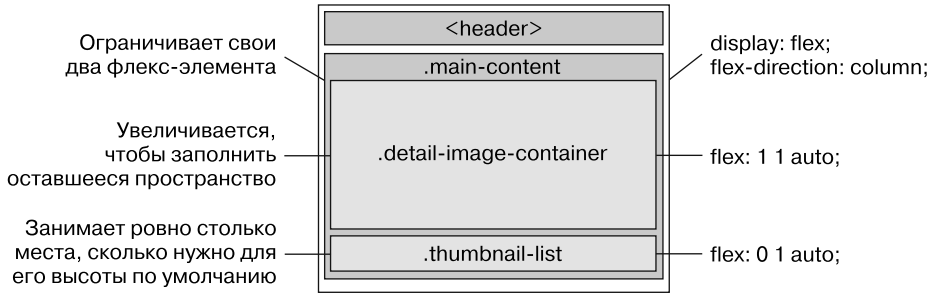


Рис. 4.15. Делаем из элемента с классом .main-content флекс-контейнер

Вносим эти изменения в файл `styles.css`, добавляя код `display: flex` и `flex-direction: column` в блок объявления класса `.main-content`,и значения свойства `flex` — в блок объявления класса `.thumbnail-list`. Пишем новый блок объявления для класса `.detail-image-container`.

```
...
.main-content {
  flex: 11 auto;
  display: flex;
  flex-direction: column;
}

...

.thumbnail-list {
  flex: 01 auto;
  list-style: none;
  padding: 0;

  white-space: nowrap;
  overflow-x: auto;
}

...

.thumbnail-title {
  ...
}

.detail-image-container {
  flex: 11 auto;
```

```
}

.detail-image {
  ...
```

Вы можете удивиться, почему мы не определили высоту блоков с классами `.thumbnail-list` и `.detail-image-container` в процентах, как мы задавали ширину для класса `.detail-image`. Задание высоты блока с классом `.thumbnail-list`, равной, например, `25%`, и высоты блока с классом `.detail-image-wrapper` — `75%` кажется логичным шагом, но это не будет работать так, как нам нужно. Взаимодействие со свойством `width` элемента с классом `.detail-image` приведет к тому, что элемент с классом `.detail-image-container` окажется слишком большим, а элемент с классом `.thumbnail-list` — или слишком большим, или слишком маленьким (в зависимости от размера окна).

Одним словом, использование свойства `flex` для задания размеров флекс-элементов совместно с единственным важным для нас фиксированным размером — шириной элемента с классом `.detail-image` — то, что надо.

Теперь займемся переносом списка миниатюр под увеличенное изображение. По умолчанию флекс-элементы отрисовываются в том порядке, в каком они встречаются в HTML. Это *порядок, который принят по умолчанию* и который среди разработчиков считается основным способом управления порядком отрисовки элементов.

Один из способов перемещения увеличенного изображения — вырезать и вставить разметку для увеличенного изображения так, чтобы она находилась до разметки с классом `.thumbnail-list` (чтобы поменять задаваемый исходным текстом порядок размещения). Однако можно это сделать и с помощью еще одного свойства флекс-блока.

Чтобы поменять порядок с помощью флекс-блоков, добавим в селектор класса `.thumbnail-list` в файле `styles.css` объявление свойства `order`:

```
...
.thumbnail-list {
  flex: 01 auto;
  order: 2;
  list-style: none;
  padding: 0;

  white-space: nowrap;
  overflow-x: auto;
}
...
```

Свойству `order` можно присвоить любое целочисленное значение. Значение по умолчанию равно `0` (указывает браузеру использовать задаваемый исходным текстом порядок). Все остальные значения, включая отрицательные, указывают

браузеру отрисовывать флекс-элемент до или после других флекс-элементов, а вставка в класс `.thumbnail-list` объявления `order: 2` — отрисовывать его *после* всех родственных элементов с более низким значением свойства `order`, например элемента с классом `.detail-image-container`, использующего значение по умолчанию.

Сохраните файл `styles.css` и перейдите в браузер Chrome. Вы увидите, что миниатюры визуализируются вдоль низа страницы (рис. 4.16).

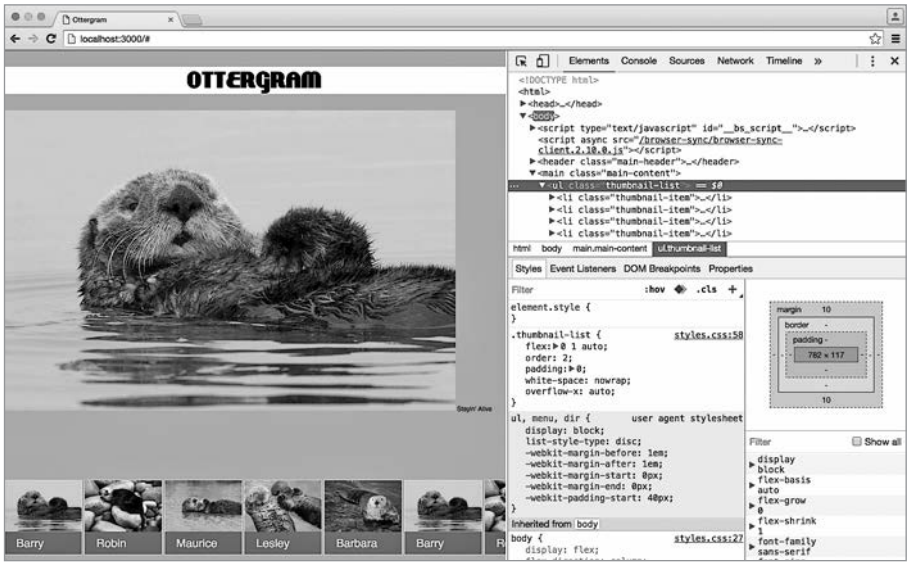


Рис. 4.16. Изменение порядка отрисовки элементов

В дальнейшем мы будем часто применять объявление `display: flex` при работе над макетом Ottergram UI. До сих пор мы имели дело с флекс-контейнерами, содержащими лишь несколько флекс-элементов. Сделаем класс `.thumbnail-list` флекс-контейнером, чтобы продолжить изучение возможностей, которые может предложить нам флекс-блок.

```
...
.thumbnail-list {
  flex: 01 auto;
  order: 2;
  display: flex;
  list-style: none;
  padding: 0;
  white-space: nowrap;
  overflow-x: auto;
}
...
```

Не паникуйте, когда после сохранения изменений обнаружите, что миниатюры стали отображаться странным образом — как на рис. 4.17.

Рис. 4.17. Миниатюры расположены неровно

Чтобы это исправить, заменим объявление свойства `width` в классе `.thumbnail-list` двумя объявлениями, одно из которых для свойства `min-width`, а другое — для свойства `max-width`. Так мы уберем различия в размере, которые привели к столь странному виду макета.

Можно также убрать блок объявления, задающий свойство `margin-top` для элементов с классом `.thumbnail-item + .thumbnail-item`. Он больше нам не нужен.

```
...
~~.thumbnail-item + .thumbnail-item {~~
  ~~margin-top: 10px;~~
~~}~~

.thumbnail-item {
  display: inline-block;
  ~~width: 120px;~~
  min-width: 120px;
  max-width: 120px;
  border: 1px solid rgb(100%, 100%, 100%);
  border: 1px solid rgba(100%, 100%, 100%, 0.8);
}
...
```

Далее мы займемся полями флекс-элементов внутри класса `.thumbnail-list`. Добавьте в файл `styles.css` объявление для свойства `justify-content` в селекторе класса `.thumbnail-list`:

```
...
.thumbnail-list {
  flex: 01 auto;
  order: 2;
  display: flex;
  justify-content: space-between;
  list-style: none;
  padding: 0;

  white-space: nowrap;
```

```
  overflow-x: auto;
}
...
```

Свойство `justify-content` позволяет флекс-контейнеру управлять отрисовкой флекс-элементов вдоль главной оси. Мы использовали в качестве его значения `space-between`, чтобы обеспечить одинаковые поля вокруг каждого отдельного флекс-элемента.

Существует пять различных значений, которые можно указать для свойства `justify-content`. Рисунок 4.18 иллюстрирует действие каждого из них.

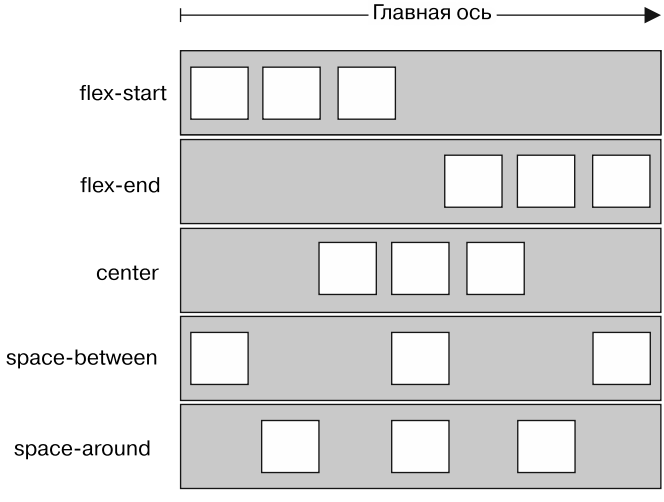


Рис. 4.18. Действие значений свойства justify-content

Мы разобрались с макетом для класса `.thumbnail-list`. Далее займемся классом `.detail-image-container` и его содержимым.

Выравниваем увеличенное изображение по центру

Увеличенное изображение должно находиться на переднем плане в центре, чтобы пользователь наверняка восхитился величием выдры. Оно также должно иметь броское название.

Чтобы выровнять увеличенное изображение по центру, сначала обернем изображение и его название в контейнер, а затем центрируем *обертку* (wrapper) внутри элемента с классом `.detail-image-container`. Эта идея проиллюстрирована на рис. 4.19.

Хотя мы можем выровнять по центру сам элемент с классом `.detail-image` внутри элемента с классом `.detail-image-container`, будет непросто правильно сместить элемент с классом `.detail-image-title`, поскольку элемент с классом `.detail-image` и элемент с классом `.detail-image-container` меняют свой размер динамически.

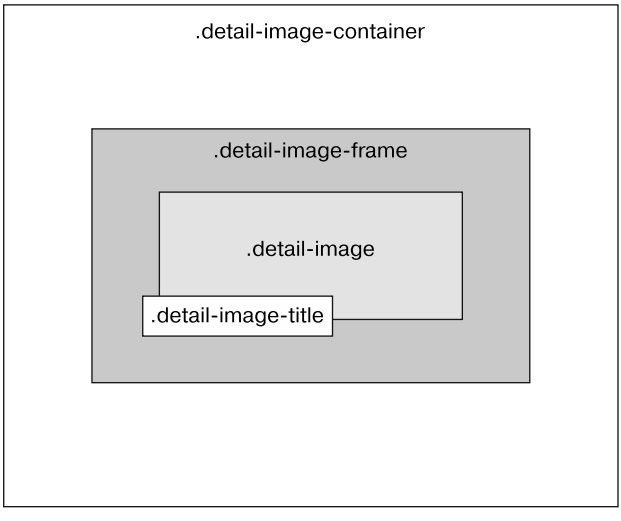


Рис. 4.19. Кадрируем элементы с классами .detail-image и .detail-image-title

Промежуточный элемент-обертка — удобный прием в такой ситуации. Он ограничит размер элемента с классом `.detail-image` и послужит в качестве точки отсчета для позиционирования элемента с классом `.detail-image-title`.

Начнем с добавления в файл `index.html` тега `<div>` с именем класса `.detail-image-frame`:

```
...
      </ul>

      <div class="detail-image-container">
        <div class="detail-image-frame">
          <img class="detail-image" src="img/otter1.jpg" alt="">
          <span class="detail-image-title">Stayin' Alive</span>
        </div>
      </div>
    </main>
  </body>
</html>
```

Сохраните файл `index.html`. Теперь в файле `styles.css` добавьте блок объявления для класса `.detail-image-frame` с единственным объявлением стиля: `text-align: center`. Это один из способов выровнять контент по центру без использования флекс-блока, но обратите внимание, что он работает только в горизонтальном направлении.

```
...
.detail-image-container {
  flex: 11 auto;
}
```

```
.detail-image-frame {
  text-align: center;
}

.detail-image {
  width: 90%;
}
```

Нам нужно центрировать элемент с классом `.detail-image-frame` внутри элемента с классом `.detail-image-container`. Для этого поменяем файл `styles.css` таким образом, чтобы класс `.detail-image-container` стал флекс-контейнером. Обеспечим отрисовку его флекс-элементов по центру главной оси (в данном случае горизонтально — по умолчанию) с помощью задания значения свойства `justify-content: center` и добавим еще одно свойство флекс-блока `align-items: center`, чтобы отрисовать его флекс-элементы по центру поперечной оси (вертикально).

```
...
.detail-image-container {
  flex: 11 auto;
  display: flex;
  justify-content: center;
  align-items: center;
}
...
```

Сохраните изменения — и можете полюбоваться великолепной выдрой, величаво расположившейся в центре элемента с классом `.detail-image-container` (рис. 4.20).

Рис. 4.20. Вид страницы после центрирования элемента с классом .detail-image-frame внутри элемента с классом .detail-image-container

Абсолютное и относительное позиционирование

Иногда нужно поместить элемент точно в определенном месте другого элемента. CSS позволяет сделать это с помощью *абсолютного позиционирования*.

Мы воспользуемся абсолютным позиционированием, чтобы поместить элемент с классом `.detail-image-title` в левом нижнем углу элемента с классом `.detail-image-frame`.

Существует три требования к абсолютно позиционированному элементу. У него должны быть:

- заданное свойство `position: absolute` — для извещения браузера о необходимости извлечь его из *обычного потока*, вместо того чтобы размещать рядом с его родственными элементами;
- координаты, указанные с помощью одного или нескольких свойств `top`, `right`, `bottom` и `left`; в качестве значений можно использовать абсолютные (например, в пикселах) или относительные (например, в процентах) расстояния;
- элемент-предок с объявленным свойством `position` со значением `relative` или `absolute`. Это важно: если ни у одного из предков не объявлено свойство `position`, абсолютно позиционированный элемент будет размещен относительно элемента `<html>` (окна браузера).

Небольшое предостережение: может показаться заманчивым использовать `position: absolute` для всего, что только можно, но лучше знать меру. Макет со сплошным абсолютным позиционированием практически невозможно сопровождать, к тому же он будет выглядеть ужасно при любом размере экрана, кроме того, для которого он создавался.

При задании координат фактически мы указываем расстояние от края элемента до края его контейнера, как показано на рис. 4.21.

На рис. 4.21 приведены два примера абсолютного позиционирования. В первом элемент позиционируется так, что его верхний край находится на расстоянии `50px` от верхнего края его контейнера, а его левый край — на расстоянии `200px` от левого края его контейнера. Второй пример демонстрирует вариант, при котором элемент позиционируется по его нижнему и левому краям.

Чтобы позиционировать элемент с классом `.detail-image-title`, начнем с объявления `position: relative` для класса `.detail-image-frame` в файле `styles.css`. Мы будем позиционировать элемент с классом `.detail-image-title` относительно него.

```
...
.detail-image-frame {
  position: relative;
  text-align: center;
}
...
```

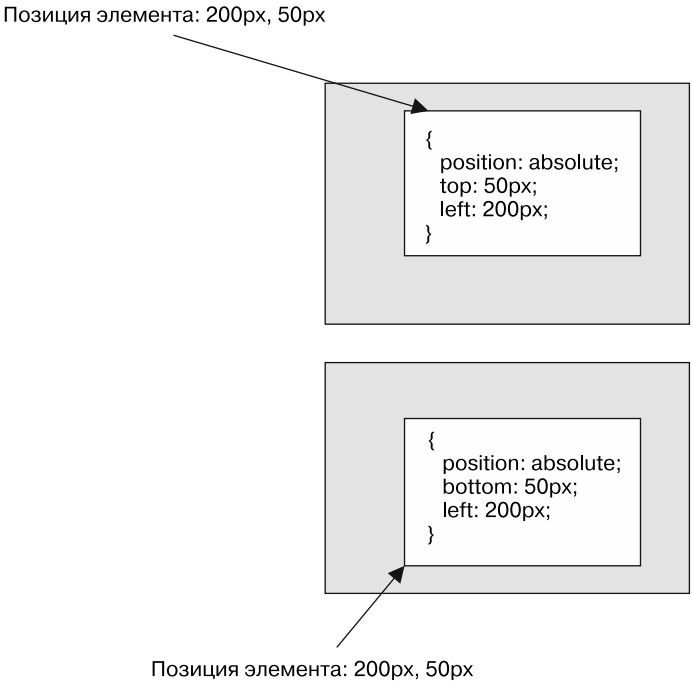


Рис. 4.21. Элементы абсолютно позиционированы по своим краям

Мы использовали `position: relative` для класса `.detail-image-frame`, потому что хотели оставить его в обычном потоке. Не помешает также, чтобы он служил контейнером для абсолютно позиционированного потомка, так что его свойство `position` должно быть явно определено.

В конце файла `styles.css` добавьте блок объявления для селектора класса `.detail-image-title`. Пока что сделаем название белым и зададим размер шрифта в четыре раза больше размера по умолчанию.

```
...
.detail-image {
  width: 90%;
}

.detail-image-title {
  color: white;
  font-size: 40px;
}
```

Пока все хорошо (рис. 4.22). Но очень примитивно.

Ради хотя бы видимости элегантности добавим в класс `.detail-image-title` текстовые эффекты. При позиционировании стилизованных текстовых элементов имейте

в виду, что блок элемента может меняться из-за визуальных характеристик гарнитуры шрифта или других эффектов. В данном примере мы выполним всю стилизацию текста для элемента с классом `.detail-image-title` до задания его позиции. Добавляем в файле `styles.css` свойство `text-shadow` в класс `.detail-image-title`.

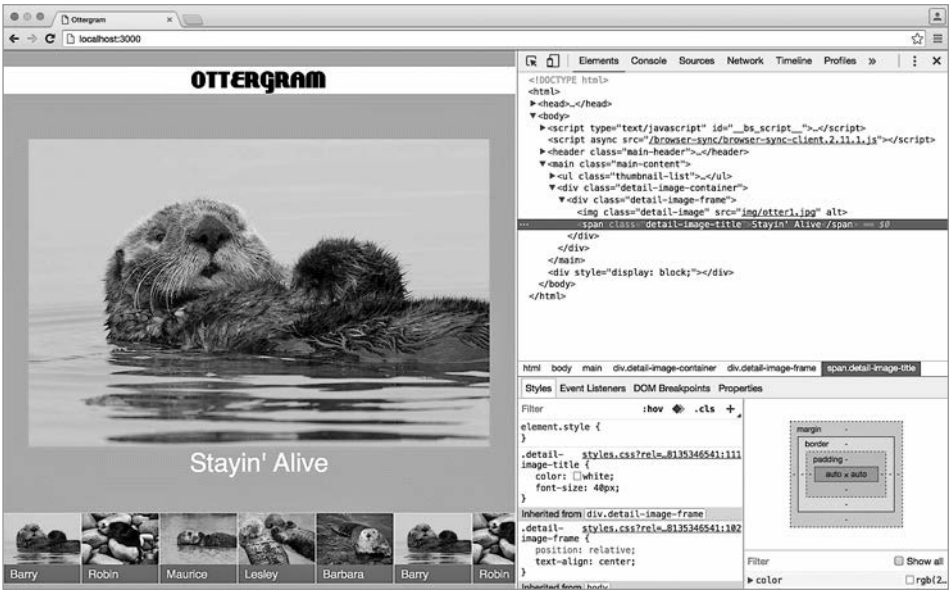


Рис. 4.22. Простейшая стилизация текста для элемента с классом .detail-image-title

```
...
.detail-image-title {
  color: white;
  text-shadow: rgba(0, 0, 0, 0.9) 1px 2px 9px;
  font-size: 40px;
}
```

Как понятно из имени, свойство `text-shadow` добавляет к тексту тень. Оно принимает на входе цвет тени, два расстояния для *смещения* (то есть падает ли тень вверху или внизу текста, слева или справа от текста), а также расстояние для *радиуса размытия* — необязательной части объявления `text-shadow`, делающей тень больше и светлее при увеличении этого значения.

Задаем для тени атрибут цвета `rgba(0, 0, 0, 0.9)`, чтобы сделать ее черной полупрозрачной. Она смещена на `1px` вправо и `2px` ниже текста (отрицательные значения поместили бы ее слева или выше текста). Последнее значение, равное `9px`, — радиус размытия. Рисунок 4.23 демонстрирует нашу обновленную тень.

Попробуйте поменять значения свойства `text-shadow` на панели стилей DevTools, чтобы понять, какой результат они дают (рис. 4.24).

Рис. 4.23. Применяем свойство text-shadow для класса .detail-image-title

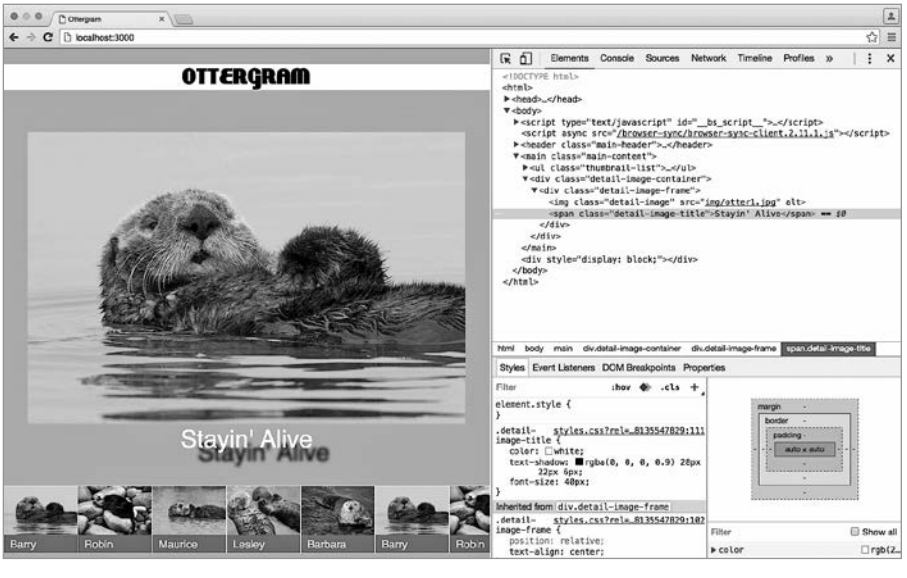


Рис. 4.24. Значительно увеличиваем тень с помощью DevTools

А теперь последний штрих — пользовательский шрифт. Как вы уже делали в главе 3, добавьте в файл `styles.css` объявление правила `@font-face`, чтобы в нашем проекте появился шрифт Airstream. Добавьте в класс `.detail-image-title` объявление `font-family: airstreamregular`, чтобы использовать этот шрифт.

```
@font-face {
    font-family: 'airstreamregular';
    src: url('fonts/Airstream-webfont.eot');
    src: url('fonts/Airstream-webfont.eot?#iefix')
         format('embedded-opentype'),
         url('fonts/Airstream-webfont.woff') format('woff'),
         url('fonts/Airstream-webfont.ttf') format('truetype'),
         url('fonts/Airstream-webfont.svg#airstreamregular') format('svg');
    font-weight: normal;
    font-style: normal;
}

@font-face {
    font-family: 'lakeshore';
    ...
}

...

.detail-image-title {
  font-family: airstreamregular;
  color: white;
  text-shadow: rgba(0, 0, 0, 0.9) 1px 2px 9px;
  font-size: 40px;
}
```

Пока что все стильно (рис. 4.25).

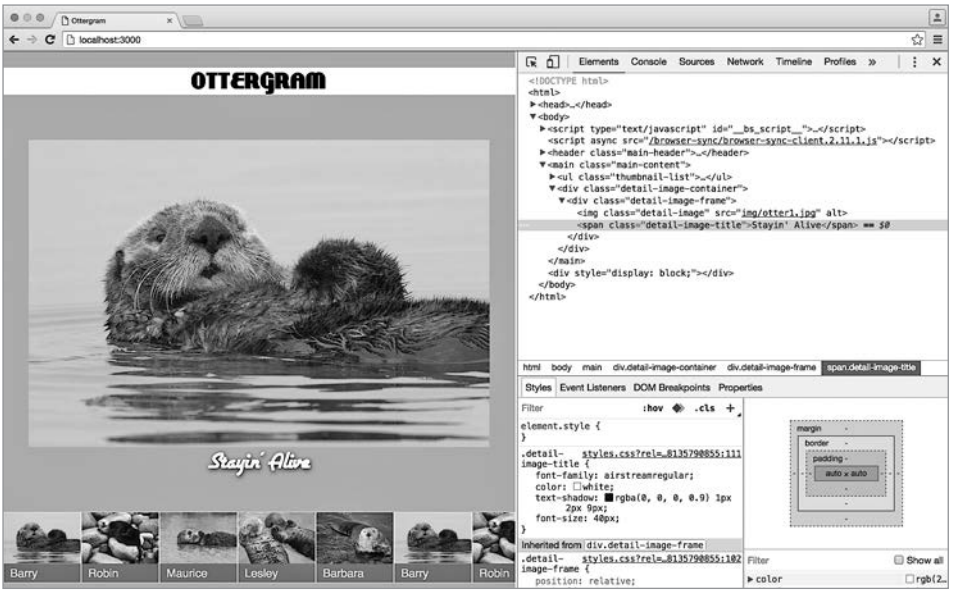


Рис. 4.25. Все стало еще лучше

Теперь, когда мы закончили стилизацию элемента с классом `.detail-image-title`, зададим для него объявление `position: absolute`, чтобы браузер поместил его в точно определенном месте элемента с классом `.detail-image-frame`. Укажем это местоположение следующим образом: `bottom: -16px` и `left: 4px`, чтобы поместить его как раз *под* нижним краем `.detail-image-frame`, сделав чуть выходящим за левый край этого элемента с классом (отрицательные значения в качестве координат вполне допустимы).

```
...
.detail-image-title {
  position: absolute;
  bottom: -16px;
  left: 4px;

  font-family: airstreamregular;
  color: white;
  text-shadow: rgba(0, 0, 0, 0.9) 1px 2px 9px;
  font-size: 40px;
}
```

Сохраните файл `styles.css` — и вы сможете увидеть в браузере, что элемент с классом `.detail-image-title` находится ниже и левее фотографии выдры. Вот теперь можно полюбоваться на действительно шикарный Ottergram (рис. 4.26).

Рис. 4.26. Великолепно!

Благодаря добавлению к нашим стилям флекс-блока получился динамичный и гибкий макет. В следующей главе нам предстоит поработать над тем, чтобы макет приспосабливался к различным размерам окна браузера.

5 Создание адаптивных макетов с помощью медиазапросов

В этой главе рассмотрим метод включения и отключения стилей в зависимости от размера окна браузера и других параметров. Мы обеспечим альтернативный макет для экранов большего размера, использовав минимальное количество кода. Браузер сможет при изменении размера окна переключаться без перезагрузки страницы между различными макетами в режиме реального времени. Рисунок 5.1 демонстрирует исходный и альтернативный макеты.

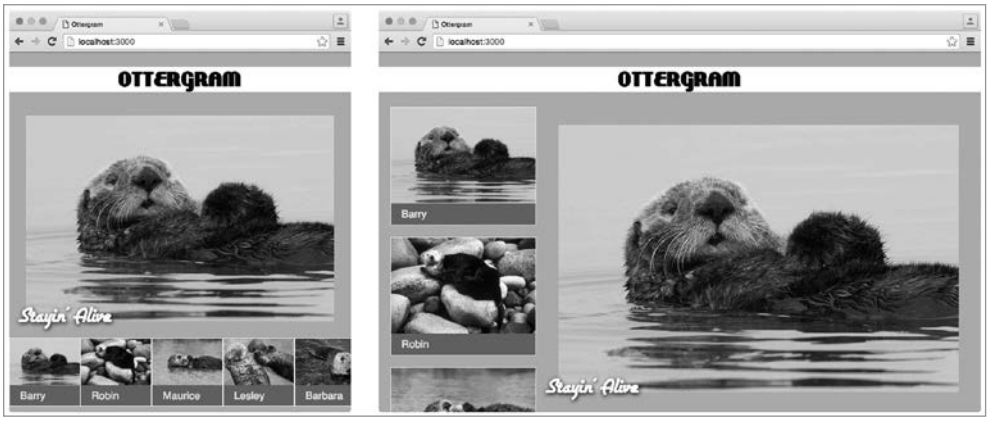


Рис. 5.1. Два макета Ottergram

Официальный термин для подобного поведения в англоязычных источниках — *responsive website*, в отечественной литературе обычно переводится как *адаптивный*

сайт. Многие думают, что это означает «быстрый сайт» или «сайт с динамическими изображениями». Мы предпочитаем называть его *адаптивным*.

Существует несколько способов использования альтернативных стилей в зависимости от текущего состояния браузера. Рекомендуемый подход — описать стили в расчете на минимальный размер экрана, после чего переопределять стили в *медиазапросах*, срабатывающих, когда *область просмотра* — видимая область браузера — больше заданной пороговой величины.

В современных браузерах (таких как Chrome, используемый нами при разработке Ottergram) область просмотра — это область, демонстрируемая окном браузера. Это интуитивно понятно. С мобильными браузерами все сложнее. У них имеется несколько областей просмотра, каждая из которых играет свою роль в визуализации страницы.

Разработчикам клиентской части необходимо сосредотачивать внимание на *области просмотра макета*, иногда называемой *фактической областью просмотра*. Область просмотра макета как бы говорит браузеру: «Сделай вид, что мой размер на самом деле 980 пикселов, после чего отрисовывай страницу».

Пользователей же больше интересует *визуальная область просмотра (visual viewport)* мобильного браузера. Именно там они могут выполнять масштабирование страницы путем сведения пальцев вместе или разведения их в стороны (рис. 5.2).

Если вы прямо сейчас посмотрите на Ottergram на вашем смартфоне, то увидите что-то вроде того, что изображено рис. 5.2. Нет смысла упоминать, что, хотя пользователь мобильного устройства может выполнить масштабирование вручную, крайне нежелательно, чтобы Ottergram вел себя так по умолчанию.

Ранее мы упоминали, что разрабатываем Ottergram исходя из принципа «сначала мобильные устройства». По большей части это справедливо. Наша разметка и стили были созданы так, чтобы не доставлять проблем при использовании на мобильных устройствах (мы применили минимум разметки и стилизовали сначала самые мелкие элементы). Теперь нам достаточно просто предоставить браузеру информацию относительно области просмотра макета, которую ему нужно использовать.

Переопределяем размер экрана

В главе 3 мы добавили в Ottergram файл `normalize.css`. Это обеспечило использование всеми браузерами, в которых будет просматриваться Ottergram, одного и того же набора стилей по умолчанию. А уже поверх этих стилей по умолчанию можно спокойно добавлять свои собственные таблицы стилей CSS, будучи уверенными, что все будет работать единообразно от браузера к браузеру.

Мы сделаем почти то же самое для области просмотра макета. Аналогично тому, как у всех браузеров могут быть различные таблицы стилей агента пользователя,

Рис. 5.2. Визуальная область просмотра по сравнению с областью просмотра макета

у них могут быть и различные области просмотра макета по умолчанию. Однако, в отличие от использования файла `normalize.css`, мы не собираемся делать область просмотра макета одинаковой для всех браузеров. Вместо этого с помощью тега `<meta>` сообщим всем браузерам о необходимости отображать Ottergram в оптимальном для физического экрана устройства размере.

Добавьте в файл `index.html` тег `<meta>`, сообщающий браузеру, что атрибут `width` области просмотра макета равен ширине экрана устройства. Не забудьте установить масштаб 100 %, задав атрибут `initial-scale` равным `1`.

```
<!doctype html>
<html>
  <head>
    <meta charset="utf-8">
```

```
    <meta name="viewport" content="width=device-width, initial-scale=1">
    <link rel="stylesheet"
      href="https://cdnjs.cloudflare.com/ajax/libs/normalize/3.0.3/
      normalize.min.css">
    <link rel="stylesheet" href="stylesheets/styles.css">
    <title>Ottergram</title>
  </head>
...
```

Сохраните изменения. Область просмотра макета задается равной *идеальной области просмотра*. Идеальная область просмотра — наилучший размер области просмотра для данного конкретного устройства, рекомендованный производителем браузера. Он значительно варьируется, поскольку существует очень много различных устройств и немало различных браузеров.

Таблица 5.1 подытоживает различные типы областей просмотра.

Таблица 5.1. Сводка различных областей просмотра

Область просмотра	Описание	Устройство
Область просмотра	Область, равная окну браузера. Служит контейнером для элемента <html>	Настольный компьютер, ноутбук
Область просмотра макета	Используемый для расчета макета страницы виртуальный экран (его размер больше, чем размер фактического экрана устройства)	Мобильное устройство
Визуальная область просмотра	Масштабируемая область, видимая пользователю на экране устройства. Масштабирование не влияет на макет страницы	Мобильное устройство
Идеальная область просмотра	Оптимальные размеры для конкретного браузера на конкретном устройстве	Мобильное устройство

Запустите утилиту browser-sync и убедитесь, что панели DevTools открыты в браузере Chrome. Посмотрите влево от элемента меню Elements (Элементы) и найдите кнопку Toggle Device Mode (Включить/выключить режим устройства), которая выглядит таким образом: . Она показана вместе с окружающими ее элементами на рис. 5.3.

Рис. 5.3. Кнопка Toggle Device Mode (Включить/выключить режим устройства)

Нажмите эту кнопку, чтобы активизировать *режим устройства*. Вы увидите, что теперь представление веб-страницы отображает Ottergram на смоделированном экране смартфона. Имеется меню для выбора между различными размерами экра-

нов, соответствующими популярным устройствам. Можете также щелкнуть на серой полоске под заранее заданными вариантами, чтобы переключиться между маленьким, средним и большим размерами экрана. Есть также меню для быстрого выбора ориентации экрана (книжной или альбомной). На рис. 5.4 показан скриншот режима устройства на момент написания данной книги. У вас все может выглядеть совсем по-другому, так как DevTools постоянно обновляются.

Рис. 5.4. Использование режима устройства для адаптивного тестирования

Можно увидеть, что благодаря нашему новому элементу `<meta>` Ottergram отлично отображается на маленьком экране, таком как у смартфона. Для устройств с большими экранами, таких как планшеты или ноутбуки, более подходящим будет немного другой макет. Далее мы начнем применять различные стили макета, используя сочетание флекс-блоков и медиазапросов.

Еще раз нажмите кнопку , чтобы выключить режим устройства, прежде чем продолжить работу.

Добавление медиазапроса

Медиазапросы позволяют группировать блоки объявлений CSS и задавать условия, при которых их необходимо использовать. Эти условия могут быть, например, такими: «если экран по меньшей мере 640 пикселов в ширину» или «если ширина экрана превышает его высоту и у него высокая плотность пикселов».

Синтаксис для них начинается с `@media`, за которым следуют условия. Далее идут фигурные скобки, в которые заключен весь блок объявления. Посмотрим, как это выглядит на деле.

Начнем написание нашего первого медиазапроса в конце файла `styles.css`. Мы создадим медиазапрос, который сделает стили активными при работе на любом устройстве с областью просмотра не менее 768 пикселов в ширину (это распространенный размер экрана планшетов):

```
...
.detail-image-title {
  ...
}

@media all and (min-width: 768px) {
  /* Здесь будут находиться стили */
}
```

За правилом `@media` следует *тип носителя (media type)* `all`. Типы носителей первоначально предназначались для того, чтобы различать устройства, такие как смарт-телевизоры и переносные устройства. К сожалению, браузеры плохо реализуют эту возможность, так что желательно всегда указывать значение `all`. Единственный случай, когда можно не использовать `all`, — при необходимости задать стили для печати (в этом случае можно без опаски использовать тип носителя `print`).

После типа носителя описываются условия применения стилей. В данном случае мы используем удобное условие `min-width`. Как видите, условия похожи на объявления стилей.

Чтобы достичь показанного на рис. 5.1 эффекта, понадобится изменить свойство `flex-direction` элемента `.main-content`. Это позволит миниатюрам и увеличенному изображению расположиться рядом. Нам не хотелось бы, чтобы миниатюры приводили к прокрутке браузера. Вместо этого они должны оставаться прокручиваемыми независимо от окна браузера. Для этого добавим объявление `overflow: hidden`.

Добавьте эти стили в медиазапрос в конце файла `styles.css`:

```
...
@media all and (min-width: 768px) {
  /* Здесь будут находиться стили */
  .main-content {
```

```
    flex-direction: row;
    overflow: hidden;
  }
}
```

Вы будете шокированы, если сохраните изменения и достаточно широко растянете окно браузера для срабатывания этого медиазапроса. В настоящий момент наша страница выглядит так, как показано на рис. 5.5. Не волнуйтесь. Мы исправим это с помощью всего нескольких строк кода.

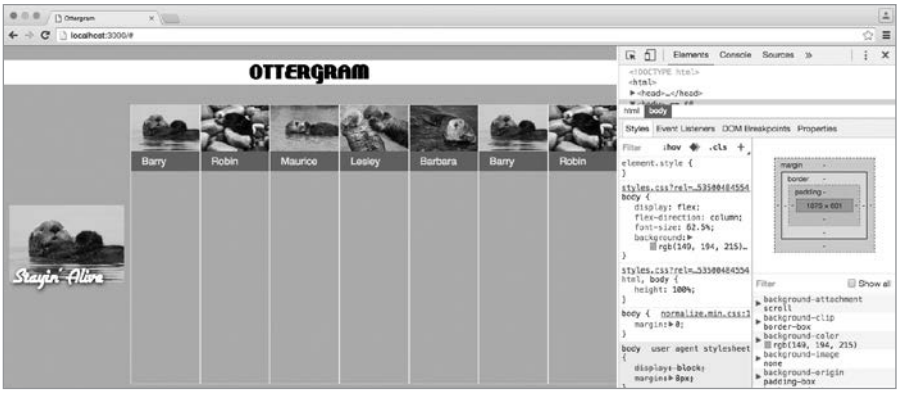


Рис. 5.5. Выдры в полном беспорядке

Миниатюры необходимо отображать в столбец, а не в ряд. Это несложно сделать, поскольку для их размещения мы использовали флекс-блок. Добавим блок объявления внутри тела медиазапроса в файле `styles.css`, задав значение свойства `flex-direction` для элементов с классом `.thumbnail-list`, равным `column`:

```
...
@media all and (min-width: 768px) {
  .main-content {
    flex-direction: row;
    overflow: hidden;
  }

  .thumbnail-list {
    flex-direction: column;
  }
}
```

Сохраните файл `styles.css`. Ситуация значительно улучшилась (рис. 5.6)!

В соответствии с нашим замыслом миниатюры должны располагаться слева. Этого можно добиться, поменяв свойство `order` класса `.thumbnail-list`. Ранее мы сделали его равным `2`, чтобы элемент с таким классом отрисовывался после `.detail-image-container`. Теперь же зададим для него значение `0`, чтобы он следовал порядку,

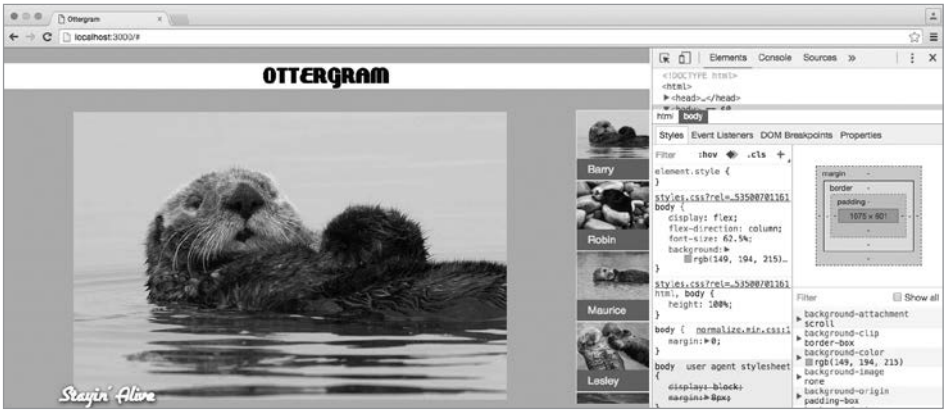


Рис. 5.6. Вид страницы после задания значения свойства flex-direction равным column

соответствующему исходному тексту, и отрисовывался перед элементом с классом `.detail-image-container`:

```
...
@media all and (min-width: 768px) {
  .main-content {
    flex-direction: row;
    overflow: hidden;
  }

  .thumbnail-list {
    flex-direction: column;
    order: 0;
  }
}
```

Сохраните изменения и убедитесь, что миниатюры отрисовываются в левой части страницы.

Мы почти закончили! Добавим еще несколько стилей для элементов с классами `.thumbnail-list` и `.thumbnail-items` в файл `styles.css`, чтобы немного улучшить вид страницы:

```
...
@media all and (min-width: 768px) {
  .main-content {
    flex-direction: row;
    overflow: hidden;
  }

  .thumbnail-list {
    flex-direction: column;
    order: 0;
```

```
    margin-left: 20px;
  }

  .thumbnail-item {
    max-width: 260px;
  }

  .thumbnail-item + .thumbnail-item {
    margin-top: 20px;
  }
}
```

Сохраните файл `styles.css` и переключитесь на браузер. Теперь наш макет отлично выглядит независимо от размера области просмотра (у́же или шире) (рис. 5.7).

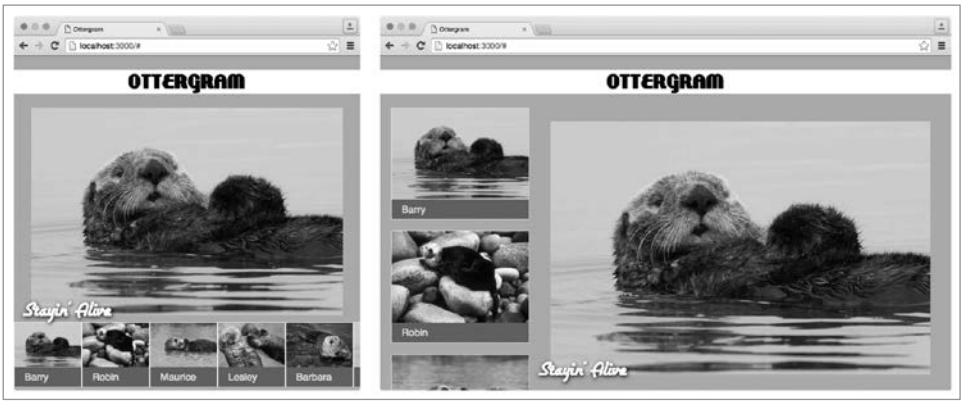


Рис. 5.7. «Адаптивные» выдры

Ottergram неизменно улучшается! Мы создали отлично выглядящий сайт с макетом, который адаптируется к различным размерам экрана. В следующей главе начнем использовать JavaScript для добавления в Ottergram интерактивного слоя.

Бронзовое упражнение: книжная ориентация

Наш текущий медиазапрос меняет макет, основываясь на ширине области просмотра. Можно взглянуть на это иначе. У нашей области просмотра может быть два режима *ориентации*. Один — для областей просмотра, чья высота превышает ширину, а другой — для областей просмотра, чья ширина превышает высоту.

Почитайте в документации MDN о медиазапросах и измените медиазапрос так, чтобы макет менялся в соответствии с ориентацией и не основывался на ширине.

Для самых любознательных: известные решения (и ошибки) при создании макетов с помощью флекс-блоков

Филипп Уолтон — разработчик, поддерживающий два очень важных ресурса по флекс-блокам. Первый — сайт Solved by Flexbox, который находится по адресу philipwalton.github.io/solved-by-flexbox и предлагает демоверсии макетов, реализованных с помощью флекс-блоков. Там также можно найти всю информацию, которая понадобится при самостоятельном их создании. Некоторые из этих макетов очень сложно сделать без флекс-блоков.

Второй ресурс — страница Flexbugs, расположенная по адресу github.com/philipwalton/flexbugs. Флекс-блоки — замечательное средство, но не идеальное. Flexbugs предлагает решения и обходные пути для распространенных проблем, с которыми сталкиваются разработчики при использовании флекс-блоков. Информация предоставляется членами сообщества разработчиков, которым встретились эти ошибки; список поддерживается в актуальном состоянии.

Золотое упражнение: макет Holy Grail

Обязательно сделайте копию своего кода, прежде чем пытаться выполнить это упражнение! Оно потребует значительных изменений в разметке и стилях. Используйте для работы копию, а оригинал оставьте в целости и сохранности для следующей главы.

Используя сайт Solved by Flexbox в качестве справочника, реализуйте макет Holy Grail в Ottergram. Сохраните вторую панель навигации с миниатюрами, но поместите ее с другой стороны области просмотра.

Не забудьте добавить нижний колонтитул внизу страницы. Задействуйте тег `<footer>` и поместите внутрь него тег `<h1>`.

6 Обработка событий с помощью JavaScript

Знаете, что особенно круто в выдрах? Помимо всего прочего, они держатся за лапки во сне, чтобы их не унесло течением. Имейте это в виду при изучении работы с обратными вызовами для событий в JavaScript.

JavaScript — язык программирования, придающий сайтам интерактивность за счет работы с элементами DOM и стилями CSS на странице. Изначально он предназначался для использования непрофессиональными программистами. Его возможности и популярность сильно выросли, и сейчас он применяется при разработке различных приложений. Такие сайты, как Gmail или Netflix, являются написанными на языке JavaScript программами. По существу, текстовый редактор Atom представляет собой написанное на JavaScript традиционное (не веб-) приложение.

Несмотря на широкие возможности и распространенность, у него имеются свои индивидуальные особенности, как и у любого другого языка программирования. При дальнейшей работе над Ottergram и другими приложениями из этой книги вы научитесь обходить проблемные места этого языка и пользоваться его сильными сторонами.

Существует несколько версий языка программирования JavaScript, и в процессе работы над проектами данной книги мы будем использовать три из них. Все они представляют собой редакции стандартной спецификации ECMAScript (или ES) (табл. 6.1).

Таблица 6.1. Используемые в данной книге версии JavaScript

Редакция ECMAScript	Дата выхода	Примечания
Третья	Декабрь 1999 года	Наиболее широко поддерживаемая версия, охватывает большинство возможностей языка, которые мы будем использовать, таких как переменные, типы и функции

Продолжение ⇩

Таблица 6.1 *(продолжение)*

Редакция ECMAScript	Дата выхода	Примечания
Пятая	Декабрь 2009 года	Обратно совместимая версия с дополнительными расширениями, такими как строгий режим (strict mode), предотвращающий использование наиболее подверженных ошибкам частей языка
Шестая	Июнь 2015 года	Включает новый синтаксис и возможности языка; на момент написания данной книги большинство браузеров еще не поддерживают ES6, но код ES6 можно транслировать в ES5, сделав его пригодным для использования в большинстве браузеров

В данной главе мы используем JavaScript, чтобы сделать Ottergram интерактивным: увеличенное изображение и название для него будут меняться при щелчке или постукивании[1] по одной из миниатюр.

Чтобы это реализовать, напишем *функцию* JavaScript (совокупность шагов, которым должен следовать браузер), читающую URL изображения и отображающую его в предназначенной для увеличенного изображения области. Далее добьемся выполнения этой функции при нажатии на миниатюре. Мы также создадим отдельную функцию для скрытия области, которая предназначена для увеличенного изображения (выполняется при нажатии клавиши Esc).

В конце этой главы Ottergram сможет отображать любую выдру в области увеличенного изображения (рис. 6.1).

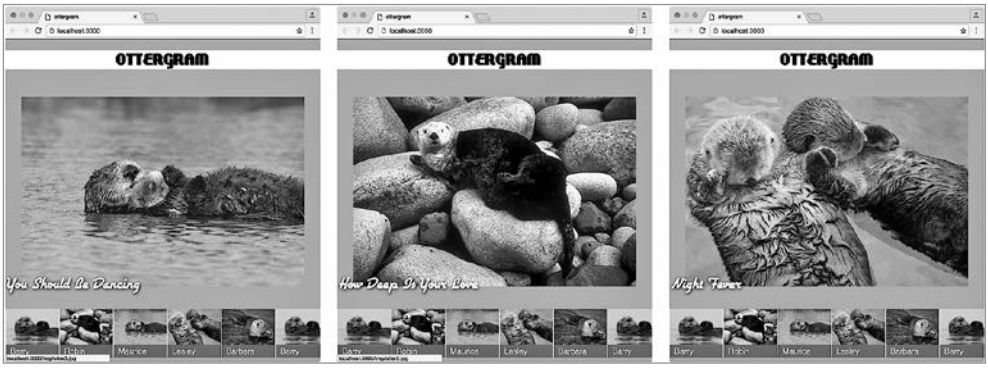


Рис. 6.1. Щелчок на миниатюре сменяет на экране увеличенное изображение и название

При создании этих функций мы будем взаимодействовать со страницей с помощью набора предопределенных интерфейсов, встроенных в браузер. Существует

[1] На устройствах с сенсорным экраном. — *Примеч. пер.*

большое их количество, но мы рассмотрим только те, которые необходимы для решения нашей задачи. Больше подробностей вы можете узнать в MDN на сайте developer.mozilla.org/en-US/docs/Web/API/Element.

Готовим теги-якоря к работе

Прежде чем начать добавлять интерактивные возможности с помощью языка JavaScript, необходимо внести несколько изменений в разметку. Наши миниатюры изображений обернуты в теги-якоря, но эти теги на самом деле не связаны ни с какими ресурсами. Вместо этого они используют значение `#` для атрибута `href`, сообщающее браузеру, что не нужно покидать страницу. Чтобы щелчок на миниатюре выполнял какие-нибудь полезные действия, необходимо это исправить.

Во-первых, в файле `index.html` уберите все элементы с классом `.thumbnail-item`, кроме пяти. Нам больше не нужны дубликаты, поскольку с макетом все в порядке.

Далее поменяйте свойства `href` тега-якоря так, чтобы больше не использовать фиктивное значение `#`. Взамен введите те же значения, что указаны для атрибутов `src` каждого из тегов `<img>`.

Редактор Atom может помочь при выполнении этих изменений, взяв самую скучную часть работы с HTML на себя. Как и любой другой текстовый редактор, он умеет искать и заменять текст. Выберите Find ▸ Find in Buffer (Найти ▸ Найти в буфере) или воспользуйтесь сочетанием горячих клавиш Command+F (Ctrl+F). Это действие откроет строку поиска на панели Buffer внизу окна редактора (рис. 6.2).

Рис. 6.2. Использование возможности «найти и заменить» редактора Atom

Введите символ `#` в текстовое поле Find in current buffer (Найти в текущем буфере) и `img/otter.jpg` — в текстовое поле Replace in current buffer (Заменить в текущем буфере). Затем нажмите кнопку Replace All (Заменить все) справа внизу.

Все теги `<a href="#">` будут заменены на `<a href="img/otter.jpg">`. Теперь осталось только вручную добавить соответствующие номера в каждый из них (`img/otter1.jpg`, `img/otter2.jpg` и т. д.).

Нажмите клавишу Esc, чтобы закрыть панель Find in Buffer (Найти в буфере). Файл `index.html` должен выглядеть следующим образом:

```
...
      <ul class="thumbnail-list">
        <li class="thumbnail-item">
          <a href="#">
          <a href="img/otter1.jpg">
            <img class="thumbnail-image" src="img/otter1.jpg" alt="">
            <span class="thumbnail-title">Barry</span>
          </a>
        </li>
        <li class="thumbnail-item">
          <a href="#">
          <a href="img/otter2.jpg">
            <img class="thumbnail-image" src="img/otter2.jpg" alt="">
            <span class="thumbnail-title">Robin</span>
          </a>
        </li>
        <li class="thumbnail-item">
          <a href="#">
          <a href="img/otter3.jpg">
            <img class="thumbnail-image" src="img/otter3.jpg" alt="">
            <span class="thumbnail-title">Maurice</span>
          </a>
        </li>
        <li class="thumbnail-item">
          <a href="#">
          <a href="img/otter4.jpg">
            <img class="thumbnail-image" src="img/otter4.jpg" alt="">
            <span class="thumbnail-title">Lesley</span>
          </a>
        </li>
        <li class="thumbnail-item">
          <a href="#">
          <a href="img/otter5.jpg">
            <img class="thumbnail-image" src="img/otter5.jpg" alt="">
            <span class="thumbnail-title">Barbara</span>
          </a>
        </li>
      </ul>
...
```

Далее нам необходимо добавить дополнительные свойства в элементы-якоря, чтобы можно было обращаться к ним из JavaScript. При стилизации с помощью CSS мы использовали селекторы имен классов для ссылок на элементы страницы. Для JavaScript мы будем использовать *атрибуты данных*.

Атрибуты данных аналогичны остальным атрибутам HTML, с которыми мы уже работали, за исключением того, что, в отличие от таких атрибутов, как `src` или `href`, атрибуты данных никакого особого смысла для браузера не несут. Единственное требование — имя атрибута должно начинаться с `data-`. Применение пользовательских атрибутов данных дает нам возможность указывать, с какими элементами HTML на странице взаимодействует наш JavaScript-код.

Формально в JavaScript можно обращаться к элементам на странице через их имена классов. Аналогично можно задействовать атрибуты данных в селекторах для стилизации. Но делать это нежелательно. Код будет более легким в сопровождении, если JavaScript и CSS не станут использовать одни и те же атрибуты.

Исправьте теги-якоря в файле `index.html`, вставив атрибуты данных. Обратите внимание, что разрывы строк в приведенном ниже коде были добавлены, чтобы все уместилось на странице. Вы можете добавлять их или не добавлять — как вам удобнее. Для браузера они не имеют значения.

```
...
        <li class="thumbnail-item">
          <a href="img/otter1.jpg" data-image-role="trigger"
                                   data-image-title="Stayin' Alive"
                                   data-image-url="img/otter1.jpg">
            <img class="thumbnail-image" src="img/otter1.jpg" alt="">
            <span class="thumbnail-title">Barry</span>
          </a>
        </li>
          <li class="thumbnail-item">
            <a href="img/otter2.jpg" data-image-role="trigger"
                                     data-image-title="How Deep Is Your Love"
                                     data-image-url="img/otter2.jpg">
            <img class="thumbnail-image" src="img/otter2.jpg" alt="">
            <span class="thumbnail-title">Robin</span>
          </a>
        </li>
        <li class="thumbnail-item">
          <a href="img/otter3.jpg" data-image-role="trigger"
                                   data-image-title="You Should Be Dancing"
                                   data-image-url="img/otter3.jpg">
            <img class="thumbnail-image" src="img/otter3.jpg" alt="">
            <span class="thumbnail-title">Maurice</span>
          </a>
        </li>
        <li class="thumbnail-item">
          <a href="img/otter4.jpg" data-image-role="trigger"
                                   data-image-title="Night Fever"
                                   data-image-url="img/otter4.jpg">
            <img class="thumbnail-image" src="img/otter4.jpg" alt="">
            <span class="thumbnail-title">Lesley</span>
          </a>
        </li>
        <li class="thumbnail-item">
          <a href="img/otter5.jpg" data-image-role="trigger"
                                   data-image-title="To Love Somebody"
                                   data-image-url="img/otter5.jpg">
            <img class="thumbnail-image" src="img/otter5.jpg" alt="">
            <span class="thumbnail-title">Barbara</span>
          </a>
        </li>
...
```

Добавьте также атрибуты данных для увеличенного изображения:

```
...
      <div class="detail-image-container">
        <div class="detail-image-wrapper">
        <img class="detail-image" data-image-role="target"
             src="img/otter1.jpg" alt="">
        <span class="detail-image-title" data-image-role="title">Stayin'
              Alive</span>
        </div>
      </div>
...
```

Наш код JavaScript может ссылаться на эти атрибуты данных, чтобы обращаться к соответствующим элементам на странице, поскольку браузер позволяет нам использовать JavaScript для выполнения запросов относительно содержимого веб-страницы. Например, можно запросить все элементы, соответствующие селектору, такому как `data-image-role="trigger"`. Если запрос обнаруживает соответствия, то возвращает *ссылки* на соответствующие элементы.

При наличии ссылки на элемент с ним можно делать все что угодно. Можно читать или менять значения его атрибутов, изменять текст внутри него и даже обращаться к окружающим его элементам. При выполнении изменений элемента через ссылку браузер обновляет страницу сразу же.

В данной главе мы напишем код JavaScript для получения ссылок на элементы якоря и увеличенного изображения, чтения значений из атрибутов данных якоря с последующим изменением значения атрибута `src` увеличенного изображения. Именно таким образом мы сделаем Ottergram интерактивным.

Вероятно, вы обратили внимание, что и у тега-якоря, и у тега `<img>` увеличенного изображения есть атрибуты `data-image-role`, но их значения различны.

Использование одинаковых имен атрибутов данных для тега-якоря и тега `<img>` необязательно, но это хорошая практика. Они напоминают вам, разработчику, что эти элементы будут частью единого поведения JavaScript.

Прежде чем начать работу над нашим JavaScript-кодом, нужно внести последнее изменение: необходимо указать HTML-коду выполнить JavaScript. Сделаем это путем добавления в файл `index.html` тега `<script>`. Этот тег `<script>` будет ссылаться на файл `scripts/main.js`, который мы вскоре создадим.

```
...
        </div>
      </div>
    </main>
    <script src="scripts/main.js" charset="utf-8"></script>
  </body>
</html>
```

Когда браузер встретит тег `<script>`, он сразу начнет выполнять код, на который этот тег ссылается. JavaScript не может получить доступ к элементу HTML до того, как браузер его визуализирует, так что размещение тега `<script>` внизу тега `<body>` гарантирует, что наш JavaScript не будет запущен до окончания синтаксического разбора всей разметки.

Теперь наш HTML готов для связи с кодом JavaScript, который мы собираемся написать. Не забудьте сохранить файл `index.html`, прежде чем продолжить.

Наш первый сценарий

Пришло время создать каталог `script` и файл `main.js`. Вспомним, что у нас есть возможность создавать каталоги из редактора Atom. Щелкните кнопкой мыши с нажатой клавишей Control (правой кнопкой мыши — в Windows) на ottergram на панели слева, а потом — на New folder (Новый каталог) во всплывающем окне. Наберите название `scripts` в появившемся приглашении ввести данные.

Далее щелкните кнопкой мыши с нажатой клавишей Control (правой кнопкой мыши — в Windows) на пункте scripts на панели слева и выберите New File (Новый файл). Приглашение на ввод будет заранее заполнено текстом `scripts/`. После него введите `main.js` и нажмите Enter.

Убедитесь, что структура каталогов выглядит так, как показано на рис. 6.3.

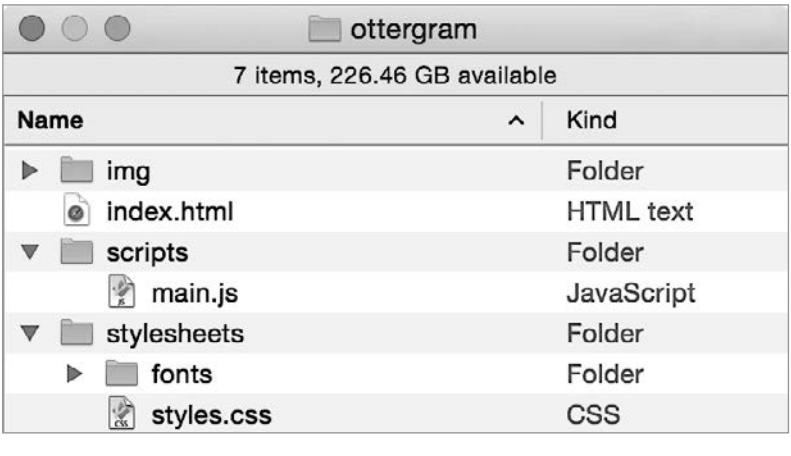


Рис. 6.3. Структура каталогов Ottergram

Название `main.js` особого значения для браузера не несет, но это распространенное условное обозначение, используемое многими разработчиками клиентской части.

Еще один момент, прежде чем углубиться в JavaScript-код. Необходимо запустить утилиту browser-sync, а для этого понадобится немного изменить команду, которую мы использовали ранее:

```
browser-sync start --server --browser "Google Chrome"
                   --files "*.html, stylesheets/*.css, scripts/*.js"
```

Мы добавили путь `scripts/*.js` в список файлов, чтобы browser-sync отслеживал изменения в коде JavaScript наряду с изменениями в коде HTML и CSS.

Обзор JavaScript для Ottergram

Прежде чем мы начнем писать код, не помешает все спланировать. Вот написанная понятным языком версия того, что нам нужно будет сделать с Ottergram.

1. Получить все миниатюры.
2. Организовать прослушивание на предмет события щелчка на каждой из них.
3. При щелчке обновлять увеличенное изображение и название информацией из соответствующей миниатюры.

   Шаг 3 можно разбить на три части:

   1) получить URL изображения из атрибута данных миниатюры;
   2) получить текст названия из атрибута данных миниатюры;
   3) задать изображение и название для увеличенного изображения.

Вот этот же план в виде схемы (рис. 6.4).

В этой главе мы создадим соответствующий код, начав с последнего шага. Это так называемый восходящий подход, и он отлично работает при написании кода JavaScript.

Объявляем строковые переменные

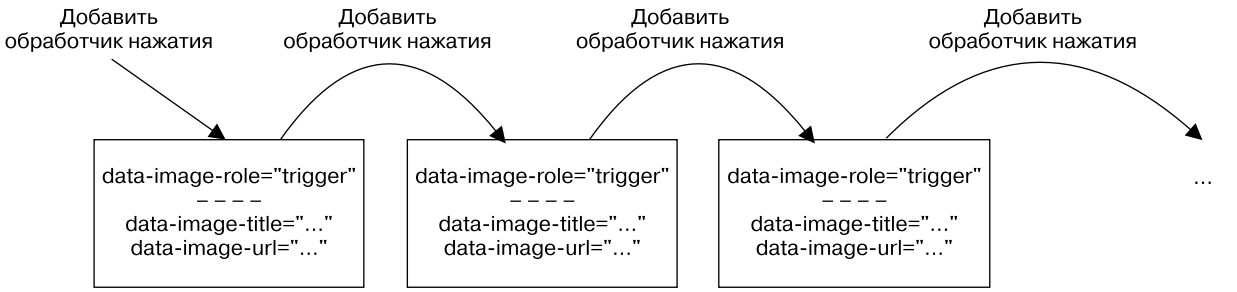
Наша первая относящаяся к JavaScript задача — создать строковые переменные для каждого из добавленных в разметку атрибутов данных (если эти термины вам непонятны, не волнуйтесь, мы все объясним через минуту).

Начните с добавления вверху файла `main.js` переменной `DETAIL_IMAGE_SELECTOR` и присвоения ей строки `'[data-image-role="target"]'`.

```
var DETAIL_IMAGE_SELECTOR = '[data-image-role="target"]';
```

Это совсем небольшой фрагмент кода, но он заслуживает пристального внимания. Начнем с середины — с символа `=`. Это *оператор присваивания*. В отличие от математического знака символ `=` в языке программирования JavaScript не означает, что две вещи равны. Он означает «возьмите значение справа от меня и дайте ему имя, находящееся слева».

Получить все миниатюры, добавить обработчик нажатия.



При нажатии на миниатюру...

... обновить увеличенное изображение и название.

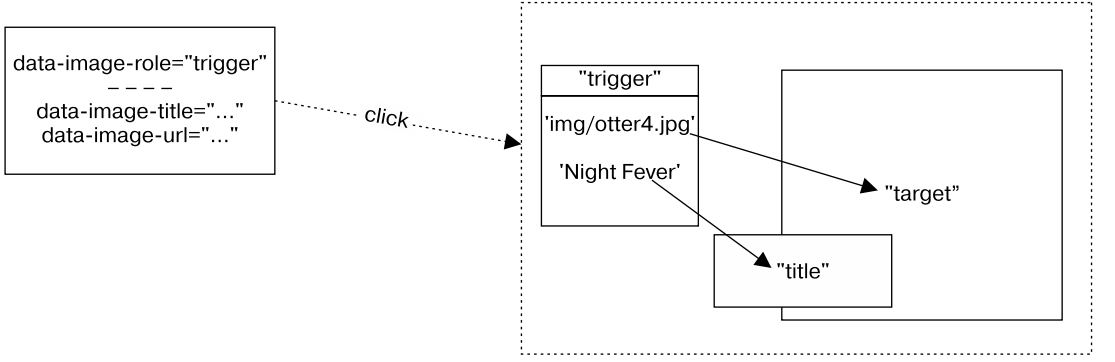


Рис. 6.4. Фронт работ для Ottergram

Справа от данного конкретного присваивания находится *строка* текста: `'[data-image-role="target"]'`. Строка — это просто последовательность символов, представляющих собой текст, она ограничена одиночными кавычками. Текст внутри одиночных кавычек — селектор атрибута для нашего увеличенного изображения. Это значит, что мы будем использовать эту строку для доступа к данному элементу.

Слева от присваивания находится *объявление переменной*. Может быть удобно представлять себе переменную как ярлык, использующийся для ссылки на какое-то значение, которое может быть числовым значением, строкой (как в нашем случае) или другим типом значения. С помощью ключевого слова `var` мы создаем переменную под названием `DETAIL_IMAGE_SELECTOR`.

Далее объявляем в файле `main.js` переменные для селектора названия увеличенного изображения и селектора якоря миниатюры, присваиваем им строковые значения:

```
var DETAIL_IMAGE_SELECTOR = '[data-image-role="target"]';
var DETAIL_TITLE_SELECTOR = '[data-image-role="title"]';
var THUMBNAIL_LINK_SELECTOR = '[data-image-role="trigger"]';
```

Как можно догадаться по названию *«переменная»*, ее значение можно переназначать, то есть оно может меняться. Написание имен переменных заглавными буквами — условное обозначение, которое разработчики иногда используют, когда значения не должны меняться. В других языках программирования имеются служащие для этой цели *константы*. JavaScript находится в процессе изменения: в версии ES5 нет констант, а в ES6 они есть, но, как мы упоминали ранее, ES6 еще не полностью поддерживается. Пока константы не станут полностью поддерживаемыми, можно следовать вышеупомянутому условному обозначению, чтобы помечать значения, которые не должны меняться.

Строки могут ограничиваться одинарными или двойными кавычками. Можно выбрать любые из них, но в данной книге используются одинарные, и мы предлагаем следовать этому соглашению, по крайней мере для проектов из этой книги.

Если вы хотите использовать двойные кавычки, вам придется *экранировать* все двойные кавычки, встречающиеся внутри строки, чтобы браузер не принял их за часть кода. Чтобы экранировать символ, необходимо поставить перед ним обратную косую черту:

```
var DETAIL_IMAGE_SELECTOR = "[data-image-role=\"target\"]";
```

Использование одинарных кавычек не гарантирует, что вам не придется экранировать символы. Если ограниченная одинарными кавычками строка содержит внутри одинарные кавычки — апострофы, вам придется их экранировать.

Сохраните файл `main.js`. Теперь, когда у нас есть эти переменные, можно поэкспериментировать с ними в DevTools браузера Chrome.

Работаем в консоли

Одна из самых полезных частей DevTools — консоль, позволяющая вводить код JavaScript и сразу же оценивать результат его выполнения. Это особенно удобно для итеративно разрабатываемого кода JavaScript, вносящего изменения на странице.

В DevTools перейдите на вкладку Console (Консоль) справа от вкладки Elements (Элементы) (рис. 6.5).

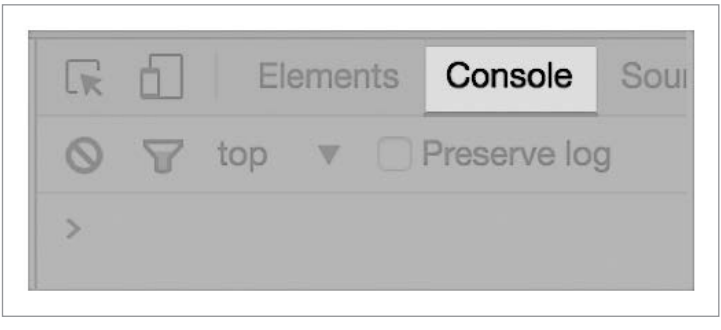


Рис. 6.5. Выбор вкладки консоли

В консоли есть приглашение командной строки, куда можно вводить код. Щелкните справа от символа > , означающего, что консоль готова к работе (рис. 6.6).

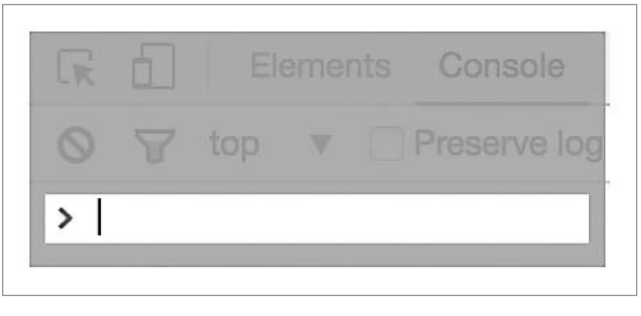


Рис. 6.6. Готовая к вводу консоль

Введите в консоль следующее математическое выражение:

```
137 + 349
```

Нажмите клавишу Enter. Консоль выведет результат (рис. 6.7).

Основная задача консоли — в максимально упрощенном виде сообщить вам результат вычисления введенного вами кода.

Рис. 6.7. Вычисление математического выражения

Как и вообще в жизни, порядок важен. Если необходимо, чтобы определенные элементы вычислялись как группа, можно заключить их в скобки. Это намного проще, чем запоминать порядок, в котором JavaScript сделает это без скобок. Введите в консоль следующее выражение:

```
3 * ( (2 * 4) + (3 + 5) )
```

Нажмите клавишу Enter — и консоль произведет математические вычисления в правильном порядке (рис. 6.8). Кстати, хотя ради удобства чтения мы добавили пробелы между числами и операторами, делать это не обязательно (для консоли это значения не имеет).

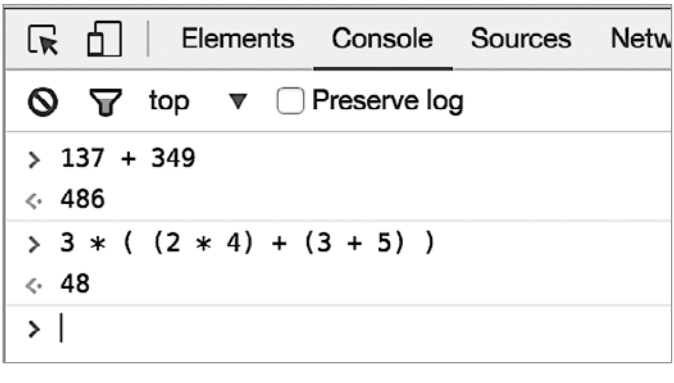


Рис. 6.8. Вычисление более сложного математического выражения

Теперь вернемся к переменным. Очистить содержимое консоли можно щелчком на значке ⊘ в левом верхнем углу панели консоли или нажатием сочетания горячих клавиш Command+K (Ctrl+K).

Начните набирать `DETAIL_IMAGE_SELECTOR`. Набрав первые буквы, вы обнаружите, что консоль уже знает, какие переменные мы создали, и предоставляет список возможных окончаний названия (рис. 6.9).

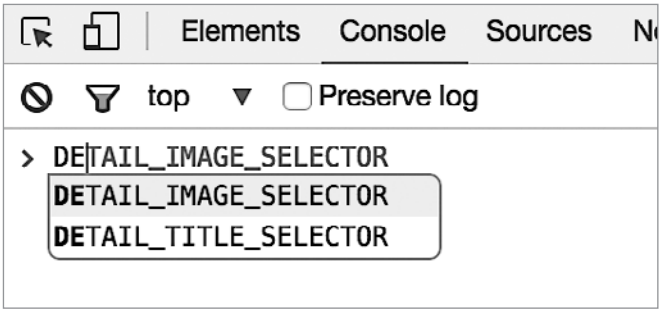


Рис. 6.9. Меню автодополнения консоли

Нажмите клавишу Tab — и консоль автоматически дополнит имя переменной за вас. После нажатия клавиши Enter консоль сообщит, что значение `DETAIL_IMAGE_SELECTOR` представляет собой строку `"[data-image-role="target"]"` (рис. 6.10).

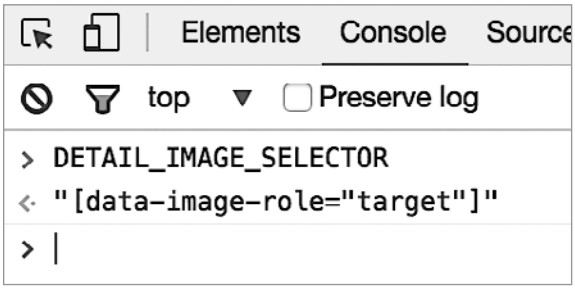


Рис. 6.10. Консоль выводит значение переменной

Консоль всегда выводит строки с двойными кавычками, хотя в действительности в файле `main.js` мы использовали одинарные.

Строки — один из пяти *простых (primitive)* типов значений в JavaScript (еще два — числа и булевы значения). Они называются простыми, поскольку представляют простые значения, в отличие от более сложных значений JavaScript, о которых вы узнаете далее.

Обращение к элементам DOM

Мы уже видели, что консоль дает нам возможность обращаться к созданным нами переменным. Ранее мы говорили, что эти переменные можно использовать для доступа к элементам на странице. Можете опробовать это сейчас. Введите в консоли следующее:

```
document.querySelector(DETAIL_IMAGE_SELECTOR);
```

Нажмите клавишу Enter. При этом будет отображен код HTML для увеличенного изображения. Проведите указателем мыши над HTML в консоли. Вы увидите, что увеличенное изображение подсвечивается на странице, как если бы вы щелкнули на HTML-коде на панели элементов (рис. 6.11).

Рис. 6.11. HTML в консоли соответствует элементу на странице

В написанной вами в консоли строке кода `document` — встроенная в браузер переменная, предоставляющая доступ к веб-странице. Ее значение не относится к какому-либо из простых типов. Это сложное значение, тип которого — *объект*.

Объект `document` соответствует всей странице целиком. Он предоставляет доступ к множеству методов для получения ссылок на элементы страницы. Тип методов — функции (они являются функциями с явным образом указанным владельцем, но вам не нужно беспокоиться об этих нюансах прямо сейчас), то есть список шагов, которым должен следовать браузер. Во введенной в консоли строке мы использовали метод `querySelector`. Оператор-точка в `document.querySelector` — способ обращения к методам объекта.

Мы попросили объект `document` использовать свой метод `querySelector` для поиска какого-либо из соответствующих строке `'[data-image-role="target"]'` элементов. Метод `querySelector` вернул ссылку на найденный элемент — увеличенное изображение (рис. 6.12).

Рис. 6.12. Обращение к странице с помощью объекта document и метода document.querySelector

А теперь немного терминологии. На самом деле мы не «попросили» страницу найти подходящие элементы. Мы *вызвали* метод `querySelector` объекта `document` и *передали* ему строку. Метод *вернул* ссылку на элемент увеличенного изображения.

Когда мы вызвали метод, мы заставили его решать ту задачу, для выполнения которой он предназначен. Нам нередко понадобится передавать методу информацию, необходимую ему для работы. Эту информацию мы поместили в скобках после названия метода. Затем (помимо выполнения своих задач) метод может вернуть значение, которое мы сможем использовать.

Как вы помните, переменной `DETAIL_IMAGE_SELECTOR` было присвоено значение `'[data-image-role="target"]'`, следовательно, именно оно и было передано методу `querySelector`.

Незаметно для нас `querySelector` использует эту строку для поиска какого-либо элемента, соответствующего данному селектору. Объект `document` не выполняет

фактический поиск по странице, он ищет в *объектной модели документа* (Document Object Model, DOM). DOM — внутреннее представление браузера для HTML-документа. Он строит это представление по мере чтения и интерпретации HTML.

В JavaScript можно взаимодействовать с DOM посредством объекта `document` и его методов, таких как `querySelector`. Для каждого тега HTML существует соответствующий элемент в DOM. С любым элементом можно взаимодействовать с помощью JavaScript (обычно, когда мы говорим об элементе, мы имеем в виду элемент DOM).

В консоли снова вызовите `document.querySelector`, передав ему переменную `DETAIL_IMAGE_SELECTOR`, чтобы получить ссылку на элемент для увеличенного изображения. Но на этот раз присвойте полученную ссылку новой переменной `detailImage`:

```
var detailImage = document.querySelector(DETAIL_IMAGE_SELECTOR);
```

Нажмите Enter (Ввод) — и консоль выведет `undefined` (рис. 6.13). Не паникуйте! Это не ошибка.

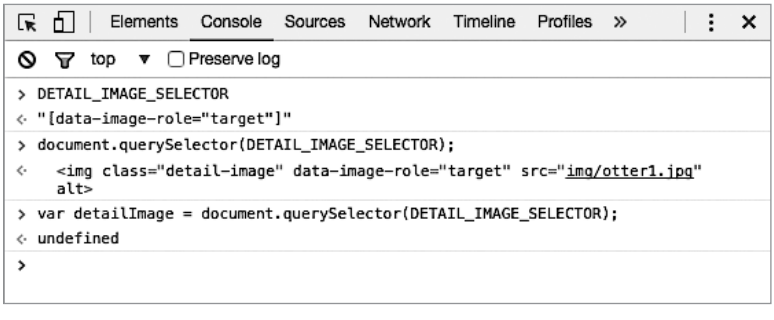


Рис. 6.13. Объявление переменной в консоли

Консоль просто делает свою работу, извещая нас об отсутствии возвращаемого значения при объявлении переменной. В JavaScript ключевое слово `undefined` означает отсутствие значения.

Это не говорит о том, что значение нашей переменной `detailImage` не было присвоено. Чтобы проверить это, наберите `detailImage` в консоли и нажмите Enter. Вы увидите HTML-представление увеличенного изображения, как и при вводе `document.querySelector(DETAIL_IMAGE_SELECTOR)` (рис. 6.14).

В чем же смысл всего этого? С помощью присваивания ссылки переменной можно использовать имя переменной всякий раз, когда необходимо сослаться на элемент. Теперь, чтобы не набирать каждый раз `document.querySelector(DETAIL_IMAGE_SELECTOR)`, можно набрать `detailImage`.

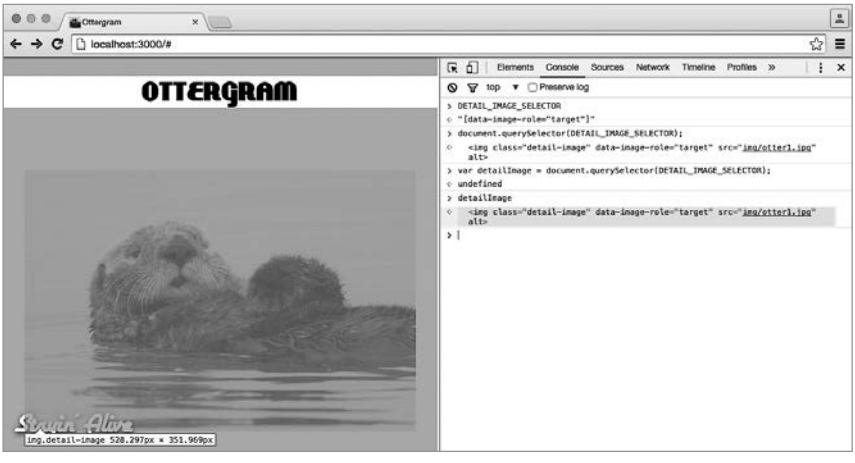


Рис. 6.14. Проверка значения переменной detailImage

Имея ссылку на увеличенное изображение, можно легко поменять значение его атрибута `src`. Присваиваем в консоли атрибуту `detailImage.src` строку `'img/otter2.jpg'`:

```
detailImage.src = 'img/otter2.jpg';
```

С помощью оператора-точки мы обратились к *свойству* `src` объекта `detailImage`. Свойство подобно переменной, но принадлежит конкретному объекту. Если вы присвоите `src` значение (или *установите* `src` в значение), равное строке `'img/otter2.jpg'`, то увидите, что в области увеличенного изображения уже находится фотография другой выдры (рис. 6.15).

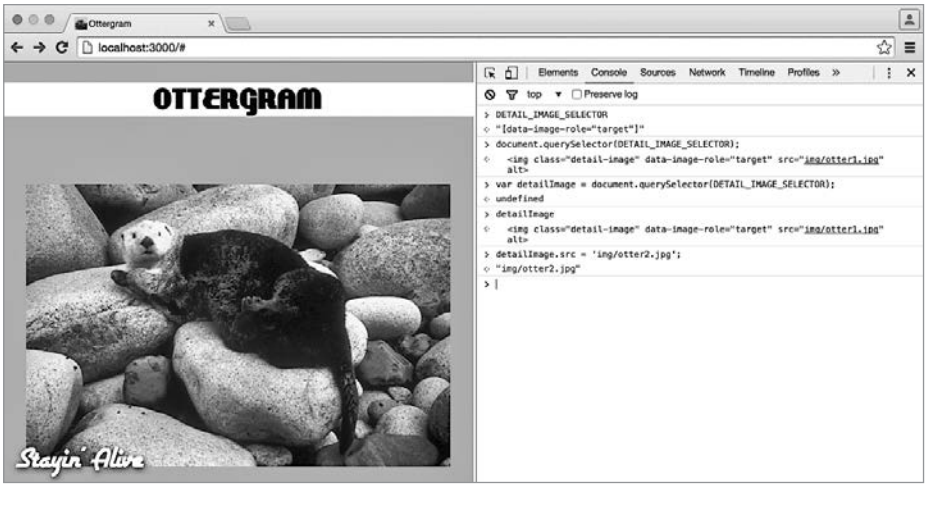


Рис. 6.15. Устанавливаем свойство src увеличенного изображения

Свойство `src` соответствует атрибуту `src` тега `<img>` в `index.html`. Поэтому еще один способ получить тот же результат — использовать метод `setAttribute` переменной `detailImage`.

Вызовите этот метод в консоли и передайте ему две строки: название атрибута и новое значение.

```
detailImage.setAttribute('src', 'img/otter3.jpg');
```

Увеличенное изображение снова поменялось (рис. 6.16).

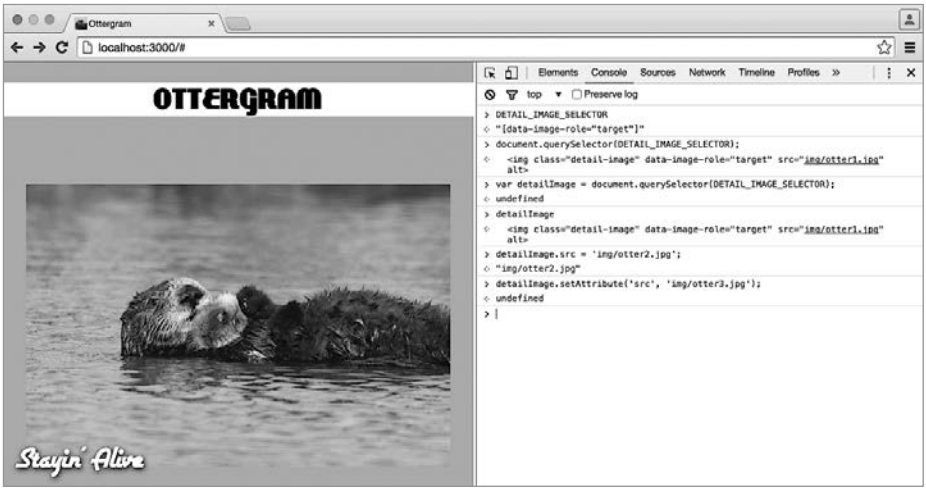


Рис. 6.16. Использование метода setAttribute для изменения изображения

Теперь у нас есть все, что нужно для автоматизации изменения увеличенного изображения. Приготовьтесь написать свою первую функцию!

Написание функции setDetails

Мы уже имели дело с методами и видели, что их можно вызвать, чтобы выполнить блок кода. Функции и методы, по сути, просто список шагов, которые хотелось бы использовать снова и снова. Вызов функции — это как будто сказать «Сделайте сендвич» вместо «Отрежьте два куска хлеба. Положите прошутто, салями и проволоне на один кусок. Положите второй кусок на первый».

В этой главе мы напишем семь функций для Ottergram. Наша первая функция будет менять увеличенное изображение и его название. Добавьте следующее *объявление функции* в файле `main.js`:

```
var DETAIL_IMAGE_SELECTOR = '[data-image-role="target"]';
var DETAIL_TITLE_SELECTOR = '[data-image-role="title"]';
```

```
var THUMBNAIL_LINK_SELECTOR = '[data-image-role="trigger"]';

function setDetails() {
  'use strict';
  // Здесь будет находиться код
}
```

Мы объявили функцию под названием `setDetails` с помощью ключевого слова `function`. При объявлении функции за названием всегда следует пара круглых скобок. Они, однако, не являются частью имени (вы скоро узнаете, для чего они нужны).

После круглых скобок следует пара фигурных. Внутри фигурных скобок находится *тело* функции. Тело будет содержать шаги, которые должна выполнять функция. Эти шаги называются *операторами*.

Первая строка нашей функции — `'use strict';`. Мы будем использовать ее в начале всех функций, чтобы сообщить браузеру, что они соответствуют последней стандартной версии JavaScript (больше информации о строгом режиме вы найдете в разделе «Для самых любознательных» в конце главы).

Вторая строка в функции `setDetails` — комментарий. Аналогично комментариям CSS комментарии JavaScript игнорируются браузером, но они полезны для разработчиков. Однострочные комментарии JavaScript можно записать вышеуказанным образом — с помощью `//`. Для комментариев, занимающих более одной строки, можно воспользоваться синтаксисом `/* */`. Оба варианта синтаксически корректны в JavaScript.

Мы уже опробовали в консоли все операторы, необходимые для изменения увеличенного изображения. Вернитесь в консоль и нажмите клавишу ↑. Вы увидите, что самый последний введенный вами оператор будет скопирован в приглашение командной строки. Стрелки ↑ и ↓ позволяют перемещаться назад и вперед по истории выполненных операторов.

С помощью клавиш со стрелками найдите оператор получения ссылки на увеличенное изображение: `var detailImage = document.querySelector (DETAIL_IMAGE_SELECTOR);`. Скопируйте эту строку из консоли и вставьте в файл `main.js` вместо комментария. Далее скопируйте и вставьте в консоли строку с вызовом метода `detailImage.setAttribute`: `detailImage.setAttribute('src', 'img/otter3.jpg');`.

Получившаяся функция `setDetails` должна выглядеть следующим образом:

```
...
function setDetails() {
  'use strict';
  // Здесь будет находиться код
  var detailImage = document.querySelector(DETAIL_IMAGE_SELECTOR);
  detailImage.setAttribute('src', 'img/otter3.jpg');
}
```

Сохраните файл `main.js` и вернитесь в консоль. Введите название функции `setDetails`, как указано ниже, и нажмите Enter для ее запуска.

```
setDetails();
```

Так мы вызовем функцию, работа которой приведет к выполнению приведенного выше кода. Вы увидите, что теперь в увеличенном изображении отображается `img/otter3.jpg` (рис. 6.17).

Рис. 6.17. Выполняем функцию setDetails для смены изображения

Функция `setDetails` поменяла увеличенное изображение, но не его название. А нам хотелось бы и того и другого. Аналогично тому, как мы поступили с увеличенным изображением, добавим операторы для получения ссылки на элемент и изменение его свойств.

В нашей функции `setDetails` в файле `main.js` снова вызываем метод `document.querySelector`, передавая ему переменную `DETAIL_TITLE_SELECTOR`. Присваиваем полученный результат новой переменной с именем `detailTitle`. Затем устанавливаем для свойства `textContent` значение `'You Should Be Dancing'` («Наверное, ты танцуешь»).

```
...
function setDetails() {
  'use strict';
  var detailImage = document.querySelector(DETAIL_IMAGE_SELECTOR);
  detailImage.setAttribute('src', 'img/otter3.jpg');

  var detailTitle = document.querySelector(DETAIL_TITLE_SELECTOR);
  detailTitle.textContent = 'You Should Be Dancing';
}
```

Свойство `textContent` представляет собой текст (не считая HTML-тегов) внутри элемента.

Сохраните изменения и выполните функцию `setDetails` в консоли. Теперь изменится и изображение, и название (рис. 6.18).

Рис. 6.18. Меняем изображение и название с помощью функции setDetails

Прием аргументов путем объявления параметров. Функция `setDetails` изменяет увеличенное изображение и его название. Но каждый раз при запуске она устанавливает значение атрибута `src` изображения равным `'img/otter3.jpg'`, а значение атрибута `textContent` названия — равным `'You Should Be Dancing'`. Но что если нам необходимы другие изображения и текст?

Нам нужно подобрать способ, чтобы сообщать функции `setDetails` при ее вызове, *какое* изображение и *какой* текст ей использовать. Для этого требуется, чтобы наша функция принимала *аргументы* (передаваемые ей значения, которые она сможет использовать). Чтобы это сделать, необходимо указать *параметры* в объявлении функции.

Добавьте два параметра в объявление функции `setDetails` в файле `main.js`:

```
...
function setDetails(imageUrl, titleText) {
  'use strict';
  var detailImage = document.querySelector(DETAIL_IMAGE_SELECTOR);
  detailImage.setAttribute('src', 'img/otter3.jpg');

  var detailTitle = document.querySelector(DETAIL_TITLE_SELECTOR);
  detailTitle.textContent = 'You Should Be Dancing';
}
```

Далее подставим эти два параметра вместо `'img/otter3.jpg'` и `'You Should Be Dancing'`:

```
...
function setDetails(imageUrl, titleText) {
  'use strict';
  var detailImage = document.querySelector(DETAIL_IMAGE_SELECTOR);
  detailImage.setAttribute('src', 'img/otter3.jpg');
  detailImage.setAttribute('src', imageUrl);

  var detailTitle = document.querySelector(DETAIL_TITLE_SELECTOR);
  detailTitle.textContent = 'You Should Be Dancing';
  detailTitle.textContent = titleText;
}
```

Два наших параметра, `imageUrl` и `titleText`, используются в качестве ярлыков для передаваемых функции `setDetails` значений. Сохраните файл `main.js` и опробуйте ее в консоли, чтобы увидеть, как она работает.

Вызовите функцию `setDetails` и передайте ей значения `'img/otter4.jpg'` и `'Night Fever'` (убедитесь, что между ними стоит запятая).

```
setDetails('img/otter4.jpg', 'Night Fever');
```

Вы должны увидеть новые изображение и текст названия, как показано на рис. 6.19.

Между аргументами и параметрами существует важное различие. *Параметры описываются как часть функции*. В JavaScript параметры в точности соответствуют переменным, объявленным внутри тела функции. *Аргументы — это значения, передаваемые функции при ее вызове*.

Кроме того, знайте, что вне зависимости от использованных для аргументов имен переменных их значения всегда привязываются к именам параметров, чтобы их

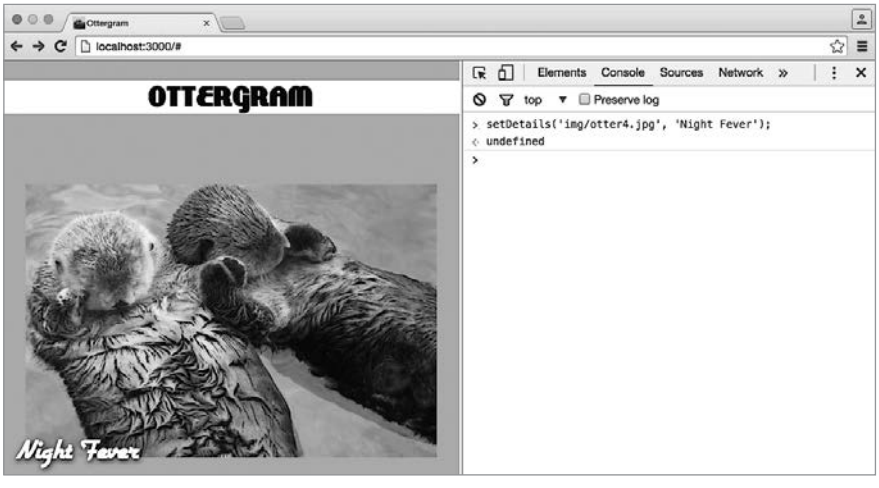


Рис. 6.19. Передача значений функции setDetails

можно было использовать внутри тела функции. Например, представьте себе, что вы использовали переменные для URL изображения и текста названия. При вызове функции `setDetails` вы передали эти две переменные в качестве аргументов.

```
var otterOneImage = 'img/otter1.jpg';
var otterOneTitle = 'Stayin\' Alive';

setDetails(otterOneImage, otterOneTitle);
```

Функция `setDetails` принимает значения, помечает их именами параметров `imageUrl` и `titleText`, после чего выполняет имеющийся в ней код. Этот код использует оба названных параметра, передавая их в качестве аргументов методу `document.querySelector`.

Аналогично именам переменных имена параметров — просто «ярлыки» для значений. Вы можете использовать такие имена параметров, какие пожелаете[1], но рекомендуется выбирать наглядные имена (как мы и сделали), чтобы облегчить чтение и сопровождение кода.

Возврат значений из функций

Мы выполнили первый (или, точнее, последний) пункт нашего плана и по пути узнали парочку секретов JavaScript. Теперь перейдем к следующим двум пунктам

[1] На самом деле на имена параметров, как и переменных, в JavaScript накладываются ограничения: 1) имя параметра может состоять только из букв, цифр и знака подчеркивания (_); 2) первый символ имени должен быть буквой или знаком подчеркивания; цифра быть первым символом имени не может; 3) имя параметра не должно быть зарезервированным словом. — *Примеч. пер.*

списка: получению URL изображения и названия из миниатюры. Для каждого из них напишем новую функцию.

Добавьте объявление функции `imageFromThumb` в файл `main.js`. Она примет один параметр — `thumbnail`, представляющий собой ссылку на элемент-якорь миниатюры, и будет извлекать и возвращать значение атрибута `data-image-url`.

```
...
function setDetails(imageUrl, titleText) {
  ...
}

function imageFromThumb(thumbnail) {
  'use strict';
  return thumbnail.getAttribute('data-image-url');
}
```

Метод `getAttribute` делает противоположное методу `setAttribute`, который мы использовали в функции `setDetails`. Он принимает только один аргумент — имя атрибута.

В отличие от `setDetails` функция `imageFromThumb` использует ключевое слово `return`. При вызове функции, имеющей ключевое слово `return`, она возвращает значение. Пример такой функции — метод `querySelector`. При вызове он вернул значение, которое мы затем присвоили переменной.

Сохраните файл `main.js` и выполните в консоли следующий код, нажимая Enter между строками:

```
var firstThumbnail = document.querySelector(THUMBNAIL_LINK_SELECTOR);
imageFromThumb(firstThumbnail);
```

Консоль показывает, что возвращенное значение было строкой `"img/otter1.jpg"`, поскольку функция `imageFromThumb` возвращает значение атрибута `data-image-url` миниатюры (рис. 6.20).

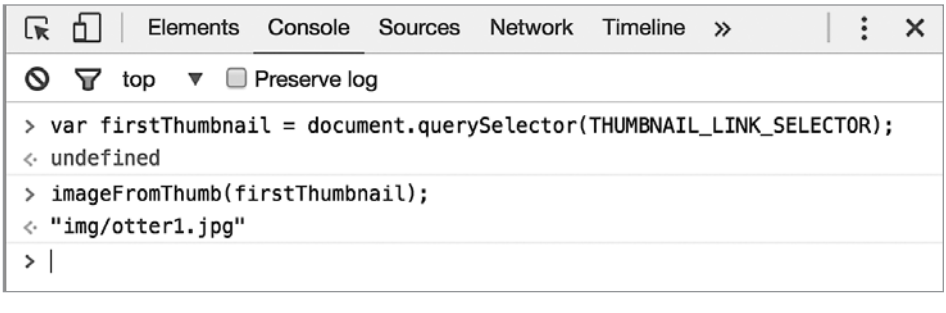


Рис. 6.20. Возвращенное из функции imageFromThumb значение

Обратите внимание, что любые находящиеся после оператора `return` операторы не будут выполнены. Оператор `return` фактически завершает выполнение функции.

Следующая функция, которую мы напишем, будет принимать ссылку на элемент миниатюры и возвращать текст названия.

Добавьте объявление функции `titleFromThumb` в файл `main.js` (с параметром `thumbnail`). Она будет возвращать значение атрибута `data-image-title`.

```
...
function imageFromThumb(thumbnail) {
  ...
}

function titleFromThumb(thumbnail) {
  'use strict';
  return thumbnail.getAttribute('data-image-title');
}
```

Сохраните файл `main.js` и оцените, как работает эта функция, в консоли (рис. 6.21):

```
var firstThumbnail = document.querySelector(THUMBNAIL_LINK_SELECTOR);
titleFromThumb(firstThumbnail);
```

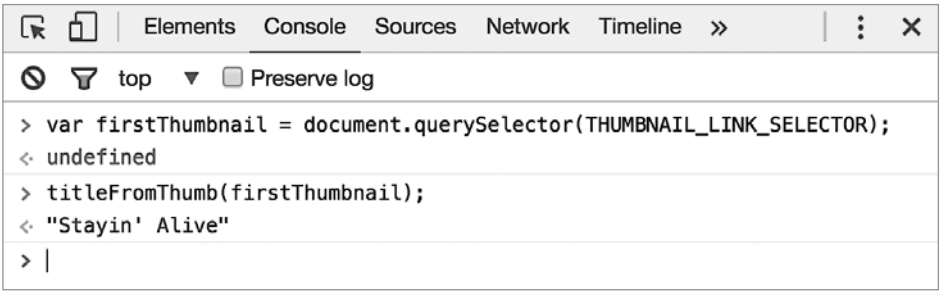


Рис. 6.21. Возвращенное из функции titleFromThumb значение

Следующая функция, которую необходимо написать, ради удобства объединяет все три предыдущие функции, так что вам не нужно вызывать их по отдельности. Она будет принимать на входе ссылку на элемент миниатюры, после чего вызывать функцию `setDetails`, передавая ей значения, полученные из вызовов функций `imageFromThumb` и `titleFromThumb`.

Добавьте в файл `main.js` объявление функции `setDetailsFromThumb`:

```
...
function titleFromThumb(thumbnail) {
  ...
}

function setDetailsFromThumb(thumbnail) {
  'use strict';
  setDetails(imageFromThumb(thumbnail), titleFromThumb(thumbnail));
}
```

Обратите внимание, что функция `setDetails` вызывается с двумя параметрами (эти параметры тоже представляют собой вызовы функций). Как это работает?

Прежде чем `setDetails` вызывается, ее аргументы предварительно сводятся к простейшим возможным значениям. Во-первых, выполняется и возвращает значение функция `imageFromThumb(thumbnail)`. Затем выполняется и возвращает значение функция `titleFromThumb(thumbnail)`. И наконец, вызывается функция `setDetails` со значениями, возвращенными вызовами функций `imageFromThumb(thumbnail)` и `titleFromThumb(thumbnail)`. Рисунок 6.22 иллюстрирует этот процесс.

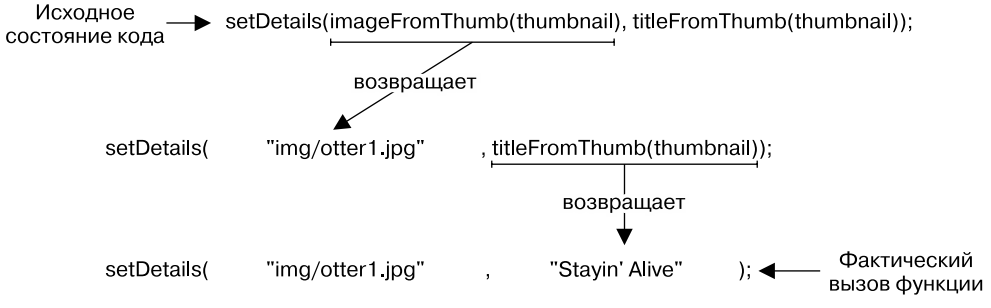


Рис. 6.22. Использование вызовов функций в качестве аргументов

Сохраните файл `main.js`. Мы создали весь код, необходимый для извлечения значений атрибутов данных из миниатюр и использования этих значений для изменения того, что отображается в увеличенном изображении и его названии.

Далее от низкоуровневых операций переходим к написанию кода, позволяющего переносить данные из миниатюры в увеличенное изображение при щелчке пользователем на миниатюре.

Добавляем прослушиватель событий

Браузеры — очень «занятые» программы. Ни один щелчок кнопкой мыши, прокручивание или нажатие клавиши не остаются не замеченными ими. Каждое из этих действий — *событие (event)*, на которое браузер должен отреагировать. Чтобы сайты были более динамичными и интерактивными, можно сделать так, чтобы тогда, когда происходит одно из этих событий, срабатывал наш собственный код. В текущем разделе мы добавим для каждой из миниатюр *прослушиватель события*.

Прослушиватель события — объект, который, как понятно из названия, «слушает», не произойдет ли определенное событие, например щелчок кнопкой мыши. Когда оно происходит, прослушиватель события запускает вызов функции в ответ на это событие. (События, связанные с мышью, такие как щелчки и двойные щелчки, и связанные с клавиатурой, такие как нажатия клавиш, — наиболее распространен-

ные типы событий. Чтобы увидеть полный список событий, загляните в справочник событий в MDN по адресу developer.mozilla.org/en-US/docs/Web/Events.)

Метод `addEventListener` доступен для каждого элемента DOM, включая объект `document`. Как и ранее, мы будем сначала экспериментировать с кодом в консоли, а затем уже использовать протестированный код для написания функций в файле `main.js`.

Переключитесь на браузер Chrome и введите в консоли следующий код. Вам нужно нажимать Shift+Enter для ввода разрывов строк. Нажмите Enter, когда закончите ввод всего кода.

```
document.addEventListener('click', function () {
  console.log('you clicked!');
});
```

Введенный вами код добавляет прослушиватель события для объекта `document`, слушающий на предмет щелчков кнопкой мыши на странице. При щелчке прослушиватель события будет выводить в консоль you clicked! (вы щелкнули кнопкой мыши!) с помощью встроенного метода `console.log` (рис. 6.23).

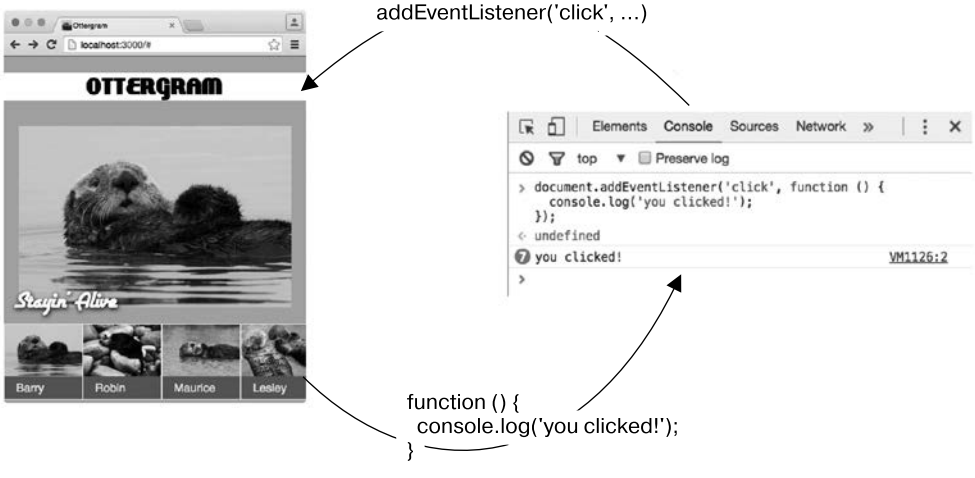


Рис. 6.23. Добавляем прослушиватель события для щелчков кнопкой мыши

Щелкните кнопкой мыши на заголовке, увеличенном изображении или фоне. Вы увидите, что в консоли появился текст you clicked! (вы щелкнули кнопкой мыши!) (не щелкайте на миниатюрах — это уведет вас со страницы `index.html` Ottergram. Когда вы не находитесь на `index.html`, никакая ваша разметка, CSS и JavaScript не будут загружаться и выполняться в браузере.)

Метод `addEventListener` принимает на входе два аргумента: строку с названием события и функцию. `addEventListener` будет выполнять эту функцию всякий раз,

когда для данного элемента происходит соответствующее событие. То, как эта функция написана, может на первый взгляд показаться странным. Это *анонимная функция*.

До сих пор мы имели дело с *поименованными функциями*, такими как `setDetails` и `titleFromThumb`. Как и следует ожидать, у поименованных функций есть имена, и создаются они с помощью объявлений функций.

Можно также создавать функциональные литеральные значения — аналогично созданию числовых литеральных значений, таких как `42`, и строковых литеральных значений, таких как `"Barry the Otter"` (Барри — выдра). Другое название функции-литерала — *анонимная функция*.

Анонимные функции часто используются в качестве аргументов других функций, как та, которую мы передали в качестве второго аргумента методу `document.addEventListener`. Практика передачи одной функции другой довольно распространена в JavaScript и известна как *паттерн обратного вызова*, поскольку передаваемая в качестве аргумента функция будет «вызвана обратно»[1] в какой-то момент в будущем.

Вполне допустимо использовать и поименованную функцию в качестве обратного вызова, но многие разработчики клиентской части используют анонимные функции, поскольку они обеспечивают большую гибкость, чем поименованные. Вскоре вы увидите, как это работает.

Теперь надо добавить прослушиватель события для отдельной миниатюры. Введите в консоли следующий код (не забывайте нажимать Shift+Enter для разрывов строк в обращении к методу `firstThumbnail.addEventListener`):

```
var firstThumbnail = document.querySelector(THUMBNAIL_LINK_SELECTOR);
firstThumbnail.addEventListener('click', function () {
  console.log('you clicked!');
});
```

Если вы щелкнете на первой миниатюре (Barry the Otter, самая дальняя слева), то браузер перенаправит вас к увеличенному изображению Барри. Что произошло? Как вы помните, каждая миниатюра обернута в тег-якорь, атрибут `href` которого указывает на изображение, например `img/otter1.jpg`. Это обычное поведение браузера при нажатии пользователем на ссылку: он открывает файл, на который указывает атрибут `href`.

Но мы же не хотим уходить из Ottergram при нажатии на ссылку, также нам не хотелось бы менять теги-якоря. К счастью, все это можно уладить изнутри функции обратного вызова.

Вспомните: ранее в данной главе говорилось, что функции делают свое дело, а нам не нужно заботиться о деталях происходящего. Обычно необходимо только

[1] Выполнена основной функцией. — *Примеч. пер.*

знать, какую информацию передать в качестве аргументов и какая информация будет возвращена. При передаче функции обратного вызова в качестве аргумента нужно знать на один пункт больше: какая информация будет передана функции обратного вызова.

При вызове метода `addEventListener` мы как бы говорим браузеру: «Когда произойдет щелчок на `firstThumbnail`, вызови вот эту функцию» — и браузер старательно ждет щелчка на этом элементе. Если щелчок происходит, браузер фиксирует все относящиеся к данному событию подробности (точное местонахождение мыши, нажималась правая или левая ее кнопка, одинарный или двойной щелчок). Затем браузер передает объект с этой информацией нашей функции. Этот объект называется *«объект-событие»*.

Эта взаимосвязь схематично изображена на рис. 6.24 с помощью вымышленной реализации метода `addEventListener`.

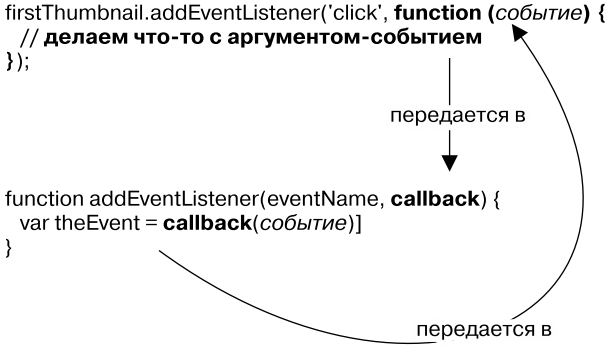


Рис. 6.24. Передача анонимной функции, ожидающей на входе аргумент

Совсем скоро мы передадим анонимную функцию в метод `addEventListener`, как и раньше. Но на этот раз наша анонимная функция будет ожидать на входе аргумент. Убедитесь, что Ottergram находится на странице `index.html`, и введите в консоли следующий код:

```
var firstThumbnail = document.querySelector(THUMBNAIL_LINK_SELECTOR);
firstThumbnail.addEventListener('click', function (event) {
  event.preventDefault();
  console.log('you clicked!');
  console.log(event);
});
```

Браузер начнет вызывать нашу анонимную функцию всякий раз при щелчке на `firstThumbnail`, причем будет передавать ей объект-событие. Далее мы вызываем метод `preventDefault` этого объекта (названного нами `event`). Это не даст перейти браузеру на другую страницу. Наконец, вызываем метод `console.log` для объекта-события, чтобы можно было изучить его в DevTools.

Теперь щелкните на первой миниатюре. Браузер останется на странице Ottergram, а событие будет записано в консоль: `MouseEvent {isTrusted: true}`. Если вы щелкнете кнопкой мыши на стрелке рядом с `MouseEvent` для разворачивания текста, то должны увидеть немало информации о событии (рис. 6.25), включая координаты мыши на странице, то, какая кнопка мыши была нажата и были ли во время щелчка нажаты какие-то специальные клавиши-модификаторы на клавиатуре.

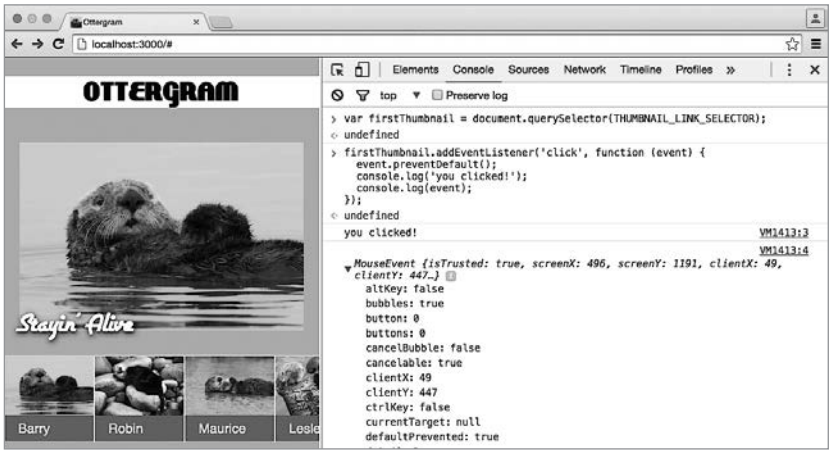


Рис. 6.25. Предотвращаем событие по умолчанию и заносим в журнал объект-событие

Пока что не концентрируйтесь на различных свойствах объекта-события. Просто запомните, что он содержит массу информации о произошедшем событии браузера.

Кстати, совсем не обязательно, что параметр функции обратного вызова должен называться `event` (он будет соответствовать передаваемому значению независимо от того, как его назвать). Вы можете использовать любое имя параметра, какое вам нравится, но хорошей практикой считается выбирать наглядные имена (как мы и поступали), чтобы код было легче читать и сопровождать.

Теперь у нас есть функция, принимающая на входе миниатюру и добавляющая прослушиватель события. Добавьте в файл `main.js` объявление функции `addThumbClickHandler`. Оно должно описывать параметр `thumb`.

Вы можете скопировать наш экспериментальный код для метода `addEventListener` из консоли и вставить его в тело функции `addThumbClickHandler`. Измените его так, чтобы вызывать метод `thumb.addEventListener`. Пока в функции обратного вызова нам понадобится вызвать только метод `event.preventDefault`.

```
...
function setDetailsFromThumb(thumbnail) {
  ...
}

function addThumbClickHandler(thumb)
  'use strict';
  thumb.addEventListener('click', function (event) {
    event.preventDefault();
  });
}
```

Внутри обратного вызова для события есть доступ к параметру `thumb`, объявленному как часть функции `addThumbClickHandler`. Передадим его в вызов функции `setDetailsFromThumb`.

```
...
function addThumbClickHandler(thumb) {
  'use strict';
  thumb.addEventListener('click', function (event) {
    event.preventDefault();
    setDetailsFromThumb(thumb);
  });
}
```

В JavaScript, как и в других языках программирования, имеются правила относительно описания и доступа к переменным и функциям. Переданная нами методу `addEventListener` анонимная функция может обращаться к функции `setDetailsFromThumb`, поскольку `setDetailsFromThumb` была объявлена в *глобальной области видимости*. Это значит, что к ней можно обратиться из любой другой функции или из консоли. Это же относится и к переменным, например `DETAIL_IMAGE_SELECTOR`, которые также объявлены в глобальной области видимости.

Однако переменные `detailImage` и `detailTitle`, объявленные *внутри* функции `setDetails`, доступны только в теле `setDetails`. К ним нельзя обратиться из консоли или из других функций. Эти переменные объявлены в *области видимости функции (локальной области видимости)* `setDetails`. Параметры функции ведут себя во многом аналогично объявленным внутри функции переменным. Они тоже являются частью области видимости этой функции.

При обычных обстоятельствах функции не могут обращаться к переменным или функциям, являющимся частью области видимости другой функции. Функция `addThumbClickHandler` интересна тем, что объявляет параметр `thumb`, к которому обращается другая функция (функция обратного вызова, переданная методу `addEventListener`). Это возможно благодаря тому, что функция обратного вызова является частью области видимости функции `addThumbClickHandler`.

Прочитать об этом подробнее вы можете в разделе «Для самых любознательных» в конце главы.

Доступ ко всем миниатюрам

Мы добавили в консоли прослушиватель события для первой миниатюры. Теперь добавим прослушиватель события для всех миниатюр с помощью нового метода DOM.

Когда мы извлекали увеличенное изображение и название для него, мы использовали метод `document.querySelector` для поиска в DOM элемента, который бы соответствовал переданному в него селектору. `document.querySelector` вернет только одно значение, даже если мы передадим в него селектор, соответствующий нескольким элементам.

Метод `document.querySelectorAll`, напротив, будет возвращать список всех подходящих элементов. Вызовите `document.querySelectorAll(THUMBNAIL_LINK_SELECTOR)` в консоли и изучите полученные результаты. Вы должны увидеть список элементов-якорей (рис. 6.26).

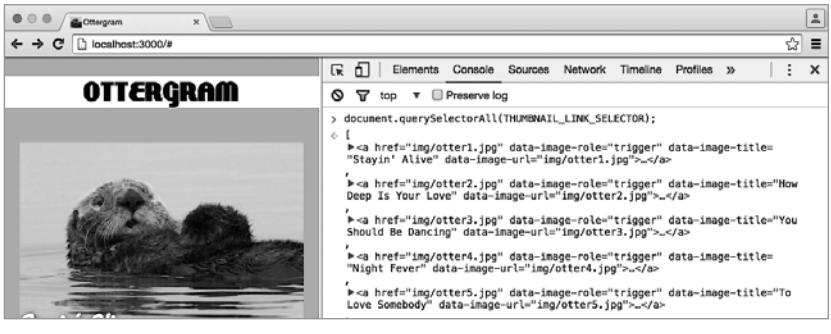


Рис. 6.26. Метод document.querySelectorAll возвращает несколько подходящих элементов

Зная это, можно проверить работу функции `setDetailsFromThumb`. Присвойте в консоли результат вызова метода `document.querySelectorAll(THUMBNAIL_LINK_SELECTOR)` переменной `thumbnails`. Воспользуйтесь *синтаксисом с квадратными скобками* для извлечения из списка `thumbnails` пятого элемента и передайте его функции `setDetailsFromThumb`. Синтаксис с квадратными скобками предоставляет возможность указывать элемент списка с помощью его числовой позиции в списке. Нумерация позиций начинается с 0, так что позиция пятого элемента равна 4.

Вот код, который вам необходимо ввести в консоли:

```
var thumbnails = document.querySelectorAll(THUMBNAIL_LINK_SELECTOR);
setDetailsFromThumb(thumbnails[4]);
```

После выполнения этих действий вы увидите, что элемент из списка `thumbnails` может быть передан функции `setDetailsFromThumb`, успешно меняя увеличенное изображение и название (рис. 6.27).

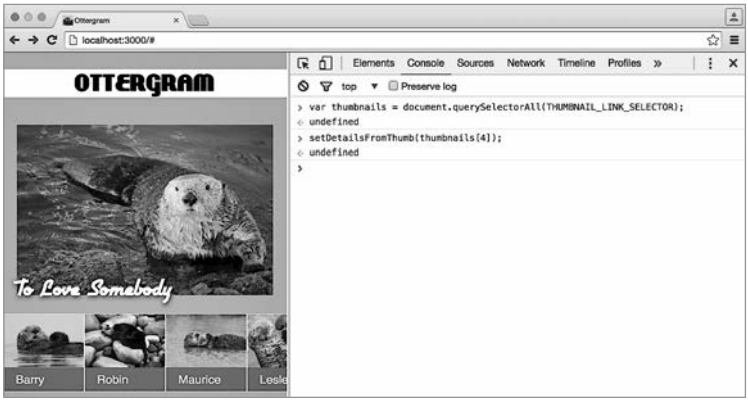


Рис. 6.27. Передаем функции setDetailsFromThumb элемент, полученный из вызова метода querySelectorAll

Добавьте в файл `main.js` функцию с именем `getThumbnailsArray` и вставьте туда код, извлекающий все соответствующие `THUMBNAIL_LINK_SELECTOR` элементы и присваивающий результат переменной `thumbnails`.

```
...
function addThumbClickHandler(thumb) {
  ...
}

function getThumbnailsArray() {
  'use strict';
  var thumbnails = document.querySelectorAll(THUMBNAIL_LINK_SELECTOR);
}
```

Прежде чем двигаться дальше, упомянем небольшой глюк, встречающийся при работе с методами DOM. Методы, возвращающие списки элементов, не возвращают *массивы*. Вместо этого они возвращают объект NodeList. И массивы, и NodeList — списки элементов, но у массивов имеется немало мощных методов для работы с наборами элементов (некоторые из них были бы полезны нам в Ottergram).

Нам понадобится преобразовать возвращенный из метода `querySelectorAll` NodeList в массив с помощью довольно необычно выглядящего кода JavaScript. Не беспокойтесь об этом синтаксисе пока что. Это обратно совместимый способ преобразования из NodeList в массив. Внесите следующие изменения в файл `main.js`:

```
...
function getThumbnailsArray() {
  'use strict';
  var thumbnails = document.querySelectorAll(THUMBNAIL_LINK_SELECTOR);
  var thumbnailArray = [].slice.call(thumbnails);
  return thumbnailArray;
}
```

Теперь можно взять все миниатюры выдр и связать их с нашим кодом прослушивания событий, который будет менять увеличенное изображение и название в ответ на щелчок кнопкой мыши.

Организация цикла по массиву миниатюр

Связывание миниатюр с кодом обработки событий будет быстрым и несложным делом. Мы напишем функцию, которая станет отправной точкой всей логики Ottergram. В других языках программирования, в отличие от JavaScript, есть встроенный механизм запуска приложения. Но не волнуйтесь — его достаточно легко можно реализовать вручную.

Начнем с добавления функции `initializeEvents` в конец файла `main.js`. Этот метод свяжет воедино все шаги по превращению Ottergram в интерактивное приложение. Во-первых, он получит массив миниатюр. Далее он пройдет в цикле по массиву, добавляя обработчик нажатий для каждой из них. После написания этой функции мы добавим вызов функции `initializeEvents` в самый конец файла `main.js` для ее запуска.

В теле нашей новой функции добавьте вызов функции `getThumbnailsArray` и присвойте результат (массив миниатюр) переменной `thumbnails`:

```
...
function getThumbnailsArray() {
  ...
}

function initializeEvents() {
  'use strict';
  var thumbnails = getThumbnailsArray();
}
```

Далее нам нужно пройти в цикле по массиву миниатюр, по одному элементу за раз. При обращении к каждому из них мы будем вызывать метод `addThumbClickHandler` и передавать ему элемент миниатюры. Это может показаться несколькими шагами, но поскольку `thumbnails` — настоящий массив, сделать все это можно с помощью вызова одного-единственного метода.

Добавьте вызов метода `thumbnails.forEach` в файл `main.js` и передайте его функции `addThumbClickHandler` в качестве обратного вызова.

```
...
function initializeEvents() {
  'use strict';
  var thumbnails = getThumbnailsArray();
  thumbnails.forEach(addThumbClickHandler);
}
```

Обратите внимание, что вы передаете в качестве обратного вызова поименованную функцию. Как вы прочтете далее, это не всегда хорошее решение. Однако в данном случае оно сработает как надо, поскольку функции `addThumbClickHandler` требуется только та информация, которая будет ей передаваться, когда ее будет вызывать метод `forEach`, — элемент массива `thumbnails`.

Наконец, чтобы увидеть все это в действии, добавьте вызов функции `initializeEvents` в самый конец файла `main.js`:

```
...
function initializeEvents() {
  'use strict';
  var thumbnails = getThumbnailsArray();
  thumbnails.forEach(addThumbClickHandler);
}

initializeEvents();
```

Помните, браузер выполняет код по мере чтения каждой строки вашего JavaScript. На протяжении большей части файла `main.js` он просто выполняет объявления переменных и функций. Но когда он дойдет до строки `initializeEvents();`, он выполнит эту функцию.

Сохраните и вернитесь в браузер. Нажмите на несколько различных миниатюр и полюбуйтесь на плоды своих трудов (рис. 6.28).

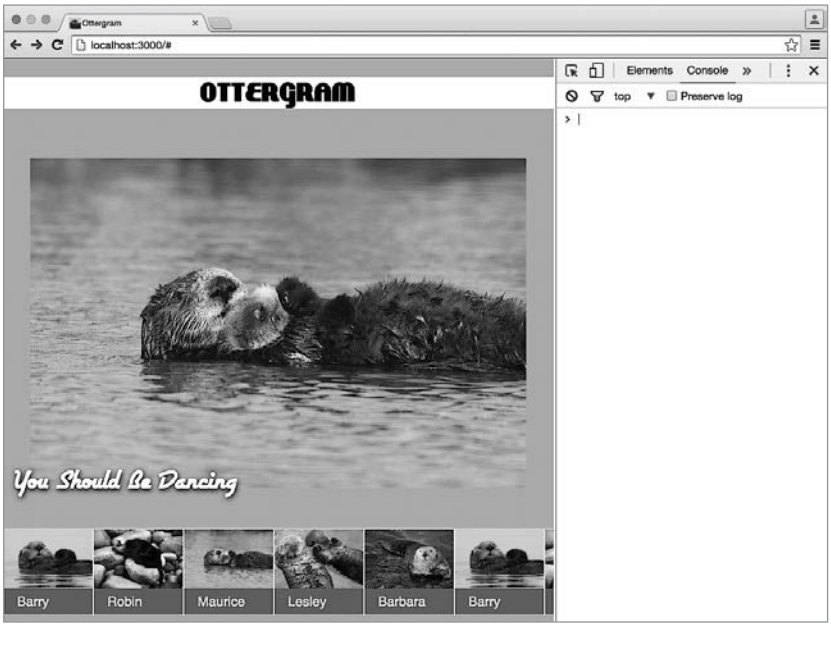


Рис. 6.28. И мы тоже можем пуститься в пляс

Откиньтесь на спинку кресла, расслабьтесь и наслаждайтесь щелчками на фото выдр! Вы немало потрудились и усвоили много нового во время создания интерактивного слоя нашего сайта. В следующей главе мы завершим создание Ottergram, добавив для пущей красоты визуальные эффекты.

Серебряное упражнение: взлом ссылок

DevTools браузера Chrome предоставляют немало возможностей для развлечений с посещаемыми страницами. Следующее упражнение будет заключаться в том, чтобы поменять все ссылки на странице результатов поиска так, чтобы они вели в никуда.

Зайдите в вашу любимую поисковую систему и выполните поиск по ключевому слову выдры. Откройте консоль DevTools. Используя написанные для Ottergram функции в качестве образца, подключите прослушиватели событий ко всем ссылкам и отключите имеющуюся по умолчанию функциональность перехода по щелчку кнопкой мыши.

Золотое упражнение: случайные выдры

Напишите функцию для изменения атрибута `data-image-url` случайно выбранной миниатюры выдры так, чтобы увеличенное изображение более не соответствовало миниатюре. Используйте URL изображения по вашему выбору (хотя можно отыскать неплохое путем поиска в Интернете по слову `tacocat`).

В качестве дополнительного упражнения напишите функцию, возвращающую миниатюрам выдр исходные значения атрибута `data-image-url` и меняющую одну из них, выбранную случайным образом.

Для самых любознательных: строгий режим

Что такое строгий режим и для чего он существует? Он был создан в качестве более «чистого» режима JavaScript, позволяет перехватывать определенные виды ошибок программирования (например, опечатки в именах переменных), удерживая разработчиков от использования некоторых подверженных ошибкам частей языка и отключая возможности языка, которые попросту явно неудачны.

У строгого режима есть немало преимуществ.

- Заставляет использовать ключевое слово `var`.
- Не требует использования операторов `with`.
- Ограничивает способы использования функции `eval`.
- Рассматривает дублирующиеся имена параметров функций как синтаксическую ошибку.

Все это можно получить всего лишь за счет размещения директивы `'use strict'` наверху функции. В качестве бонуса директива `'use strict'` игнорируется не поддерживающими ее старыми версиями браузеров (эти браузеры просто рассматривают эту директиву как строку).

Прочесть больше о строгом режиме можно на MDN по адресу developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Strict_mode.

Для самых любознательных: замыкания

Ранее мы упоминали, что разработчики часто предпочитают использовать анонимные функции для обратных вызовов вместо поименованных. Функция `addThumbClickHandler` демонстрирует, почему анонимная функция — лучший вариант, чем поименованная.

Предположим, мы попытались использовать для обратного вызова поименованную функцию `clickFunction`. Внутри этой функции мы можем обращаться к объекту `event`, поскольку его будет передавать туда метод `addEventListener`. Но у тела функции `clickFunction` нет доступа к объекту `thumb`. Этот параметр доступен только *внутри* функции `addThumbClickHandler`.

```
function clickFunction (event) {
  event.preventDefault();
  setDetailsFromThumb(thumb); // <--- Это вызовет ошибку
}

function addThumbClickHandler(thumb) {
  thumb.addEventListener('click', clickFunction);
}
```

С другой стороны, используемая анонимная функция получает доступ к параметру `thumb`, поскольку она также находится внутри функции `addThumbClickHandler`. Когда функция описана внутри другой функции, она может использовать любые параметры и переменные этой внешней функции. В терминах теории вычислительной техники это известно как *замыкание*.

При запуске функции `addThumbClickHandler` вызывается метод `addEventListener`, который ставит в соответствие функции обратного вызова событие щелчка кнопкой мыши. Браузер отслеживает эти ассоциации, сохраняя ссылки на функции обратного вызова и выполняя обратные вызовы при наступлении соответствующих событий.

Когда обратный вызов наконец выполняется, формально переменных и параметров функции `addThumbClickHandler` уже нет. Их существование прекращается с завершением выполнения `addThumbClickHandler`. Но обратный вызов сохраняет значения переменных и параметров `addThumbClickHandler` и использует их во время своего выполнения.

Для получения более полной информации изучите раздел о замыканиях в MDN.

Для самых любознательных: NodeList and HTMLCollection

Существует два способа извлечения имеющихся в DOM списков элементов. Первый — метод `document.querySelectorAll`, возвращающий `NodeList`. Второй — метод `document.getElementsByTagName`, отличающийся от `document.querySelectorAll` тем, что ему можно передать только строку с именем тега, например `"div"` или `"a"`, и он возвращает интерфейс HTMLCollection.

Ни NodeList, ни HTMLCollection не являются настоящими массивами, так что у них нет таких методов массивов, как `forEach`, но у них все же есть весьма интересные свойства.

HTMLCollection представляет собой набор *динамических узлов*. Это значит, что при внесении изменений в DOM для изменения содержимого HTMLCollection не нужно опять вызывать метод `document.getElementsByTagName`.

Чтобы увидеть, как это работает, выполните в консоли следующее:

```
var thumbnails = document.getElementsByTagName("a");
thumbnails.length;
```

После получения всех элементов-якорей в виде HTMLCollection мы вывели длину (`length`) этого списка в консоль.

Теперь удалим часть тегов-якорей со страницы с помощью панели элементов в DevTools. Щелкните кнопкой мыши с нажатой клавишей Control (правой кнопкой мыши — в Windows) на одном из элементов списка и выберите Delete Element (Удалить элемент) (рис. 6.29).

Сделайте это несколько раз, после чего еще раз введите в консоли `thumbnails.length`. Вы должны увидеть при этом, что значение `length` изменилось (рис. 6.30).

Преобразование NodeList и HTMLCollection в массивы не только облегчает работу с ними благодаря использованию методов массивов, но и гарантирует, что элементы массива не поменяются, даже если изменится DOM.

Для самых любознательных: типы данных JavaScript

В этой главе мы создавали переменные, чтобы ссылаться на данные в наших функциях. Мы рассказали вам в самом начале, что строки, числа и булев тип — три из пяти *простых типов данных*. Остальные два типа: `null` и `undefined`.

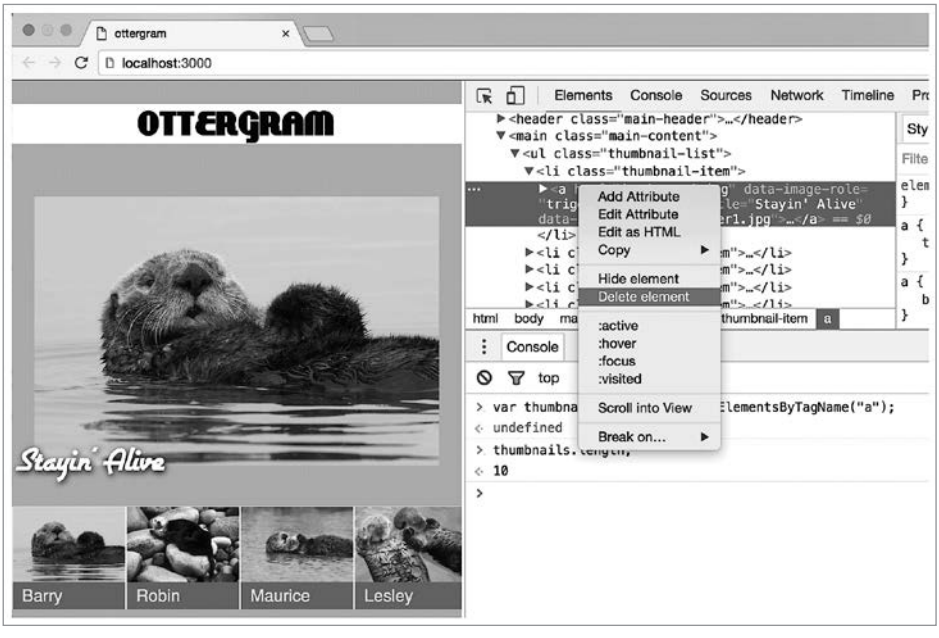


Рис. 6.29. Удаление элемента DOM с помощью DevTools

Рис. 6.30. Значение length изменилось после удаления элементов

В табл. 6.2 кратко описаны свойства пяти простых типов данных.

Все остальные типы в JavaScript считаются *составными*, или *сложными*. Они включают массивы и объекты, которые могут содержать внутри объекты других типов.

Например, мы написали функцию, которая создала массив объектов миниатюр. У массивов тоже есть свойства (например, `length`) и методы (такие как `forEach`).

Таблица 6.2. Простые типы данных в JavaScript

Тип	Пример	Описание
string	«И вы получаете $100! И вы получаете $100! И...»	Буквы, числа или символы, заключенные в парные кавычки
number	42, 3.14159, –1	Целые числа и десятичные дроби
Boolean	true, false	Ключевые слова true и false, соответствующие логической истине и лжи
null	null	Обозначает недопустимое значение
undefined	undefined	Значение переменной, которой еще ничего не было присвоено

Мы будем работать с простыми и сложными типами данных и дальше, на протяжении всей книги.

7 CSS и визуальные эффекты

В предыдущей главе мы наделили Ottergram способностью отвечать на действия пользователя изменением увеличенного изображения при щелчке на миниатюре. В текущей главе мы добавим в Ottergram еще три различных визуальных эффекта.

Первый визуальный эффект — простое изменение макета, включающее скрытие увеличенного изображения. При этом миниатюры будут занимать всю ширину страницы. При щелчке пользователя на миниатюре увеличенное изображение будет появляться снова, а миниатюры — возвращаться к своему первоначальному размеру.

Остальные два эффекта будут использовать CSS для создания визуальной анимации миниатюр и увеличенного изображения (рис. 7.1).

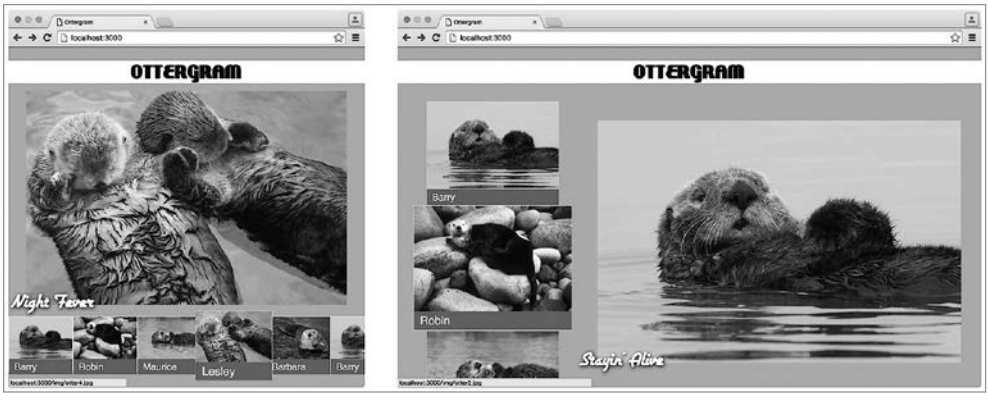


Рис. 7.1. Ottergram с эффектами перехода

Скрытие и отображение увеличенного изображения

Пользователи Ottergram хотели бы иметь возможность прокрутки миниатюр без наличия на странице увеличенного изображения (рис. 7.2).

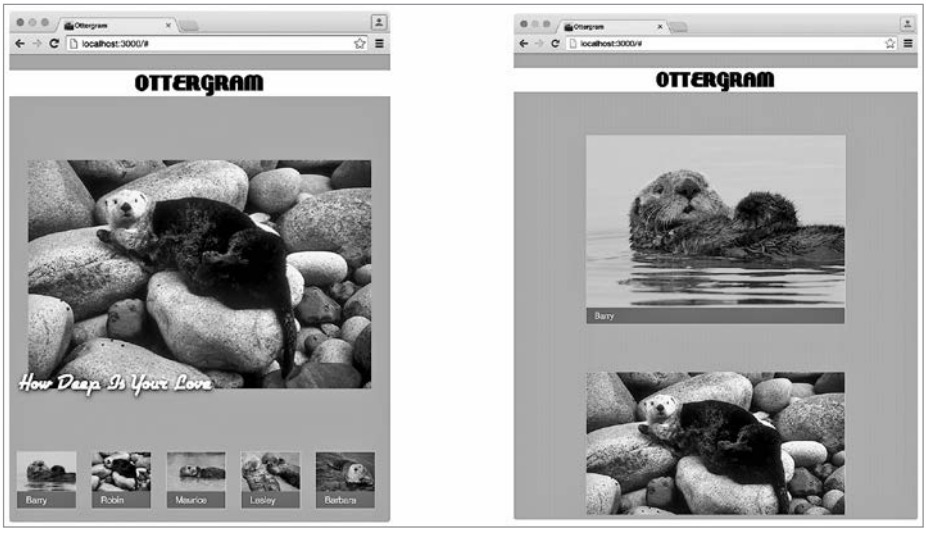


Рис. 7.2. Увеличенное изображение: видимое и скрытое

Чтобы этого добиться, необходимо уметь применять к элементам с классами `.thumbnail-list` и `.detail-image-container` стили в зависимости от условия, которое будет время от времени меняться при работе сайта. Сделать это можно, создав новые селекторы классов — `.thumbnail-list-no-detail` и `.hidden-detail-image-container` — и добавив эти классы в целевые элементы с помощью JavaScript. Однако такой подход не самый эффективный.

То же событие, в результате которого увеличенное изображение будет скрыто, должно *одновременно* приводить к изменению местоположения списка миниатюр. Это единое событие. Добавление классов в элементы `<ul>` и `<div>` по отдельности не отражает этого.

Лучший подход — воспользоваться JavaScript для добавления *единого* селектора класса, воздействующего на макет в целом. Тогда в дальнейшем мы сможем выбирать элементы с классами `.thumbnail-list` и `.detail-image-container`, если они будут потомками этого нового селектора.

Мы собираемся динамически добавлять имя класса в элемент `<body>` для скрытия увеличенного изображения и увеличения миниатюр, после чего динамически удалять имя класса для возвращения к текущим стилям (рис. 7.3).

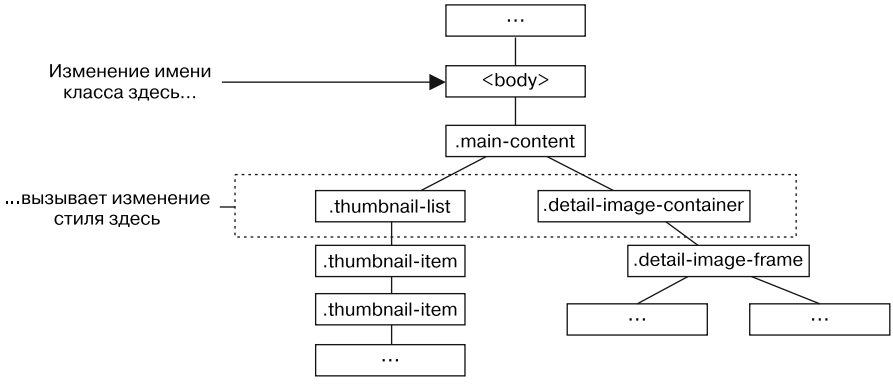


Рис. 7.3. Изменение стилей потомков при смене класса на предка

Этот метод в двух моментах схож с тем, который мы применяли для медиазапросов. Во-первых, в нем используются стили, становящиеся активными, когда предок соответствует определенному условию. Для медиазапросов предком была область просмотра, а условием — минимальная ширина. В новом коде предком будет любой элемент, единый для целевых элементов, а условием — конкретное имя класса предка.

Второе сходство — необходимость размещать условные стили *после* остальных объявлений для затрагиваемых ими элементов в таблице стилей (поскольку эти условные стили должны при использовании перекрывать предыдущие объявления).

Мы разобьем дальнейшие действия на три шага.

1. Описать в CSS стили, создающие необходимый нам визуальный эффект. Кроме того, протестировать наши стили в DevTools.
2. Написать функции JavaScript для добавления и удаления имени класса для элемента `<body>`.
3. Добавить прослушиватель события для запуска функции JavaScript.

Создание стилей для скрытия увеличенного изображения

Чтобы скрыть элемент с классом `.detail-image-container`, добавим объявление, задающее `display: none` для этого элемента. Свойство `display: none` сообщает браузеру, что элемент не нужно визуализировать.

Мы назовем класс, который будем динамически добавлять в элемент `<body>`, `hidden-detail`. Следовательно, нам нужно использовать `display: none` для элемента с классом `.detail-image-container` только тогда, когда он является потомком элемента с классом `.hidden-detail`.

Добавим стиль для скрытия увеличенного изображения в файл `styles.css`:

```
...
.detail-image-title {
  ...
}

.hidden-detail .detail-image-container {
  display: none;
}

@media all and (min-width: 768px) {
  ...
}
```

Теперь подумаем о том, как будет выглядеть наш элемент с классом `.thumbnail-list`. При нынешних стилях это будет столбец вдоль левой стороны на широких экранах и горизонтальная строка сверху более узких экранов. При скрытом увеличенном изображении лучший вариант — столбец по центру (независимо от размера экрана).

Добавляем в файл `styles.css` стили для элементов с классами `.thumbnail-list` и `.thumbnail-item` в тех случаях, когда они являются потомками `.hidden-detail`.

```
...
.hidden-detail .detail-image-container {
  display: none;
}

.hidden-detail .thumbnail-list {
  flex-direction: column;
  align-items: center;
}

.hidden-detail .thumbnail-item {
  max-width: 80%;
}

@media all and (min-width: 768px) {
  ...
}
```

Теперь элементы с классом `.thumbnail-list` будут всегда отображаться в виде столбца при скрытом элементе с классом `.detail-image-container`.

Мы также добавили объявление, которое при скрытом увеличенном изображении устанавливает ширину элементов с классом `.thumbnail-item` равной `max-width: 80%`. Это объявление перекроет заданные где-либо еще для элементов с классом `.thumbnail-items` стили `max-width`, поэтому оно окажется в приоритете.

Если элементы с классами `.detail-image-container`, `.thumbnail-list` и `.thumbnail-item` вложены где-то внутри элемента с классом `hidden-detail`, будут задействоваться эти новые стили.

Обратите внимание, что мы вставили их *перед* нашими медиазапросами. Как вы уже знаете, порядок кода CSS важен, ведь стили, находящиеся дальше в файле, перекрывают находящиеся раньше. В целом для одного и того же селектора браузер использует стили, которые встретились ему позднее. В данном случае, однако, наши новые стили используют более приоритетные селекторы, чем те, которые были у нас в медиазапросах, и приоритетность оказывается важнее их новизны.

В общем, лучше размещать медиазапросы в конце файла. Медиазапросы обычно используют селекторы из существующих стилей, а размещение их в конце файла гарантирует перекрытие этих стилей. Кроме того, оно облегчит поиск медиазапросов (их всегда нужно будет искать в конце файла).

Сохраните файл. Прежде чем писать код JavaScript на основании добавленных нами стилей, неплохо было бы их протестировать. Запустите утилиту browser-sync (с помощью команды `browser-sync start --server --browser "Google Chrome" --files "*.html, stylesheets/*.css, scripts/*.js"`) и откройте DevTools. На панели элементов щелкните кнопкой мыши с нажатой клавишей Control (правой кнопкой мыши — в Windows) на элементе `<body>` и выберите в появившемся меню пункт Add Attribute (Добавить атрибут) (рис. 7.4).

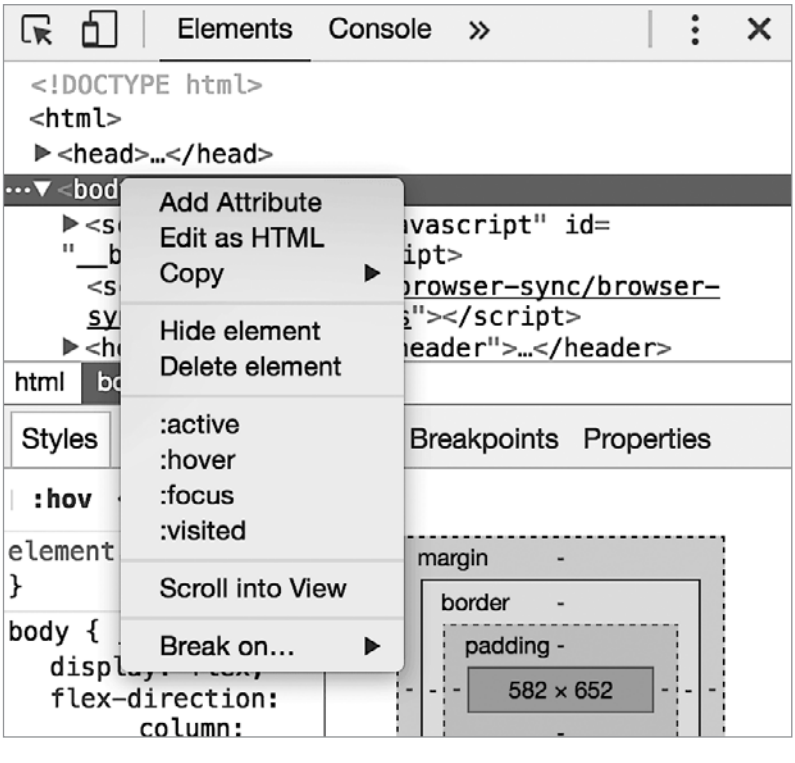


Рис. 7.4. Контекстное меню, с помощью которого можно добавить атрибут

DevTools предоставляет пустое место внутри тега `<body>` для набора текста. Введите `class="hidden-detail"` и нажмите Enter (рис. 7.5).

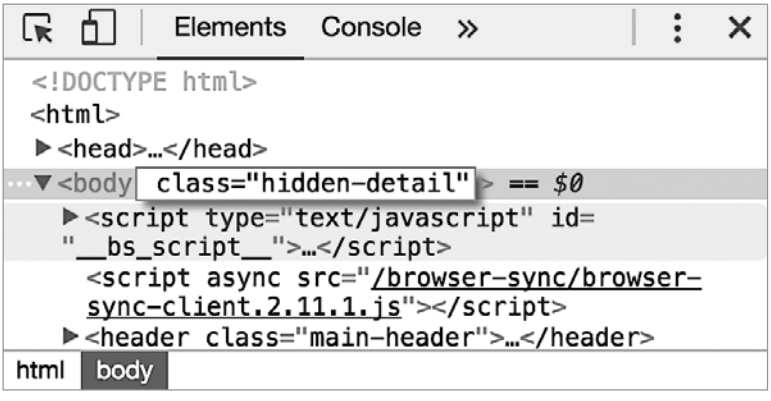


Рис. 7.5. Добавление атрибута класса hidden-detail

После того как вы добавите на панели DevTools класс `hidden-detail` в элемент `<body>`, увеличенное изображение исчезнет и миниатюры станут гораздо крупнее — именно так, как мы и задумывали (рис. 7.6).

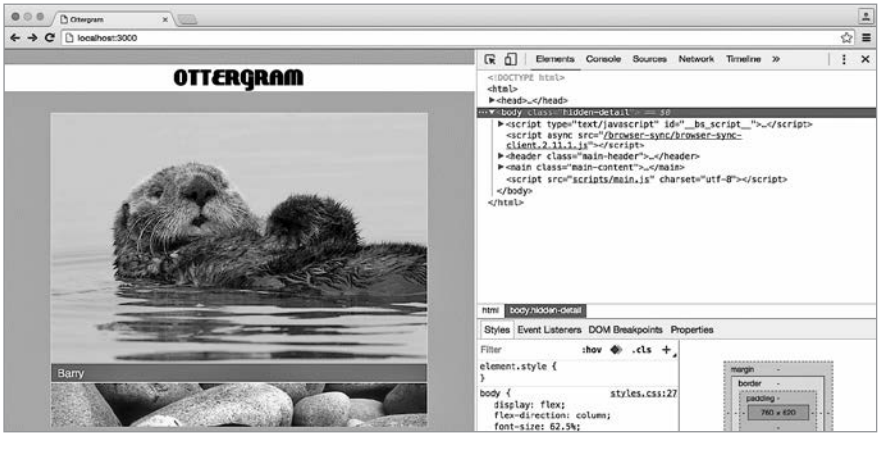


Рис. 7.6. Изменение макета после применения класса hidden-detail

Написание кода JavaScript для скрытия увеличенного изображения

Далее напишем JavaScript-код, меняющий состояние класса `hidden-detail` для элемента `<body>`.

Добавьте в файл `main.js` переменную `HIDDEN_DETAIL_CLASS`.

```
var DETAIL_IMAGE_SELECTOR = '[data-image-role="target"]';
var DETAIL_TITLE_SELECTOR = '[data-image-role="title"]';
var THUMBNAIL_LINK_SELECTOR = '[data-image-role="trigger"]';
var HIDDEN_DETAIL_CLASS = 'hidden-detail';
...
```

Теперь напишем в файле `main.js` новую функцию под названием `hideDetails`. Ее задача — добавление имени класса в элемент `<body>`. Для работы с именем класса воспользуемся методом DOM `classList.add`.

```
...
function getThumbnailsArray() {
  ...
}

function hideDetails() {
  'use strict';
  document.body.classList.add(HIDDEN_DETAIL_CLASS);
}

function initializeEvents() {
  ...
}
...
```

Мы обратились к элементу `<body>` с помощью свойства `document.body`. Этот элемент DOM соответствует тегу `<body>` в нашей разметке. Как и все элементы DOM, он предоставляет удобный способ работы с его именами классов.

Мы также вызвали метод `add` свойства `document.body`, чтобы добавить в `<body>` класс `hidden-detail`.

Прослушивание на предмет события нажатия клавиши

Теперь нам нужен способ инициировать скрытие увеличенного изображения. Как и ранее, мы воспользуемся прослушивателем события, но на этот раз наш прослушиватель события будет «слушать» нажатия клавиши вместо щелчка кнопкой мыши.

Мы используем термин «нажатие клавиши» (keypress) для обозначения процесса, состоящего из нажатия и отпускания клавиши, но этот простой процесс на самом деле вызывает срабатывание множества событий. Когда сначала клавиша нажимается, посылается событие `keydown`. Если это клавиша с символом (а не клавиша-модификатор наподобие Shift), то посылается также событие `keypress`. Когда клавишу отпускают, посылается событие `keyup`.

Для Ottergram эти различия несущественны. Мы будем использовать событие `keyup`.

Добавьте в файл `main.js` функцию под названием `addKeyPressHandler`, вызывающую метод `document.body.addEventListener` с передачей ему строки `'keyup'` и анонимной функции, объявляющей параметр `event`. Внутри тела этой анонимной функции не забываем вызвать метод `preventDefault` для `event`, а затем вывести в консоль (`console.log`) `keyCode` для `event`.

```
...
function hideDetails() {
  ...
}

function addKeyPressHandler() {
  'use strict';
  document.body.addEventListener('keyup', function (event) {
    event.preventDefault();
    console.log(event.keyCode);
  });
}

function initializeEvents() {
  ...
}
...
```

У всех событий нажатий клавиш есть свойство `keyCode`, соответствующее вызвавшей срабатывание события клавише. `keyCode` — целочисленное значение, например 13 для Enter, 32 для пробела и 38 — для стрелки вверх.

Поменяем в файле `main.js` функцию `initializeEvents` таким образом, чтобы она вызывала `addKeyPressHandler`. Это необходимо сделать, чтобы элемент `<body>` мог прослушивать на предмет событий клавиатуры при загрузке страницы.

```
...
function initializeEvents() {
  'use strict';
  var thumbnails = getThumbnailsArray();
  thumbnails.forEach(addThumbClickHandler);
  addKeyPressHandler();
}

initializeEvents();
```

Сохраните изменения и вернитесь в браузер. Убедитесь, что консоль видима, после чего щелкните на странице, чтобы фокус не оказался на DevTools (в противном случае прослушиватель события не сработает). Теперь нажмите какие-нибудь клавиши на клавиатуре. Вы увидите выводимые в консоль числа (рис. 7.7).

Мы хотели бы скрывать увеличенное изображение при нажатии клавиши Esc, а не любой клавиши. Если нажать Esc, то можно видеть, что значение соответствующего свойства `event.keyCode` равно 27. Воспользуемся этим, чтобы сделать прослушиватель события более приоритетным.

Рис. 7.7. Журналирование свойства keyCode в консоли

Добавляем вверху файла `main.js` переменную для значения, соответствующего клавише Esc:

```
var DETAIL_IMAGE_SELECTOR = '[data-image-role="target"]';
var DETAIL_TITLE_SELECTOR = '[data-image-role="title"]';
var THUMBNAIL_LINK_SELECTOR = '[data-image-role="trigger"]';
var HIDDEN_DETAIL_CLASS = 'hidden-detail';
var ESC_KEY = 27;
...
```

Теперь изменим прослушиватель события `keyup` таким образом, чтобы вызывать функцию `hideDetails` в случае равенства значения свойства `event.keyCode` значению переменной `ESC_KEY`:

```
...
function addKeyPressHandler() {
  'use strict';
  document.body.addEventListener('keyup', function (event) {
    event.preventDefault();
    console.log(event.keyCode);
    if (event.keyCode === ESC_KEY) {
      hideDetails();
    }
  });
}
...
```

Мы использовали в этом фрагменте кода *оператор строгого равенства* (`===`) для сравнения значения свойства `event.keyCode` и переменной `ESC_KEY`. При их равенстве вызываем функцию `hideDetails`.

Мы могли вместо этого воспользоваться для сравнения значений *оператором нестрогого равенства*, но лучше обратиться к оператору строгого равенства. Главное различие между ними состоит в том, что оператор нестрогого равенства автоматически преобразует один тип значений в другой, а оператор строгого равенства не выполняет это преобразование. При строгом равенстве, если типы различаются, результат сравнения всегда равен `false`.

Многие разработчики клиентской части называют автоматическое преобразование типов *приведением типов*. Оно выполняется, если необходимо сравнить значения (при использовании оператора равенства), сложить их или выполнить конкатенацию (при работе со строками).

Вследствие этого автоматического преобразования при сложении строки `"27"` с числом `42` не происходит синтаксической ошибки, хотя результат может отличаться от ожидаемого (рис. 7.8).

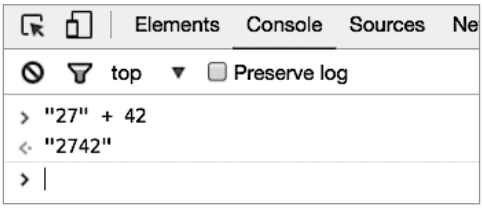


Рис. 7.8. JavaScript автоматически выполняет преобразование типов

Это чрезвычайно важно при работе с вводимыми пользователями данными (этим мы будем заниматься в главе 10).

Сохраните файл `main.js` и проверьте новую функциональность в браузере (рис. 7.9).

Рис. 7.9. Ух ты! Нажатие Esc скрывает увеличенное изображение и название

Вновь отображаем увеличенное изображение

Осталось выполнить одно маленькое, но важное действие: сделать увеличенное изображение снова видимым. Оно станет таким при щелчке на миниатюре.

Мы использовали метод `classList.add` для добавления имени класса в элемент `<body>`. Для удаления этого имени класса при щелчке на миниатюре мы воспользуемся методом `classList.remove`. Добавим в файл `main.js` новую функцию `showDetails`:

```
...
function hideDetails() {
  ...
}

function showDetails() {
  'use strict';
  document.body.classList.remove(HIDDEN_DETAIL_CLASS);
}

function addKeyPressHandler() {
  ...
}
...
```

Теперь добавим вызов функции `showDetails` в нашу функцию `addThumbClickHandler` (добавлять новый прослушиватель события нет необходимости):

```
...
function addThumbClickHandler(thumb) {
  'use strict';
  thumb.addEventListener('click', function (event) {
    event.preventDefault();
    setDetailsFromThumb(thumb);
    showDetails();
  });
}
...
```

Сохраните файл `main.js` и переключитесь на браузер. Оцените новую функциональность: скройте увеличенное изображение, затем щелкните на миниатюре для его возврата (рис. 7.10). Выдры смотрят на вас с одобрением, не правда ли?

Теперь Ottergram умеет динамически адаптировать свой макет в зависимости от области просмотра, используя медиазапросы (в том числе в качестве реакции на действия пользователя).

В настоящий момент изменения макета происходят моментально. В следующем разделе мы сгладим этот переход с помощью CSS.

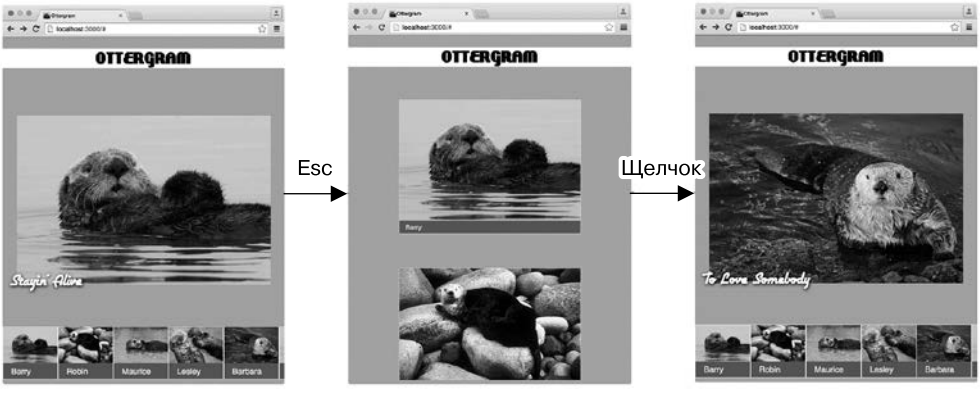


Рис. 7.10. Нажатие клавиши Esc скрывает увеличенное изображение; щелчок на миниатюре его отображает

Изменение состояния с помощью CSS-переходов

CSS-переходы позволяют создавать плавный переход от одного визуального состояния к другому. Это именно то, что нам нужно, чтобы сделать эффект Ottergram по скрытию/отображению более элегантным.

При создании CSS-перехода мы как бы говорим браузеру: «Хотелось бы, чтобы свойства стилей вот этого элемента поменялись на вот такие, причем это изменение длилось столько, сколько я укажу».

Распространенный пример CSS-перехода — всплывающие меню на многих сайтах, таких как предназначенная для маленьких экранов версия сайта `bignerdranch.com`. Щелчок на пункте меню в браузере с узкой областью просмотра приводит к появлению вверху страницы навигационного меню, но это не происходит внезапно. Оно плавно перемещается вниз от заголовка — с визуальной анимацией перехода от начального состояния (скрыто) к конечному (отображено полностью) (рис. 7.11). Щелчок на пункте меню еще раз приводит к плавному перемещению меню навигации обратно до тех пор, пока оно не скроется снова.

Прежде чем создавать эффект перехода для отображения и скрытия увеличенного изображения, необходимо создать более простой переход для миниатюр.

Переходы создают в три этапа.

1. Следует определиться, каким должно быть конечное состояние. Один из способов — добавить объявления CSS для конечного состояния в целевой элемент. Это даст возможность увидеть их в браузере и убедиться, что они выглядят так, как нужно.
2. Переместить объявления из существующего блока объявлений целевого элемента в новый блок объявлений CSS. Возможно, понадобится использовать новый класс для селектора этого нового блока.

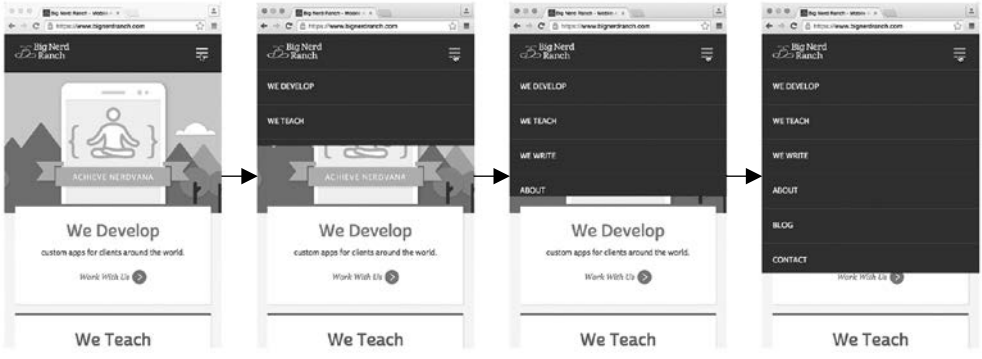


Рис. 7.11. Всплывающее меню навигации на сайте bignerdranch.com

3. Добавить объявление свойства `transition` в целевой элемент. Свойство `transition` сообщает браузеру о необходимости визуальной анимации изменений, начиная от текущих значений CSS до значений CSS конечного состояния, а также то, что переход должен длиться определенный период времени.

Работаем со свойством transform

Наш первый переход будет увеличивать размер миниатюры при задержке над ней указателя мыши (рис. 7.12). Однако мы не будем непосредственно менять стили свойств `width` или `height`. Мы воспользуемся свойством `transform`, которое вносит изменения в форму, размер, местоположение элемента, степень его поворота без нарушения потока элементов вокруг него.

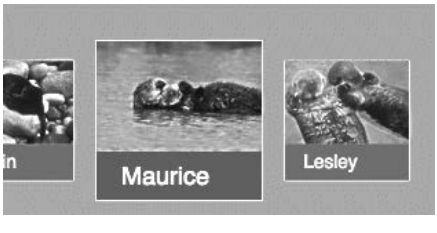


Рис. 7.12. Миниатюра с эффектом изменения масштаба

Целевым элементом для этого перехода является элемент с классом `.thumbnail-item`. Мы начнем с добавления объявления свойства `transform` непосредственно в класс `.thumbnail-item`.

После того как мы проверим, что все работает так, как нам нужно, перенесем это преобразование в новый блок объявления `.thumbnail-item:hover`[1]. Наконец, добавим в класс `.thumbnail-item` объявление свойства `transition`.

Начнем с добавления в файле `styles.css` объявления `transform` в класс `.thumbnail-item`:

[1] От англ. hover — «задерживаться, зависать, нависать». — *Примеч. пер.*

```
...
.thumbnail-item {
  display: inline-block;
  min-width: 120px;
  max-width: 120px;
  border: 1px solid rgb(100%, 100%, 100%);
  border: 1px solid rgba(100%, 100%, 100%, 0.8);

  transform: scale(2.2);
}
...
```

Объявление `transform: scale(2.2);` сообщает браузеру о необходимости отрисовать данный элемент в масштабе 220 % от его исходного размера. Существует множество значений, которые можно использовать для свойства `transform`, включая расширенные 3D-эффекты. В MDN представлен отличный их обзор по адресу developer.mozilla.org/en-US/docs/Web/CSS/transform.

Сохраните изменения и посмотрите на результат в браузере (рис. 7.13).

Рис. 7.13. Чересчур большие миниатюры выдр

Как видите, миниатюры стали намного больше, чем раньше, точнее, слишком велики. Поменяем значение, чтобы они были лишь чуть-чуть больше:

```
...
.thumbnail-item {
  display: inline-block;
  min-width: 120px;
  max-width: 120px;
  border: 1px solid rgb(100%, 100%, 100%);
  border: 1px solid rgba(100%, 100%, 100%, 0.8);

  transform: scale(2.2);
  transform: scale(1.2);
}
...
```

После сохранения вы должны увидеть, что миниатюры выдр лишь чуть-чуть больше своего первоначального размера (рис. 7.14).

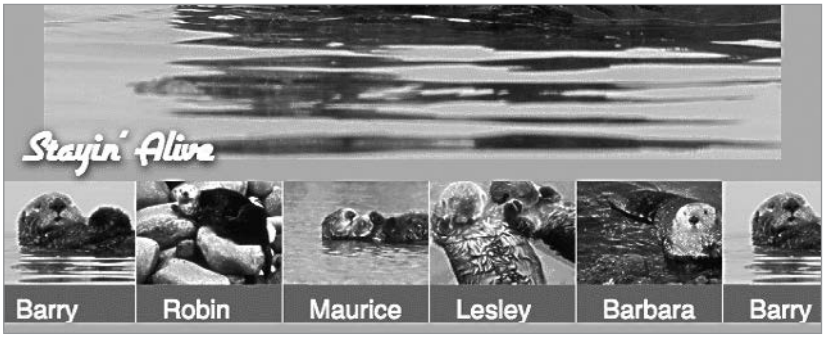


Рис. 7.14. Умеренно большие миниатюры выдр

Масштаб миниатюр вполне подходящий, так что можно перейти к следующему шагу.

Добавление CSS-перехода

Настало время переместить стиль конечного состояния в новое объявление стиля и настроить переход для элемента с классом `.thumbnail-item`.

Когда пользователь задерживает указатель мыши над миниатюрой, она должна увеличить свой масштаб (`scale`) до 120 %. Добавим в файл `styles.css` блок объявления, использующий модификатор `:hover` для обозначения стилей, которые необходимо применять только тогда, когда пользователь задерживает указатель над миниатюрой.

```
...
.thumbnail-item {
  display: inline-block;
  min-width: 120px;
  max-width: 120px;
  border: 1px solid rgb(100%, 100%, 100%);
  border: 1px solid rgba(100%, 100%, 100%, 0.8);

  ~~transform: scale(1.2);~~
}

.thumbnail-item:hover {
  transform: scale(1.2);
}
...
```

Такой модификатор называется *псевдоклассом*. Псевдокласс `:hover` соответствует элементу тогда, когда пользователь держит над ним указатель мыши. Есть несколько ключевых слов для псевдоклассов, чтобы описывать различные состояния элемента. С некоторыми из них мы столкнемся в данной книге в дальнейшем, при работе с формами. Вы можете обратиться к MDN, чтобы узнать о них больше.

Далее делаем из этого изменения переход путем добавления в файл `styles.css` объявления свойства `transition` в класс `.thumbnail-item`. Нам нужно указать, какое свойство необходимо анимировать и сколько времени должна длиться анимация.

```
...
.thumbnail-item {
  display: inline-block;
  min-width: 120px;
  max-width: 120px;
  border: 1px solid rgb(100%, 100%, 100%);
  border: 1px solid rgba(100%, 100%, 100%, 0.8);

  transition: transform 133ms;
}

.thumbnail-item:hover {
  transform: scale(1.2);
}
...
```

Мы задали переход `transition` для свойства `transform`. Это сообщает браузеру о необходимости анимировать изменение, но только для свойства `transform`. Мы также указали, что переход должен длиться 133 миллисекунды.

Сохраните новый переход и оцените его в деле. Вы должны увидеть, как каждая миниатюра увеличивается, когда вы задерживаете над ней указатель. Когда вы убираете указатель, переход выполняется в обратном направлении — и миниатюра возвращается к первоначальному размеру (рис. 7.15).

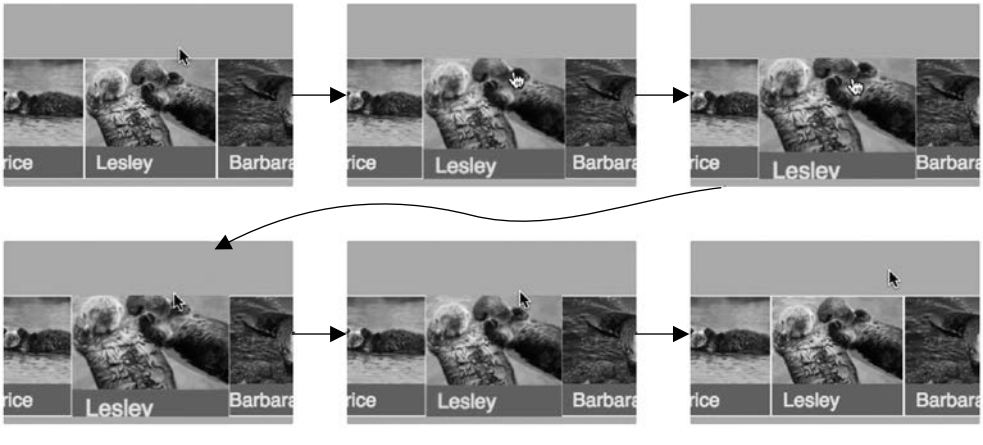


Рис. 7.15. При зависании указателя мыши над миниатюрой происходит переход, инвертируемый при движении указателя мыши дальше

DevTools предоставляет удобную возможность для тестирования псевдоклассов. Перейдите на панель элементов и разворачивайте теги до тех пор, пока не отобра-

зится один из тегов <li>. Щелкните на этом теге, чтобы он был выделен, и слева вы увидите овал. Щелкните на нем и выберите в появившемся контекстном меню пункт :hover из списка псевдоклассов (рис. 7.16).

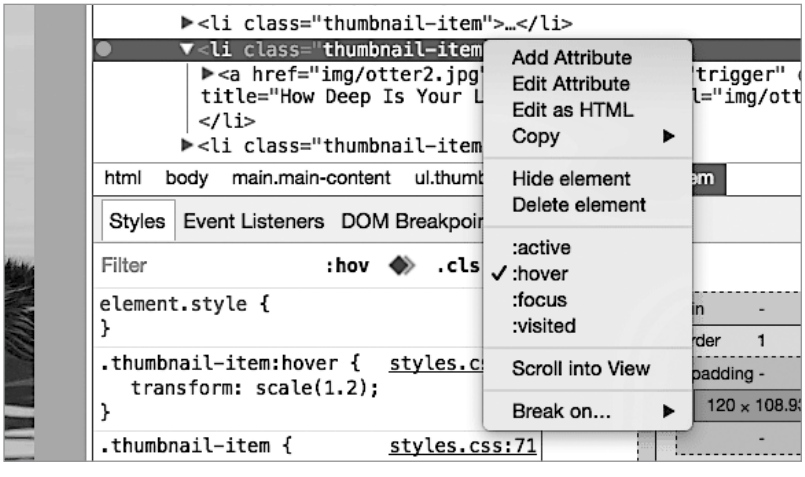


Рис. 7.16. Включение псевдокласса на панели элементов

Слева от тега <li> на панели элементов появится оранжевый круг, извещая вас, что один из псевдоклассов стал активным с помощью DevTools. Соответствующая миниатюра останется в состоянии :hover, даже если вы проведете над ней указателем мыши, а затем уберете его.

Опять откройте контекстное меню, щелкнув на оранжевом круге, и отключите состояние :hover, прежде чем продолжить.

Наш переход неплох, но в нем есть небольшая ошибка. В настоящий момент эффект «нависания» указателя приводит к тому, что определенные части миниатюры обрезаются. Это происходит из-за того, что используемое для элемента с классом .thumbnail-item свойство transform не заставляет свой родительский элемент адаптировать *его* размер. Для решения этой проблемы необходимо добавить небольшие поля в элемент с классом .thumbnail-list. Измените в файле styles.css вертикальные поля для класса .thumbnail-list:

```
...
.thumbnail-list {
  flex: 01 auto;
  order: 2;
  display: flex;
  justify-content: space-between;
  list-style: none;
  ~~padding: 0;~~
  padding: 20px 0;
```

```
  white-space: nowrap;
  overflow-x: auto;
}
...
```

Для свойства `padding` мы воспользовались сокращенной записью. Первое значение `20px` определяет верхнее и нижнее поля, а второе — левое и правое. Внесите аналогичную правку внутри запроса `@media`, но добавьте дополнительные поля по `35px` слева и справа.

```
...
@media all and (min-width: 768px) {
  .main-content {
  ...
  }

  .thumbnail-list {
    flex-direction: column;
    order: 0;
    margin-left: 20px;

    padding: 0 35px;
  }
  ...
```

Сохраните изменения и проверьте результат в браузере. Мы добились нужного эффекта (рис. 7.17).

Рис. 7.17. Дополнительное место для выполнения эффекта задержки указателя в книжной и альбомной ориентациях экрана

Использование временной функции

Наш эффект выглядит отлично! Но ему недостает визуального шика. С помощью CSS-переходов можно не только задать длительность перехода, но и сделать так, чтобы переход осуществлялся с разной скоростью в разные моменты времени.

Существует несколько *функций управления временем* (timing function) (в дальнейшем мы будем называть их *временными функциями*), которые можно использовать

для переходов. По умолчанию выбирается *линейная временная функция*: анимация перехода происходит с одинаковой постоянной скоростью. Остальные временные функции интереснее, они придают переходу ускорение или замедление.

Измените переход в файле `styles.css`, чтобы он использовал временную функцию `ease-in-out`. В итоге переход замедлится в начале и ускорится в середине.

```
...
.thumbnail-item {
  display: inline-block;
  min-width: 120px;
  max-width: 120px;
  border: 1px solid rgb(100%, 100%, 100%);
  border: 1px solid rgba(100%, 100%, 100%, 0.8);

  transition: transform 133ms ease-in-out;
}
...
```

Сохраните изменения и задержите указатель над одной из миниатюр. Эффект незначителен, но заметен.

Существует множество доступных для использования временных функций. Их список можно посмотреть в MDN по адресу `developer.mozilla.org/en-US/docs/Web/CSS/transition-timing-function`.

Наше свойство `transition` использует одно и то же значение длительности и временную функцию для перехода и *к* конечному состоянию, и *от* конечного состояния. Это не обязательно должно оставаться так — можно указать различные значения в зависимости от направления перехода. Если задать свойство `transition` в объявлении как начального состояния, так и конечного состояния, браузер будет использовать то объявление, в направлении которого происходит переход.

Возможно, лучше увидеть это в действии. Для быстрой демонстрации добавьте в файл `styles.css` объявление свойства `transition` в `.thumbnail-item:hover` (вы удалите его после проверки в браузере).

```
...
.thumbnail-item:hover {
  transform: scale(1.2);
  transition: transform 1000ms ease-in;
}
...
```

Сохраните изменения и еще раз задержите указатель над одной из миниатюр в браузере. Эффект масштабирования будет сильно замедленным и займет почти секунду. Это происходит из-за того, что он использует объявленное для элемента с классом `.thumbnail-item:hover` значение. Уберите указатель с миниатюры. На этот раз переход займет 133 миллисекунды — это значение объявлено для класса `.thumbnail-item`.

Не забудьте удалить объявление свойства `transition` из `.thumbnail-item:hover`, прежде чем продолжить.

```
...
.thumbnail-item:hover {
  transform: scale(1.2);
  transition: transform 1000ms ease-in;
}
...
```

Переход при изменении класса

Благодаря нашему второму переходу элемент с классом `.detail-image-frame` будет выглядеть так, будто приближается издалека.

Вместо того чтобы использовать для запуска перехода селектор псевдокласса, на этот раз мы будем добавлять и удалять имена классов с помощью JavaScript. Почему так? Потому, что не существует псевдокласса, соответствующего событию щелчка кнопкой мыши. Использование JavaScript предоставляет нам гораздо более широкий контроль над тем, как и когда будут происходить изменения пользовательского интерфейса.

Кроме того, мы зададим различную длительность для начала и окончания перехода. В результате при щелчке на миниатюре для увеличенного изображения будет использовано соответствующее изображение выдры, которое мгновенно уменьшится до размеров маленькой точки в центре области увеличенного изображения, а затем перейдет к полному размеру (рис. 7.18).

Рис. 7.18. Щелчок на миниатюре приводит к ее масштабированию от очень маленького до полного размера

Начнем с добавления в файл `styles.css` описания стиля для нового класса `is-tiny`.

```
...
.detail-image-frame {
  ...
}

.is-tiny {
  transform: scale(0.001);
```

```
  transition: transform 0ms;
}

.detail-image {
  ...
```

Мы добавили два стиля для класса `.is-tiny`. Первый масштабирует (`scale`) элемент до размера, составляющего лишь крошечную часть первоначального. Второй определяет, что любой переход (`transition`) для свойства `transform` должен длиться 0 миллисекунд и применять стиль практически мгновенно. Другими словами, переход увеличенного изображения к стилям класса `.is-tiny` фактически будет отсутствовать. Поскольку он длится 0 миллисекунд, необходимости задавать временную функцию нет.

Далее добавим еще одно объявление `transition` длительностью 333 миллисекунды. Это значение будет использоваться при переходе от класса `.is-tiny`, что приведет к масштабированию увеличенного изображения до нормального размера за треть секунды. Добавьте это объявление `transition` в класс `.detail-image-frame` в файле `styles.css`.

```
...
.detail-image-frame {
  position: relative;
  text-align: center;

  transition: transform 333ms;
}
...
```

Сохраните файл `styles.css`, прежде чем двигаться дальше.

Запуск переходов с помощью JavaScript

Теперь, когда наши стили переходов готовы, необходимо обеспечить их работу с помощью JavaScript. Для этого добавим атрибут данных в элемент с классом `.detail-image-frame` в файле `index.html`.

```
...
      <div class="detail-image-container">
        <div class="detail-image-frame" data-image-role="frame">
        <img class="detail-image" data-image-role="target"
             src="img/otter1.jpg" alt="">
        <span class="detail-image-title" data-image-role="title">Stayin'
              Alive</span>
        </div>
      </div>
...
```

Сохраните файл `index.html`. Теперь нам нужно добавить в файл `main.js` переменные для класса `.is-tiny` и селектора `data-image-role="frame"`, после чего мы

исправим функцию `showDetails` для выполнения изменений имени класса, которые обеспечат срабатывание перехода.

Начнем с переменных. Добавьте переменную `DETAIL_FRAME_SELECTOR` для строки селектора `'[data-image-role="frame"]'`. Кроме того, добавьте переменную `TINY_EFFECT_CLASS` для имени класса `is-tiny`:

```
var DETAIL_IMAGE_SELECTOR = '[data-image-role="target"]';
var DETAIL_TITLE_SELECTOR = '[data-image-role="title"]';
var DETAIL_FRAME_SELECTOR = '[data-image-role="frame"]';
var THUMBNAIL_LINK_SELECTOR = '[data-image-role="trigger"]';
var HIDDEN_DETAIL_CLASS = 'hidden-detail';
var TINY_EFFECT_CLASS = 'is-tiny';
var ESC_KEY = 27;
...
```

Порядок переменных не обязательно должен быть таким (для браузера это значения не имеет). Но согласитесь, что все должно быть упорядочено. В файле `main.js` за сгруппированными вместе всеми переменными селекторов следуют переменные классов, а за ними — числовое значение для клавиши Esc.

Теперь исправим функцию `showDetails` в файле `main.js`, чтобы получить в ней ссылку на элемент `[data-image-role="frame"]`. Для запуска перехода понадобится добавить и удалить класс `TINY_EFFECT_CLASS`.

```
...
function showDetails() {
  'use strict';
  var frame = document.querySelector(DETAIL_FRAME_SELECTOR);
  document.body.classList.remove(HIDDEN_DETAIL_CLASS);
  frame.classList.add(TINY_EFFECT_CLASS);
  frame.classList.remove(TINY_EFFECT_CLASS);
}
...
```

Если вы сохраните изменения и заглянете в браузер, то выполнения перехода не увидите. Почему? Потому что класс `TINY_EFFECT_CLASS` добавляется, а затем сразу же удаляется. В итоге нет изменения класса, которое можно было бы визуализировать. Так функционирует выполняемая браузером оптимизация.

Необходимо добавить небольшую задержку перед удалением класса `TINY_EFFECT_CLASS`. В JavaScript, однако, нет встроенной функции задержки или ожидания, как в некоторых языках программирования. Настало время для небольшого обходного маневра!

Воспользуемся методом `setTimeout`, принимающим на входе функцию и задержку (в миллисекундах). После задержки функция становится в очередь на выполнение браузером.

Добавьте вызов `setTimeout` после вызова метода `frame.classList.add` в файле `main.js`. Передайте ей два аргумента: функцию со списком шагов, которые нужно

выполнить, и количество миллисекунд ожидания перед вызовом первого аргумента. В нем будет выполняться всего один шаг — удаление класса `TINY_EFFECT_CLASS`.

```
...
function showDetails() {
  'use strict';
  var frame = document.querySelector(DETAIL_FRAME_SELECTOR);
  document.body.classList.remove(HIDDEN_DETAIL_CLASS);
  frame.classList.add(TINY_EFFECT_CLASS);
  setTimeout(function () {
    frame.classList.remove(TINY_EFFECT_CLASS);
  }, 50);
}
...
```

Посмотрим повнимательнее, что делает этот код. Во-первых, он добавляет класс `.is-tiny` в элемент `frame`. При этом применяется наше преобразование `transform: scale(0.001)`.

Затем браузеру указывается подождать 50 миллисекунд, после чего добавить в очередь выполнения анонимную функцию. Выполнение функции `showDetails` завершается. Через 50 миллисекунд анонимная функция становится в очередь на выполнение (фактически она становится в очередь на процессорное время, ожидая выполнения всех остальных функций, уже находящихся в очереди).

Запустившись, эта анонимная функция удаляет `TINY_EFFECT_CLASS` из списка класса `frame`. Это приводит к выполнению перехода `transform` в течение 333 миллисекунд, в результате чего `frame` увеличивается до своего обычного размера.

Сохраните изменения и полюбуйтесь результатами. Пощелкайте на миниатюрах и насладитесь тем, как эти загадочные выдры вырастают на глазах.

Пользовательские временные функции

А теперь вишенка на торте Ottergram! Существует возможность создавать собственные временные функции для переходов, а не только ограничиваться встроенными.

Можно нарисовать графики временных функций для визуализации выполнения перехода с течением времени. Графики встроенных временных функций (с сайта cubic-bezier.com) показаны на рис. 7.19.

Формы на этих графиках известны как *кубические кривые Безье*. Линии на графиках описывают поведение анимации с течением времени. Они определяются четырьмя точками. Вы можете создать пользовательские переходы, задав четыре точки, определяющие кривую. Попробуйте использовать `cubic-bezier` в качестве части объявления свойства `transition` для класса `.detail-image-frame` в файле `styles.css`.

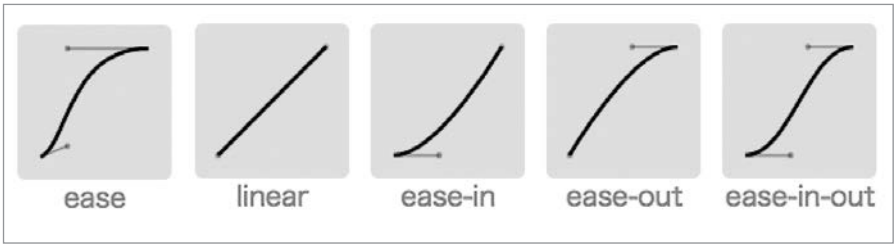


Рис. 7.19. Встроенные временные функции

```
...
.detail-image-frame {
  position: relative;
  text-align: center;
  transition: transform 333ms cubic-bezier(1,.06,.28,1);
}
...
```

Сохраните изменения и щелкните на какой-нибудь миниатюре в браузере, чтобы посмотреть, чем отличается переход.

Благодаря Лиа Веру и ее сайту cubic-bezier.com создание пользовательских временных функций не представляет сложностей (рис. 7.20).

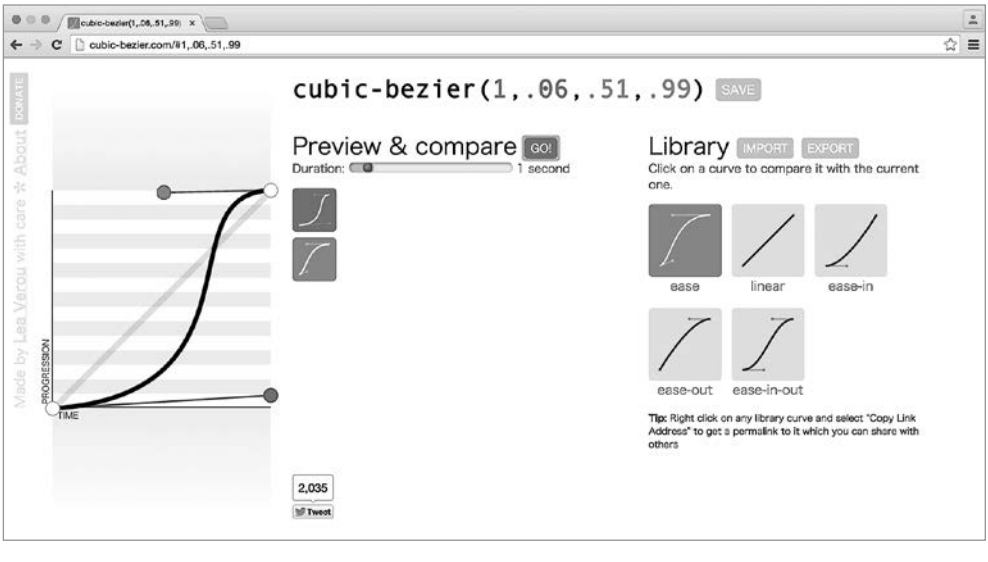


Рис. 7.20. Создание пользовательской временной функции с помощью cubic-bezier.com

Слева — кривая с красным и синим перетаскиваемыми маркерами. Кривая — график, отражающий то, какая часть перехода произошла за данный промежуток вре-

мени. Щелкните и перетащите маркеры для изменения кривой. При ее изменении меняются также и десятичные значения вверху страницы.

Справа находятся встроенные временные функции: `ease`, `linear`, `ease-in`, `ease-out` и `ease-in-out`. Сначала щелкните на одной из них, а затем нажмите кнопку GO! рядом с заголовком Preview & compare (Предварительный просмотр и сравнение). Значки, представляющие две временные функции — пользовательскую `cubic-bezier` и встроенную функцию, — будут анимированы, что даст вам возможность увидеть свою пользовательскую временную функцию в действии и сравнить ее со встроенной.

Создайте пользовательскую временную функцию и, когда она вас полностью будет устраивать, скопируйте и вставьте значения с сайта в наш код в файле `styles.css`:

```
...
.detail-image-frame {
  position: relative;
  text-align: center;
  transition: transform 333ms cubic-bezier(поместите сюда ваши значения);
}
...
```

Поздравляем! Ottergram полнофункционален! Сохраните файл и любуйтесь плодом своего труда. Мы превратили Ottergram из простой статической веб-страницы в интерактивную адаптивную страницу с динамическими визуальными эффектами.

Мы прошли долгий путь и надеемся, что вам доставило удовольствие изучение основ разработки клиентской части. Пришло время попрощаться с выдрами, ведь в следующей главе нам предстоит новый проект.

Для наиболее любознательных: правила приведения типов

Как упоминалось в главе 6, JavaScript был изначально создан для того, чтобы не только профессиональные программисты могли делать веб-страницы интерактивными. Считалось, что «простые смертные» не должны волноваться о том, является ли значение числом, объектом или бананом (это была шутка — типа «банан» в JavaScript нет).

Один из способов достижения этого — приведение типов. С помощью приведения типов можно сравнивать два значения независимо от их типов, используя оператор `==` и выполняя конкатенацию двух значений посредством оператора `+`. JavaScript сам находит способ, как добиться этого, даже если ему приходится делать при этом что-то немного странное, например преобразовывать строку `"2"` в число `2`.

Такое положение дел озадачило как программистов, так и непрограммистов. Большинство программистов согласны, что лучше использовать строгое сравнение

с помощью оператора `===`. Однако правила приведения типов четко определены в JavaScript — и не помешает их знать.

Допустим, нам нужно сравнить две переменные: `x == y`. Если у них одинаковые тип и значение, результатом сравнения будет булево значение `true`. Существует только одно исключение из этого правила: если какая-либо из величин `x` или `y` равна `NaN` (константа языка JavaScript, означающая «не является числом»), то результатом будет `false`.

Однако если `x` и `y` разного типа, все усложняется. Вот некоторые правила, используемые JavaScript в подобном случае:

- результатом следующих сравнений будет `true`: `null == undefined` и `undefined == null`;
- при сравнении строки и числа сначала строка преобразуется к своему числовому эквиваленту. Это значит, что `"3" == 3` равно `true`, а `"dog" == 20` равно `false`;
- при сравнении булева значения со значением другого типа сначала булево значение преобразуется в число: `true` преобразуется в число `1`, а `false` — в число `0`. Это значит, что `false == 0` равно `true` и `true == 1` тоже равно `true`;
- наконец, при сравнении строки или числа с объектом сначала выполняется попытка преобразовать объект в простое значение. Если это преобразование не удается, выполняется попытка преобразовать объект в строку.

Для получения более подробной информации почитайте обсуждение MDN по адресу developer.mozilla.org/en-US/docs/Web/JavaScript/Equality_comparisons_and_sameness.

Часть II. Модули, объекты и формы

8 Модули, объекты и методы

На протяжении следующих семи глав мы будем создавать приложение, работающее по принципу корзины для виртуальных покупок. Оно будет называться CoffeeRun и предназначаться для управления заказами кофе в автокафе. CoffeeRun будет состоять из трех слоев кода: UI, внутренней логики и связи с сервером.

В этой главе мы создадим внутреннюю логику и поработаем с ней с помощью консоли DevTools, как показано на рис. 8.1.

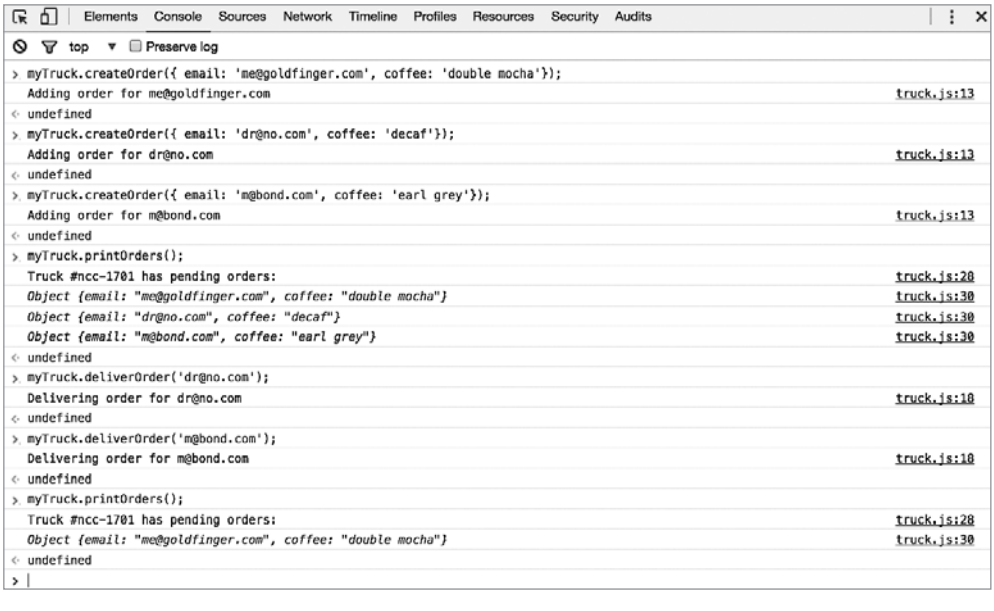


Рис. 8.1. Содержимое CoffeeRun

Модули

Приложение CoffeeRun сложнее, чем Ottergram, так что важно правильно организовать код, чтобы его было легче отлаживать и расширять. CoffeeRun будет разбит на компоненты, схематично изображенные на рис. 8.2.

Рис. 8.2. Обзор компонентов и взаимодействий CoffeeRun

У каждой части приложения будет своя сфера ответственности. Код внутренней логики нашего приложения будет работать с данными, код UI — отвечать за обработку событий и манипуляции DOM (аналогично коду Ottergram), код связи с сервером будет взаимодействовать с удаленным сервером, сохраняя и извлекая данные по сети.

Язык программирования JavaScript был создан для написания очень маленьких сценариев, вносящих в приложение крошечные кусочки интерактивности, а не для создания сложных приложений. Приложение CoffeeRun не очень сложное, но оно выходит за пределы того, что можно выполнить в одном файле сценария.

Ради правильного разделения нашего кода создадим три файла кода JavaScript для подмножеств функциональности приложения. У нас будут отдельные файлы для внутренней логики, пользовательского интерфейса и кода связи с сервером. Мы добьемся такого разделения путем написания кода в отдельных компонентах программы — *модулях*.

Способ группировки кода целиком выбирается по усмотрению разработчика. Чаще всего код группируется около каких-либо абстрактных понятий, например «запасы» или «меню». Говоря в терминах кода, модули группируют вместе взаимосвязанные функции. Некоторые из этих функций будут доступны извне, другие будут использоваться только внутри модуля.

Представьте себе ресторан. На кухне полно посуды и разных ингредиентов для приготовления блюд, но посетители видят только меню из нескольких позиций. Это меню — пользовательский интерфейс к приготавливающему еду модулю под названием «кухня».

Аналогично если в ресторане есть бар, то приспособления для приготовления напитков и ингредиенты являются внутренними элементами; у посетителя есть доступ только к карте напитков, являющейся их интерфейсом к делающему напитки модулю под названием «бар».

Как посетитель вы не можете одолжить на кухне нож мясника или воспользоваться блендером. Вы не можете взять дополнительный кусочек масла из холодильника или плеснуть себе еще виски в коктейль. Вы ограничены использованием предоставляемых кухней и баром интерфейсов.

Аналогично часть функциональности каждого из модулей приложения CoffeeRun будет оставаться закрытой. Другая часть его функциональности будет общедоступной, чтобы другие модули могли с ним взаимодействовать.

В CoffeeRun мы продолжим работать с ES5 — наиболее поддерживаемой версией JavaScript на момент написания данной книги (в следующем проекте, Chattrbox, мы будем использовать последнюю версию — ES6). В ES5 отсутствует формальный способ организации кода по самостоятельным модулям, но достичь такой организации можно, поместив взаимосвязанный код (переменные и функции) в тело функции.

Паттерн модуля

Код можно организовать с помощью функций, но распространенной практикой является использование несколько отличающейся разновидности обычной функции для этой цели. Прежде чем начать наш новый проект, взглянем на *паттерн модуля (module pattern)* организации кода, чтобы понять, как можно переписать функцию из Ottergram в виде модуля.

Вот код простейшего модуля:

```
(function () {
  'use strict';
  // Здесь находится код
})();
```

Если вы видите этот паттерн в первый раз, то, вероятно, он кажется вам довольно странным. Он известен под названием *немедленно выполняемого функционального выражения* (IIFE), и лучше начать его изучение с внутреннего уровня.

Основная его часть — анонимная функция:

```
function () {
  'use strict';
  // Здесь находится код
}
```

Вы уже имели дело с анонимными функциями в Ottergram, так что все должно выглядеть привычно.

Однако здесь анонимная функция заключена в скобки:

```
(function () {
  'use strict';
```

```
  // Здесь находится код
})
```

Эти скобки очень важны, поскольку как бы говорят браузеру: «Пожалуйста, не интерпретируй этот код как объявление функции».

Браузер видит скобки и отвечает: «Хорошо. Я понял, что вижу анонимную функцию. Я воздержусь от выполнения с ней каких-либо действий».

Большую часть времени мы используем анонимные функции, передавая их в качестве аргумента. В данном случае мы вызываем ее немедленно. Это осуществляется с помощью пустой пары скобок:

```
(function () {
  'use strict';
  // Здесь находится код
})()
```

Когда браузер обнаруживает пустые скобки, он понимает, что мы хотим вызвать находящийся перед ними код, и говорит: «Понятно, мне досталась анонимная функция, которую я могу выполнить!»

Вы можете подумать: «Это не только странно, но и бесполезно». На самом деле мы уже писали код, который работает аналогичным образом. Вспомните функцию `initializeEvents` из приложения Ottergram. Мы вызвали ее сразу же после объявления, после чего больше к ней не обращались. Вот соответствующий код для сравнения:

```
function initializeEvents() {                  // Функция объявлена
  'use strict';
  var thumbnails = getThumbnailsArray();
  thumbnails.forEach(addThumbClickHandler);

  addKeyPressHandler();
}

initializeEvents();                            // Функция вызвана
```

Задача этой функции заключалась в том, чтобы сгруппировать вместе определенные действия и выполнить их при загрузке страницы. Вот аналогичный код, переписанный в виде IIFE (менять код Ottergram не нужно, мы просто иллюстрируем идею):

```
~~function initializeEvents() {~~
(function () {
  'use strict';
  var thumbnails = getThumbnailsArray();
  thumbnails.forEach(addThumbClickHandler);
  addKeyPressHandler();
~~}~~
})();

~~initializeEvents();~~
```

IIFE удобны, если нужно выполнить какой-то код однократно, без создания глобальных переменных или функций. Чтобы понять, почему это так важно, рассмотрим написанные нами для приложения Ottergram переменные и функции.

В Ottergram мы создали набор полезных функций, после чего вызывали их по мере необходимости. У этих функций были имена, например `getThumbnailsArray` и `addKeyPressHandler`. К счастью, эти имена уникальны. Если бы вы попытались объявить две функции с одинаковым именем, вторая бы просто заменила первую.

При объявлении функций или переменных они по умолчанию вносятся в *глобальное пространство имен*, которое представляет собой реестр браузера для всех имен функций и переменных вашей JavaScript-программы наряду со встроенными функциями и переменными. В общем случае *пространство имен* — средство организации кода (код упорядочивается в пространствах имен аналогично тому, как файлы на вашем компьютере упорядочиваются по каталогам).

В CoffeeRun у нас может быть некоторое количество функций с одинаковым именем, например `add` или `addClickHandler`. Вместо добавления их в глобальное пространство имен, где они могут оказаться перекрыты, мы объявляем их внутри функции. Это защищает их от обращения или перезаписи из находящегося вне функции кода.

При группировке кода вместе в модуле может понадобиться сделать часть функциональности — но не всю — доступной для внешнего (по отношению к модулю) мира. Для этого можно воспользоваться тем, что IIFE, как и любые другие функции, могут принимать аргументы.

Изменение объекта с помощью IIFE

IIFE хорошо подходят не только для выполнения готового кода, как в случае функции `initializeEvents` из Ottergram, но и для выполнения кода, вносящего в объект, обычно передаваемый в виде аргумента, изменения и дополнения. Чтобы проиллюстрировать, как это работает, воспользуемся нашей старой знакомой — функцией `initializeEvents`.

Вот версия функции `initializeEvents`, задача которой заключается в изменении вида миниатюр путем добавления обработчика щелчков (нажатий) (для упрощения вызов функции `addKeyPressHandler` был убран):

```
function initializeEvents() {
  'use strict';
  var thumbnails = getThumbnailsArray();
  thumbnails.forEach(addThumbClickHandler);
}

initializeEvents();
```

В этом виде `initializeEvents` меняет массив миниатюр с помощью функции `addThumbClickHandler`. Но она может также получать этот массив в качестве аргу-

мента. Для этого нужно объявить параметр в описании функции. После этого при вызове можно передать массив:

```
function initializeEvents(thumbnails) {
  'use strict';
  var thumbnails = getThumbnailsArray();
  thumbnails.forEach(addThumbClickHandler);
}

var thumbnails = getThumbnailsArray();
initializeEvents(thumbnails);
```

Чтобы переписать этот фрагмент кода в виде IIFE, необходимо убрать имя функции, заключить функцию в скобки и добавить пару пустых скобок для ее вызова:

```
(function initializeEvents(thumbnails) {
  'use strict';
  thumbnails.forEach(addThumbClickHandler);
})();

var thumbnails = getThumbnailsArray();
initializeEvents(thumbnails);
```

Однако нам по-прежнему необходимо передавать массив миниатюр в качестве аргумента. Это можно осуществить путем переноса вызова в функцию `getThumbnailsArray`. Вместо того чтобы присваивать результат переменной, мы будем передавать его нашему IIFE:

```
(function (thumbnails) {
  'use strict';
  thumbnails.forEach(addThumbClickHandler);
})(getThumbnailsArray());

var thumbnails = getThumbnailsArray();
```

При таком варианте кода массив (результат вызова функции `getThumbnailsArray`) передается IIFE. IIFE получает массив и присваивает ему «ярлык» `thumbnails`. Внутри тела IIFE каждая миниатюра в массиве получает прослушиватель события (рис. 8.3).

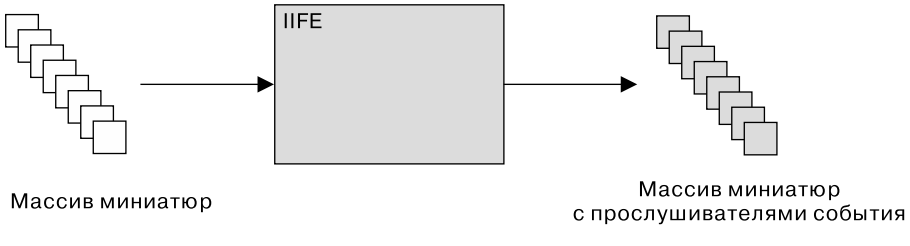


Рис. 8.3. IIFE меняет свои аргументы

IIFE можно передать для изменения все что угодно. Мы будем передавать всем IIFE проекта CoffeeRun объект `window`. Но вместо привязки кода своих модулей непосредственно к глобальному пространству имен они будут привязывать этот код к отдельному свойству `App` *в* глобальном пространстве имен. Каждый модуль CoffeeRun будет находиться в собственном файле и загружаться в браузер с помощью отдельного тега `<script>`. Обзор этого процесса показан на рис. 8.4.

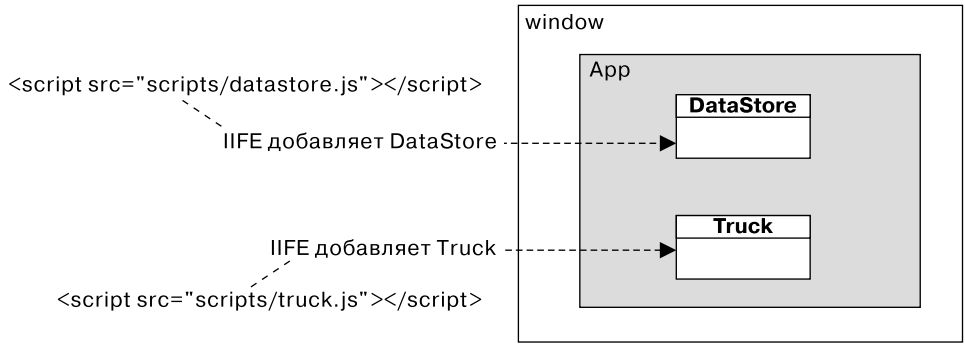


Рис. 8.4. Теги <script> загружают модули, изменяющие свойство window.App

Настройка приложения CoffeeRun

Хватит теории — приступим к практике. Поскольку это новый проект, начнем с создания нового каталога. Откройте текстовый редактор Atom и выберите File ▸ Add Project Folder (Файл ▸ Добавить каталог проекта). Выберите каталог `front-end-dev-book` и щелкните на команде New Folder (Новый каталог). Назовите новый каталог `coffeerun` и нажмите кнопку Open (Открыть).

Далее щелкните кнопкой мыши с нажатой клавишей Control (правой кнопкой мыши — в Windows) на нашем новом каталоге `coffeerun` на панели навигации Atom. Выберите New File (Новый файл) и введите `index.html` в качестве имени файла. Щелкните кнопкой мыши с нажатой клавишей Control (правой кнопкой мыши — в Windows) на `coffeerun` и выберите New Folder (Новый каталог). Назовите каталог `scripts`.

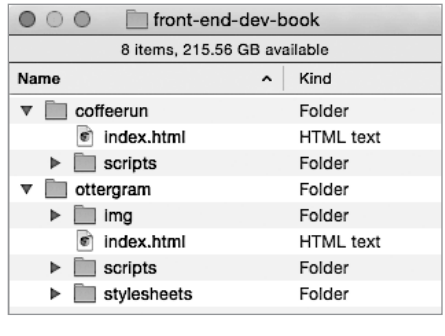


Рис. 8.5. Создание файлов и каталогов для CoffeeRun

В каталоге `front-end-dev-book` нам нужны подкаталоги для каждого из наших проектов — `ottergram` и `coffeerun` — с аналогичными файловыми структурами (рис. 8.5).

Если у вас уже открыт сеанс терминала и запущена утилита browser-sync, закрой-

те browser-sync нажатием Control+C. Если нет, то откройте новое окно терминала. В любом случае перейдите в каталог `coffeerun` (посмотрите описание команд в главе 1, если не вполне уверены, как это делается) и запустите browser-sync снова. Напомним команду для его запуска: `browser-sync start --server --files "stylesheets/*.css, scripts/*.js, *.html"`.

Добавьте в файл `index.html` простейший каркас для нашего документа (помните, что автодополнение Atom может сделать большую часть этой работы за вас, просто начните набирать `html`):

```
<!doctype html>
<html>
  <head>
    <meta charset="utf-8">
    <title>coffeerun</title>
  </head>
  <body>

  </body>
</html>
```

Теперь мы готовы к созданию нашего первого модуля!

Создание модуля DataStore

Первый модуль, который мы напишем, будет хранить информацию о заказах кофе в простой базе данных, не сильно отличающейся от записи заказов вручную. Все заказы будут храниться с указанием адреса электронной почты посетителя. Для начала мы будем отслеживать только текстовое описание заказа, например `quadruple espresso` (четверной эспрессо) (рис. 8.6).

Электронная почта	Заказ
caquino@bignerdranch.com	Четверной эспрессо
tgandee@bignerdranch.com	Черный кофе
jreece@bignerdranch.com	Смузи

Рис. 8.6. Первоначальная структура базы данных CoffeeRun

В дальнейшем мы будем также отслеживать объем, вкус и крепость (содержание кофеина) каждого заказа кофе. Адрес электронной почты посетителя будет служить уникальным идентификатором всего заказа, так что все подробности заказа будут связаны с единым адресом электронной почты. (Простите нас, кофеманы! Только один заказ для одного посетителя.)

Создайте новый файл `scripts/datastore.js`. Далее добавьте тег `<script>` в файл `index.html`, чтобы включить этот новый файл в наш проект:

```
<!doctype html>
<html>
  <head>
    <meta charset="utf-8">
    <title>coffeerun</title>
  </head>
  <body>
    <script src="scripts/datastore.js" charset="utf-8"></script>
  </body>
</html>
```

Сохраните файл `index.html`. Начнем в файле `scripts/datastore.js` с простого IIFE для создания структуры нашего модуля:

```
(function (window) {
  'use strict';
  // Здесь будет находиться код
})();
```

Теперь, когда каркас модуля готов и у него есть соответствующий тег `<script>`, пришло время включить его в пространство имен нашего приложения.

Добавляем модули в пространство имен

Во многих других языках программирования существует специальный синтаксис для создания модулей и компоновки их воедино. В ES5 — нет. Вместо этого мы собираемся добиться аналогичной структуры с помощью объектов.

Объекты можно использовать, чтобы ассоциировать данные любого типа с именем ключа. По сути, именно так мы и собираемся организовывать наши модули. Точнее, мы будем использовать отдельный объект в качестве пространства имен для приложения CoffeeRun. В этом пространстве имен отдельные модули будут регистрироваться, чтобы стать доступными для использования в остальном коде нашего приложения.

Чтобы использовать IIFE для регистрации модулей в пространстве имен, необходимо выполнить следующие действия.

1. Получить ссылку на пространство имен, если оно существует.
2. Создать код модуля.
3. Связать код модуля с этим пространством имен.

Посмотрим, как это выглядит на практике. Измените IIFE в файле `datastore.js`, как показано ниже. Мы дадим пояснения к этому коду после того, как вы его введете.

```
(function (window) {
  'use strict';
  // Здесь находится код
  var App = window.App || {};

  function DataStore() {
    console.log('running the DataStore function');
  }

  App.DataStore = DataStore;
  window.App = App;
})(window);
```

В теле IIFE мы объявили локальную переменную `App`. Если у объекта `window` уже есть свойство `App`, мы присвоим его локальной переменной `App`. Если нет, «ярлык» `App` будет ссылаться на новый пустой объект, представленный `{}`. `||` — *оператор умолчания*, известный также как *оператор логического ИЛИ*. Его можно использовать для предоставления допустимого значения (в данном случае `{}`), если первый операнд (`window.App`) еще не был создан.

Все наши модули будут выполнять ту же самую проверку. Это все равно, что сказать: «Кто бы ни добрался до этого места первым — вперед, создайте новый объект. Все остальные будут этот объект использовать».

Далее мы объявили функцию `DataStore`. Вскоре мы добавим в нее больше кода.

Наконец, мы связали `DataStore` с объектом `App` и присвоили глобальному свойству `App` только что измененную переменную `App` (если свойство не существовало и пришлось создать переменную в виде пустого объекта, необходимо их связать).

Сохраните файлы и перейдите в браузер. Откройте DevTools, щелкните на вкладке консоли и вызовите нашу функцию `DataStore` с помощью следующего кода:

```
App.DataStore();
```

Функция `DataStore` выполнится и выведет текст в консоль (рис. 8.7).

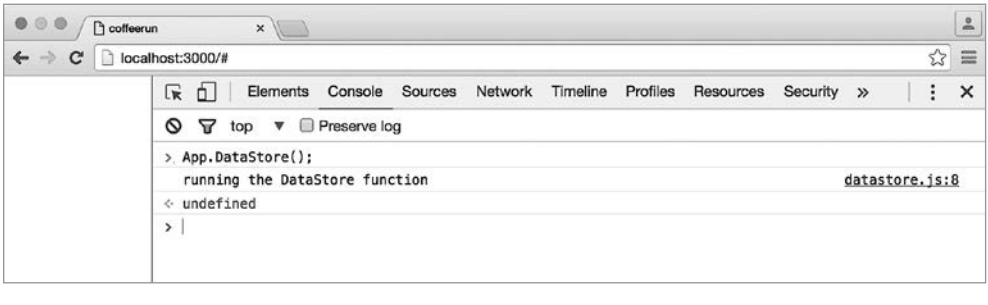


Рис. 8.7. Выполнение функции App.DataStore

Обратите внимание, что нам не пришлось писать `window.App.DataStore();`. Причина в том, что объект `window` является глобальным пространством имен. Все его свойства доступны любому коду JavaScript, который вы только напишете, в том числе и в консоли.

Конструкторы

IIFE дает нам возможность воспользоваться областью видимости функции, чтобы создать пространства имен для организации крупных частей кода. Есть и иной способ применения функций — они становятся фабриками объектов с одинаковыми свойствами и методами. В других языках программирования для этой цели можно воспользоваться классами. Строго говоря, в языке программирования JavaScript нет классов, но он все же предоставляет возможность создания пользовательских типов.

Мы уже начали создание типа `DataStore`. Теперь нам предстоит в два этапа адаптировать его к нашей конкретной задаче. На первом этапе добавим в него свойство, которое будет использоваться для внутреннего хранения данных. На втором этапе он получит набор методов для взаимодействия с этими данными. Давать другим объектам прямой доступ к этим данным нет необходимости, так как данный тип будет предоставлять внешний интерфейс в виде набора методов.

Функции-фабрики объектов называются в языке JavaScript *конструкторами*.

Добавьте следующий код в тело функции `DataStore` в файле `datastore.js`.

```
(function (window) {
  'use strict';
  var App = window.App || {};

  function DataStore() {
    console.log('running the DataStore function');
    this.data = {};
  }

  App.DataStore = DataStore;
  window.App = App;
})(window);
```

Задача конструктора состоит в создании нового объекта и его адаптации к конкретной задаче. Внутри тела конструктора можно ссылаться на этот объект с помощью ключевого слова `this`. Мы воспользовались оператором-точкой для создания в нашем новом объекте поименованного свойства `data` и присваивания ему пустого объекта.

Вы могли обратить внимание, что первая буква названия `DataStore` заглавная. Таково соглашение о наименованиях конструкторов в языке JavaScript. Это не обязательно, но считается хорошей привычкой (как способ известить

других разработчиков о том, что данную функцию следует использовать как конструктор).

Чтобы провести различие между конструктором и обычной функцией, мы использовали при его вызове ключевое слово `new`. Оно указывает языку JavaScript создать новый объект, настроить ссылку `this` для этого нового объекта и *неявно* вернуть этот объект. Это значит, что объект будет возвращен без явного наличия оператора `return` в конструкторе.

Сохраните изменения и вернитесь в консоль. Чтобы научиться использовать конструкторы, мы создадим два объекта — *экземпляра* — `DataStore` и добавим в них значения. Начнем с создания экземпляров:

```
var dsOne = new App.DataStore();
var dsTwo = new App.DataStore();
```

Мы создали эти экземпляры `DataStore` посредством вызова конструктора `DataStore`. Пока что свойства `data` у них обоих пусты. Добавим в них значения:

```
dsOne.data['email'] = 'james@bond.com';
dsOne.data['order'] = 'black coffee';
dsTwo.data['email'] = 'moneypenny@bond.com';
dsTwo.data['order'] = 'chai tea';
```

Затем проверим значения:

```
dsOne.data;
dsTwo.data;
```

Результаты показывают нам, что каждый экземпляр содержит различную информацию (рис. 8.8).

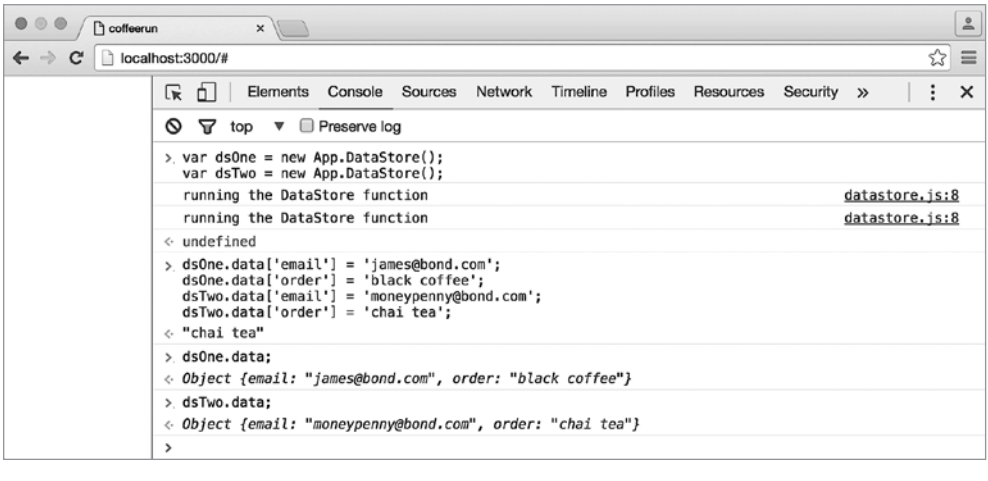


Рис. 8.8. Сохранение значений в экземплярах, созданных с помощью конструктора DataStore

Прототип конструктора

С помощью экземпляра `DataStore` мы можем вручную сохранять и извлекать данные. Но в нынешнем виде `DataStore` — всего лишь обходной путь создания объекта-литерала, и любой использующий экземпляр `DataStore` модуль придется программировать на непосредственное использование свойства `data`.

Это отнюдь не лучший способ проектирования программного обеспечения. Лучше было бы, если бы тип `DataStore` предоставлял общедоступный интерфейс для добавления, удаления и извлечения данных, держа все подробности выполнения в секрете.

Вторая часть создания нашего пользовательского типа `DataStore` и заключается в предоставлении этих методов для работы с данными. Их задача — служить интерфейсом для других модулей при работе с экземпляром `DataStore`. Чтобы добиться этого, воспользуемся замечательной возможностью функций JavaScript — свойством `prototype`.

Функции в языке JavaScript также являются объектами. Это значит, что у них могут быть свойства. В JavaScript у всех созданных посредством конструктора экземпляров есть доступ к общему набору свойств и методов: свойству конструктора `prototype`.

Для создания этих экземпляров мы использовали ключевое слово `new` при вызове конструктора. Ключевое слово `new` не только создает экземпляр и возвращает его, но и особую ссылку из экземпляра на свойство `prototype` конструктора. Эта ссылка существует для любого экземпляра, созданного с помощью конструктора с ключевым словом `new`.

Если мы добавляем свойство в `prototype` и присваиваем ему функцию, то каждый созданный с помощью соответствующего конструктора экземпляр будет иметь доступ к этой функции. Мы можем использовать внутри тела этой функции ключевое слово `this` (оно будет ссылаться на соответствующий экземпляр).

Чтобы оценить это на практике, создайте в файле `datastore.js` функцию `add` как свойство прототипа. Можете также удалить вызов `console.log`.

```
(function (window) {
  'use strict';
  var App = window.App || {};

  function DataStore() {
    ~~console.log('running the DataStore function');~~
    this.data = {};
  }

  DataStore.prototype.add = function (key, val) {
    this.data[key] = val;
  };

  App.DataStore = DataStore;
```

```
  window.App = App;
})(window);
```

Мы задали `DataStore.prototype` свойство `add` и присвоили ему функцию. Эта функция принимает на входе два аргумента: `key` и `val`. Эти свойства были использованы в теле функции для изменения свойства `data`.

Экземпляр `DataStore` будет сохранять информацию о заказе (`val`), используя адрес электронной почты посетителя (`key`).

Мы не создали настоящую базу данных, но `DataStore` работает достаточно хорошо для удовлетворения потребностей CoffeeRun. Он умеет сохранять определенную информацию — свойство `val` — в соответствии с уникальным идентификатором, задаваемым свойством `key`. Поскольку мы используем для хранения объект JavaScript, каждый `key` гарантированно является уникальной записью в базе данных (в объекте JavaScript имя свойства всегда уникально — аналогично именам функций в пространстве имен. Если вы попробуете сохранить несколько значений с одним и тем же ключом, то просто замените предыдущие значения для этого ключа новыми).

Эта особенность объектов JavaScript удовлетворяет важнейшему требованию к любой базе данных: раздельное хранение отдельных элементов данных.

Сохраните код и переключитесь обратно на браузер. Создайте в консоли экземпляр `DataStore` и воспользуйтесь его методом `add` для сохранения информации.

```
var ds = new App.DataStore();
ds.add('email', 'q@bond.com');
ds.add('order', 'triple espresso');
ds.data;
```

Проверьте значение свойства `data`, чтобы убедиться, что все работает (рис. 8.9).

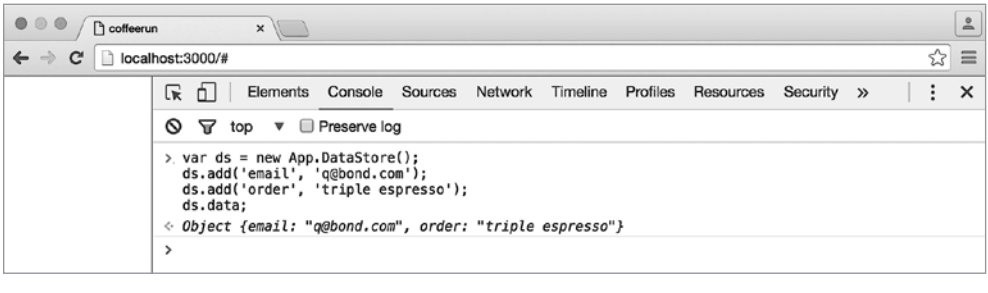


Рис. 8.9. Вызов метода прототипа

Добавляем в конструктор методы

Следующее, что нам нужно сделать, — добавить методы для доступа к данным. Добавим в файл `datastore.js` один метод для поиска значения по заданному ключу и еще один — для поиска всех ключей и значений:

```
...
  DataStore.prototype.add = function (key, val) {
    this.data[key] = val;
  };

  DataStore.prototype.get = function (key) {
    return this.data[key];
  };

  DataStore.prototype.getAll = function () {
    return this.data;
  };

  App.DataStore = DataStore;
  window.App = App;
})(window);
```

Мы создали метод `get`, который принимает на входе ключ, выполняет поиск значения для него в свойстве `data` экземпляра и возвращает (`return`) это значение. Мы также создали метод `getAll`. Он практически аналогичен, но вместо поиска значения для отдельного ключа возвращает ссылку на свойство `data`.

Теперь мы можем вносить данные в экземпляр `DataStore` и извлекать их из него. Для полноты цикла нам нужен еще метод для удаления информации. Добавим его в файл `datastore.js` прямо сейчас:

```
...
  DataStore.prototype.getAll = function () {
    return this.data;
  };

  DataStore.prototype.remove = function (key) {
    delete this.data[key];
  };

  App.DataStore = DataStore;
  window.App = App;
})(window);
```

При вызове нашего нового метода `remove` оператор `delete` удаляет пару «ключ/значение» из объекта.

Итак, мы закончили создание модуля `DataStore` — важнейшей части приложения CoffeeRun. Он может хранить данные, выдавать их по запросу и удалять ненужные данные по команде.

Чтобы посмотреть все наши методы в действии, сохраните код и перейдите в консоль после того, как утилита browser-sync обновит браузер. Введите следующий код, использующий все методы экземпляра `DataStore`:

```
var ds = new App.DataStore();
ds.add('m@bond.com', 'tea');
```

```
ds.add('james@bond.com', 'eshpressho');
ds.getAll();
ds.remove('james@bond.com');
ds.getAll();
ds.get('m@bond.com');
ds.get('james@bond.com');
```

Как показано на рис. 8.10, методы экземпляра `DataStore` работают так, как полагается. Именно с их помощью другие модули будут взаимодействовать с базой данных нашего приложения.

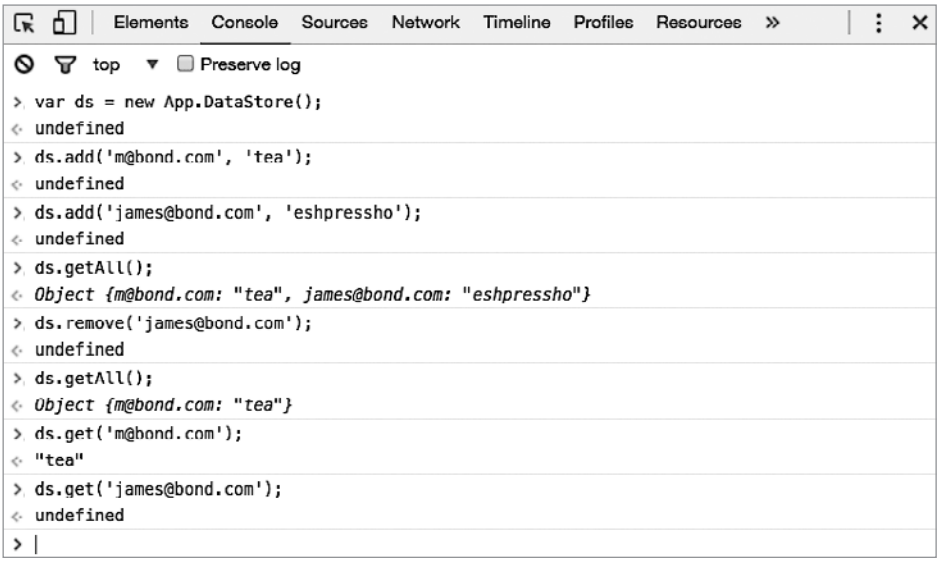


Рис. 8.10. Работа с экземпляром DataStore с помощью одних только методов прототипа

У нашего следующего модуля будет аналогичная структура: IIFE с параметром пространства имен для модификации. Но его функциональность будет коренным образом отличаться от функциональности модуля `DataStore`.

Создание модуля Truck

Следующий модуль, который мы создадим, носит название `Truck` и обеспечивает всю необходимую для управления автокафе функциональность. Он будет содержать методы для создания/выдачи заказов и для печати списка заказов, ожидающих выполнения. Рисунок 8.11 демонстрирует, как модуль `Truck` будет взаимодействовать с модулем `DataStore`.

Когда создается экземпляр `Truck`, он получает объект `DataStore`. У `Truck` имеются методы для работы с заказами кофе, но его не волнуют вопросы хранения этой

информации и управления ею. Он передает эти обязанности объекту `DataStore`. Например, при вызове метода `createOrder` экземпляра `Truck` он задействует метод `add` объекта `DataStore`.

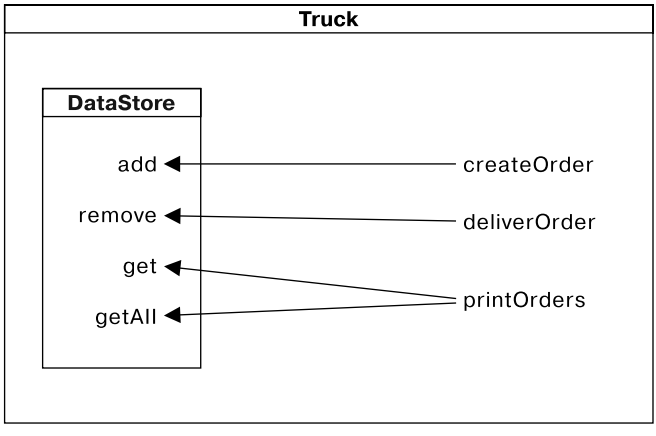


Рис. 8.11. Взаимодействие модулей Truck и DataStore

Создайте файл `scripts/truck.js` и добавьте тег `<script>` для него в файл `index.html`:

```
<!doctype html>
<html>
  <head>
    <meta charset="utf-8">
    <title>coffeerun</title>
  </head>
  <body>
    <script src="scripts/datastore.js" charset="utf-8"></script>
    <script src="scripts/truck.js" charset="utf-8"></script>
  </body>
</html>
```

Сохраните файл `index.html`. Сделайте в файле `truck.js` IIFE для нашего модуля и конструктор для типа `Truck`:

```
(function (window) {
  'use strict';
  var App = window.App || {};

  function Truck() {
  }

  App.Truck = Truck;
  window.App = App;
})(window);
```

Далее мы добавим параметры в наш конструктор, чтобы у каждого экземпляра был уникальный идентификатор и свой экземпляр `DataStore`. Идентификатор — просто имя, чтобы можно было отличать один экземпляр `Truck` от другого. Экземпляр `DataStore` будет играть намного более важную роль.

Добавим новые параметры в файл `truck.js`.

```
(function (window) {
  'use strict';
  var App = window.App || {};

  function Truck(truckId, db) {
    this.truckId = truckId;
    this.db = db;
  }

  App.Truck = Truck;
  window.App = App;
})(window);
```

Мы объявили параметры `truckId` и `db`, после чего присвоили их свойствам только что созданного экземпляра.

Экземплярам `Truck` понадобятся методы для управления заказами кофе — и мы их добавим. Данные о заказах будут включать адрес электронной почты и подробное описание напитка.

Добавляем заказы

Первый метод, который нам предстоит добавить, — `createOrder`. При вызове этого метода с его свойством `db` будет взаимодействовать экземпляр `Truck` (посредством объявленных ранее методов `DataStore`). То есть мы будем вызывать метод `add` объекта `DataStore`, чтобы сохранить заказ кофе (используя соответствующий заказу адреса электронной почты).

Объявляем этот новый метод прототипа в файле `truck.js`:

```
...
  function Truck(truckId, db) {
    this.truckId = truckId;
    this.db = db;
  }

  Truck.prototype.createOrder = function (order) {
    console.log('Adding order for ' + order.emailAddress);
    this.db.add(order.emailAddress, order);
  };

  App.Truck = Truck;
```

```
  window.App = App;
})(window);
```

Мы вывели сообщение в консоль в методе `createOrder`, после чего сохранили информацию о заказе с помощью метода `add` объекта `db`.

Для использования метода `add` достаточно сослаться на переменную `db` экземпляра `Truck` и вызвать `add`. Нет необходимости указывать пространство имен экземпляра `App.DataStore` или упоминать конструктор `DataStore` где-либо в этом модуле. Экземпляры `Truck` спроектированы так, чтобы работать с чем угодно, имеющим те же имена методов, что и экземпляр `DataStore`. Экземпляру `Truck` не нужна никакая информация, помимо этой.

Сохраните файл и протестируйте метод `createOrder` в консоли с помощью следующего кода:

```
var myTruck = new App.Truck('007', new App.DataStore());
myTruck.createOrder({ emailAddress: 'dr@no.com', coffee: 'decaf'});
myTruck.createOrder({ emailAddress: 'me@goldfinger.com',
                      coffee: 'double mocha'});
myTruck.createOrder({ emailAddress: 'm@bond.com', coffee: 'earl grey'});
myTruck.db;
```

Полученные результаты должны выглядеть так, как показано на рис. 8.12.

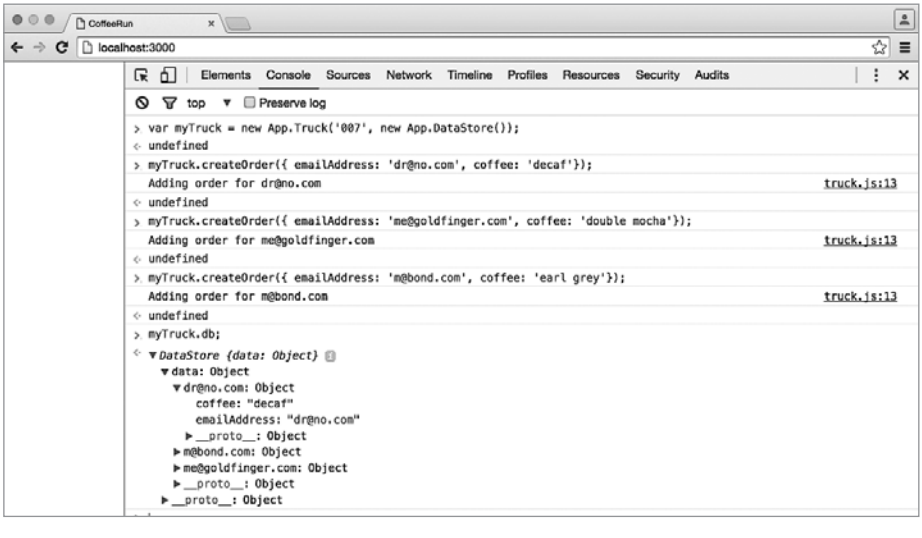


Рис. 8.12. Тест-драйв метода Truck.prototype.createOrder

После того как в консоль будет выведено значение свойства `myTruck.db`, необходимо щелкнуть на значке ▶, чтобы увидеть вложенные свойства (например, свойство `dr@no.com` объекта `data`).

Удаляем заказы

После выдачи заказа экземпляр `Truck` должен удалить его из базы данных. Добавляем новый метод `deliverOrder` в объект `Truck.prototype` в файле `truck.js`.

```
...
  Truck.prototype.createOrder = function (order) {
    console.log('Adding order for ' + data.emailAddress);
    this.db.add(data.emailAddress, order);
  };

  Truck.prototype.deliverOrder = function (customerId) {
    console.log('Delivering order for ' + customerId);
    this.db.remove(customerId);
  };

  App.Truck = Truck;
  window.App = App;

})(window);
```

Мы присвоили методу `Truck.prototype.deliverOrder` функциональное выражение. Эта функция принимает на входе аргумент `customerId`, который затем передает методу `this.db.remove`. Значением `customerId` должен быть соответствующий заказу адрес электронной почты.

Как и `createOrder`, метод `deliverOrder` вызывает имеющийся у объекта `this.db` метод `remove`. Ему не требуется какая-либо дополнительная информация о том, как метод `remove` работает.

Сохраните изменения и переключитесь на консоль. Создайте экземпляр `Truck`, добавьте с помощью метода `createOrder` несколько заказов, после чего убедитесь, что метод `deliverOrder` удаляет их из свойства `db` экземпляра (можете нажимать Enter или Shift+Enter после каждого вызова `createOrder` и `deliverOrder`, но не забудьте нажать Enter после каждой записи `myTruck.db`):

```
var myTruck = new App.Truck('007', new App.DataStore());
myTruck.createOrder({ emailAddress: 'm@bond.com', coffee: 'earl grey'});
myTruck.createOrder({ emailAddress: 'dr@no.com', coffee: 'decaf'});
myTruck.createOrder({ emailAddress: 'me@goldfinger.com',
                      coffee: 'double mocha'});
myTruck.db;
myTruck.deliverOrder('m@bond.com');
myTruck.deliverOrder('dr@no.com');
myTruck.db;
```

Введя эти тестовые команды, вы увидите, что информация о заказах в `myTruck.db` меняется после вызова метода `deliverOrder` (рис. 8.13).

Обратите внимание, что консоль отображает состояние данных *на момент* щелчка на значке ▶. Если не посмотреть значения в свойстве `myTruck.db` до того, как

вызвать метод `deliverOrder`, может показаться, что данные никогда и не добавлялись (рис. 8.14).

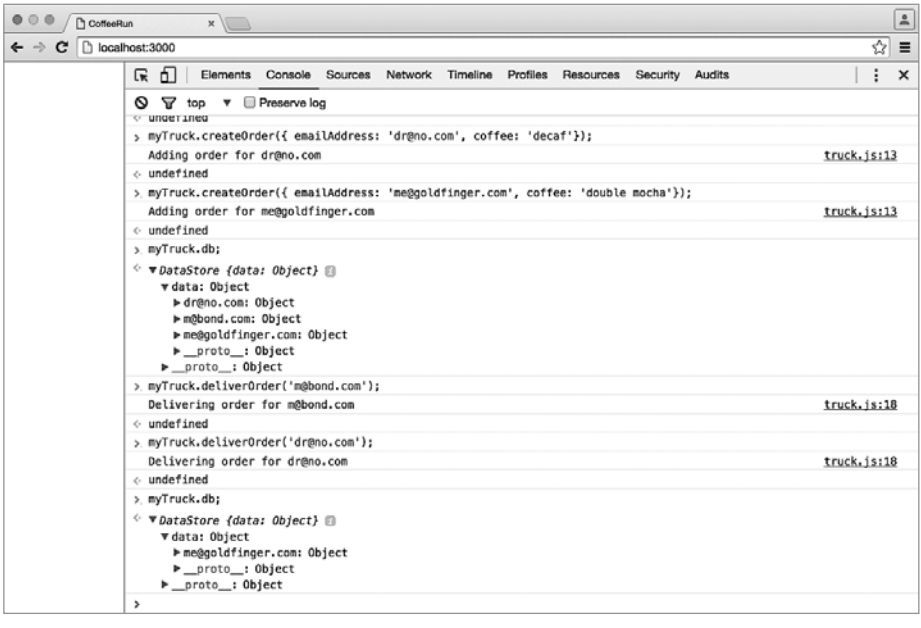


Рис. 8.13. Удаление данных о заказах с помощью метода Truck.prototype.deliverOrder

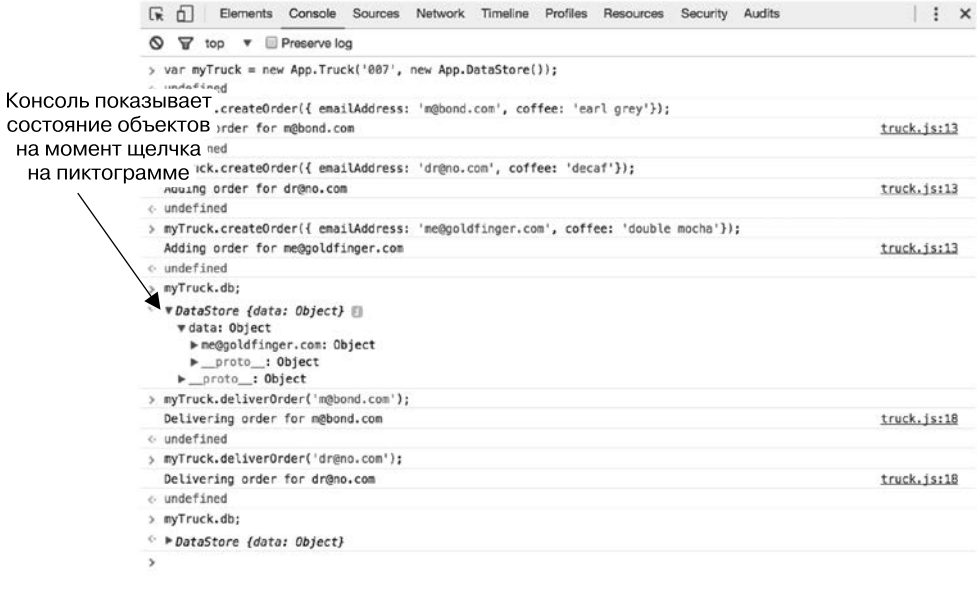


Рис. 8.14. Консоль отображает значения на момент щелчка на значке со стрелкой

Отладка

Последний метод, который нам нужно добавить в объект `Truck.prototype`, — `printOrders`. Он будет получать массив всех адресов электронной почты пользователей, выполнять итерацию по массиву и выводить в консоль (`console.log`) информацию о заказе.

Код этого метода очень похож на код других написанных нами функций и методов. Но в самом начале его разработки мы столкнемся с ошибкой, которую сможем обнаружить с помощью средств отладки браузера Chrome.

Идем дальше. Начнем с создания в файле `truck.js` простейшей версии метода `printOrders`. В его теле мы извлечем все заказы кофе из объекта `db`. Затем воспользуемся методом `Object.keys` для получения массива, содержащего адреса электронной почты заказов. Наконец, пройдем в цикле по массиву адресов электронной почты и выполним функцию обратного вызова для каждого элемента этого массива.

```
...
  Truck.prototype.deliverOrder = function (customerId) {
    console.log('Delivering order for ' + customerId);
    this.db.remove(customerId);
  };

  Truck.prototype.printOrders = function () {
    var customerIdArray = Object.keys(this.db.getAll());

    console.log('Truck #' + this.truckId + ' has pending orders:');
    customerIdArray.forEach(function (id) {
      console.log(this.db.get(id));
    });
  };

  App.Truck = Truck;
  window.App = App;

})(window);
```

Внутри нового метода `printOrders` мы вызываем метод `this.db.getAll`, чтобы извлечь все заказы в виде пар «ключ/значение» и передать их методу `Object.keys`, который вернет содержащий одни только ключи массив. Это массив мы присваиваем переменной `customerIdArray`.

При итерации по этому массиву мы передаем методу `forEach` функцию обратного вызова. В теле этого обратного вызова мы пытаемся получить (`get`) соответствующий `id` (адресу электронной почты посетителя) заказ.

Сохраните изменения и вернитесь в консоль. Создайте новый экземпляр `Truck` и добавьте еще несколько заказов кофе. После этого попробуйте выполнить наш новый метод `printOrders`.

```
var myTruck = new App.Truck('007', new App.DataStore());
myTruck.createOrder({ emailAddress: 'm@bond.com', coffee: 'earl grey'});
myTruck.createOrder({ emailAddress: 'dr@no.com', coffee: 'decaf'});
myTruck.createOrder({ emailAddress: 'me@goldfinger.com',
                      coffee: 'double mocha'});
myTruck.printOrders();
```

Вместо списка заказов кофе вы увидите ошибку Uncaught TypeError: Cannot read property 'db' of undefined (Неперехваченная ошибка типа: не могу прочитать принадлежащее undefined свойство ‘db’) (рис. 8.15).

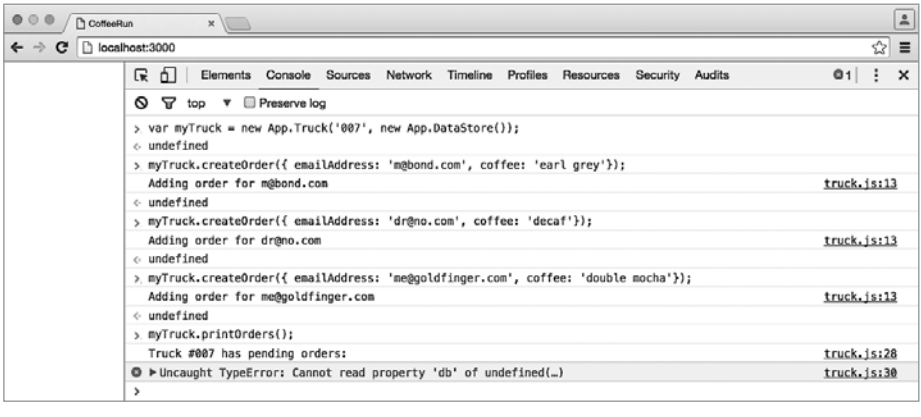


Рис. 8.15. Ошибка при выполнении метода printOrders

Это одна из самых распространенных ошибок, с которыми вы столкнетесь при написании кода на языке JavaScript. Многих разработчиков она сильно раздражает из-за сложности поиска вызвавшей ее причины. Умение работать с отладчиком (этому вы сейчас научитесь) — ключ к решению проблемы.

Поиск причины ошибки с помощью DevTools

Отладка требует многократного воспроизведения ошибки по мере постепенного выделения глючного кода. Отладчик браузера Chrome делает этот процесс несложным.

В случае появления ошибки консоль отображает имя файла и номер строки кода, вызвавшего ошибку (на рис. 8.15; она ссылается на `truck.js:30`; номер строки в вашем случае может отличаться). Щелкните на этом тексте, чтобы открыть проблемную строку кода в средствах отладки (рис. 8.16).

Вы видите сейчас панель исходного кода DevTools. Щелкните на красном значке в проблемной строке, чтобы увидеть информацию об ошибке (рис. 8.17).

Рис. 8.16. Просмотр ошибки в средствах отладки

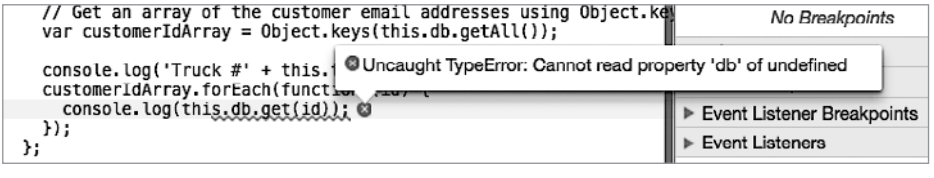


Рис. 8.17. Вызвавшая ошибку строка на панели исходного кода

Сообщение об ошибке говорит о том, что браузер полагает, будто вы пытаетесь прочитать свойство db несуществующего объекта.

Следующий шаг: выполнить код вплоть до вызвавшей ошибку строки, после чего проверить, каково значение этого объекта. На панели исходного кода щелкните на номере строки слева от строки с флагом ошибки. Это приведет к созданию *точки останова* для отладчика, указывающей браузеру приостановиться как раз перед тем, как попытаться выполнить эту строку. При задании точки останова номер строки слева становится синим, а на панель точек останова справа добавляется соответствующая запись (рис. 8.18).

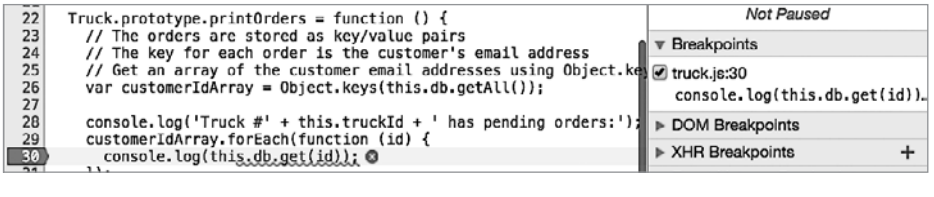


Рис. 8.18. Создание точки останова

Нажмите клавишу Esc, чтобы отобразить консоль под панелью исходного кода, известной под названием *всплывающей панели* (рис. 8.19). Вам пригодится возможность одновременно видеть код на панели исходного кода и работать с консолью.

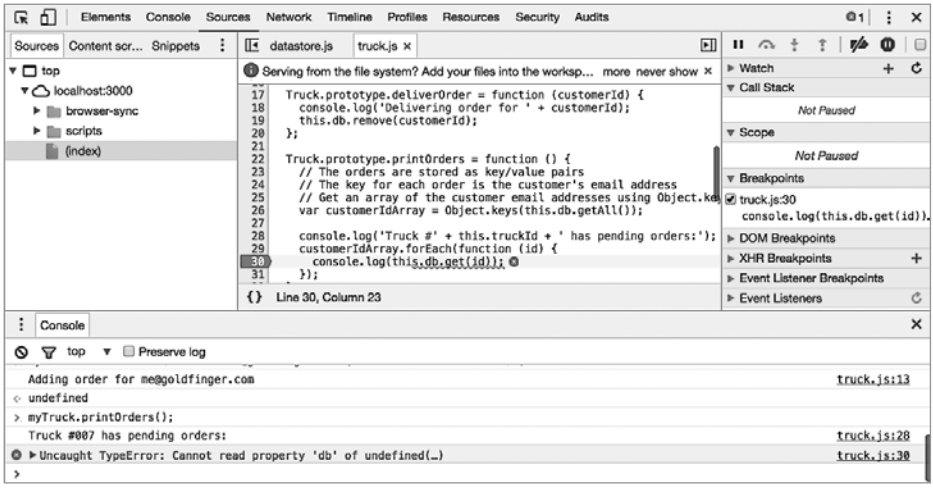


Рис. 8.19. Отображение всплывающей панели консоли

Выполните в консоли оператор `myTruck.printOrders();` еще раз. Браузер запустит отладчик, и реализация кода временно прервется на точке останова (рис. 8.20).

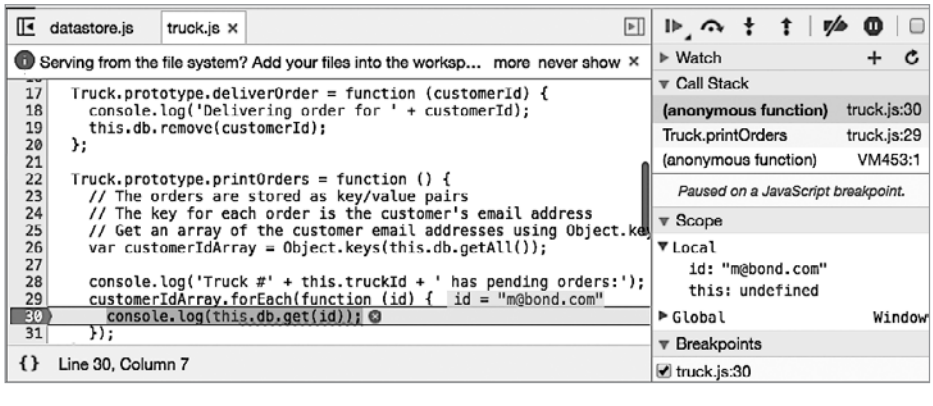


Рис. 8.20. Отладчик приостанавливает работу на точке останова

Во время приостановки отладчиком работы у вас имеется доступ ко всем переменным, какие только существуют на данном этапе. С помощью консоли вы можете изучить значения всех переменных в поисках признаков проблемы.

Попробуйте воспроизвести ошибку путем вычисления значений частей строки кода с флагом ошибки. Начните с кода с максимальным уровнем вложенности внутри скобок. В нашем случае это переменная `id`. Когда вы введете ее в консоли, вы получите ответ, что значение равно `m@bond.com` (рис. 8.21).

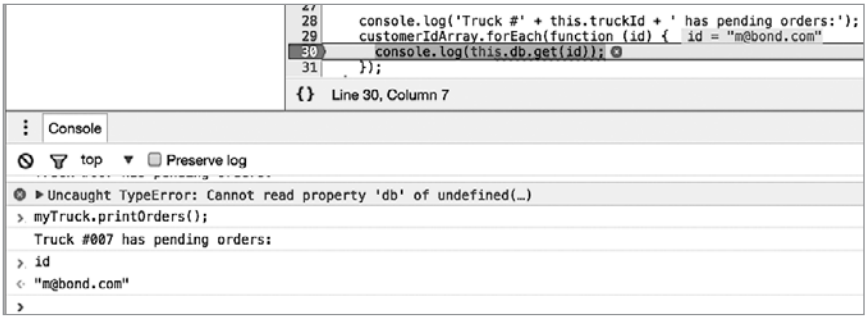


Рис. 8.21. Проверка самого глубоко вложенного значения

Поскольку при этом ошибка не воспроизводится, переходите к коду сразу за этим набором скобок, а именно `this.db.get(id)`. Вычислите его значение с помощью консоли. Вы увидите, что при этом возвращается ошибка (рис. 8.22).

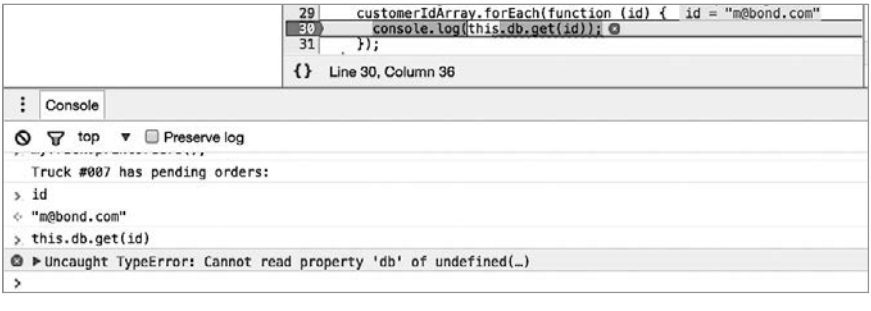


Рис. 8.22. Воспроизведение ошибки

Теперь можно дальше искать причину ошибку. Начните вычислять значение частей кода, постепенно убирая его части справа. Делайте это, пока не перестанет выводиться ошибка. Начните с `this.db.get`. После этого введите `this.db`. Консоль продолжает выводить ошибку (рис. 8.23).

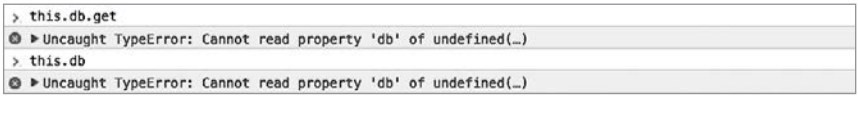


Рис. 8.23. Продолжаем искать

Наконец, введите `this`. Теперь мы находимся на стадии, на которой ошибки не происходит (рис. 8.24).

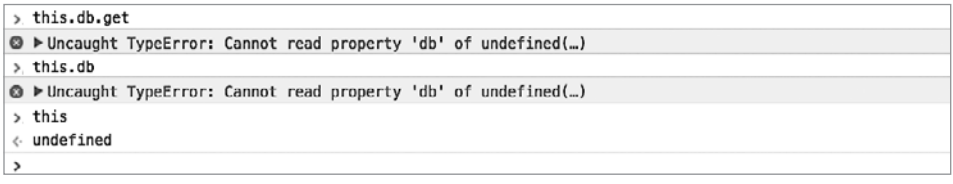


Рис. 8.24. Обрезаем код для поиска причины ошибки

Почему же значение ссылки `this` равно `undefined` в обратном вызове? Дело в том, что `this` не присвоено значение какого-либо объекта внутри функции обратного вызова. Необходимо выполнить присваивание явным образом.

Эта ситуация отличает от той, которая имела место с методами `Truck.prototype` (когда ссылка `this` указывала на экземпляр `Truck`). Несмотря на то что обратный вызов находится внутри метода `Truck.prototype.printOrder`, у него своя собственная переменная `this`, значение которой *не* присвоено, а значит, она `undefined`.

Прежде чем исправлять код, хорошо было бы ознакомиться с двумя другими способами поиска места ошибки. При задержке указателя мыши над различными частями кода на панели исходного кода отладчик будет отображать их значения. Когда указатель будет находиться над `this`, отладчик покажет, что его значение равно `undefined` (рис. 8.25).

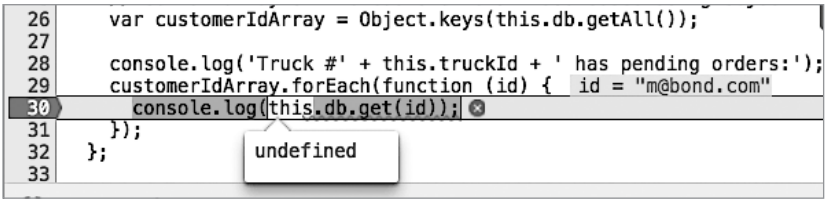


Рис. 8.25. Показ значений при задержке указателя мыши

Справа от кода находится панель области видимости, содержащая список имеющихся в наличии переменных. Можно видеть, как отображаются значения для переменных `id` и `this`, причем значение для `this` опять же равно `undefined` (рис. 8.26).

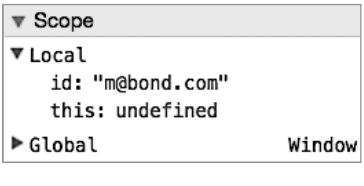


Рис. 8.26. Значения переменных отображаются на панели области видимости

Щелкните на синей кнопке ▶ вверху правой панели (рис. 8.27). Это отменит приостановку кода и позволит продолжить его выполнение.

Рис. 8.27. Панель управления отладчика

Прежде чем продолжить, удалите точку останова, опять щелкнув на номере строки, подсвеченной синим. Синий индикатор исчезнет (рис. 8.28).

Рис. 8.28. Щелкните на номере строки для отключения точки останова

Теперь самое время исправить эту досадную ошибку!

Задаем значение ссылки this с помощью метода bind

В языке программирования JavaScript ключевому слову `this` внутри функции при ее вызове автоматически присваивается значение. Для функций-конструкторов и методов прототипа значение `this` равно объекту экземпляра. Экземпляр называется *владельцем* вызова функции. Использование ключевого слова `this` предоставляет нам доступ к свойствам владельца.

Как мы уже упоминали, в случае функций обратного вызова `this` не присваивается автоматически значение объекта. Можно вручную указать, какой объект должен быть владельцем, с помощью метода `bind` функции (не забывайте, что функции JavaScript, по сути, являются объектами и у них могут быть свои свойства и методы, например `bind`).

Метод `bind` принимает на входе объектный аргумент и возвращает новую версию этой функции. При вызове новой версии переданный методу `bind` объектный аргумент будет использован в теле этой функции в качестве значения переменной `this`.

Внутри обратного вызова метода `forEach`, `this` имеет значение `undefined`, поскольку у обратного вызова отсутствует владелец. Исправим это, вызвав метод `bind` и передав ему ссылку на экземпляр `Truck`:

```
...
  Truck.prototype.printOrders = function () {
    var customerIdArray = Object.keys(this.db.getAll());

    console.log('Truck #' + this.truckId + ' has pending orders:');
```

```
    customerIdArray.forEach(function (id) {
      console.log(this.db.get(id));
    }.bind(this));
  };
...
```

Вне тела обратного вызова `forEach` ключевое слово `this` ссылается на экземпляр `Truck`. Добавляя `.bind(this)` сразу после анонимной функции (но внутри скобок вызова `forEach`), мы передаем `forEach` исправленную версию анонимной функции, использующую в качестве владельца экземпляр `Truck`.

Сохраните и убедитесь, что заказы выводятся правильно. Для этого вам понадобится снова объявить переменную `myTruck` и еще раз вызвать метод `createOrder`.

Результат должен выглядеть так, как показано на рис. 8.29.

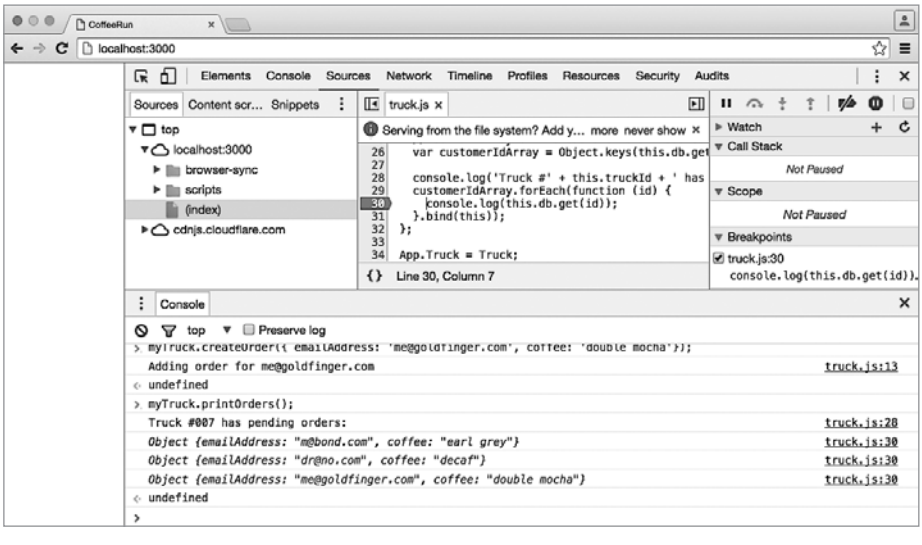


Рис. 8.29. После использования bind(this) метод printOrders заработал

Инициализация CoffeeRun при загрузке страницы

Наши модули `DataStore` и `Truck` работают как надо. Мы создали в консоли новый экземпляр `Truck`, предоставив ему новый объект `DataStore`.

Теперь нужно создать модуль, который бы выполнял то же самое при загрузке страницы. Создайте файл `scripts/main.js` и добавьте в файл `index.html` еще один тег `<script>`:

```
<!doctype html>
<html>
  <head>
```

```
    <meta charset="utf-8">
    <title>coffeerun</title>
  </head>
  <body>
    <script src="scripts/datastore.js" charset="utf-8"></script>>
    <script src="scripts/truck.js" charset="utf-8"></script>>
    <script src="scripts/main.js" charset="utf-8"></script>
  </body>
</html>
```

Сохраните файл `index.html`. Мы собираемся добавить в файл `main.js` IIFE, как и для других модулей, но на этот раз нам не придется экспортировать какие-либо новые свойства в `window.App`. Добавьте в `main.js` следующее:

```
(function (window) {
  'use strict';
  var App = window.App;
  var Truck = App.Truck;
  var DataStore = App.DataStore;
})(window);
```

Задача этого модуля заключается в получении объекта `window` для использования в теле функции. Он также извлекает конструкторы, описанные нами в качестве части пространства имен `window.App`.

С технической точки зрения мы могли просто написать весь наш код, используя полные имена (например, `App.Truck` и `App.DataStore`), но с короткими именами он получается более удобочитаемым.

Создание экземпляра Truck. Полностью аналогично тому, как мы делали в консоли, создадим экземпляр `Truck`, передав ему `id` и экземпляр `DataStore`.

Вызовите конструктор `Truck` в файле `main.js`, передав ему `ncc-1701`[1] в качестве аргумента `id` и новый экземпляр `DataStore`:

```
(function (window) {
  'use strict';
  var App = window.App;
  var Truck = App.Truck;
  var DataStore = App.DataStore;
  var myTruck = new Truck('ncc-1701', new DataStore());
})(window);
```

Это почти такой же код, как и тот, который вы вводили в консоли ранее, но необходимости указывать `App` перед `Truck` или `DataStore` нет, поскольку мы создали указывающие на объекты `App.Truck` и `App.DataStore` локальные переменные.

В настоящий момент наше приложение практически закончено. Однако мы все еще не можем взаимодействовать с экземпляром `Truck`. Почему? Дело в том, что

[1] NCC-1701 — реестровый номер космического корабля «Энтерпрайз» из научно-фантастического телесериала «Звездный путь». — *Примеч. пер.*

соответствующая переменная объявлена в модуле `main` внутри функции. Функции защищают свои переменные от доступа из кода, находящегося вне функции, включая код, вводимый в консоли.

Так что для взаимодействия с экземпляром `Truck` экспортируем его в файле `main.js` в глобальное пространство имен:

```
(function (window) {
  'use strict';
  var App = window.App;
  var Truck = App.Truck;
  var DataStore = App.DataStore;
  var myTruck = new Truck('ncc-1701', new DataStore());
  window.myTruck = myTruck;
})(window);
```

Сохраните изменения и вернитесь в консоль. Перезагрузите страницу вручную, чтобы точно очистить все, что было ранее сделано в консоли.

Начните набирать `myTruck` — и увидите, что консоль пытается автоматически дополнить этот текст (рис. 8.30). Это значит, что она нашла экспортированную нами в качестве свойства объекта `window` переменную `myTruck`.

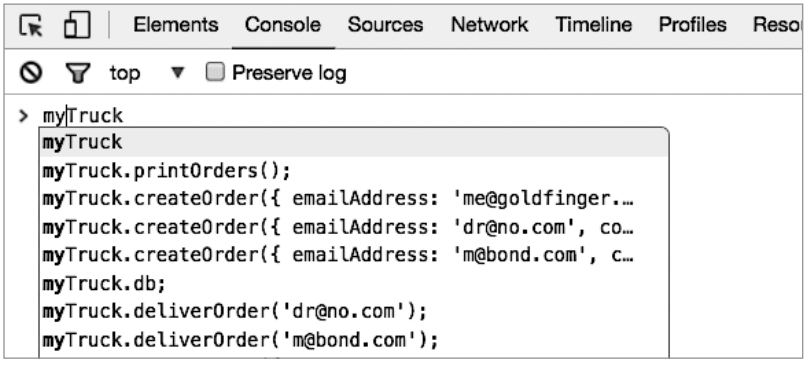


Рис. 8.30. Консоль находит переменную myTruck в глобальном пространстве имен

Вызовите несколько раз метод `myTruck.createOrder` с какими-нибудь тестовыми данными. Это можно легко сделать, позволив консоли автоматически дополнить текст вашими предыдущими вызовами метода `createOrder` (рис. 8.31).

В качестве альтернативы можете ввести следующий код, чтобы убедиться, что все работает так, как и должно:

```
myTruck.createOrder({ emailAddress: 'me@goldfinger.com',
                      coffee: 'double mocha'});
myTruck.createOrder({ emailAddress: 'dr@no.com', coffee: 'decaf'});
myTruck.createOrder({ emailAddress: 'm@bond.com', coffee: 'earl grey'});
```

```
myTruck.printOrders();
myTruck.deliverOrder('dr@no.com');
myTruck.deliverOrder('m@bond.com');
myTruck.printOrders();
```

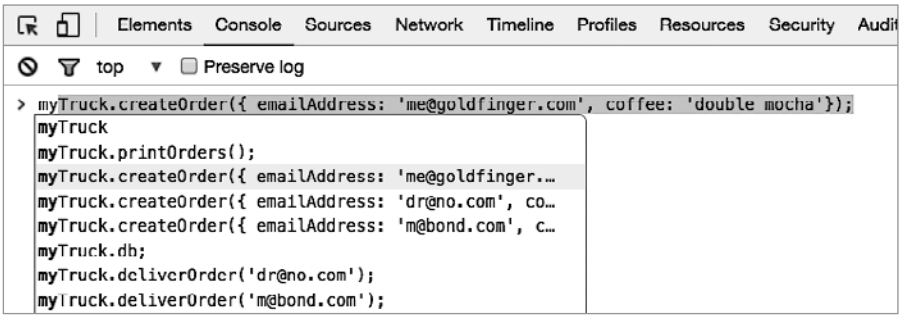


Рис. 8.31. Консоль автоматически дополняет текст на основе предыдущих вызовов метода createOrder

После испытания методов `createOrder`, `printOrders` и `deliverOrder` вы должны увидеть что-то похожее на рис. 8.32.

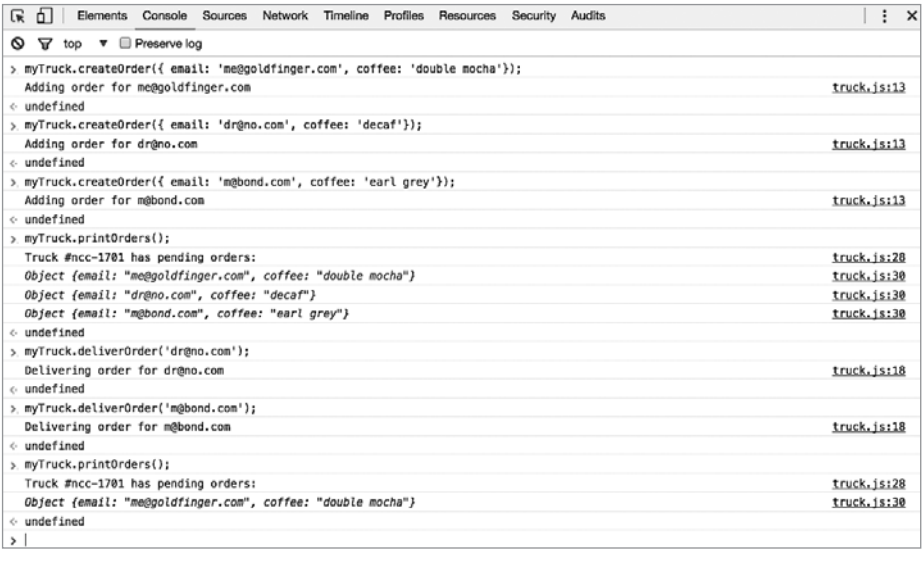


Рис. 8.32. Загруженное заказами автокафе

Ура! Мы закончили фундамент CoffeeRun. У него пока еще нет UI, но мы добавим его в следующих главах. И нам не потребуется вносить изменения в ядро приложения, поскольку UI просто будет обращаться к уже написанным и проверенным нами методам `Truck.prototype`.

Именно в этом заключается преимущество модульного подхода: мы можем работать над приложением слоями, где каждый новый слой основывается на работающем коде базовых модулей.

Бронзовое упражнение: идентификатор автокафе для не фанатов сериала «Звездный путь»

Передайте в файле `main.js` в качестве аргумента `truckId` другую строку.

(Несколько неплохих вариантов: «Серенити», «КИТТ» или «Галактика»[1]. Использовать «ЭАЛ»[2], полагаем, — не самая хорошая идея.)

Для самых любознательных: закрытые данные модулей

Внутри модуля у конструкторов и методов прототипа есть доступ к любым переменным, объявленным внутри IIFE. Существует альтернатива добавлению свойств в свойство `prototype` для целей совместного использования данных экземплярами с одновременным скрытием их от любого кода вне модуля. Выглядит она вот так:

```
(function (window) {
  'use strict';
  var App = window.App || {};
  var launchCount = 0;

  function Spaceship() {
    // Здесь находится код инициализации
  }

  Spaceship.prototype.blastoff = function () {
    // Область видимости замыкания позволяет обращаться
    // к переменной launchCount
    launchCount++;
    console.log('Spaceship launched!')
  }
```

[1] Названия вымышленных средств передвижения. «Серенити» — космический корабль из культового американского научно-фантастического телесериала «Светлячок». «КИТТ» — автомобиль с искусственным интеллектом из научно-фантастического телесериала «Рыцарь дорог». «Галактика» — корабль из фантастического телесериала «Звездный крейсер “Галактика”». — *Примеч. пер.*

[2] Взбунтовавшийся компьютер из научно-фантастического романа Артура Кларка «2001: Космическая одиссея». — *Примеч. пер.*

```
  Spaceship.prototype.reportLaunchCount = function () {
    console.log('Total number of launches: ' + launchCount);
  }

  App.Spaceship = Spaceship
  window.App = App;
})(window);
```

В других языках программирования есть возможность объявлять *закрытые* переменные, но в JavaScript — нет. Можно также воспользоваться областью видимости замыкания (использование функцией объявленных во внешней области видимости переменных) для объявления закрытых переменных.

Серебряное упражнение: делаем данные закрытыми

Измените модуль `DataStore`, чтобы свойство `data` было закрытым для модуля.

Существуют ли какие-то причины, почему это было бы нежелательно? Что произойдет, если вы объявите несколько экземпляров `DataStore`?

Для самых любознательных: делаем то же самое в обратном вызове метода forEach

Мы немножко обманули вас ранее. Использование метода `bind` — не *единственный* способ задать значение переменной `this` в обратном вызове для `forEach`.

Загляните в MDN в документацию для `Array.prototype.forEach` (developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/forEach). Вы увидите, что метод `forEach` принимает на входе необязательный второй аргумент и использует его в качестве значения ссылки `this` в обратном вызове.

Это значит, что можно было написать метод `printOrders` следующим образом:

```
...
Truck.prototype.printOrders = function () {
  var customerIdArray = Object.keys(this.db.getAll());

  console.log('Truck #' + this.truckId + ' has pending orders:');
  customerIdArray.forEach(function (id) {
    console.log(this.db.get(id));
  }, this);
};
...
```

Однако `bind` — удобный метод, который еще встретится нам в следующих главах. На примере метода `Truck.prototype.printOrders` было удобно познакомить вас с его синтаксисом.

9 Введение в фреймворк Bootstrap

В этой главе мы создадим HTML-разметку для нашего UI. Мы воспользуемся стилями, которые предоставляет популярный CSS-фреймворк Bootstrap, чтобы немного отшлифовать наш пользовательский интерфейс, не создавая при этом таблицы стилей CSS вручную. Так мы сможем сконцентрировать все внимание на логике приложения в JavaScript, которую будем программировать в главе 10.

Наш пользовательский интерфейс для приложения CoffeeRun будет состоять из двух частей. Первая содержит форму, в которую пользователь может внести заказ кофе со всеми подробностями (рис. 9.1). Во второй заказы кофе будут отображаться в виде списка. У каждой из этих частей будет соответствующий JavaScript-модуль для обработки взаимодействия пользователя с системой.

Добавляем фреймворк Bootstrap в приложение

Библиотека CSS фреймворка Bootstrap предоставляет набор стилей, которые можно использовать для сайтов и приложений. Из-за ее популярности нежелательно применять Bootstrap для интерфейса находящегося в промышленной эксплуатации сайта без определенных изменений внешнего вида. Иначе все кончится тем, что ваш сайт будет выглядеть так же, как и у всех других. Однако Bootstrap отлично подходит для быстрого создания предварительных образцов.

Как и файл `normalize.css` в приложении Ottergram, получить фреймворк Bootstrap можно, скачав его с `cdnjs.com`. Используйте версию Bootstrap 3.3.6, находящуюся по адресу cdnjs.com/libraries/twitter-bootstrap/3.3.6 (чтобы найти последнюю версию для ваших собственных проектов, выполните поиск на `cdnjs.com` по ключевым словам `twitter bootstrap`).

Рис. 9.1. Стилизация приложения CoffeeRun с помощью фреймворка Bootstrap

Убедитесь, что вы воспользовались ссылкой на файл `bootstrap.min.css` (рис. 9.2), а не ссылкой на тему или шрифты.

После копирования ссылки откройте файл `index.html` и добавьте тег `<link>` с соответствующим URL (нам пришлось перенести атрибут `href` на вторую строку, чтобы он поместился на странице, но вы вводите его одной строкой):

```
...
  <head>
    <meta charset="utf-8">
    <title>coffeerun</title>
    <link rel="stylesheet" href="https://cdnjs.cloudflare.com/ajax/libs/
          twitter-bootstrap/3.3.6/css/bootstrap.min.css">
  </head>
...
```

Как работает Bootstrap. Фреймворк Bootstrap предоставляет адаптивные стили для сайтов и веб-приложений. В большинстве случаев необходимо просто включить CSS-файл, после чего добавить классы в разметку. Один из основных классов, которые мы будем использовать, — `container`.

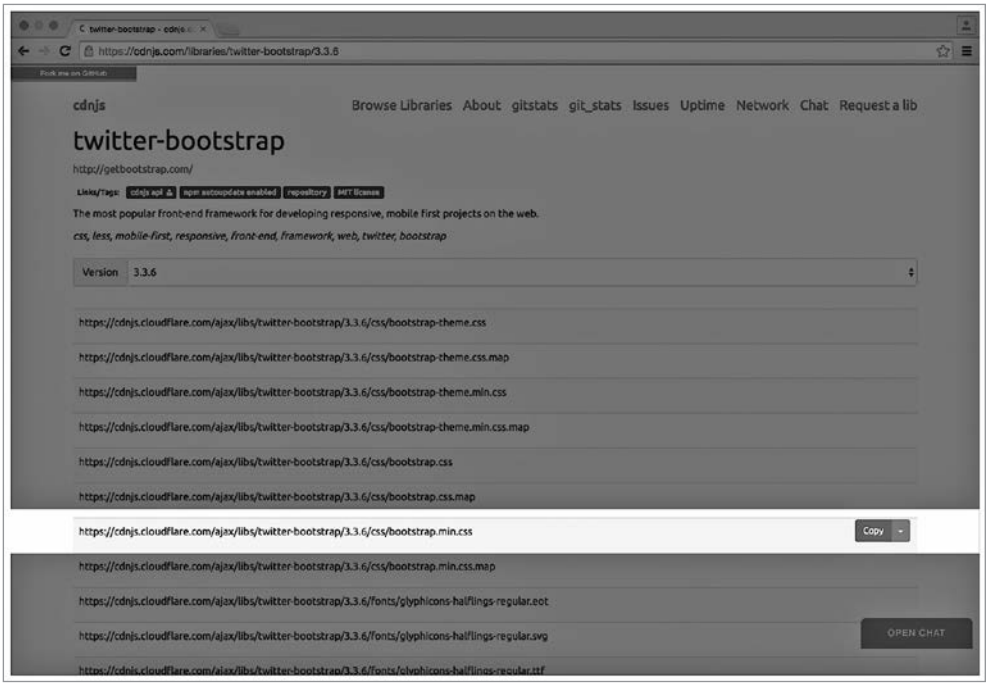


Рис. 9.2. Страница cdnjs.com для twitter-bootstrap

Добавьте класс `container` в элемент `<body>` в файле `index.html`. И раз уж вы находитесь тут, добавьте также на страницу заголовок:

```
...
    <title>coffeerun</title>
    <link rel="stylesheet" href="https://cdnjs.cloudflare.com/ajax/libs/
      twitter-bootstrap/3.3.6/css/bootstrap.min.css">
  </head>
  <body>
  <body class="container">
    <header>
      <h1>CoffeeRun</h1>
    </header>
    <script src="scripts/datastore.js" charset="utf-8"></script>
    <script src="scripts/truck.js" charset="utf-8"></script>
    <script src="scripts/main.js" charset="utf-8"></script>
  </body>
</html>
```

Класс `container` выступает в качестве обертки для всего контента, который должен приспосабливаться к размеру области просмотра. Он служит основой адаптивности поведения макета.

Сохраните файл `index.html`, убедитесь, что утилита browser-sync запущена, и просмотрите страницу. Она должна выглядеть так, как показано на рис. 9.3.

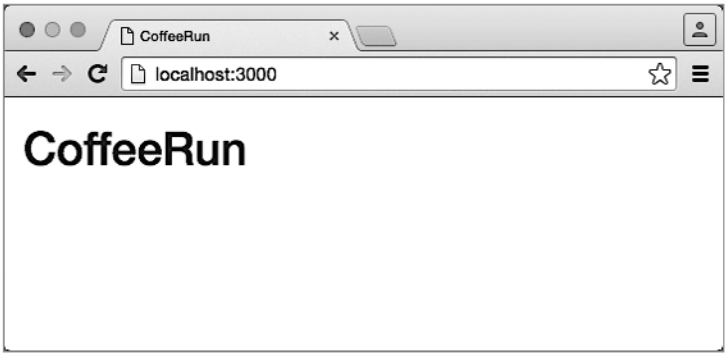


Рис. 9.3. Стилизованный с помощью фреймворка Bootstrap заголовок

Хотя на странице пока почти ничего нет, обратите внимание, что вокруг заголовка есть поля достаточного размера и для него используется стиль шрифта.

В Bootstrap существуют стили для огромного количества различных визуальных элементов. Приложение CoffeeRun затронет лишь верхушку айсберга, но в следующих главах мы рассмотрим еще парочку стилей. А пока что нужно добавить разметку для формы заказа.

Создание формы заказа

Добавьте тег `<section>`, два тега `<div>` и тег `<form>` в файл `index.html` под только что созданным элементом `<header>`:

```
...
    <header>
      <h1>CoffeeRun</h1>
    </header>
    <section>
      <div class="panel panel-default">
        <div class="panel-body">
          <form data-coffee-order="form">
            <!-- Здесь будут находиться элементы для ввода данных -->
          </form>
        </div>
      </div>
    </section>
    <script src="scripts/datastore.js" charset="utf-8"></script>
...
```

Именно в теге `<form>` будет происходить все основное действие. Мы задали для него атрибут `data-coffee-order` со значением `form`. В приложении CoffeeRun мы будем использовать атрибуты данных для доступа к элементам DOM из кода JavaScript аналогично тому, как делали в Ottergram.

Для макета мы добавили два тега `<div>`. В принципе, не имеет значения, что это именно теги `<div>`. Важно то, что к ним применяются классы `panel`, `panel-default` и `panel-body`. Это классы фреймворка Bootstrap, которые будут вызывать срабатывание стилей.

Не забывайте, что теги `<div>` — просто универсальные контейнеры уровня блока для остальной разметки. Они занимают столько места по горизонтали, сколько предоставляет им охватывающий их родительский элемент. Они часто будут использоваться в приложении CoffeeRun, и вы неоднократно встретите их в примерах из документации по фреймворку Bootstrap.

Возможно, вы недоумеваете, почему наши теги `<div>` и `<form>` обертывает тег `<section>`. У тегов `<div>` нет семантического значения, а у тегов `<section>` — есть: они логически группируют другую разметку. Конкретно в этом теге `<section>` будет располагаться пользовательский интерфейс для формы. На странице вполне могут быть и другие теги `<section>` для каких-либо иных частей UI.

Добавляем текстовые поля для ввода

Главная информация, которая нас заботит, — сам заказ кофе. Пока что мы воспользуемся простым однострочным текстовым полем для представления заказа. Позднее добавим еще поля для ввода дополнительной информации о заказе.

При использовании Bootstrap для форм приходится добавлять дополнительные элементы `<div>`, предназначенные исключительно для применения описанных в библиотеке фреймворка Bootstrap стилей.

Добавьте в файл `index.html` еще один тег `<div>` с классом `form-group`. Класс `form-group` фреймворка Bootstrap обеспечивает одинаковые вертикальные поля для элементов формы. Затем добавьте элементы `<label>` и `<input>`:

```
...
      <div class="panel panel-default">
        <div class="panel-body">
          <form data-coffee-order="form">
            <!-- Здесь будут находиться элементы для ввода данных -->
            <div class="form-group">
              <label>Coffee Order</label>
              <input class="form-control" name="coffee">
            </div>
          </form>
        </div>
      </div>
...
```

Класс `form-control` — еще один определяемый фреймворком Bootstrap класс. Он обеспечивает для наших элементов формы макет и типографское оформление.

Сохраните файл `index.html` и посмотрите на результаты в браузере (рис. 9.4).

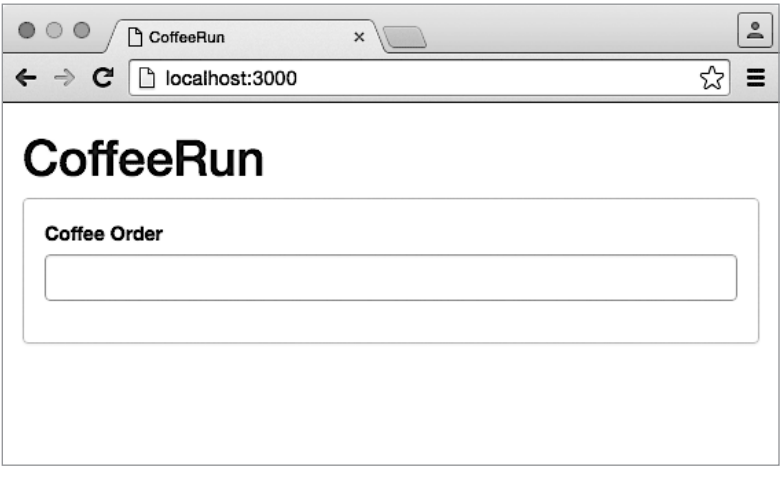


Рис. 9.4. Поле ввода для заказа кофе

По умолчанию наш элемент `<input>` представляет собой однострочное текстовое поле. Помимо класса `form-control`, у него есть только один атрибут — `name`. При подтверждении отправки формы данные отправляются на сервер, а с ними и атрибут `name`. Если рассматривать данные формы как пару «ключ/значение», то атрибут `name` будет ключом, а вводимые пользователем данные — значением.

Связываем метку и элемент формы

Теги `<label>` — важное расширение, делающее элементы формы удобными и простыми в использовании. Чтобы указать, для какого элемента служит меткой тег `<label>`, можно задать значение его атрибута `for` равным атрибуту `id` этого элемента формы.

Добавьте атрибуты `for` и `id` в элементы формы `<label>` и `<input>` соответственно в файле `index.html`. Установите значение обоих атрибутов равным `coffeeOrder`:

```
...
      <div class="panel panel-default">
        <div class="panel-body">
          <form data-coffee-order="form">
            <div class="form-group">
              <label for="coffeeOrder">Coffee Order</label>
              <input class="form-control" name="coffee" id="coffeeOrder">
            </div>
          </form>
        </div>
      </div>
...
```

После связывания тега `<label>` с элементом формы можно щелкнуть на тексте тега `<label>` на странице — и он сделает активным связанный с ним элемент формы. Следует всегда связывать теги `<label>` с их элементами форм.

Чтобы посмотреть, как это работает, переключитесь на браузер и щелкните на тексте метки `Coffee Order`. При этом фокус окажется на элементе `<input>`, и можно будет вводить текст (рис. 9.5).

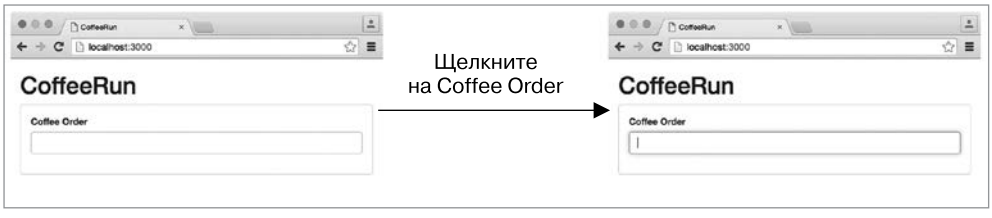


Рис. 9.5. Щелчок кнопкой мыши на связанной метке приводит к получению полем ввода фокуса

Добавление автофокуса

Поскольку это первое поле на экране, хотелось бы, чтобы пользователь мог, не выполняя каких-либо щелчков кнопкой мыши, вводить текст в него сразу после загрузки страницы.

Чтобы добиться этого, добавим атрибут `autofocus` в поле `<input>` в файле `index.html`:

```
...
        <div class="form-group">
          <label for="coffeeOrder">Coffee Order</label>
          <input class="form-control" name="coffee" id="coffeeOrder"
                 autofocus>
        </div>
...
```

Сохраните изменения в файле `index.html` и вернитесь в браузер. Вы увидите, что указатель будет находиться в текстовом поле ввода и оно будет выделено сразу, как только страница загрузится (рис. 9.6).

Обратите внимание, что у атрибута `autofocus` нет значения. Оно ему не требуется. Само присутствие атрибута `autofocus` в теге `<input>` говорит браузеру о необходимости сделать это поле активным. `autofocus` — *булев* атрибут. Это значит, что его возможные значения — только `true` и `false`. Достаточно просто добавить имя атрибута в тег, чтобы задать его значение. Когда он присутствует в теге, его значение равно `true`. Когда он отсутствует, считается, что он равен `false`.

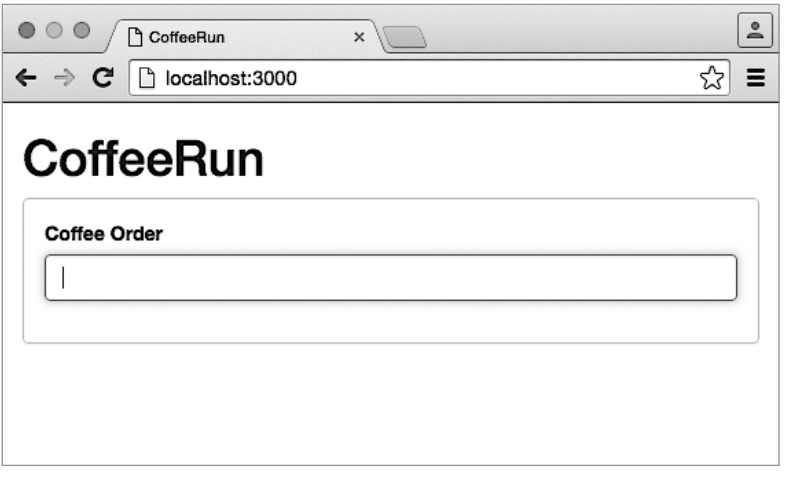


Рис. 9.6. Поле ввода с автофокусом при загрузке страницы

Добавление поля ввода для электронной почты

Когда мы создавали модули `Truck` и `DataStore`, то отслеживали заказы по адресу электронной почты пользователя. Теперь мы будем получать эту информацию с помощью еще одного элемента `<input>`.

Добавьте в файл `index.html` еще один элемент `.form-group` со своими элементами `<label>` и `<input>`. Задайте тип элемента `<input>` равным `email`, `name` — `emailAddress` и `id` — `emailInput`. Добавьте также атрибут `value` с равным пустой строке значением. Это гарантирует пустоту данного поля при загрузке страницы. Наконец, свяжите `<input>` и `<label>` с помощью атрибута `id`.

```
...
        <form data-coffee-order="form">
          <div class="form-group">
            <label for="coffeeOrder">Coffee Order</label>
            <input class="form-control" name="coffee" id="coffeeOrder"
                   autofocus>
          </div>
          <div class="form-group">
            <label for="emailInput">Email</label>
            <input class="form-control" type="email" name="emailAddress"
              id="emailInput" value="">
          </div>
        </form>
...
```

Сохраните файл `index.html` и загляните в браузер, чтобы посмотреть на новое поле формы (рис. 9.7).

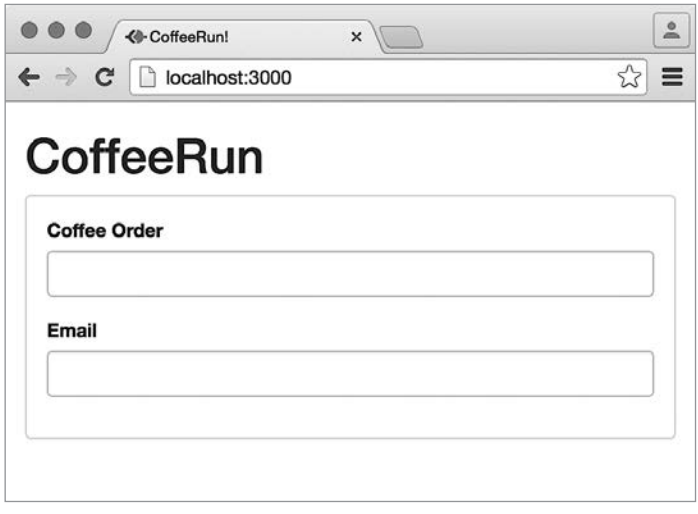


Рис. 9.7. Поле ввода для адреса электронной почты

Отображение примера ввода с помощью текста-заполнителя

Иногда пользователям нравятся советы насчет того, что именно им следует вводить в текстовое поле. Чтобы создать пример текста, задействуем атрибут `placeholder`.

Добавьте атрибут `placeholder` в наш новый элемент `<input>` в файле `index.html`:

```
...
          <div class="form-group">
            <label for="emailInput">Email</label>
            <input class="form-control" type="email" name="emailAddress"
              id="emailInput" value="" placeholder="dr@who.com">
          </div>
...
```

Сохраните файл. Результат будет выглядеть так, как показано на рис. 9.8.

Значение атрибута `placeholder` видно в текстовом поле до тех пор, пока пользователь не начнет вводить какой-то текст (в этот момент оно исчезнет). Если пользователь удалит весь текст в поле, то текст-заполнитель появится снова.

Возможность выбора и переключатели

Хотелось бы дать пользователям возможность указывать размер порции кофе — выбирать между маленькой, средней и большой (причем чтобы выбрать можно было не более одного варианта). Для такого типа вводимых данных можно использовать поля `<input>` со значением атрибута `type`, равным `radio`.

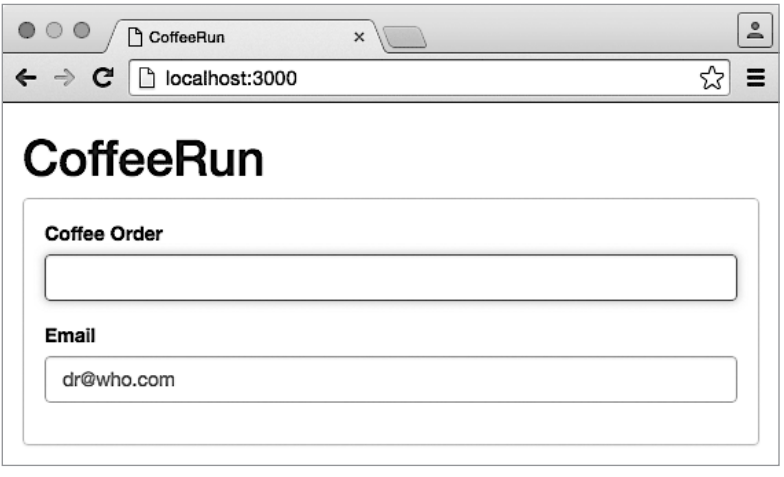


Рис. 9.8. Текст-заполнитель в поле ввода для электронной почты

Разметка для наших переключателей будет отличаться от разметки для других полей `<input>`. Каждый переключатель будет состоять из поля `<input>`, *обернутого* в элемент `<label>`. А `<label>`, в свою очередь, будет обернут в тег `<div>`, *тоже* с классом `radio`.

Если вы недоумеваете, почему HTML для переключателей отличается, то объясним: это происходит потому, что фреймворк Bootstrap стилизует их не так, как другие элементы формы.

При написании своего собственного кода вы можете выбрать: обертывать ли элемент `<input>` в `<label>` или использовать атрибут `<for>` (оба варианта правильные). Но при использовании фреймворка Bootstrap необходимо следовать его паттернам и соглашениям, чтобы стили работали так, как ожидается. Обратитесь к документации по Bootstrap, чтобы посмотреть примеры, как следует структурировать HTML (getbootstrap.com/css/#forms).

Добавьте в файл `index.html` разметку для наших переключателей сразу после поля `<input>` для электронной почты:

```
...
          <div class="form-group">
            <label for="emailInput">Email</label>
            <input class="form-control" type="email" name="emailAddress"
              id="emailInput" value="" placeholder="dr@who.com">
          </div>
          <div class="radio">
            <label>
              <input type="radio" name="size" value="short">
               Short
            </label>
          </div>
          <div class="radio">
```

```
            <label>
              <input type="radio" name="size" value="tall" checked>
               Tall
            </label>
          </div>
          <div class="radio">
            <label>
              <input type="radio" name="size" value="grande">
               Grande
            </label>
          </div>
        </form>
...
```

У всех трех наших переключателей одинаковое значение атрибута `name` (`size`). Это указывает браузеру, что за раз может быть выбран (отмечен) только один из них. У переключателя `Tall` имеется булев атрибут `checked`. Он работает аналогично `autofocus`: при его наличии значение атрибута равно `true`, а при отсутствии — `false`.

Сохраните файл `index.html` и посмотрите на наши новые переключатели (рис. 9.9).

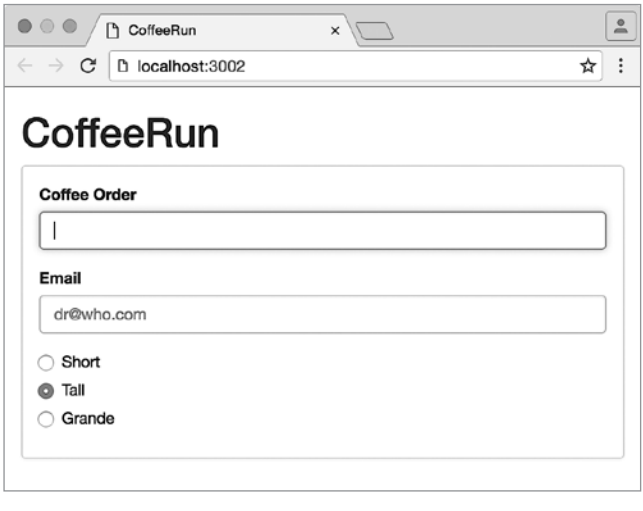


Рис. 9.9. Переключатели для выбора размера порции кофе

Попробуйте пощелкать кнопкой мыши или на переключателях, или на тексте рядом с ними. В любом случае переключатель покажет, что он был выбран.

Добавляем выпадающее меню

Некоторые люди просто обожают ароматизированный кофе. Хотелось бы, чтобы у них была возможность выбирать из нескольких различных ароматизаторов. По умолчанию ароматизатор добавляться не будет.

Можно воспользоваться для этого набором переключателей, но количество ароматизаторов в списке может оказаться довольно большим. Чтобы варианты ароматизаторов не загромождали пользовательский интерфейс, лучше ввести выпадающее меню.

Для создания стилизованного с помощью фреймворка Bootstrap выпадающего меню добавьте в файл `index.html` тег `<div>` с классом `form-group`. Создайте элемент `<select>` с классом `form-group`. Bootstrap стилизует его как выпадающий. Свяжите этот элемент с соответствующим элементом `<label>` с помощью `id`, равного `flavorShot`. Внутри `<select>` добавьте по элементу `<option>` для каждого из пунктов меню, которые вы хотели бы отобразить, указав для каждого соответствующее значение.

```
...
          <div class="radio">
            <label>
              <input type="radio" name="size" value="grande">
               Grande
            </label>
          </div>
          <div class="form-group">
            <label for="flavorShot">Flavor Shot</label>
            <select id="flavorShot" class="form-control" name="flavor">
              <option value="">None</option>
              <option value="caramel">Caramel</option>
              <option value="almond">Almond</option>
              <option value="mocha">Mocha</option>
            </select>
          </div>
        </form>
      </div>
    </div>
...
```

Каждый из элементов `<option>` соответствует одному из возможных значений, а элемент `<select>` задает `name`.

Сохраните файл `index.html` и проверьте, что выпадающее меню отображается корректно — со всеми добавленными нами пунктами (рис. 9.10).

По умолчанию выбран первый элемент `<option>`. Можно добавить в один из элементов `option` булев атрибут `selected`, если необходимо, чтобы автоматически выбирался какой-либо из них, отличный от первого.

Для первого пункта выпадающего меню мы задали атрибут `value` равным пустой строке. Если бы мы вообще опустили атрибут `value`, то браузер использовал бы в качестве значения строку `"None"`. Лучше всегда задавать значение атрибута `value`, чтобы не надеяться на то, что браузер сделает все так, как вы ожидаете.

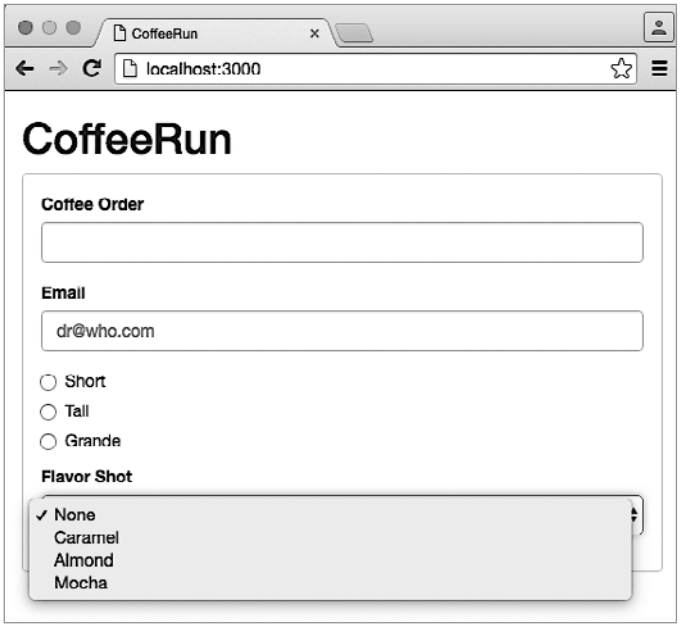


Рис. 9.10. Выпадающее меню ароматизаторов для кофе

Добавляем слайдер выбора диапазона

Не все любят убийственно крепкий кофе. Желательно, чтобы посетители могли выбрать крепость кофе в виде значения от 0 до 100. С другой стороны, нежелательно заставлять их вводить точное значение.

Для этого добавьте в файл `index.html` элемент `<input>` с типом `range`. С его помощью вы создали *слайдер выбора диапазона*. Элементы `<input>` и `<label>` должны быть связаны между собой и обернуты в тег `<div>` с классом `form-group`. Пожалейте покупателей кофе и установите значение по умолчанию равным 30.

```
...
            <option value="mocha">Mocha</option>
          </select>
        </div>
        <div class="form-group">
          <label for="strengthLevel">Caffeine Rating</label>
          <input name="strength" id="strengthLevel" type="range" value="30">
        </div>
      </form>
...
```

Сохраните файл `index.html` и опробуйте наш новый слайдер в браузере. Он будет выглядеть так, как показано на рис. 9.11.

Рис. 9.11. Слайдер для крепости кофе

Добавляем кнопки Submit и Reset

Последнее, что необходимо сделать в разметке, — добавить кнопку Submit (Подтвердить). Ради удобства использования следует также добавить кнопку Reset (Сброс) для очистки формы (на тот случай, если пользователь хочет оформить заказ с начала).

Обычно кнопка Submit — просто элемент `<input>`, тип которого — `submit`. Аналогично кнопка Reset — просто элемент `<input>`, тип которого — `reset`. Однако, чтобы воспользоваться CSS Bootstrap, мы будем применять элемент `<button>`.

Добавьте в файл `index.html` два элемента `<button>` с именем класса `btn btn-default`. Задайте тип первого равным `submit`, а второго — `reset`. Между открывающим и закрывающим тегами поместите в качестве описания Submit и Reset соответственно:

```
...
          <div class="form-group">
            <label for="strengthLevel">Caffeine Rating</label>
            <input name="strength" id="strengthLevel" type="range" value="30">
          </div>
          <button type="submit" class="btn btn-default">Submit</button>
          <button type="reset" class="btn btn-default">Reset</button>
        </form>
...
```

После сохранения изменений браузер добавит эти кнопки внизу формы (рис. 9.12).

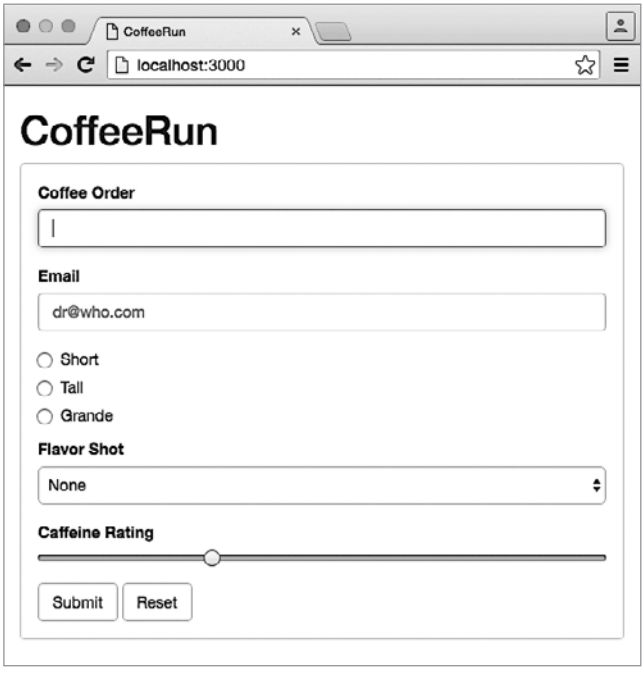


Рис. 9.12. Кнопки Submit и Reset

Кнопка Submit пока что ничего не будет делать. Этим мы займемся в следующей главе. Однако кнопка Reset будет сбрасывать все к значениям по умолчанию.

У этих кнопок есть два класса, которые могут показаться избыточными. Таковы условные обозначения фреймворка Bootstrap, служащие исключительно для стилизации. Класс `btn` обеспечивает все стандартные визуальные свойства кнопки Bootstrap, включая закругленные углы и поля. Класс `btn-default` добавляет белый цвет фона.

Мы воспользовались фреймворком пользовательского интерфейса Bootstrap для стилизации нашего приложения CoffeeRun. Благодаря паттерну Bootstrap (для разметки и имен классов) приложение будет иметь единообразный вид и адаптироваться к множеству размеров экранов и версий браузера.

Чтобы узнать больше о возможностях Bootstrap, загляните в его документацию на сайте `getbootstrap.com/css`.

Фреймворк Bootstrap особенно хорошо подходит для быстрой стилизации приложения, позволяя сосредоточить свое внимание на его логике. Именно ею мы и займемся в следующих главах.

10 Обработка форм с помощью JavaScript

Разработка приложения CoffeeRun началась неплохо. У него есть два модуля JavaScript для обработки его внутренней логики и HTML-форма, стилизованная с помощью фреймворка Bootstrap. В данной главе мы создадим более сложный модуль, связывающий форму с логикой, что даст возможность использовать форму для ввода заказов кофе.

Как вы помните из главы 2, браузеры взаимодействуют с серверами путем отправки запросов на информацию для конкретного URL. То есть для каждого файла, который браузер должен загрузить, он отправляет серверу запрос `GET` на данный файл.

Когда же браузеру нужно отправить информацию на сервер, например, если пользователь заполняет и отправляет форму, браузер берет данные формы и помещает их в запрос `POST`. Сервер получает запрос, обрабатывает данные, после чего отправляет обратно ответ (рис. 10.1).

В CoffeeRun нам не понадобится отправлять данные форм на сервер для обработки. Наши модули `Truck` и `DataStore` служат для той же цели, что и традиционный серверный код. Их задача заключается в обработке бизнес-логики и хранении данных для нашего приложения.

Поскольку этот код находится в браузере, а не на сервере, приходится перехватывать данные формы до их отправки. В этой главе специально для этой цели мы создадим новый модуль `FormHandler`. Кроме того, добавим в приложение CoffeeRun библиотеку jQuery, которая поможет нам в работе. При создании CoffeeRun в следующих нескольких главах мы воспользуемся и другими возможностями jQuery.

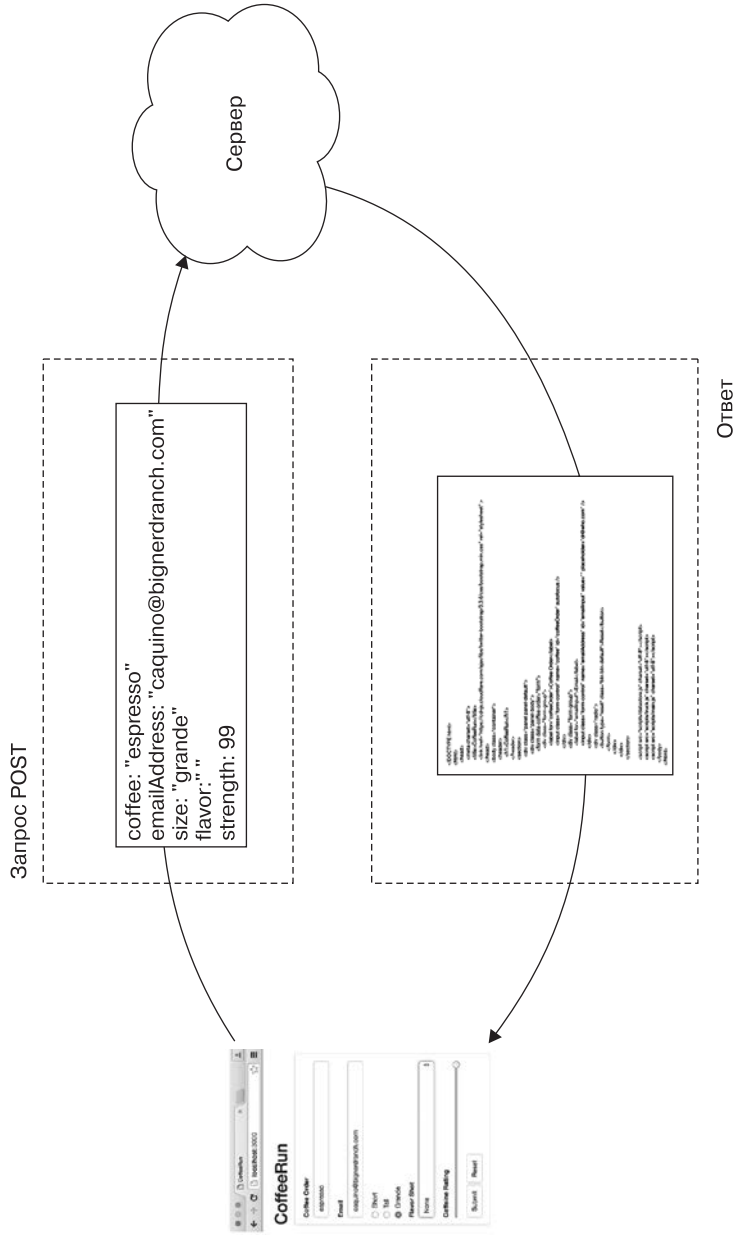


Рис. 10.1. Традиционная обработка форм на стороне сервера

Создаем модуль FormHandler

Модуль `FormHandler` будет предотвращать отправку браузером данных формы на сервер. Взамен он станет читать значения из формы при нажатии пользователем кнопки Submit. Затем он отправит эти данные экземпляру `Truck` с помощью метода `createOrder`, написанного нами в главе 8 (рис. 10.2).

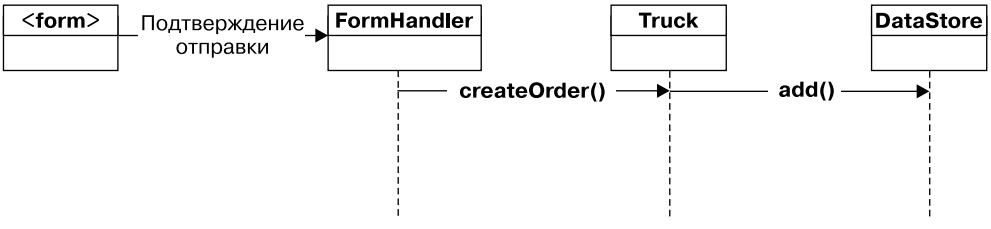


Рис. 10.2. Архитектура приложения CoffeeRun с App.FormHandler

Создайте новый файл с названием `formhandler.js` в каталоге `scripts` и добавьте для него тег `<script>` в файл `index.html`.

```
...
          </form>
        </div>
      </div>
    </section>
    <script src="scripts/formhandler.js" charset="utf-8"></script>
    <script src="scripts/datastore.js" charset="utf-8"></script>
    <script src="scripts/truck.js" charset="utf-8"></script>
    <script src="scripts/main.js" charset="utf-8"></script>
  </body>
</html>
```

Как и остальные наши модули, `FormHandler` будет использовать IIFE для инкапсуляции кода и связывания конструктора со свойством `window.App`.

Откройте файл `scripts/formhandler.js` и создайте IIFE. В IIFE создайте переменную `App` и присвойте ей имеющееся значение `window.App`. Если свойство `window.App` пока еще не существует, присвойте ей пустой объектный литерал. Объявите функцию — конструктор `FormHandler` — и экспортируйте ее в свойство `window.App`:

```
(function (window) {
  'use strict';
  var App = window.App || {};
  function FormHandler() {
    // Здесь будет находиться код
  }
  App.FormHandler = FormHandler;
  window.App = App;
})(window);
```

Пока что этот код соответствует привычному паттерну, который мы использовали в модулях `Truck` и `DataStore`. Однако скоро все изменится, поскольку он будет импортировать и использовать для своей работы библиотеку jQuery.

Введение в библиотеку jQuery

Библиотека jQuery была создана Джоном Резигом в 2006 году. Это одна из наиболее популярных универсальных библиотек JavaScript с открытым исходным кодом. Помимо прочего, она предоставляет удобную сокращенную запись для манипуляций DOM, создания элементов, взаимодействия с сервером и обработки событий.

Нелишне познакомиться поближе с jQuery, ведь с ее помощью было написано так много кода. Кроме того, многие библиотеки взяли за образец условные соглашения jQuery. По сути, библиотека jQuery непосредственно оказала влияние на API DOM (два примера этого влияния — методы `document.querySelector` и `document.querySelectorAll`).

Мы не будем детально рассматривать jQuery прямо сейчас. Вместо этого мы познакомим вас с различными ее возможностями по мере того, как это понадобится для создания более сложных частей CoffeeRun. Если же вы захотите изучить ее глубже, то всегда можете заглянуть в документацию по адресу jquery.com.

Аналогично тому, как мы поступили с фреймворком Bootstrap, мы добавим копию библиотеки jQuery в наш проект с сайта cdnjs.com. Перейдите по адресу cdnjs.com/libraries/jquery, чтобы найти версию 2.1.4, и скопируйте ее адрес (там может быть более новая версия, но лучше воспользуйтесь версией 2.1.4, чтобы избежать каких-либо проблем совместимости).

Добавьте jQuery в тег `<script>` в файле `index.html`.

```
...
      </div>
    </section>
    <script src="https://cdnjs.cloudflare.com/ajax/libs/jquery/2.1.4/
            jquery.min.js" charset="utf-8"></script>
    <script src="scripts/formhandler.js" charset="utf-8"></script>
    <script src="scripts/datastore.js" charset="utf-8"></script>
    <script src="scripts/truck.js" charset="utf-8"></script>
    <script src="scripts/main.js" charset="utf-8"></script>
...
```

Сохраните файл `index.html`.

Импортируем библиотеку jQuery

Модуль `FormHandler` будет импортировать jQuery аналогично тому, как он импортировал свойство `App`. Причина в том, чтобы сделать явным использование нашим модулем кода, описанного где-то в другом месте. Это рекомендуемая практика для

согласованной работы с другими членами команды и для будущего сопровождения проекта.

Создайте в файле `formhandler.js` локальную переменную `$` и присвойте ей значение `window.jQuery`:

```
(function (window) {
  'use strict';
  var App = window.App || {};
  var $ = window.jQuery;

  function FormHandler() {
    // Здесь будет находиться код
  }

  App.FormHandler = FormHandler;
  window.App = App;
})(window);
```

При добавлении тега `<script>` для библиотеки jQuery создается функция с именем `jQuery`, а также указывающая на нее переменная с именем `$`. Большинство разработчиков предпочитают использовать в своем коде `$`. Если придерживаться этой практики, необходимо импортировать `window.jQuery` и присвоить его локальной переменной `$`.

Недоумеваете, почему для имени переменной используется `$`? Переменные JavaScript могут содержать буквы, цифры, знак подчеркивания (`_`) и знак доллара (`$`). Отметим, что *начинаться* они могут только с буквы, знака подчеркивания (`_`) или знака доллара (`$`), но не с цифры. Создатель библиотеки jQuery выбрал имя переменной `$` из-за его краткости и малой вероятности, что оно будет применяться в другом коде проекта.

Конфигурация экземпляров модуля FormHandler с помощью селекторов

Необходимо, чтобы наш модуль `FormHandler` можно было использовать с любым элементом `<form>`. Чтобы добиться этого, мы будем передавать в конструктор модуля `FormHandler` селектор, соответствующий элементу `<form>` в файле `index.html`.

Изменим файл `formhandler.js`, добавляя параметр `selector` в конструктор `FormHandler`. Генерируем экземпляр `Error`, если он не передан:

```
(function (window) {
  'use strict';
  var App = window.App || {};
  var $ = window.jQuery;

  function FormHandler(selector) {
```

```
    // Здесь будет находиться код
    if (!selector) {
      throw new Error('No selector provided');
    }
  }

  App.FormHandler = FormHandler;
  window.App = App;
})(window);
```

`Error` — встроенный тип, извещающий, что в коде имеет место неожиданное значение или ситуация. Пока что наш экземпляр `Error` будет просто выводить сообщение в консоли.

Сохраните и попробуйте создать новый объект `FormHandler` без передачи ему аргумента (рис. 10.3).

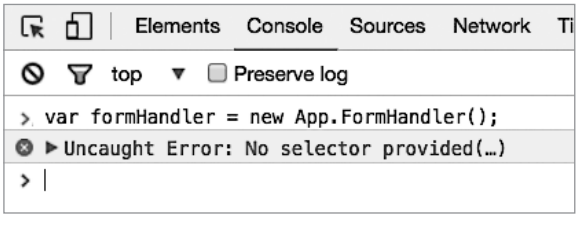


Рис. 10.3. Создание нового объекта FormHandler без передачи ему аргумента

Это первый шаг на пути к тому, чтобы сделать модуль `FormHandler` более пригодным для повторной работы. В приложении Ottergram мы создавали переменные для используемых в коде DOM селекторов. С модулем `FormHandler` мы так делать не будем. Вместо этого мы будем использовать переданный в конструктор селектор, а также библиотеку jQuery для поиска подходящих элементов.

jQuery чаще всего применяется для поиска элементов в DOM. Для этого вызывается функция `$`, которой передается селектор в виде строки. Фактически она используется так же, как мы использовали метод `document.querySelectorAll` (хотя jQuery внутри работает иначе, мы скоро об этом расскажем). Это часто называют выборкой элементов из DOM с помощью библиотеки jQuery.

Объявите в файле `formhandler.js` переменную `$formElement`. Затем найдите соответствующий элемент в DOM с помощью этого селектора и присвойте полученный результат переменной `this.$formElement`:

```
(function (window) {
  'use strict';
  var App = window.App || {};
  var $ = window.jQuery;

  function FormHandler(selector) {
```

```
    if (!selector) {
      throw new Error('No selector provided');
    }

    this.$formElement = $(selector);
  }

  App.FormHandler = FormHandler;
  window.App = App;

})(window);
```

Знак `$` перед именем переменной указывает, что она ссылается на выбранные с помощью jQuery элементы. Этот префикс необязателен при работе с библиотекой jQuery, но является распространенным условным обозначением, используемым многими разработчиками клиентской части.

При использовании функции `$` библиотеки jQuery для выборки элементов она не возвращает ссылки на элементы DOM подобно тому, как это делает метод `document.querySelectorAll`. Вместо этого она возвращает один *объект*, который содержит ссылки на выбранные элементы. У него имеются также специальные методы для работы с коллекцией ссылок. Этот объект называют выборкой, обернутой jQuery, или коллекцией, обернутой jQuery.

Нам хотелось бы быть уверенными, что выборка успешно извлекает элемент из DOM. jQuery будет возвращать пустую выборку, если ничего не нашла (если селектор ничему не соответствует, ошибки генерироваться не будет). Придется проверять это вручную, ведь функция `FormHandler` не сможет выполнить свою задачу без элемента.

Свойство `length` обернутой jQuery выборки сообщает, сколько подходящих элементов было найдено. Поменяйте файл `formhandler.js` так, чтобы проверять свойство `length` у `this.$formElement`. Если оно равно `0`, необходимо сгенерировать `Error`.

```
(function (window) {
  'use strict';
  var App = window.App || {};
  var $ = window.jQuery;

  function FormHandler(selector) {
    if (!selector) {
      throw new Error('No selector provided');
    }

    this.$formElement = $(selector);
    if (this.$formElement.length === 0) {
      throw new Error('Could not find element with selector: ' + selector);
    }
  }

  App.FormHandler = FormHandler;
```

```
  window.App = App;

})(window);
```

Наш конструктор `FormHandler` можно настроить так, чтобы он работал с любыми элементами `<form>` в зависимости от переданного селектора. Он также будет хранить ссылку на этот элемент `<form>` в виде переменной экземпляра. Благодаря этому код гарантированно не должен будет выполнять лишние запросы к DOM. Это рекомендуемая практика с точки зрения производительности (альтернативный вариант: вызывать функцию `$` снова и снова, выбирая при этом одни и те же элементы).

Добавляем обработчик события submit

Следующий этап разработки модуля `FormHandler` — добавить прослушивание на предмет события `submit` (подтверждение отправки) для элемента `<form>` и выполнение обратного вызова при его наступлении.

Чтобы модуль `FormHandler` годился для повторного использования, мы не будем «жестко зашивать» в него код обработчика `submit`. Вместо этого напишем метод, который принимает на входе функциональный аргумент, добавляет прослушиватель `submit`, после чего вызывает вышеупомянутый функциональный аргумент внутри прослушивателя.

Во-первых, добавьте в файл `formhandler.js` метод прототипа `addSubmitHandler`:

```
...
    if (this.$formElement.length === 0) {
      throw new Error('Could not find element with selector: ' + selector);
    }
  }

  FormHandler.prototype.addSubmitHandler = function () {
    console.log('Setting submit handler for form');
    // Здесь будет находиться еще код
  };

  App.FormHandler = FormHandler;
...
```

Вместо использования метода `addEventListener` (как в Ottergram) мы воспользуемся методом `on` библиотеки jQuery. Он похож на метод `addEventListener`, но предоставляет дополнительные возможности. Пока что, однако, мы будем использовать так же, как использовали бы метод `addEventListener` (некоторым дополнительным возможностям мы найдем применение в следующей главе).

```
...
    if (this.$formElement.length === 0) {
      throw new Error('Could not find element with selector: ' + selector);
```

```
    }
  }

  FormHandler.prototype.addSubmitHandler = function () {
    console.log('Setting submit handler for form');
    // Здесь будет находиться еще код
    this.$formElement.on('submit', function (event) {
      event.preventDefault();
    });
  };
...
```

Метод `on` принимает на входе имя события и обратный вызов, который необходимо будет запустить при срабатывании события. Мы вызвали функцию `event.preventDefault` — это гарантирует, что подтверждение отправки формы не уведет пользователя со страницы CoffeeRun (мы делали то же самое со ссылками миниатюр в Ottergram).

Извлечение данных

При отправке формы наш код должен читать введенные пользователем данные из формы, после чего *делать* что-то с ними. Создайте в файле `formhandler.js` (в обработчике `submit`) новую переменную `data`. Присвойте ей объект-литерал. Она будет содержать значение (`value`) каждого элемента формы.

```
...
  FormHandler.prototype.addSubmitHandler = function () {
    console.log('Setting submit handler for form');
    this.$formElement.on('submit', function (event) {
      event.preventDefault();

      var data = $(this).serializeArray();
      console.log(data);

    });
  };
...
```

Внутри обратного вызова нашего обработчика `submit` объект `this` является ссылкой на элемент `form`. Библиотека jQuery предоставляет удобный метод `serializeArray` для извлечения данных из формы. Чтобы использовать `serializeArray`, необходимо обернуть форму с помощью jQuery. Благодаря вызову `$(this)` мы получаем обернутый объект, у которого есть доступ к методу `serializeArray`.

Метод `serializeArray` возвращает данные формы в виде массива объектов. Мы присвоили его временной переменной `data` и вывели в консоль. Чтобы лучше понять, что представляет собой `serializeArray`, сохраните файл и выполните следующий код в консоли:

```
var fh = new App.FormHandler('[data-coffee-order="form"]');
fh.addSubmitHandler();
```

Далее заполните форму какими-нибудь тестовыми данными и нажмите кнопку `Submit`. Вы увидите выведенный в консоль массив. Щелкните на ▶ рядом с двумя из элементов `Object` массива. Результат будет похож на показанный на рис. 10.4.

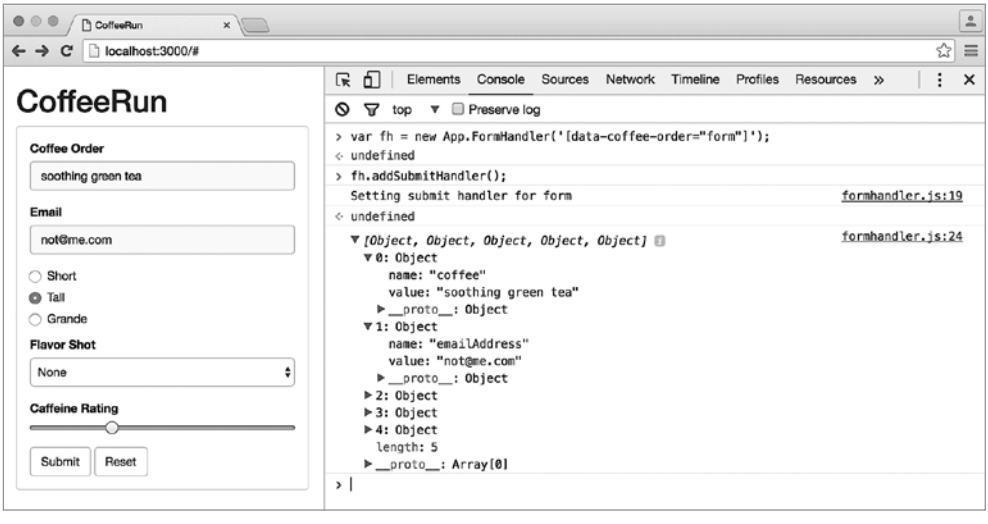


Рис. 10.4. Метод serializeArray возвращает данные формы в виде массива объектов

Вы можете видеть, что у каждого объекта массива есть ключ, соответствующий атрибуту `name` элемента `<form>`, и введенное пользователем значение `value` для этого элемента.

Теперь мы можем организовать цикл по этому массиву и скопировать значения всех элементов. Добавьте в метод `serializeArray` в файле `formhandler.js` вызов метода `forEach` и передайте в него обратный вызов. Поскольку обратный вызов выполняется для каждого объекта массива, он будет использовать свойства `name` и `value` объекта для создания нового свойства объекта `data`.

```
...
  FormHandler.prototype.addSubmitHandler = function () {
    console.log('Setting submit handler for form');
    this.$formElement.on('submit', function (event) {
      event.preventDefault();

      var data = ~~$(this).serializeArray();~~ {};
      $(this).serializeArray().forEach(function (item) {
        data[item.name] = item.value;
        console.log(item.name + ' is ' + item.value);
      });
```

```
      console.log(data);

    });
  };
...
```

Чтобы увидеть это в действии, сохраните изменения и снова выполните тестовый код в консоли, прежде чем заполнять форму:

```
var fh = new App.FormHandler('[data-coffee-order="form"]');
fh.addSubmitHandler();
```

После заполнения формы и нажатия кнопки Submit вы должны увидеть, что введенная информация скопирована в объект `data` и выведена в консоль (рис. 10.5).

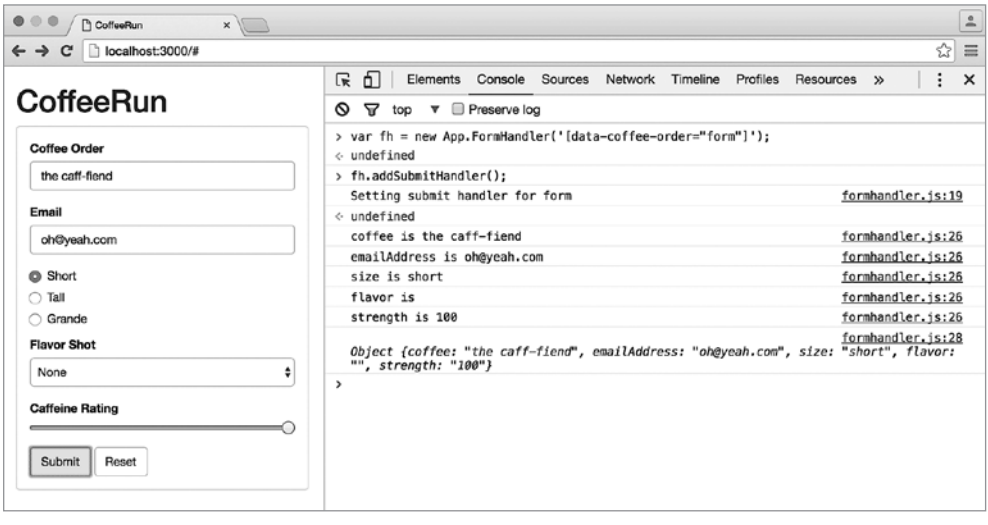


Рис. 10.5. Данные формы скопированы в обратный вызов итератора

Прием и выполнение функции обратного вызова

Теперь, когда мы получили данные формы в виде единого объекта, необходимо передать этот объект в метод `createOrder` нашего экземпляра `Truck`. Но у экземпляра `FormHandler` нет доступа к экземпляру `Truck` (и создание нового экземпляра `Truck` делу не поможет).

Эту проблему можно решить, добавив в метод `addSubmitHandler` параметр-функцию, которую он сможет вызвать в обработчике события.

В файле `formhandler.js` добавьте параметр `fn`:

```
...
  FormHandler.prototype.addSubmitHandler = function (fn) {
```

```
    console.log('Setting submit handler for form');
    this.$formElement.on('submit', function (event) {
      event.preventDefault();
...
```

Обратный вызов обработчика `submit` будет выполняться всякий раз при срабатывании события `submit` в браузере. В этот момент должна вызываться функция `fn`.

Вызовем функцию `fn` внутри обратного вызова обработчика `submit` в файле `formhandler.js` и передадим ей объект `data`, содержащий введенные пользователем данные:

```
...
  FormHandler.prototype.addSubmitHandler = function (fn) {
    ...
      console.log(data);
      fn(data);
    });
  };
...
```

Теперь, когда экземпляр `FormHandler` создан, можно передать `addSubmitHandler` любой обратный вызов. После этого при подтверждении отправки формы будет выполняться обратный вызов с передачей всех данных, введенных пользователем в форму.

Использование экземпляра FormHandler

Нам необходимо создать в файле `main.js` экземпляр `FormHandler` и передать ему селектор для элемента `<form>`: `[data-coffee-order="form"]`. Создайте для этого селектора переменную вверху файла `main.js`, чтобы при необходимости можно было использовать его повторно:

```
(function (window) {
  'use strict';
  var FORM_SELECTOR = '[data-coffee-order="form"]';
  var App = window.App;
...
```

Далее создайте локальную переменную `FormHandler` и присвойте ей значение `App.FormHandler`:

```
(function (window) {
  'use strict';
  var FORM_SELECTOR = '[data-coffee-order="form"]';
  var App = window.App;
  var Truck = App.Truck;
  var DataStore = App.DataStore;
  var FormHandler = App.FormHandler;
  var myTruck = new Truck('ncc-1701', new DataStore());
  ...
```

В конце модуля `main.js` вызовите конструктор `FormHandler` и передайте ему переменную `FORM_SELECTOR`. Благодаря этому экземпляр `FormHandler` точно будет работать с соответствующим данному селектору элементом DOM. Присвойте этот экземпляр новой переменной `FormHandler`:

```
...
  var Truck = App.Truck;
  var DataStore = App.DataStore;
  var FormHandler = App.FormHandler;
  var myTruck = new Truck('ncc-1701', new DataStore());
  window.myTruck = myTruck;
  var formHandler = new FormHandler(FORM_SELECTOR);

  formHandler.addSubmitHandler();
  console.log(formHandler);
})(window);
```

Когда вы сохраните код и вернетесь в браузер, консоль выведет `Setting submit handler for form`, демонстрируя, что при загрузке страницы был вызван метод `addSubmitHandler`. Однако если вы заполните форму и подтвердите ее отправку, то получите ошибку (рис. 10.6).

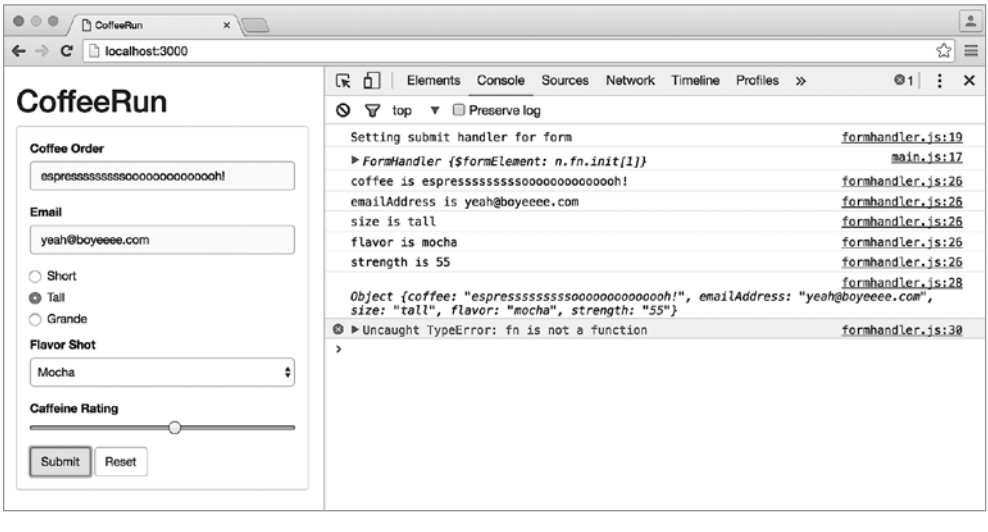


Рис. 10.6. Вызов метода addSubmitHandler при загрузке страницы

Это происходит потому, что мы пока ничего не передаем методу `addSubmitHandler`. Мы исправим это в следующем разделе.

Регистрация метода createOrder в качестве обработчика события submit. Нам хотелось бы, чтобы метод `createOrder` вызывался каждый раз, когда происходит событие `submit`. Но мы не можем просто передавать ссылку на `createOrder` методу

`formHandler.addSubmitHandler`. Причина в том, что при вызове `createOrder` внутри обратного вызова обработчика события меняется его *владелец*. При другом владельце значением переменной `this` внутри тела метода `createOrder` уже не будет экземпляр `Truck`, что приводит к ошибке при вызове `createOrder`.

Вместо этого мы передадим методу `formHandler.addSubmitHandler` связанную ссылку на `myTruck.createOrder`.

Внесите в файл `formhandler.js` следующие изменения. Не забудьте привязать (`bind`) ссылку на метод так, чтобы ее владельцем гарантированно оказался `myTruck`.

```
...
  window.myTruck = myTruck;
  var formHandler = new FormHandler(FORM_SELECTOR);

  formHandler.addSubmitHandler(myTruck.createOrder.bind(myTruck));
  console.log(formHandler);
})(window);
```

Могли ли мы просто добавить вызов метода `bind` в описание исходного метода прототипа? При описании методов прототипа у нас есть доступ к экземпляру, но только *внутри* тела метода. Для `bind` требуется, чтобы у нас была ссылка на намеченного владельца вызова (ссылка должна быть доступна вне тела метода). Поскольку у нас нет возможности сослаться на экземпляр снаружи тела метода, мы не можем привязать (`bind`) исходный метод прототипа.

Сохраните и заполните форму. После подтверждения ее отправки у нас будет возможность вызвать метод `myTruck.printOrders` и увидеть, что введенные в форму данные были добавлены в список заказов, ожидающих выполнения, как показано на рис. 10.7.

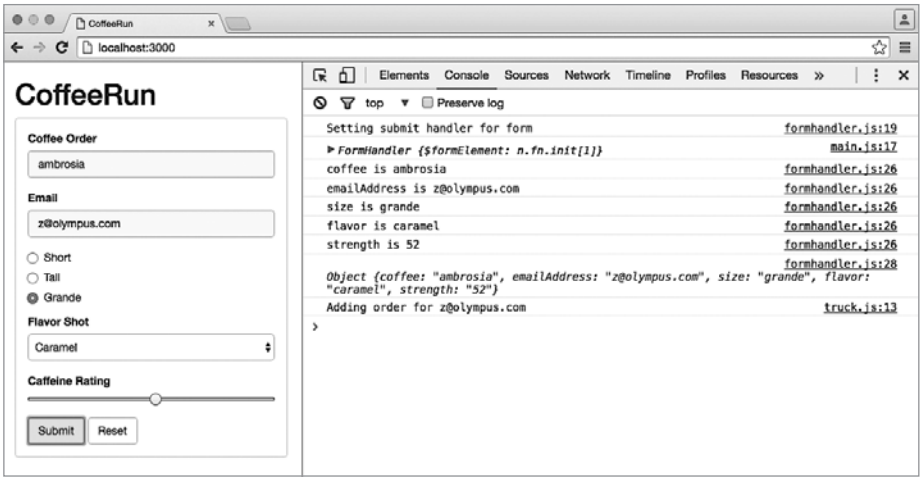


Рис. 10.7. При подтверждении отправки формы вызывается метод createOrder

Расширения UI

Было бы неплохо, если бы форма очищалась ото всех старых данных при отправке, чтобы пользователь мог немедленно начать вводить следующий заказ. Для сброса формы достаточно вызвать метод `reset` элемента `<form>`.

Найдите метод `FormHandler.prototype.addSubmitHandler` в файле `formhandler.js`. Добавьте в конце обратного вызова `this.$formElement.on('submit'...)` вызов метода `reset` формы:

```
...
  FormHandler.prototype.addSubmitHandler = function (fn) {
    console.log('Setting submit handler for form');
    this.$formElement.on('submit', function (event) {
      event.preventDefault();

      var data = {};
      $(this).serializeArray().forEach(function (item) {
        data[item.name] = item.value;
        console.log(item.name + ' is ' + item.value);
      });
      console.log(data);
      fn(data);
      this.reset();
    });
  };
...
```

Сохраните и введите в форму какие-нибудь данные. После подтверждения отправки формы вы должны увидеть, что данные были очищены.

Наконец, последний штрих в UI. Когда поле формы готово к вводу, на нем находится фокус, как мы видели в прошлой главе. Чтобы установить фокус на конкретное поле формы, можно вызвать его метод `focus` (добавленный нами в поле заказа кофе атрибут `autofocus` срабатывает только при первоначальной загрузке страницы).

Удобно работать с отдельными полями формы с помощью свойства `elements` формы. Свойство `elements` представляет собой массив полей формы, к которым можно обращаться по их позициям в массиве, начиная с 0.

В файле `formhandler.js` сразу после обращения к методу `this.reset` в обратном вызове обработчика события `submit` вызовите метод `focus` для первого поля:

```
...
  FormHandler.prototype.addSubmitHandler = function (fn) {
    console.log('Setting submit handler for form');
    this.$formElement.on('submit', function (event) {
      event.preventDefault();

      var data = {};
```

```
      $(this).serializeArray().forEach(function (item) {
        data[item.name] = item.value;
        console.log(item.name + ' is ' + item.value);
      });
      console.log(data);
      fn(data);
      this.reset();
      this.elements[0].focus();
    });
  };
...
```

CoffeeRun теперь работает при поддержке библиотеки jQuery и может принимать вводимые пользователем данные. Мы ликвидировали разрыв между нашим HTML-кодом и модулями JavaScript. В следующей главе дополним картину, создав интерактивные элементы DOM, работающие на основе перехватываемых данных формы.

Бронзовое упражнение: порции огромного размера

Добавьте еще один вариант размера порции для заказа кофе — с названием, намекающим на огромный размер, например «Кофе-зилла».

Добавьте новый заказ с этим размером порции и проверьте данные приложения в консоли, чтобы убедиться, что он был сохранен корректно.

Серебряное упражнение: отображение значения при изменении слайдера

Создайте обработчик для события изменения (`change`) слайдера. При изменении значения слайдера отображайте число рядом с меткой для слайдера.

В качестве дополнительного упражнения реализуйте изменение цвета числа (или метки) для отражения крепости кофе. Используйте зеленый цвет для слабого кофе, желтый — для обычного и красный — для крепкого.

Золотое упражнение: добавляем достижения

Когда пользователи подтверждают заказ на самую большую порцию самого крепкого кофе с ароматизатором, сообщите им о новом достижении, выведя модальное окно фреймворка Bootstrap с извещением о невероятной крепости напитка и об их приверженности определенному вкусу. Спросите, хотят ли они воспользоваться

своим достижением, и если ответ положительный — добавьте дополнительное поле формы, которое видно, только если адрес электронной почты введен в поле `email`. Оно должно предоставлять возможность выбора из одного или нескольких вариантов дальнейших усовершенствований их кофе, например путешествие во времени, чтение мыслей или даже код без ошибок.

Обратитесь к документации на сайте `getbootstrap.com/javascript/` за информацией о том, как включить и заставить работать модальное поведение фреймворка Bootstrap (вам понадобится добавить тег `<script>` с сайта `cdnjs.com` для JavaScript-кода Bootstrap).

11 От данных к DOM

В предыдущей главе мы создали модуль `FormHandler`. Он связывает форму, с которой взаимодействует пользователь, с остальной частью нашего кода. Перехватывая событие `submit`, мы отправляем вводимые пользователем данные в модуль `Truck`, который затем сохраняет их в своем экземпляре `DataStore`.

В данной главе мы создадим еще один фрагмент кода пользовательского интерфейса — модуль `CheckList`. Аналогично модулю `Truck`, он будет получать данные от `FormHandler`, но его задача заключается в добавлении на страницу перечня заказов, ожидающих выполнения. При щелчке кнопкой мыши на пункте перечня `CheckList` будет убирать этот пункт со страницы и посылать сигнал `Truck` о необходимости удалить его из экземпляра `DataStore`. Рисунок 11.1 демонстрирует приложение CoffeeRun, оснащенное перечнем заказов, ожидающих выполнения.

Настраиваем перечень заказов

Мы продолжим использовать классы фреймворка `Bootstrap` для стилизации элементов формы. Начнем с добавления в файл `index.html` двух элементов `<div>` с именами классов `Bootstrap` `panel`, `panel-default` и `panel-body`, как мы сделали для формы заказа кофе. Внутри них добавим заголовок и еще один тег `<div>`, в котором будут размещаться сами элементы перечня. Эту разметку необходимо добавить после тех элементов `<div>`, в которых находится наша форма.

```
...
    <header>
      <h1>CoffeeRun</h1>
    </header>
    <section>
      <div class="panel panel-default">
        <div class="panel-body">
          <form data-coffee-order="form">
            ...
         </form>
```

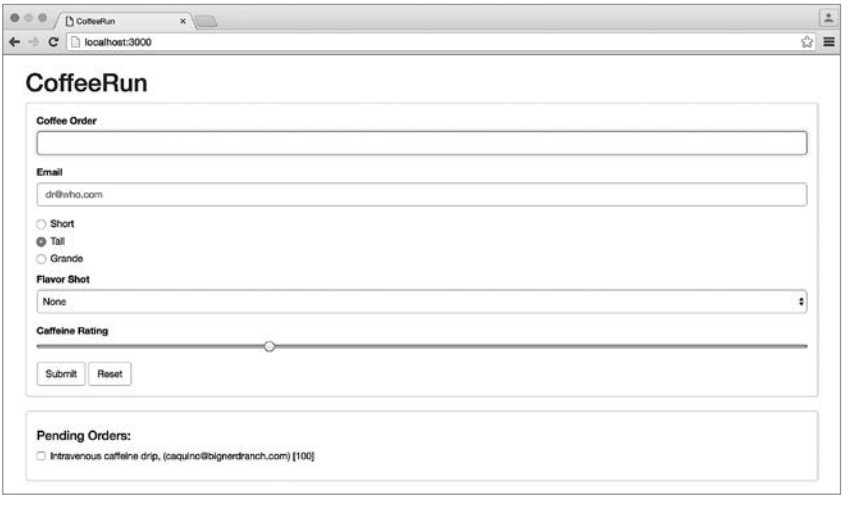


Рис. 11.1. Заказы все поступают и поступают!

```
        </div>
      </div>

      <div class="panel panel-default">
        <div class="panel-body">
          <h4>Pending Orders:</h4>
          <div data-coffee-order="checklist">
          </div>
        </div>
      </div>
    </section>
...
```

Как и ранее, мы добавили элементы `<div>`, в которых будут содержаться стили, обеспечиваемые фреймворком Bootstrap. Основная часть нашего перечня — элемент `[data-coffee-order="checklist"]`. Именно он станет целевым объектом для кода JavaScript, создающего отдельные пункты перечня заказов кофе и добавляющего их в DOM.

Сохраните файл `index.html`, запустите утилиту browser-sync и убедитесь, что CoffeeRun отображает пустую область Pending Orders (рис. 11.2).

Рис. 11.2. Результат после добавления разметки для элементов перечня

Теперь мы готовы углубиться в код JavaScript.

Создание модуля CheckList

Создайте новый файл `checklist.js` в каталоге `scripts` и добавьте ссылку на него в файл `index.html`:

```
...
    <script src="https://cdnjs.cloudflare.com/ajax/libs/jquery/2.1.4/jquery.js"
      charset="utf-8"></script>
    <script src="scripts/checklist.js" charset="utf-8"></script>
    <script src="scripts/formhandler.js" charset="utf-8"></script>
    <script src="scripts/datastore.js" charset="utf-8"></script>
    <script src="scripts/truck.js" charset="utf-8"></script>
    <script src="scripts/main.js" charset="utf-8"></script>
...
```

Сохраните файл `index.html`. Добавьте в файл `checklist.js` обычный код модуля с помощью IIFE. Импортируйте пространство имен `App` и jQuery, присвоив и то и другое локальной переменной. Создайте конструктор `CheckList`, не забыв передать в него селектор, который соответствует хотя бы одному элементу в DOM. В конце IIFE экспортируйте конструктор `CheckList` как часть пространства имен `App`.

```
(function (window) {
  'use strict';

  var App = window.App || {};
  var $ = window.jQuery;

  function CheckList(selector) {
    if (!selector) {
      throw new Error('No selector provided');
    }

    this.$element = $(selector);
    if (this.$element.length === 0) {
      throw new Error('Could not find element with selector: ' + selector);
    }
  }

  App.CheckList = CheckList;
  window.App = App;
})(window);
```

Чтобы выполнять свою задачу, модулю `CheckList` понадобятся три метода. Один будет служить для создания пункта перечня, включая кнопки-флажки и текстовое описание. Можно рассматривать эту группу элементов как строку таблицы.

Второй метод будет удалять строку из таблицы, а третий — добавлять прослушиватель для событий `click`, чтобы наш код знал, *когда* удалять строки.

Сначала мы займемся методом, создающим строку для нового заказа. На рис. 11.3 показано, каким образом модуль `CheckList` будет добавлять элементы на страницу при подтверждении отправки формы.

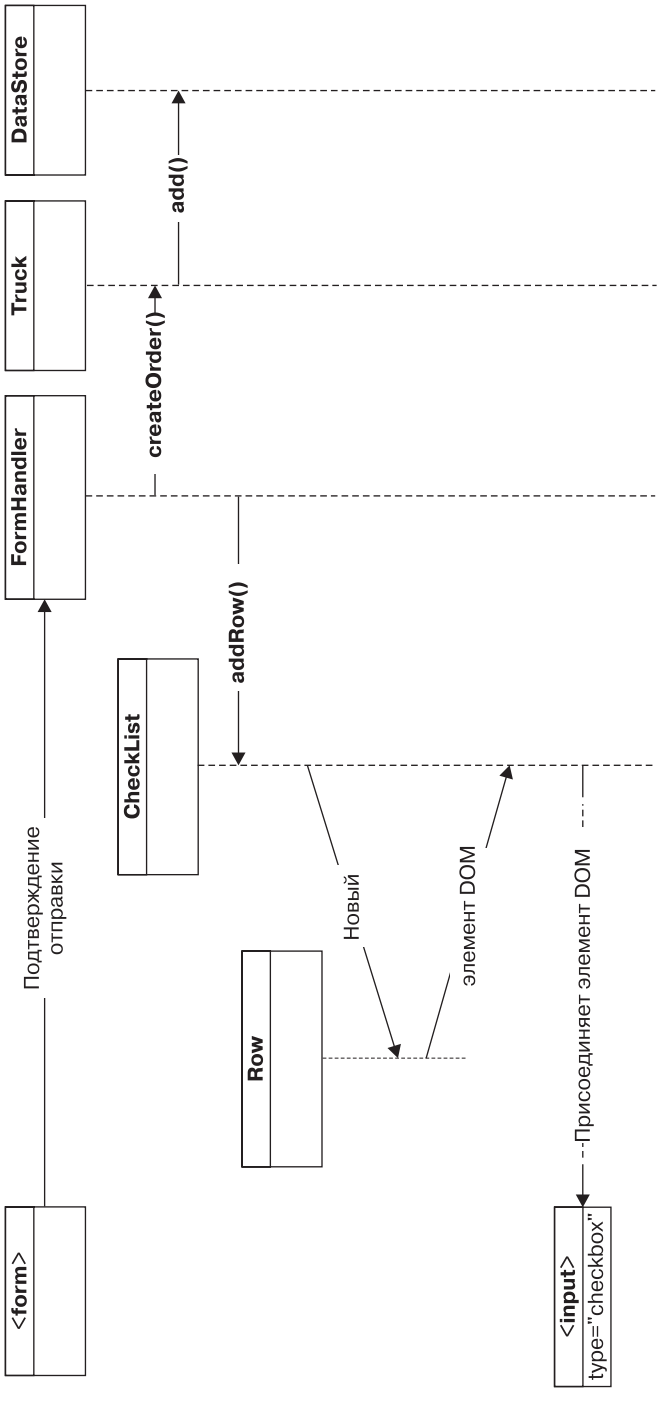


Рис. 11.3. Очередность событий при подтверждении отправки формы

Создание конструктора Row

Мы не можем создать разметку для пунктов перечня в файле `index.html`, поскольку их нужно добавить после визуализации страницы в ответ на подтверждение отправки формы. Вместо этого мы добавим в модуль `CheckList` конструктор `Row`.

Конструктор `Row` будет отвечать за создание всех элементов DOM, необходимых для представления отдельного заказа кофе, включая кнопку-флажок и текстовое описание. Но конструктор `Row` не будет экспортироваться в пространство имен `App`. Он будет использоваться только внутри — одним из методов прототипа `CheckList.prototype`.

Добавьте конструктор `Row` в файл `checklist.js` прямо перед оператором `App.CheckList = CheckList;`. Он должен принимать на входе аргумент `coffeeOrder`, представляющий собой те же самые данные, которые передаются в метод `Truck.prototype.createOrder`.

```
...
    this.$element = $(selector);
    if (this.$element.length === 0) {
      throw new Error('Could not find element with selector: ' + selector);
    }
  }

  function Row(coffeeOrder) {
    // Здесь будет находиться код конструктора
  }

  App.CheckList = CheckList;
  window.App = App;
})(window);
```

Создаем элементы DOM с помощью библиотеки jQuery. Наш конструктор `Row` будет использовать библиотеку jQuery для создания элементов DOM. Мы объявим переменные для отдельных элементов, составляющих элемент перечня. После этого конструктор объединит их в *поддерево* элементов DOM, как показано на рис. 11.4. `CheckList` присоединит это поддерево к дереву DOM страницы в качестве дочернего по отношению к `[data-coffee-order="checklist"]` элемента.

(`[39x]` в описании заказа обозначает крепость кофе.)

Созданное конструктором `Row` поддерево DOM на рис. 11.4 соответствует следующей разметке:

```
<div data-coffee-order="checkbox" class="checkbox">
  <label>
    <input type="checkbox" value="chewie@rrwwwgg.com">
    tall mocha iced coffee, (chewie@rrwwwgg.com) [39x]
  </label>
</div>
```

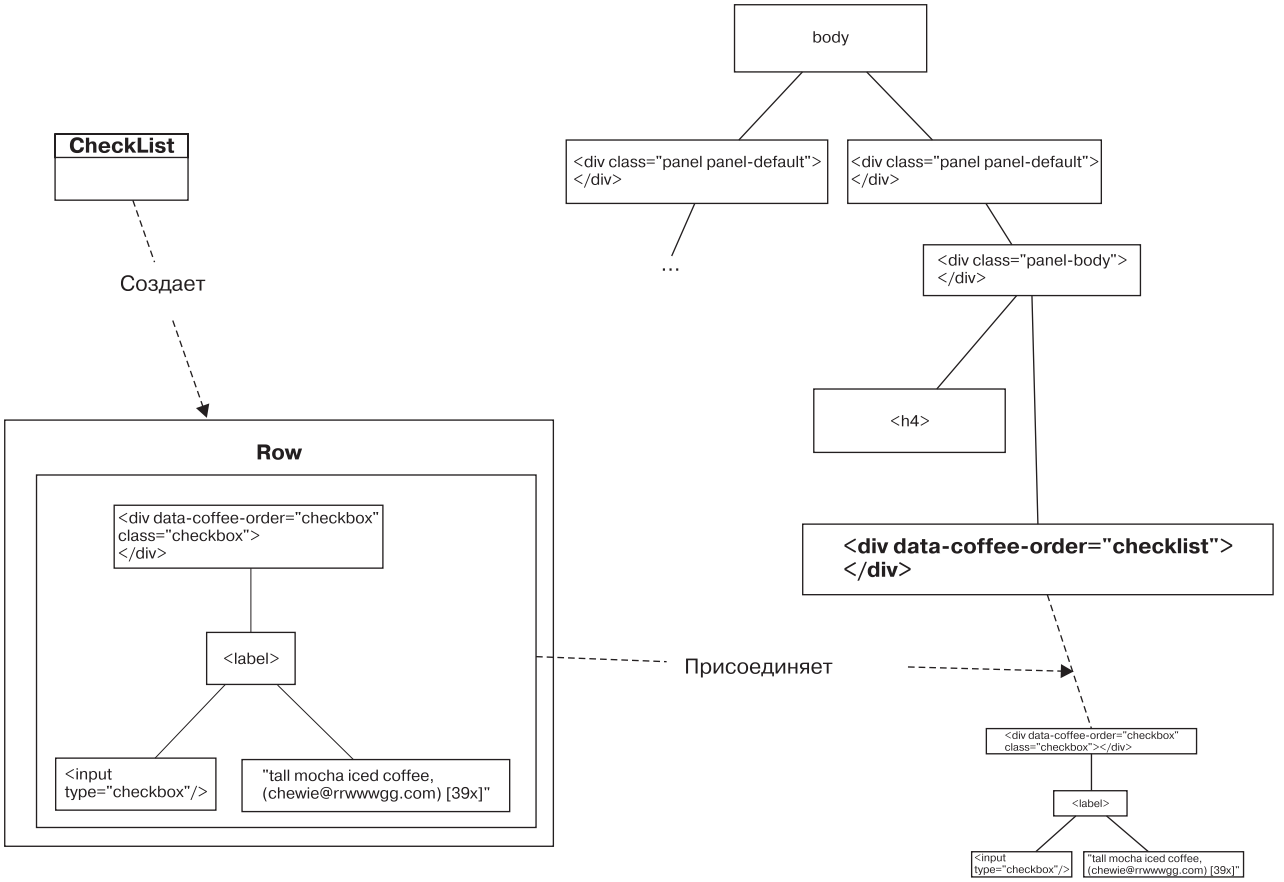


Рис. 11.4. CheckList создает строку и присоединяет ее элементы DOM

Тег `<div>` с классом `checkbox` используется для размещения элементов `<label>` и `<input>`. Класс `checkbox` служит для применения к `<div>` соответствующих стилей фреймворка `Bootstrap`. Атрибут `data-coffee-order` будет использоваться в нашем коде JavaScript, когда понадобится запустить для флажка действие щелчка кнопкой мыши.

Обратите внимание, что значение атрибута `type` нашего поля `<input>` тоже равно `checkbox`. Так мы сообщаем браузеру о необходимости отрисовать поле ввода в виде элемента формы — флажка. Текстовое описание заказа следует сразу за полем `<input>`. Тег `<Label>` содержит как поле `input` в виде флажка, так и текстовое описание. Это превращает текст и `input` в целевой объект флажка для щелчка кнопкой мыши.

Мы создадим элементы `<label>`, `<div>` и `<input>` по очереди. После этого вручную поместим их внутрь друг друга для создания поддерева DOM, которое присоединим к активному дереву DOM (тому, которое отображается в текущий момент на странице). Мы также создадим строчную переменную для текстового описания заказа, например `"tall mocha iced coffee, (chewie@rrwwwgg.com) [39x]."` (средняя порция кофе мокко со льдом (chewie@rrwwwgg.com) [39x]).

Для создания этих элементов обратимся к функции `$` библиотеки jQuery. До сих пор мы использовали функцию `$` только для выборки элементов из DOM, но она подходит и для их создания.

Во-первых, мы хотим создать элемент `<div>` путем вызова в файле `checklist.js` функции `$` в конструкторе `Row`. Мы будем передавать ей два аргумента, описывающих создаваемый элемент DOM. Первый аргумент будет строкой с тегом HTML элемента DOM, в данном случае `'<div></div>'`. Второй аргумент сделаем объектом, задающим атрибуты, которые библиотека jQuery должна будет добавить в элемент `<div>`. Пары «ключ/значение» объекта-литерала будут преобразовываться в атрибуты нового элемента.

В результате мы получим созданный jQuery элемент DOM, который присвоим новой переменной `$div`. Это будет не переменная экземпляра (это `$div`, а не `this.$div`). Стоящий перед ней знак `$` указывает на то, что это не простой элемент DOM, а тот, ссылку на который создала библиотека jQuery.

Выполните все это в файле `checklist.js`:

```
...
  function Row(coffeeOrder) {
    // Здесь будет находиться код конструктора
    var $div = $('<div></div>', {
      'data-coffee-order': 'checkbox',
      'class': 'checkbox'
    });
  }
...
```

Обратите внимание, что имена наших двух свойств заключены в одинарные кавычки. Вы могли бы подумать, что необходимо всегда использовать одинарные кавычки для имен свойств при создании элементов DOM с помощью библиотеки jQuery, но это не так. В кавычки должны заключаться имена свойств, содержащие специальные символы (например, тире), в противном случае это считается синтаксической ошибкой. Допустимые символы, которые можно использовать в имени свойства (или переменной) без одинарных кавычек, — буквы алфавита, цифры, знак подчеркивания (`_`) и знак доллара (`$`).

`'class'` заключен в одинарные кавычки, потому что class — зарезервированное слово JavaScript, так что одинарные кавычки необходимы, чтобы браузер не интерпретировал его как код JavaScript (что также привело бы к синтаксической ошибке).

Далее создайте в файле `checklist.js` элемент `<label>` с функцией `$`, но без объектного аргумента. Ему не требуются какие-либо дополнительные атрибуты.

```
...
  function Row(coffeeOrder) {
    var $div = $('<div></div>', {
      'data-coffee-order': 'checkbox',
      'class': 'checkbox'
    });

    var $label = $('<label></label>');
  }
...
```

Теперь создайте элемент `<input>` для флажка путем вызова функции `$`, передав ей HTML для тега `<input>`. В качестве второго аргумента укажите, что его тип должен быть `checkbox`, а значение (`value`) должно быть адресом электронной почты посетителя. Поскольку ни одно из этих имен свойств не содержит специальные символы, заключать их в одинарные кавычки нет необходимости.

```
...
  function Row(coffeeOrder) {
    var $div = $('<div></div>', {
      'data-coffee-order': 'checkbox',
      'class': 'checkbox'
    });

    var $label = $('<label></label>');

    var $checkbox = $('<input></input>', {
      type: 'checkbox',
      value: coffeeOrder.emailAddress
    });
  }
...
```

Задавая значение `value` равным адресу электронной почты посетителя, мы связываем кнопку-флажок с заказом кофе этого посетителя. Позднее, когда мы добавим обработчик события `click`, сможем определить, по какому заказу был выполнен щелчок кнопкой мыши, основываясь на адресе электронной почты в атрибуте `value`.

Последнее, что нам осталось создать, — текстовое описание, которое будет отображаться рядом с кнопкой-флажком. Мы сконструируем строку для этого путем конкатенации отдельных частей с помощью оператора `+=`.

Создайте в файле `checklist.js` переменную `description`. Присвойте ей значение свойства `size` заказа, после чего добавьте запятую и пробел. Если был указан ароматизатор, выполните его конкатенацию, используя `+=`. Затем добавьте значения свойств `coffee`, `emailAddress` и `strength`. `emailAddress` необходимо заключить в круглые скобки, а `strength` — в квадратные, причем перед скобкой следует поставить букву `x` (круглые и квадратные скобки нужны не из-за требований синтаксиса, а для форматирования текста).

```
...
  function Row(coffeeOrder) {
    ...

    var $checkbox = $('<input></input>', {
      type: 'checkbox',
      value: coffeeOrder.emailAddress
    });

    var description = coffeeOrder.size + ' ';
    if (coffeeOrder.flavor) {
      description += coffeeOrder.flavor + ' ';
    }

    description += coffeeOrder.coffee + ', ';
    description += ' (' + coffeeOrder.emailAddress + ')';
    description += ' [' + coffeeOrder.strength + 'x]';
  }
...
```

Оператор конкатенации `+=` выполняет сложение и присваивание за один шаг. Это значит, что следующие две строки кода эквивалентны:

```
description += coffeeOrder.flavor + ' ';
description = description + coffeeOrder.flavor + ' ';
```

Теперь у нас есть все отдельные части элемента перечня — пора объединить их друг с другом (рис. 11.5).

Мы выполним это в три этапа.

1. Присоединим значение переменной `$checkbox` к переменной `$label`.
2. Присоединим значение переменной `description` к `$label`.
3. Присоединим значение `$label` к переменной `$div`.

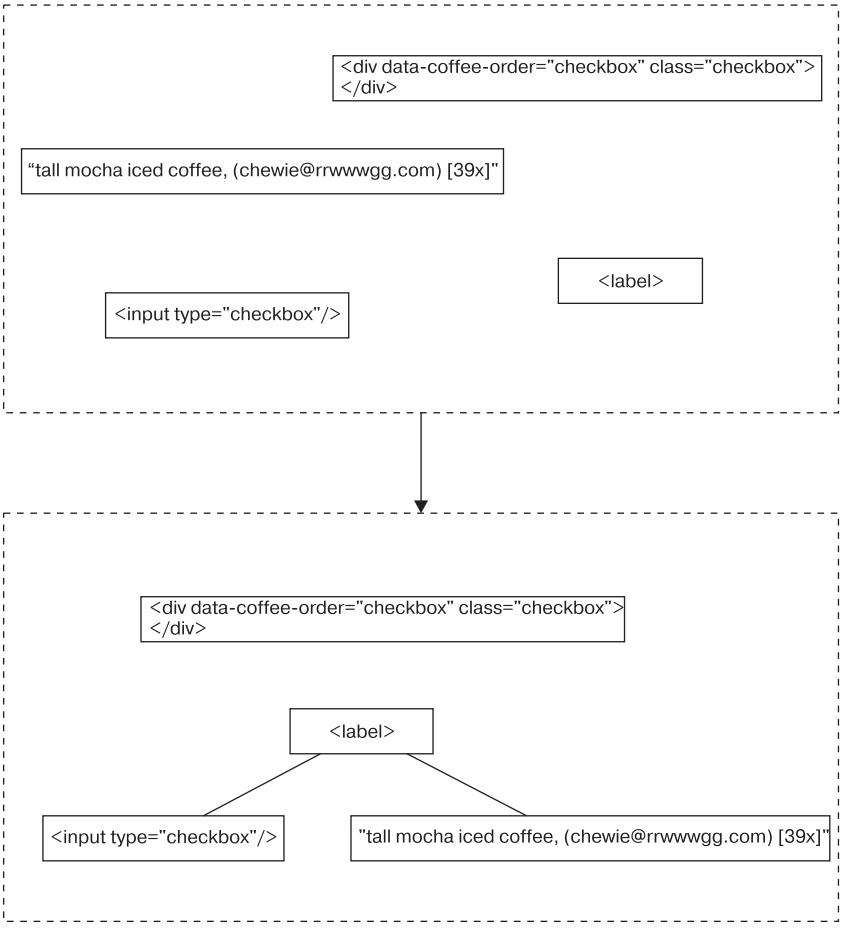


Рис. 11.5. Собираем отдельные элементы DOM в поддерево

В общем случае поддерево строится слева направо и снизу вверх. Такой подход напоминает процесс разработки CSS для приложения Ottergram в главе 3 (начиная с самых маленьких и внутренних элементов и двигаясь наружу).

Воспользуемся в файле `checklist.js` методом `append` библиотеки jQuery, чтобы связать элементы воедино. Этот метод принимает на входе или элемент DOM, или обернутую с помощью jQuery коллекцию и добавляет его в качестве дочернего элемента.

```
...
  function Row(coffeeOrder) {
    ...
    description += coffeeOrder.coffee + ', ';
    description += ' (' + coffeeOrder.emailAddress + ')';
```

```
    description += ' [' + coffeeOrder.strength + 'x]';

    $label.append($checkbox);
    $label.append(description);
    $div.append($label);
  }
...
```

Теперь наш конструктор `Row` может создавать и компоновать поддерево элементов на основе передаваемых в него данных о заказе. Но, поскольку `Row` будет использоваться в качестве конструктора, а не обычной функции, он не может просто возвращать это поддерево (на деле в конструкторах *никогда* не должно быть оператора `return`; JavaScript автоматически возвращает значение при использовании с конструктором ключевого слова `new`).

Вместо этого сделаем поддерево доступным в виде свойства экземпляра, присвоив его в файле `checklist.js` свойству `this.$element` (такое название было выбрано, чтобы соответствовать условным обозначениям, применявшимся для других наших конструкторов; в нем самом не содержится какого-либо особого смысла).

```
...
  function Row(coffeeOrder) {
    ...
    $label.append($checkbox);
    $label.append(description);
    $div.append($label);

    this.$element = $div;
  }
...
```

Конструктор `Row` готов к работе. Он может создавать поддеревья DOM, необходимые для представления отдельных заказов кофе с флажком. Он связан с этим представлением DOM через переменную экземпляра.

Создание строк CheckList при подтверждении отправки формы

Далее мы добавим в модуль `CheckList` метод, который будет использовать конструктор `Row` для создания экземпляров `Row`. Он будет присоединять все экземпляры `Row` к активному дереву DOM страницы.

Добавьте в файл `checklist.js` метод `addRow` к `CheckList.prototype`. Он должен принимать на входе аргумент `coffeeOrder`, представляющий собой объект, содержащий все данные отдельного заказа кофе.

В этом новом методе создайте новый экземпляр `Row`, вызвав конструктор `Row` и передав ему объект `coffeeOrder`. Присвойте этот новый экземпляр переменной

rowElement. Затем присоедините (append) содержащее поддерево DOM свойство $element переменной rowElement к свойству $element экземпляра CheckList (являющемуся ссылкой на контейнер элементов перечня):

```
...
  function CheckList(selector) {
    ...
  }

  CheckList.prototype.addRow = function (coffeeOrder) {
    // Создаем новый экземпляр строки на основе информации о заказе кофе
    var rowElement = new Row(coffeeOrder);

    // Добавляем свойство $element нового экземпляра строки в перечень
    this.$element.append(rowElement.$element);
  };
  function Row(coffeeOrder) {
    ...
```

Вот и все, что нужно сделать для добавления соответствующего Row поддерева DOM в страницу. Сохраните файл checklist.js.

Добавьте в файл main.js переменную для селектора, соответствующего всей области перечня, [data-coffee-order="checklist"]. После чего импортируйте модуль CheckList из пространства имен App в локальную переменную CHECKLIST_SELECTOR:

```
(function (window) {
  'use strict';
  var FORM_SELECTOR = '[data-coffee-order="form"]';
  var CHECKLIST_SELECTOR = '[data-coffee-order="checklist"]';
  var App = window.App;
  var Truck = App.Truck;
  var DataStore = App.DataStore;
  var FormHandler = App.FormHandler;
  var CheckList = App.CheckList;
  var myTruck = new Truck('ncc-1701', new DataStore());
...
```

Теперь вы можете создать экземпляр CheckList для добавления элементов перечня.

Вам может показаться соблазнительным просто добавить еще одно обращение к методу formHandler.addSubmitHandler, но это не будет работать так, как вы рассчитываете. Почему? При каждом вызове метод addSubmitHandler регистрирует новый обратный вызов, очищающий форму (путем вызова метода this.reset).

Рассмотрим следующий код:

```
...
// Создаем новый экземпляр CheckList
var checkList = new CheckList(CHECKLIST_SELECTOR);

var formHandler = new FormHandler(FORM_SELECTOR);
```

```
formHandler.addSubmitHandler(myTruck.createOrder.bind(myTruck));

// Это будет делать не то, что вам бы хотелось!
formHandler.addSubmitHandler(checkList.addRow.bind(checkList));
...
```

Этот код регистрирует два обратных вызова, которые будут выполняться при подтверждении отправки формы. После вызова первого обработчика события `submit` (метод `myTruck.createOrder`) форма очищается. При вызове второго обработчика события `submit` (метод `checkList.addRow`) в форме не остается никакой информации. В результате данные попадают в экземпляр `DataStore`, но элемент перечня не добавляется на страницу.

Чтобы избежать этого, необходимо передать методу `formHandler.addSubmitHandler` одну анонимную функцию и сделать так, чтобы она вызывала как метод `myTruck.createOrder`, так и метод `checkList.addRow`.

Помимо этого, необходимо, чтобы каждый из этих методов был привязан к конкретному экземпляру (то есть должно быть задано значение его переменной `this`). Мы уже использовали метод `bind` для указания значения `this`, но здесь мы обратимся к другому методу.

Меняем переменную this с помощью метода call. Метод `call` работает аналогично методу `bind` при указании значения ссылки `this`. Отличаются они тем, что, хотя `bind` возвращает новую версию функции или метода, он их не вызывает. Метод `call` на самом деле вызывает эту функцию/метод и позволяет передать значение ссылки `this` в виде первого аргумента (если необходимо передать еще какие-либо аргументы в функцию, достаточно добавить их в список аргументов). Метод `call` выполняет тело функции и возвращает то значение, которое было бы возвращено при обычном вызове.

Здесь нужно использовать `call` вместо `bind`, поскольку нам требуется вызвать методы `myTruck.createOrder` и `checkList.addRow` (помимо указания значения переменной `this`).

Удалите из файла `main.js` существующий вызов метода `formHandler.addSubmitHandler`. Добавьте новый вызов метода `formHandler.addSubmitHandler` и передайте ему анонимную функцию, получающую на входе один аргумент — объект `data`. Внутри этой анонимной функции используйте метод `call` методов `myTruck.createOrder` и `checkList.addRow` для указания значения переменной `this`, передавая объект `data` в качестве второго аргумента.

```
...
  var myTruck = new Truck('ncc-1701', new DataStore());
  window.myTruck = myTruck;
  var checkList = new CheckList(CHECKLIST_SELECTOR);
  var formHandler = new FormHandler(FORM_SELECTOR);

  formHandler.addSubmitHandler(~~myTruck.createOrder.bind(myTruck)~~);
                               function (data) {
```

```
    console.log(formHandler);
    myTruck.createOrder.call(myTruck, data);
    checkList.addRow.call(checkList, data);
  });
})(window);
```

Мы создали одну функцию-обработчик события `submit`, вызывающую как метод `createOrder`, так и метод `addRow`. При их вызове она передает в них нужное значение ссылки `this` и объекта `data` из формы.

Сохраните изменения и опробуйте новую функциональность перечня в браузере, введя какие-нибудь данные и подтвердив отправку формы. Вы увидите, как при подтверждении отправки каждого заказа он добавляется в перечень `Pending Orders` (заказы, ожидающие выполнения) (рис. 11.6).

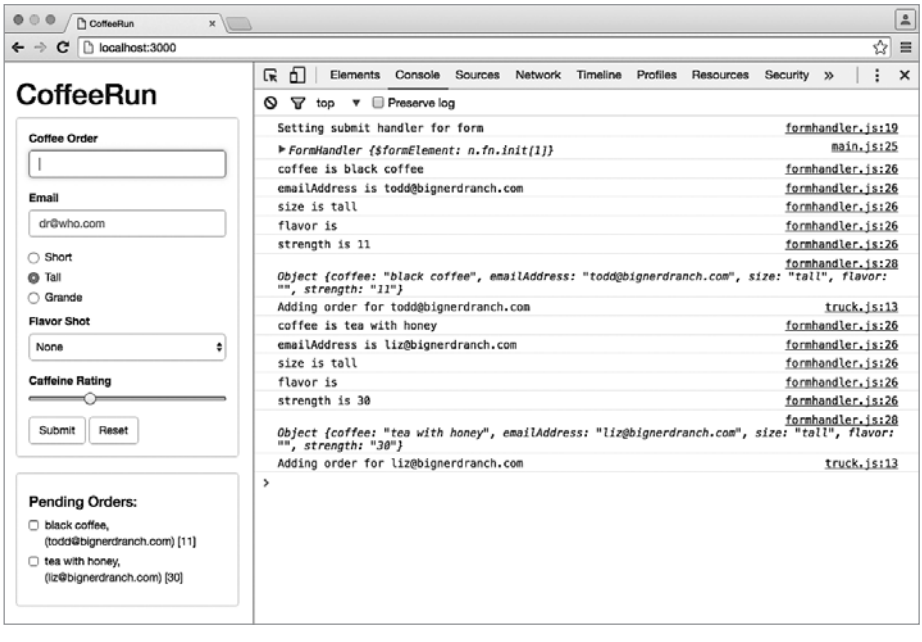


Рис. 11.6. Подтверждение отправки формы приводит к добавлению элемента в перечень

Выдача заказа с помощью щелчка на строке

Мы почти закончили! Пользователи CoffeeRun могут заполнять форму для добавления заказов. При подтверждении ими отправки формы в базу данных приложения добавляется информация о заказе и отрисовывается элемент перечня для этого заказа.

Далее пользователи должны иметь возможность помечать элементы перечня как выполненные. Щелчок кнопкой мыши на элементе перечня означает, что заказ был выдан и информацию о заказе нужно удалить из базы данных, а элемент перечня — убрать со страницы. Этот процесс показан на рис. 11.7.

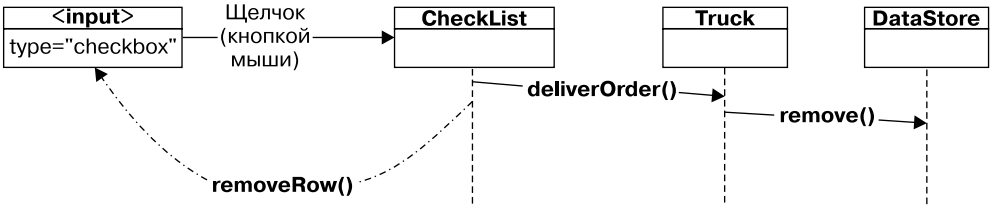


Рис. 11.7. Схема последовательности действий: щелчок на элементе перечня

Для начала создадим функциональность для удаления пунктов перечня со страницы.

Создание метода CheckList.prototype.removeRow

Когда мы создаем `Row`, атрибуту `value` элемента `<input>` присваивается значение адреса электронной почты посетителя. Метод `removeRow` будет использовать аргумент адреса электронной почты при поиске нужного элемента `CheckList` для удаления из пользовательского интерфейса. Он будет осуществлять это путем создания селектора атрибута для поиска элемента `<input>`, чей атрибут `value` соответствует данному адресу электронной почты.

При нахождении подходящего элемента он начнет перемещаться вверх по DOM до тех пор, пока не найдет `[data-coffee-order="checkbox"]`. Это элемент `<div>`, содержащий все являющиеся частями строки элементы.

Наконец, после выборки с помощью библиотеки jQuery этого тега `<div>` можно вызвать его метод `remove` и удалить элемент из DOM, а также убрать все прослушиватели события, связанные с любым элементом данного поддерева DOM.

Добавьте в файл `checklist.js` метод `removeRow`, определив параметр `emailAddress`. Воспользуйтесь свойством экземпляра `$element` для поиска каких-либо элементов-потомков, чей атрибут `value` соответствует параметру `email`. Добавьте для этого элемента вызов метода `closest` для поиска предка, чей атрибут `data-coffee-order` равен `"checkbox"`. Наконец, вызовите метод `remove` для этого элемента (вы обнаружите в этом коде новый синтаксис, который мы разъясним после того, как вы его введете):

```
...
  CheckList.prototype.addRow = function (coffeeOrder) {
    ...
  };
```

```
CheckList.prototype.removeRow = function (email) {
  this.$element
    .find('[value="' + email + '"]')
    .closest('[data-coffee-order="checkbox"]')
    .remove();
};

function Row(coffeeOrder) {
  ...
```

В этом коде мы *объединили в цепочку* вызовы нескольких методов. Библиотека JQuery построена таким образом, что можно описать несколько вызовов методов для объекта как список шагов. Нужно только поставить точку с запятой в конце самого последнего вызова.

Есть обязательное требование для объединения в цепочку: следующий метод может быть добавлен в цепочку, если предыдущий возвращает обернутую jQuery выборку. И метод `find`, и метод `closest` возвращают обернутую jQuery выборку, что позволяет нам объединить вызовы всех трех методов в цепочку.

Обратите внимание, что мы использовали метод `this.$element.find`, выполнив выборку *по области видимости*: вместо поиска по всему DOM поиск выполняется только по потомкам перечня, на который мы сослались с помощью свойства `this.$element`.

Удаление перезаписанных записей

Сохраните файл и переключитесь в браузер. С помощью нашей формы введите два заказа на один и тот же адрес электронной почты. Пусть значение свойства `coffee` для первого будет `"order 1"` (заказ 1), а для второго — `"order 2"` (заказ 2). После подтверждения отправки обоих заказов вызовите в консоли метод `myTruck.printOrders`. Результат показан на рис. 11.8.

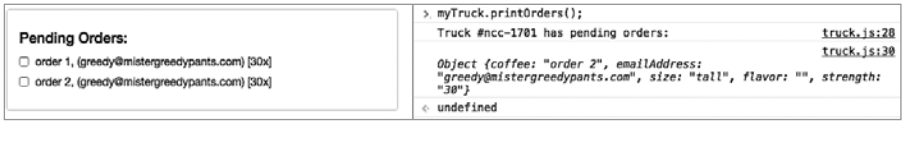


Рис. 11.8. Заказы для одного и того же адреса электронной почты остаются в UI

Мы в самом начале решили принимать только один заказ для одного адреса электронной почты. Поскольку мы используем для наших данных простое хранилище типа «ключ/значение», все последующие заказы для того же адреса электронной почты посетителя перезаписывают существующий. Как демонстрирует консоль, заказ 2 — единственный ожидающий выполнения заказ (заказ 1 был перезаписан).

Но перечень этого не отражает — он все еще отображает строки как для заказа 1, так и для заказа 2. При добавлении строки для заказа нужно убедиться, что любые имеющиеся строки, связанные с тем же адресом электронной почты, удалены.

Сейчас, когда мы умеет удалять строки по их адресу электронной почты, это несложная задача. Исправьте в файле `checklist.js` метод прототипа `addRow`, чтобы первое, что он делал, — вызывал метод `removeRow`, передавая ему адрес электронной почты посетителя.

```
...
  CheckList.prototype.addRow = function (coffeeOrder) {
    // Удаляем все имеющиеся строки, соответствующие данному адресу
    // электронной почты
    this.removeRow(coffeeOrder.emailAddress);

    // Создаем новый экземпляр строки на основе информации о заказе кофе
    var rowElement = new Row(coffeeOrder);

    // Добавляем свойство $element нового экземпляра строки в перечень
    this.$element.append(rowElement.$element);
  };
...
```

Сохраните файл `checklist.js` и проверьте в браузере, что пункт перечня для первого заказа удалился при подтверждении отправки формы для второго заказа с тем же адресом электронной почты.

Теперь, когда мы умеем удалять строки перечня из пользовательского интерфейса, переключим внимание на обработку события `click` для перечня.

Написание метода addClickHandler

Для обработки щелчков кнопкой мыши на перечне воспользуемся тем же приемом регистрации обработчиков событий, который применяли в модуле `FormHandler`.

Метод `FormHandler.prototype.addSubmitHandler` принимает на входе функциональный аргумент `fn`, а затем регистрирует анонимную функцию для обработки события `submit` для `this.$formElement`. Внутри этой анонимной функции вызывается `fn`. Вот определение этого метода:

```
FormHandler.prototype.addSubmitHandler = function (fn) {

  console.log('Setting submit handler for form');
  this.$formElement.on('submit', function (event) {
    event.preventDefault();

    var data = {};
    $(this).serializeArray().forEach(function (item) {
      data[item.name] = item.value;
```

```
      console.log(item.name + ' is ' + item.value);
    });
    console.log(data);

    fn(data);
    this.reset();
    this.elements[0].focus();
  });
};
```

Это придает гибкость методу `FormHandler.prototype.addSubmitHandler`, поскольку ему можно передать любую функцию, которую нужно вызывать при подтверждении отправки формы. Таким образом, методу `FormHandler.prototype.addSubmitHandler` не нужно знать подробностей реализации этой функции и какие действия она выполняет.

Мы добавим в модуль `CheckList` метод прототипа `addClickHandler`, который будет работать аналогично методу `addSubmitHandler` модуля `FormHandler`. То есть он будет делать следующее.

1. Принимать на входе функциональный аргумент.
2. Регистрировать обратный вызов обработчика события.
3. Вызывать функциональный аргумент внутри обратного вызова обработчика события.

Метод `CheckList.prototype.addClickHandler` отличается от метода `FormHandler.prototype.addSubmitHandler` тем, что он будет прослушивать на предмет события `click` и привязывать (`bind`) обратный вызов к экземпляру `CheckList`.

Добавьте в файл `checklist.js` метод `addClickHandler`, описав для него функциональный параметр `fn`. Мы будем прослушивать на предмет события `click` с помощью метода `on` библиотеки jQuery.

Объявите в функции обработчика события локальную переменную `email` и присвойте ей значение `event.target.value` (адрес электронной почты посетителя). Затем вызовите метод `removeRow`, передав ему переменную `email`. После этого вызовите `fn` и тоже передайте ему `email`. Не забудьте использовать вызов `bind(this)`, чтобы установить объект контекста для функции обработчика события.

```
...
  function CheckList(selector) {
    ...
  }

  CheckList.prototype.addClickHandler = function (fn) {
    this.$element.on('click', 'input', function (event) {
      var email = event.target.value;
      this.removeRow(email);
      fn(email);
```

```
    }.bind(this));
  };

  CheckList.prototype.addRow = function (coffeeOrder) {
    ...
```

При регистрации обратного вызова обработчика события с помощью метода `this.$element.on` мы указали `click` в качестве имени события. Но мы также передали в качестве второго аргумента *фильтрующий селектор*. Он указывает обработчику события выполнить функцию обратного вызова *только в том случае, если* событие было вызвано элементом `<input>`.

Такой паттерн носит название *делегирования событий* и вступает в дело, поскольку некоторые события, такие как щелчки кнопкой мыши и нажатия клавиш, *транслируются* по DOM. Это значит, что каждый элемент-предок извещается об этом событии.

Когда вам понадобится слушать события для динамически создаваемых и удаляемых элементов, таких как элементы перечней, следует использовать делегирование событий. Удобнее и эффективнее добавить отдельный прослушиватель в контейнер динамических элементов, после чего выполнить функцию-обработчик в зависимости от того, какой элемент вызвал событие.

Обратите внимание, что мы *не* вызывали метод `event.preventDefault` в обработчике события. Почему? Если бы мы вызвали `event.preventDefault`, флажок не поменял бы своего визуального состояния и не отобразил бы «галочку».

Обратите также внимание, что мы привязали (`bind`) обратный вызов обработчика события к переменной `this`, ссылающейся на экземпляр `CheckList`.

Вызов метода addClickHandler

Метод `addClickHandler` необходимо связать с методом `deliverOrder`. Для этого перейдите в файл `main.js`. Передайте связанную версию `deliverOrder` методу `CheckList.addClickHandler`.

```
...
  var myTruck = new Truck('ncc-1701', new DataStore());
  window.myTruck = myTruck;
  var checkList = new CheckList(CHECKLIST_SELECTOR);
  checkList.addClickHandler(myTruck.deliverOrder.bind(myTruck));
  var formHandler = new FormHandler(FORM_SELECTOR);
...
```

Сохраните изменения и добавьте еще несколько заказов кофе в форму. Щелкните или на флажке, или на тексте одного из пунктов перечня — и он будет удален (рис. 11.9).

Мы научились создавать динамические элементы формы и работать с генерируемыми ими событиями, сумели связать каждый из них с конкретным заказом кофе,

используя адрес электронной почты в качестве идентификатора. Применив эти приемы, мы завершили разработку управляющих пользовательским интерфейсом модулей, превратив консольное приложение в пригодное для реальной работы.

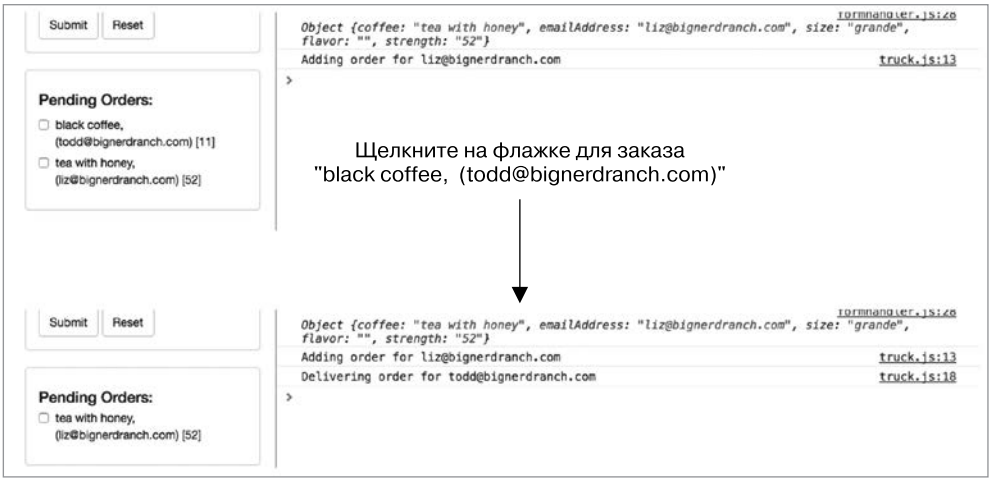


Рис. 11.9. Щелчок на пункте перечня удаляет его

Мы закончили две из трех основных частей приложения CoffeeRun. Внутренняя логика управляет данными в приложении, а элементы формы, модули `FormHandler` и `CheckList` обеспечивают интерактивный пользовательский интерфейс. Следующие главы будут касаться подготовки данных и их обмена с удаленным сервером.

Бронзовое упражнение: добавление крепости кофе в описание

Вы решили, что крепость кофе — более важная информация и должна быть первой частью описания.

Измените код для описания заказов так, чтобы крепость кофе находилась в начале текста описания.

Серебряное упражнение: цветовая маркировка в зависимости от ароматизатора

Добавьте цветовую маркировку заказов в зависимости от выбранного ароматизатора. Основываясь на указанных для каждого кофе параметрах, отображайте строки в перечне, используя различные фоновые цвета.

Проверьте, чтобы текст был достаточно контрастным по сравнению с цветом фона.

Золотое упражнение: предоставление возможности редактирования заказов

Предоставьте пользователю возможность редактировать заказы. Для этого вам придется изменить работу перечня.

Если пользователь выполняет двойной щелчок на заказе, загружайте его обратно в форму для редактирования. Если пользователь щелкает только один раз, затеняйте строку. Спустя несколько секунд считайте элемент выданным и удаляйте его из перечня и из данных приложения.

В качестве дополнительного задания сделайте так, чтобы после завершения пользователем редактирования существующая строка обновлялась на месте, а не удалялась и заменялась новой.

12 Проверка данных форм

CoffeeRun кое-как работает. Пользователи могут вводить заказы кофе в форму, и информация о заказе обрабатывается и сохраняется. Но что будет с нашим приложением — и нашим автокафе, — если кто-то отправит форму с недостающей или некорректной информацией?

Не волнуйтесь. Эти сценарии легко можно обработать с помощью небольшого кода, необходимого, чтобы убедиться, что данные подходят для использования нашим приложением. На самом деле этот этап обязателен при любой отправке данных на сервер. Практически каждый современный браузер подходит для проверки данных формы при ее отправке. Все, что от нас требуется, — задать правила проверки.

В данной главе мы изучим два способа проверки данных форм. Первый способ — добавить атрибуты проверки в HTML, позволив далее работать встроенным механизмам браузера. Второй — написать свой собственный код проверки на языке программирования JavaScript с помощью API проверки ограничений (Constraint Validation API).

Атрибут required

Простейшая форма проверки — убедиться, что у поля имеется значение, причем не пустое. Такой вид проверки не имеет смысла для полей со значением по умолчанию, таким как порция, ароматизатор и крепость кофе. Но вполне подходит для полей заказа и адреса электронной почты (безусловно, нам не хотелось бы, чтобы заказы отправлялись с этими незаполненными полями).

Добавьте в файл `index.html` булев атрибут `required` в поля заказа и адреса электронной почты:

```
...
          <div class="form-group">
            <label for="coffeeOrder">Order</label>
            <input class="form-control" name="coffee" id="coffeeOrder"
              autofocus required />
          </div>

          <div class="form-group">
            <label for="emailInput">Email</label>
            <input class="form-control" type="email" name="emailAddress"
            id="emailInput" value="" placeholder="dr@who.com"
              required />
          </div>
...
```

Помните, что булеву атрибуту нельзя присваивать значение. Если вы случайно напишете что-то вроде `required="false"`, значение окажется `true` и поле будет обязательным для заполнения! Браузер интересует только существование атрибута, он игнорирует любое присвоенное ему значение.

Не помешает повторить это еще раз: если булев атрибут существует как элемент, браузер считает, что его значение равно `true`, независимо от присвоенного ему значения.

Сохраните файл `index.html`, убедитесь, что утилита browser-sync запущена, и загрузите в браузере приложение CoffeeRun. Попробуйте подтвердить отправку формы, не заполнив одно или оба обязательных поля. Вы увидите предупреждение, продемонстрированное на рис. 12.1.

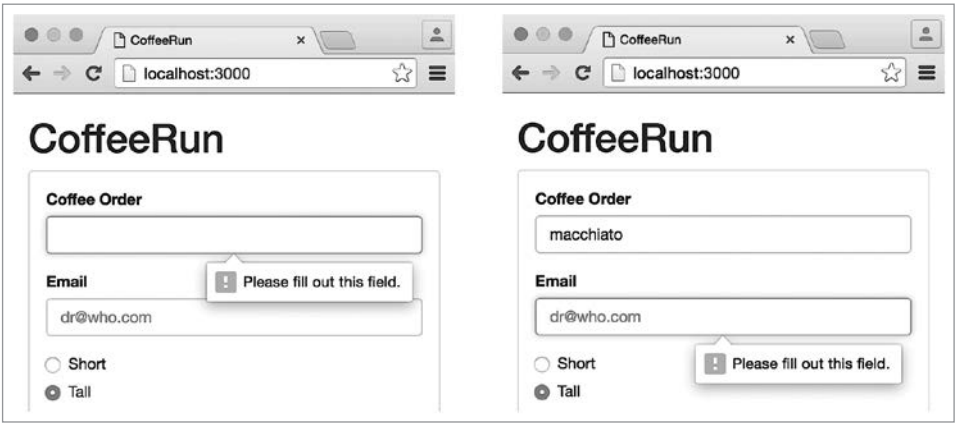


Рис. 12.1. Ошибки, которые выводятся, если обязательные поля не заполнены

Обратите также внимание на отсутствие консольных сообщений от наших обработчиков события `submit`. Событие `submit` генерируется только *после* проверки браузером формы (рис. 12.2).

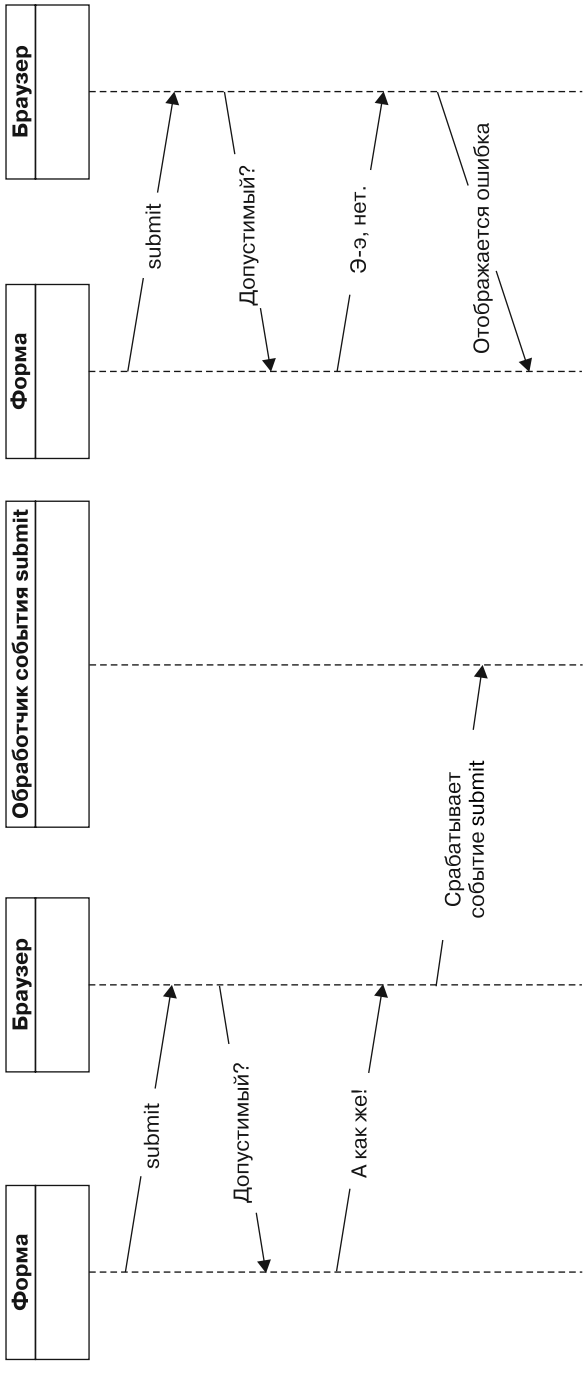


Рис. 12.2. Две возможные последовательности событий при проверке формы

Проверка с помощью регулярных выражений

Сделать поле обязательным — удобный способ гарантировать, что пользователь не оставит поле незаполненным. Но что, если нам нужно указать точнее, что может в нем содержаться? Такая разновидность проверки требует использования атрибута `pattern`.

Добавьте после атрибута `required` атрибут `pattern` в поле `<input>` нашего заказа. Присвойте ему особым образом сформированную строку, называемую *регулярным выражением*.

```
...
          <div class="form-group">
            <label for="coffeeOrder">Order</label>
            <input class="form-control" name="coffee" id="coffeeOrder"
              autofocus required pattern="[a-zA-Z\s]+" />
          </div>
...
```

Регулярное выражение — это последовательность символов для проверки соответствия шаблону. Регулярное выражение `[a-zA-Z\s]+` соответствует любым символам из набора, состоящего из латинских букв в нижнем регистре (`a-z`), заглавных букв (`A-Z`) и символов пробела (`\s`), повторенных один раз или более (`+`).

Короче говоря, при подтверждении отправки формы это поле считается заполненным правильно, только если будет состоять из букв или пробелов.

Сохраните и перезагрузитесь. Посмотрите, что произойдет, если вы поместите не буквенные символы или цифры в поле заказа и попытаетесь отправить форму.

API проверки ограничений

Наиболее устойчивый к ошибкам способ проверки полей форм в браузере заключается в написании функции проверки. Можно использовать функции проверки совместно с API проверки ограничений, чтобы запускать встроенное проверочное поведение.

Но есть нюанс, и не маленький: браузер Safari фирмы Apple плохо поддерживает API проверки ограничений.

Несмотря на эту проблему, лучше написать код, нацеленный на стандартное поведение, после чего добавить библиотеку JavaScript, которая обеспечит поддержку несовместимых браузеров (прочесть об этом подробнее вы можете в разделе «Для наиболее любознательных: библиотека Webshims» в конце этой главы).

Допустим, наше автокафе предназначено только для работников нашей фирмы, так что мы хотели бы удостовериться, что посетитель — наш сотрудник. Один из способов сделать это — убедиться, что введенный адрес электронной почты относится к нашему корпоративному домену.

Мы *можем* воспользоваться для поля `emailAddress` атрибутом `pattern`. Но эта задача — удобный случай для изучения API проверки ограничений (кроме того, после проработки следующей главы мы сможем выйти за пределы простой проверки адреса электронной почты и запрашивать удаленный сервер, чтобы узнать, действительно ли существует такой адрес).

Сохраните новый файл `scripts/validation.js` для наших функций проверки. Добавьте в файл `index.html` тег `<script>` для нашего нового модуля:

```
...
      </div>
    </section>
    <script src="https://cdnjs.cloudflare.com/ajax/libs/jquery/2.1.4/jquery.js"
      charset="utf-8"></script>
    <script src="scripts/validation.js" charset="utf-8"></script>
    <script src="scripts/checklist.js" charset="utf-8"></script>
    <script src="scripts/formhandler.js" charset="utf-8"></script>
    <script src="scripts/datastore.js" charset="utf-8"></script>
    <script src="scripts/truck.js" charset="utf-8"></script>
    <script src="scripts/main.js" charset="utf-8"></script>
  </body>
</html>
```

Сохраните файл `index.html`. Добавьте в файл `validation.js` модуль IIFE, который бы создавал пустой объект-литерал, присваивал его переменной `Validation`, после чего экспортировал эту переменную в пространство имен `App`:

```
(function (window) {
  'use strict';
  var App = window.App || {};

  var Validation = {
  };

  App.Validation = Validation;
  window.App = App;

})(window);
```

Наш новый модуль `Validation` будет использоваться только для организации функций, так что конструктор не нужен.

Добавьте метод под названием `isCompanyEmail`. Он будет проверять адрес электронной почты на соответствие регулярному выражению и возвращать `true` или `false` (при необходимости можете спокойно поменять указанный домен электронной почты).

```
(function () {
  'use strict';
  var App = window.App || {};

  var Validation = {
```

```
    isCompanyEmail: function (email) {
      return /.+@bignerdranch\.com$/.test(email);
    }
  };

  App.Validation = Validation;
  window.App = App;
})(window);
```

Мы создали регулярное выражение-литерал, вставив шаблон между прямыми косыми чертами `//`. Внутри них мы задали строку, состоящую из одного или нескольких символов (`.+`), за которыми следует `@bignerdranch.com`. Мы воспользовались обратной косой чертой, чтобы указать, что точку в регулярном выражении следует рассматривать как точку-литерал (в обычных условиях точка в регулярном выражении — джокерный символ, соответствующий любому символу). Символ `$` в конце регулярного выражения означает, что `@bignerdranch.com` должно находиться в конце строки, то есть после него не должно быть никаких символов.

Регулярное выражение — это объект, и у него имеется метод `test`. Методу `test` можно передать строку, и он вернет булево значение: `true` — если регулярное выражение соответствует строке, `false` — если нет (список других методов регулярных выражений см. на сайте developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/RegExp).

Проверьте работу функции `App.Validation.isCompanyEmail` в консоли (рис. 12.3).

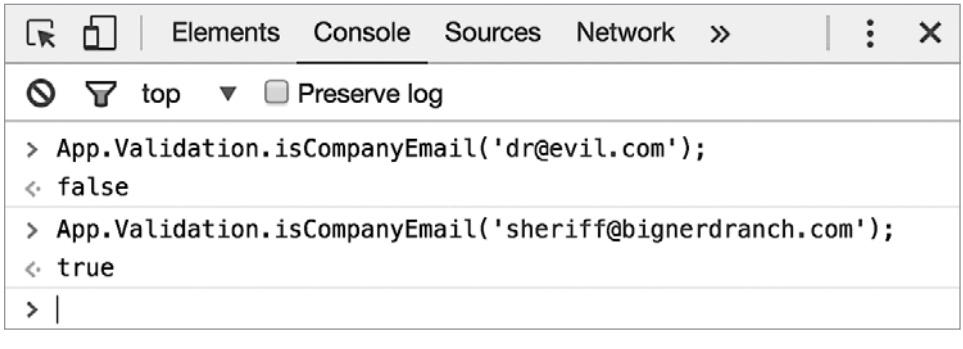


Рис. 12.3. Проверка работы функции App.Validation.isCompanyEmail в консоли

Теперь у нас есть функция для проверки корректности адресов электронной почты. Далее нужно связать ее с формой.

Прослушиватель для события input

Когда следует использовать эту функцию? Существует несколько событий, генерируемых полем ввода при заполнении пользователем формы. Одно происходит при наборе пользователем каждого символа. Другое — когда пользователь убирает

фокус с поля. Или же можно выполнять функцию при подтверждении отправки формы.

API проверки ограничений требует, чтобы некорректно заполненные поля помечались до отправки. Если какие-либо из полей заполнены некорректно, браузер останавливается, не доходя до генерации события `submit`. Так что выполнять проверку по `submit` слишком поздно.

Событие, генерируемое, когда пользователь убирает фокус с поля, называется *событием потери фокуса*. Это тоже не очень хороший вариант для проверки. Допустим, курсор пользователя находится в поле ввода электронной почты, так что фокус расположен на нем. Если пользователь затем нажимает клавишу Enter, произойдет подтверждение отправки формы, но событие `blur` не будет сгенерировано и никакой привязанной к нему проверки не произойдет.

Так что проверка должна происходить при вводе пользователем каждого символа. Добавьте в модуль `FormHandler` из файла `formhandler.js` метод прототипа `addInputHandler`. Он должен добавлять прослушиватель для события `input` формы. Аналогично методу `addSubmitHandler` он должен принимать на входе функциональный аргумент.

```
...
  FormHandler.prototype.addSubmitHandler = function (fn) {
    ...
  };

  FormHandler.prototype.addInputHandler = function (fn) {
    console.log('Setting input handler for form');
  };

  App.FormHandler = FormHandler;
  window.App = App;
...
```

Привяжем прослушиватель для события `input` с помощью метода `on` библиотеки jQuery. Не забудьте воспользоваться паттерном делегирования событий, чтобы отфильтровать все события, кроме сгенерированных полем `[name="emailAddress"]`.

```
...
  FormHandler.prototype.addInputHandler = function (fn) {
    console.log('Setting input handler for form');
    this.$formElement.on('input', '[name="emailAddress"]', function (event) {
      // Здесь будет находиться код обработчика события
    });
  };

  App.FormHandler = FormHandler;
  window.App = App;
...
```

Внутри этого обработчика события извлеките значение поля электронной почты из объекта `event.target`. Затем выведите в консоль (`console.log`) результаты выполнения функционального аргумента `fn` метода `addInputHandler`, передав ему значение из поля электронной почты.

```
...
  FormHandler.prototype.addInputHandler = function (fn) {
    console.log('Setting input handler for form');
    this.$formElement.on('input', '[name="emailAddress"]', function (event) {
      // Здесь будет находиться код обработчика события
      var emailAddress = event.target.value;
      console.log(fn(emailAddress));
    });
  };

  App.FormHandler = FormHandler;
  window.App = App;
...
```

Сохраните файл `formhandler.js`.

Связываем проверку допустимости с событием input

В файле `main.js` импортируйте объект `Validation` из пространства имен `App` и присвойте его локальной переменной.

```
...
  var Truck = App.Truck;
  var DataStore = App.DataStore;
  var FormHandler = App.FormHandler;
  var Validation = App.Validation;
  var CheckList = App.CheckList;
  var myTruck = new Truck('ncc-1701', new DataStore());
...
```

После импорта объекта `Validation` можно использовать его в вызове нового метода `addInputHandler` модуля `FormHandler`.

В конце файла `main.js` передайте свойство `Validation.isCompanyEmail` методу `addInputHandler` экземпляра `FormHandler`.

```
...
  formHandler.addSubmitHandler(function (data) {
    myTruck.createOrder.call(myTruck, data);
    checkList.addRow.call(checkList, data);
  });

  formHandler.addInputHandler(Validation.isCompanyEmail);

})(window);
```

Сохраните изменения и перезагрузите страницу. Заполните поле электронной почты и посмотрите, что появилось в консоли. По мере набора вами корректного адреса электронной почты консоль отобразит сначала множество результатов `false`, выведенных строкой `console.log(fn(emailAddress));` в методе `FormHandler.prototype.addInputHandler`. Когда вы закончите ввод корректного адреса электронной почты, вы увидите выведенную в консоль надпись `true` (рис. 12.4).

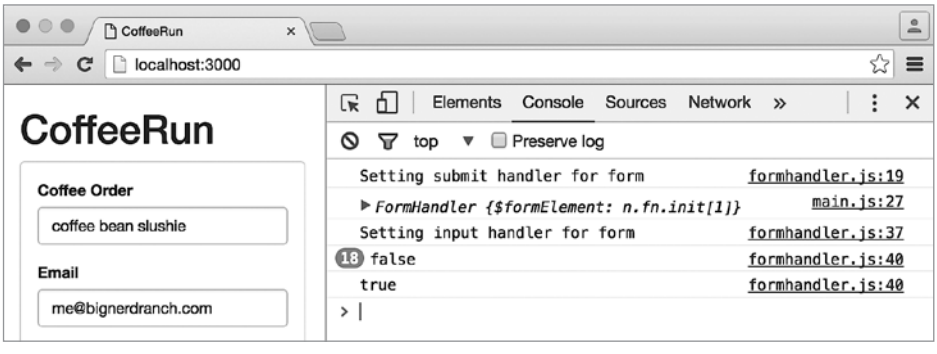


Рис. 12.4. Журналирование проверки допустимости адреса электронной почты

Наша функция проверки выполняется каждый раз, когда в поле электронной почты набирается (или удаляется из него) символ. Убедившись, что она правильно проверяет вводимые данные, мы сможем использовать ее для отображения пользовательского сообщения об ошибке.

Запуск проверки допустимости

Теперь, когда у нас есть возможность надежно проверить, относится ли адрес электронной почты к нашему корпоративному домену, неплохо бы оповещать пользователя, если проверка не пройдена. Мы воспользуемся методом `setCustomValidity` объекта `event.target`, чтобы помечать адрес как неправильный.

Удалите в файле `formhandler.js` оператор `console.log`, заменив его переменной для предупреждающего сообщения и выражением `if/else`. Если вызов функционального аргумента `fn(emailAddress)` возвращает `true`, обнулите пользовательскую отметку о допустимости значения. Если же он возвращает `false`, присвойте переменной `message` строку с предупреждающим сообщением и добавьте пользовательскую отметку о допустимости равной `message`.

```
...
  FormHandler.prototype.addInputHandler = function (fn) {
    console.log('Setting input handler for form');
    this.$formElement.on('input', '[name="emailAddress"]', function (event) {
      var emailAddress = event.target.value;
      console.log(fn(emailAddress));
```

```
      var message = '';
      if (fn(emailAddress)) {
        event.target.setCustomValidity('');
      } else {
        message = emailAddress + ' is not an authorized email address!'
        event.target.setCustomValidity(message);
      }
    });
  };
...
```

Мы передали в метод сообщение об ошибке, которое должно отображаться пользователю. Если ошибки нет, все равно нужно вызвать метод `setCustomValidity`, но с пустой строкой в качестве аргумента. Это равносильно тому, чтобы пометить поле как имеющее допустимое значение.

Проверка допустимости, выполняющаяся при наборе, только помечает поле как имеющее допустимое или недопустимое значение. Она не отображает сообщения об ошибке. Когда вы нажимаете кнопку Submit, браузер ищет поля с недопустимыми значениями и, если найдет, отображает сообщение о неудачной проверке.

Чтобы протестировать это, подтвердите отправку формы, введя в нее сначала адрес электронной почты с неподходящим доменом. Сразу после нажатия Submit вы увидите, как появится предупреждающее пользовательское сообщение о неудачной проверке (рис. 12.5).

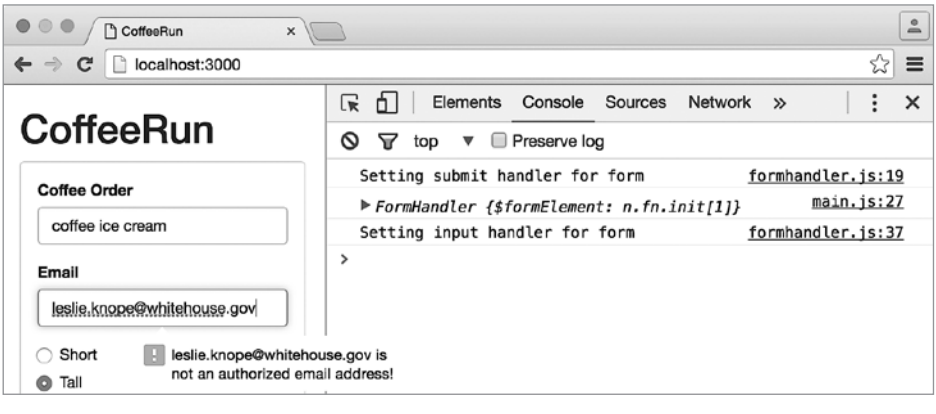


Рис. 12.5. Только допустимые адреса электронной почты!

Стилизация элементов с допустимым и недопустимым значением

В настоящий момент приложение CoffeeRun проверяет как поле заказа, так и поле адреса электронной почты. Настало время усовершенствовать пользовательский

интерфейс, визуально отмечая поля с недопустимым значением. Это очень короткий фрагмент CSS, для которого в файле `index.html` нужно добавить следующий набор правил в тег `<style>` из элемента `<head>`:

```
...
  <head>
    <meta charset="utf-8">
    <title>coffeerun</title>
    <link rel="stylesheet" href="https://cdnjs.cloudflare.com/ajax/libs/
          twitter-bootstrap/3.3.6/css/bootstrap.min.css">
    <style>
      form :invalid {
        border-color: #a94442;
      }
    </style>
  </head>
...
```

Этот код добавляет границу к любому полю с псевдоклассом `:invalid` внутри нашей формы. Данный псевдокласс автоматически добавляется браузером при выполнении формой проверок допустимости.

Сохраните изменения и вернитесь в браузер. Нажмите клавишу Tab несколько раз (или щелкните кнопкой мыши вне полей для ввода текста), чтобы фокус оказался на каких-то других элементах формы (не на поле заказа и не на поле электронной почты). Границы двух обязательных полей приобретут при этом красный цвет (рис. 12.6).

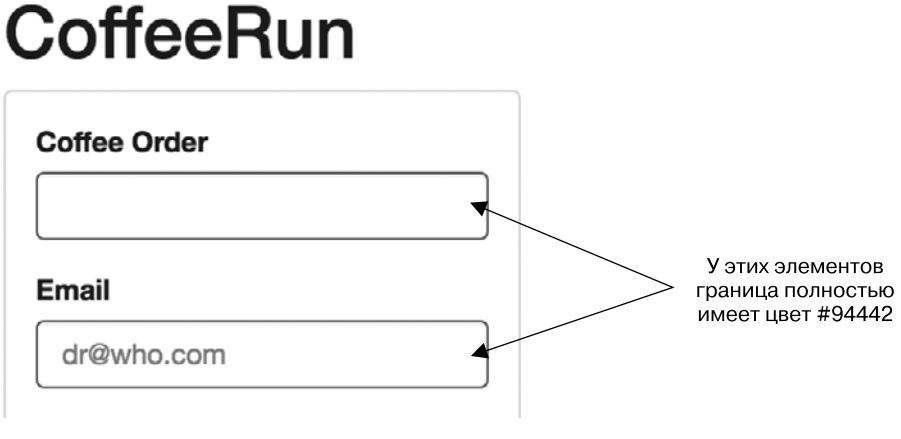


Рис. 12.6. Поверьте: цвет этих границ — красный

Было бы уместнее, чтобы граница отображалась только у тех полей, которые являются обязательными и на которых находится фокус. Добавим еще два псевдокласса в наш селектор в файле `index.html`:

```
  <head>
    <meta charset="utf-8">
    <title>coffeerun</title>
    <link rel="stylesheet" href="https://cdnjs.cloudflare.com/ajax/libs/
          twitter-bootstrap/3.3.6/css/bootstrap.min.css">
    <style>
      form :focus:required:invalid {
        border-color: #a94442;
      }
    </style>
  </head>
...
```

Теперь у полей с тремя псевдоклассами — `:focus`, `:required` и `:invalid` — одновременно будет новый цвет границы (рис. 12.7).

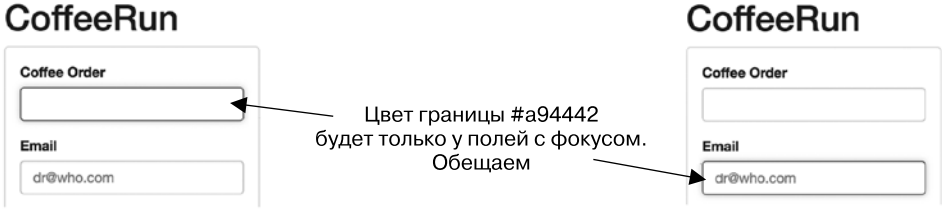


Рис. 12.7. Цвет границы :invalid будет только у полей с фокусом

CoffeeRun превратилось в полнофункциональное веб-приложение. В следующих двух главах мы будем синхронизировать данные с удаленным сервером с помощью технологии Ajax.

Серебряное упражнение: пользовательская проверка допустимости для Decaf

Добавьте еще одну функцию в модуль `Validation`. Она должна принимать на входе два аргумента: строку и целое число. Если строка содержит слово `decaf` и число больше 20, функция должна возвращать `false`.

Добавьте прослушиватели для текстового поля заказа кофе и слайдера крепости кофе. Организуйте запуск пользовательской проверки допустимости для любого редактируемого в настоящий момент поля, которое вызывает ошибку проверки.

Для наиболее любознательных: библиотека Webshims

Как уже упоминалось ранее, заслуживающим внимания браузером, не поддерживающим API проверки ограничений, является Safari от Apple. Если вам необходимо работать с Safari, можно воспользоваться библиотекой (так называемым *полифиллом*), имитирующей API для тех браузеров, которые его не реализуют.

Одна из библиотек, обеспечивающих проверку ограничений в Safari, — Webshims Lib (ее можно скачать по адресу github.com/aFarkas/webshim).

Фактически библиотека Webshims Lib может работать как полифилл для множества разных функциональных возможностей. Ее установка, настройка и использование весьма просты (однако она может выполнять еще множество различных задач, так что потеряться в документации несложно).

Вот как следует использовать ее для приложения CoffeeRun, чтобы браузер Safari работал с нашим модулем `Validation`. Прежде всего скачайте ZIP-файл архива со страницы проекта: github.com/aFarkas/webshim/releases/latest. Разархивируйте его и поместите каталог `js-webshim/webshim` в наш каталог `coffeerun` (рядом с `index.html` и нашим каталогом `scripts`).

Добавьте тег `<script>` для файла `webshim/polyfiller.js` в файл `index.html`:

```
...
      </div>
    </section>
    <script src="https://cdnjs.cloudflare.com/ajax/libs/jquery/2.1.4/jquery.js"
      charset="utf-8"></script>
    <script src="webshim/polyfiller.js" charset="utf-8"></script>
    <script src="scripts/validation.js" charset="utf-8"></script>
    <script src="scripts/checklist.js" charset="utf-8"></script>
    <script src="scripts/formhandler.js" charset="utf-8"></script>
    <script src="scripts/datastore.js" charset="utf-8"></script>
    <script src="scripts/truck.js" charset="utf-8"></script>
    <script src="scripts/main.js" charset="utf-8"></script>
  </body>
</html>
```

Затем добавьте следующие строки в файл `main.js`:

```
...
  var Validation = App.Validation;
  var CheckList = App.CheckList;
  var webshim = window.webshim;
  var myTruck = new Truck('ncc-1701', new DataStore());
  ...
  formHandler.addInputHandler(Validation.isCompanyEmail);
  webshim.polyfill('forms forms-ext');
  webshim.setOptions('forms', { addValidators: true, lazyCustomMessages: true });
}(window));
```

Этот код импортирует библиотеку `webshim`, после чего настраивает ее для работы с формами.

Наконец, еще один относящийся к этой библиотеке нюанс, который вам нужно знать. Где бы вы ни использовали метод `setCustomValidity`, необходимо обертывать объекты с помощью jQuery. В случае CoffeeRun следует в файле `formhandler.js` обернуть объекты `event.target` функции `addInputHandler`:

```
...
  FormHandler.prototype.addInputHandler = function (fn) {
    console.log('Setting input handler for form');
    this.$formElement.on('input', '[name="emailAddress"]', function (event) {
      var emailAddress = event.target.value;
      var message = '';
      if (fn(emailAddress)) {
        $(event.target).setCustomValidity('');
      } else {
        message = emailAddress + ' is not an authorized email address!'
        $(event.target).setCustomValidity(message);
      }
    });
  };
...
```

Разработчики Webshims приняли решение полностью реализовать функциональность полифиллов в виде расширения jQuery. Помимо вышеупомянутого обертывания, никаких изменений в коде делать не надо.

После сохранения изменений можно оценить работу проверки в браузере Safari. Вы увидите, что он тоже сообщит о проблеме, если вы забудете заполнить поле заказа кофе или введете недопустимый адрес электронной почты (рис. 12.8).

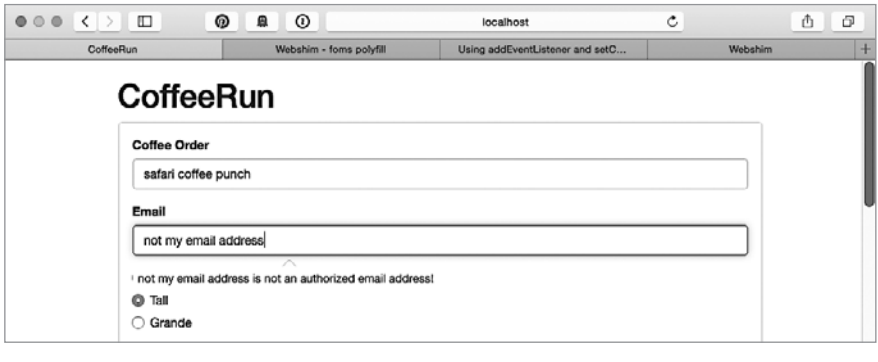


Рис. 12.8. Используем библиотеку Webshims в качестве полифилла для браузера Safari

Webshim способна на большее, чем простая проверка допустимости значений форм. Рекомендуем вам пролистать документацию, чтобы увидеть, чем еще она может помочь вам в ваших проектах.

13 Ajax

В предыдущей главе мы использовали встроенную проверку браузера, чтобы гарантировать соответствие введенных пользователем данных параметрам приложения CoffeeRun. После выполнения этих проверок можно спокойно отправлять данные на сервер.

В настоящий момент метод `FormHandler.prototype.addSubmitHandler` вызывает метод `preventDefault` объекта события, чтобы не допустить отправку браузером запроса к серверу. В обычных условиях сервер возвращает ответ, который вызывает перезагрузку страницы. Вместо этого мы извлекаем данные, введенные пользователем в форму, и обновляем форму и перечень с помощью JavaScript-кода.

В данной главе мы создадим модуль `RemoteDataStore`, отправляющий запрос к серверу и обрабатывающий его ответ (рис. 13.1). Но делать это он будет в фоновом режиме с помощью технологии Ajax (без перезагрузки страницы сервером).

Технология Ajax предлагает методику связи с удаленным сервером посредством JavaScript-кода. JavaScript-код обычно меняет содержимое веб-страницы на основе возвращаемых сервером данных без ее перезагрузки браузером. Это позволяет сделать использование веб-приложения более удобным для пользователя.

Изначально термин Ajax был аббревиатурой выражения Asynchronous JavaScript and XML (асинхронный JavaScript и XML), но сейчас используется как обобщение для асинхронного стиля обмена сообщениями независимо от используемых технологий (асинхронный обмен сообщениями означает, что приложение, отправив запрос, не должно ждать ответа от сервера, прежде чем приступить к выполнению других задач). Ajax в настоящий момент — стандартный механизм для фоновой отправки и получения данных.

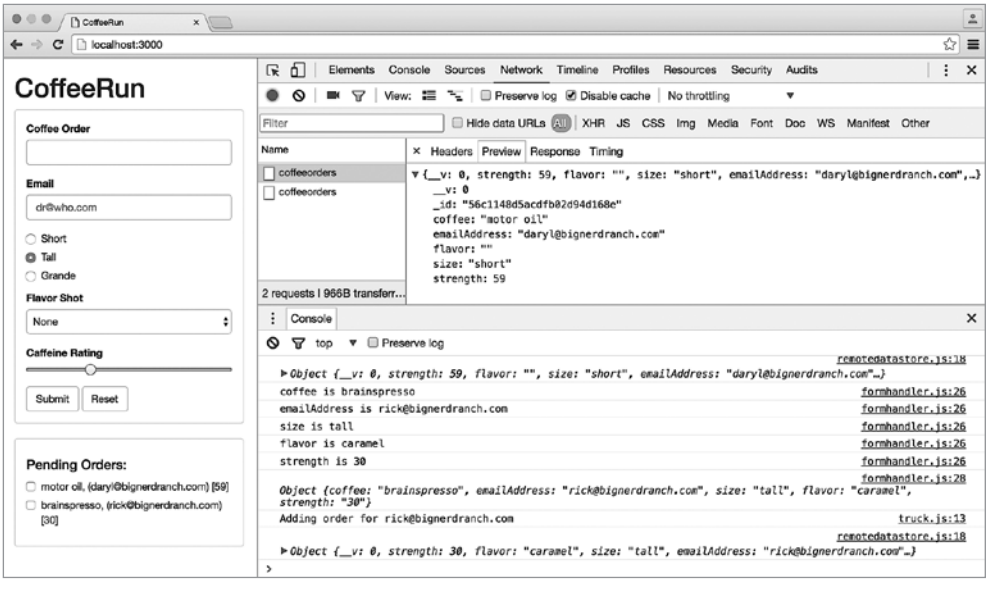


Рис. 13.1. Приложение CoffeeRun по состоянию на конец данной главы

Объекты XMLHttpRequest

В основе технологии Ajax лежит API `XMLHttpRequest`. В современных браузерах можно создавать новые экземпляры объектов `XMLHttpRequest`, предоставляющих возможность отправки запросов серверу без перезагрузки страницы. Они выполняют свою работу в фоновом режиме.

С помощью объектов `XMLHttpRequest` можно связывать обратные вызовы с различными этапами цикла «запрос/ответ», в принципе, аналогично тому, как мы прослушивали события объектов DOM. Можно также проверять свойства объектов `XMLHttpRequest` ради получения информации о состоянии цикла «запрос/ответ». Два полезных свойства — `response` и `status` — обновляются, как только происходят какие-либо изменения. Свойство `response` содержит данные (например, HTML, XML, JSON или в каком-либо другом формате), возвращаемые сервером. `status` — числовой код, сообщающий о том, был ли успешен HTTP-запрос. Официально они носят название *кодов состояния HTTP*.

Коды состояния сгруппированы по диапазонам, объединяемым общим смыслом. Например, коды состояния в диапазоне 200–299 — это коды успешного завершения, в то время как коды состояния в диапазоне 500–599 означают, что произошла ошибка сервера. Мы часто будем видеть, как на эти диапазоны ссылаются обобщенно, например `2xx` или `3xx`.

В табл. 13.1 показаны некоторые распространенные коды.

Таблица 13.1. Наиболее распространенные коды состояния HTTP

Код состояния	Текст состояния	Описание
200	OK	Запрос был выполнен успешно
400	Bad request (Плохой запрос)	Сервер не понял запрос
404	Not Found (Не найдено)	Ресурс не удалось найти (зачастую из-за того, что файл или путь не соответствует данным, имеющимся на сервере)
500	Internal Server Error (Внутренняя ошибка сервера)	Сервер столкнулся с ошибкой, например неперехваченным исключением в коде на стороне сервера
503	Service Unavailable (Сервис недоступен)	Сервер не может обработать запрос (зачастую из-за того, что он перегружен или отключен для обслуживания)

В библиотеке jQuery есть набор методов для создания и управления объектами `XMLHttpRequest`. Она также предоставляет небольшой, обратно совместимый кроссбраузерный API. Это не единственная библиотека, пригодная для управления запросами Ajax, но множество других библиотек просто следуют ее примеру. Мы будем использовать методы `get` и `post` библиотеки jQuery для работы с запросами `GET` и `POST` Ajax, а также метод `ajax` библиотеки jQuery для обработки запросов `DELETE`.

Воплощающие REST веб-сервисы

Мы хотим усовершенствовать приложение CoffeeRun, воспользовавшись удаленным веб-сервисом для хранения данных нашего приложения. Сервер, который мы будем использовать, был создан специально для данной книги.

Сервер приложения CoffeeRun предоставляет воплощающий REST веб-сервис. REST расшифровывается как «передача состояния представления» (representational state transfer) и представляет собой вид веб-сервиса, основанного на HTTP-глаголах (`GET`, `POST`, `PUT` и `DELETE`) и URL, идентифицирующих ресурсы на сервере.

Зачастую пути URL (часть, следующая за именем сервера) ссылаются или на набор объектов (например, `/coffeeorders`) или на отдельные объекты, определяемые идентификаторами (например, `/coffeeorders/[customer email]`).

Это различие влияет на то, как используются HTTP-глаголы. Например, при работе с набором запрос `GET` извлекает список всех элементов в наборе. При работе с отдельным элементом запрос `GET` извлекает детальную информацию об этом элементе.

Относящиеся к URL и глаголам HTTP паттерны кратко подытожены в табл. 13.2.

Таблица 13.2. Примеры воплощающих REST паттернов URL и глаголов HTTP

Путь URL	GET	POST	PUT	DELETE
/coffeeorders	Получить список всех записей	Создать одну запись	–	Удалить все записи
/coffeeorders/a@b.com	Получить запись	–	Изменить запись	Удалить запись

Модуль RemoteDataStore

Сейчас мы создадим новый модуль под названием `RemoteDataStore`. Задача `RemoteDataStore` — связаться с сервером от имени всего остального приложения. Его методы (такие же, как и у модуля `DataStore`) — `add`, `get`, `getAll` и `remove` — будут использоваться для связи с сервером (рис. 13.2).

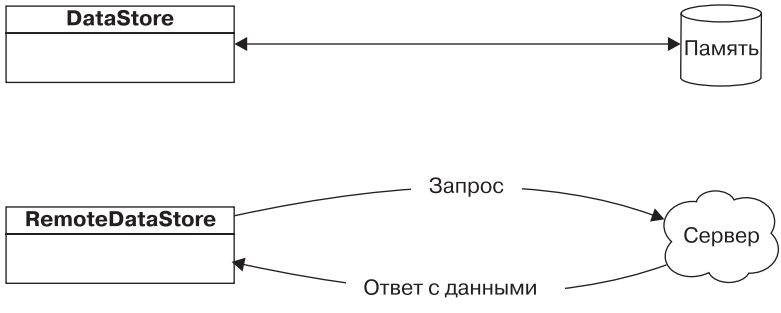


Рис. 13.2. DataStore по сравнению с RemoteDataStore

Мы сможем использовать экземпляр `RemoteDataStore` вместо `DataStore` без необходимости изменения модулей `Truck`, `FormHandler` или `CheckList` (однако мы не будем удалять наш модуль `DataStore`. Нам может понадобиться в дальнейшем усовершенствовать приложение CoffeeRun, дав ему возможность переключаться между двумя модулями хранения в зависимости от того, работает приложение в онлайн- или офлайн-режиме).

Методы модуля `RemoteDataStore` будут взаимодействовать с сервером асинхронно — путем отправки запросов в фоновом режиме. У браузера при получении ответа от сервера будет возможность выполнить обратный вызов.

Каждый из методов модуля `RemoteDataStore` станет принимать на входе функциональный аргумент, который будет вызываться после получения ответа от сервера с данными.

Создайте новый файл `scripts/remotedatastore.js` и добавьте в файл `index.html` тег `<script>` для него:

```
...
    <script src="scripts/validation.js" charset="utf-8"></script>
    <script src="scripts/checklist.js" charset="utf-8"></script>
    <script src="scripts/formhandler.js" charset="utf-8"></script>
    <script src="scripts/remotedatastore.js" charset="utf-8"></script>
    <script src="scripts/datastore.js" charset="utf-8"></script>
    <script src="scripts/truck.js" charset="utf-8"></script>
    <script src="scripts/main.js" charset="utf-8"></script>
  </body>
</html>
```

Сохраните файл `index.html`. Импортируйте в файле `remotedatastore.js` пространство имен `App` и библиотеку jQuery, после чего создайте модуль IIFE с конструктором `RemoteDataStore`. Конструктор должен принимать на входе аргумент для URL удаленного сервера и генерировать ошибку, если URL не передан. В конце описания модуля экспортируйте `RemoteDataStore` в пространство имен `App`:

```
(function (window) {
  'use strict';
  var App = window.App || {};
  var $ = window.jQuery;

  function RemoteDataStore(url) {
    if (!url) {
      throw new Error('No remote URL supplied.');
    }

    this.serverUrl = url;
  }

  App.RemoteDataStore = RemoteDataStore;
  window.App = App;

})(window);
```

Отправка данных на сервер

Первый метод, который мы создадим, — метод `add` для сохранения данных о сделанном посетителем заказе на удаленном веб-сервисе.

Добавьте в модуль `RemoteDataStore` метод прототипа. Аналогично методу `add` модуля `DataStore` он будет принимать на входе аргументы `key` и `val`. Обратите внимание, что использовать те же имена не обязательно, но разумно придерживаться единообразных названий.

```
...
  function RemoteDataStore(url) {
    ...
  }

  RemoteDataStore.prototype.add = function (key, val) {
    // Здесь будет находиться код
  };

  App.RemoteDataStore = RemoteDataStore;
  window.App = App;
...
```

Использование метода $.post библиотеки jQuery

В модуле `RemoteDataStore` мы будем использовать метод `$.post` библиотеки jQuery. Он отправляет запрос `POST` в фоновом режиме в виде объекта `XMLHttpRequest` (рис. 13.3).

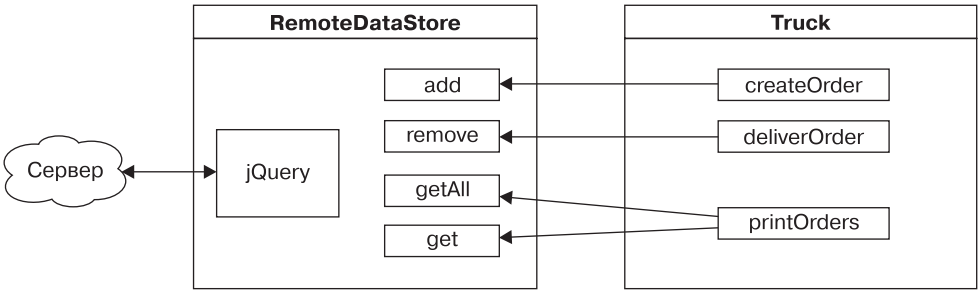


Рис. 13.3. RemoteDataStore использует jQuery для Ajax

Методу `$.post` требуются только два элемента данных: URL сервера для отправки запроса и то, какие данные включить в запрос.

В файле `remotedatastore.js` измените тело метода `add`, чтобы он вызывал метод `$.post`, передавая ему `this.serverUrl` и `val`:

```
...
  RemoteDataStore.prototype.add = function (key, val) {
  // Здесь будет находиться код
    $.post(this.serverUrl, val);
  };

  App.RemoteDataStore = RemoteDataStore;
  window.App = App;
...
```

Обратите внимание, что аргумент `key` не используется. Он остается частью объявления метода, так что метод `add` из модуля `RemoteDataStore` идентичен методу `add` из

модуля DataStore. Оба принимают на входе информацию о заказе кофе в качестве второго аргумента. Для RemoteDataStore это играет ключевую роль.

Добавление обратного вызова

Аналогично множеству других методов jQuery $.post может принимать дополнительные необязательные аргументы. Мы будем передавать ему функцию обратного вызова в качестве третьего аргумента. При получении ответа от сервера эта функция будет вызываться с передачей в нее полученных в ответе данных.

Это похоже на написанный нами ранее код обработки событий — мы регистрируем функцию, которая будет запускаться в определенный момент в будущем. При обработке событий этим моментом может быть щелчок кнопкой мыши или отправка формы. При обработке удаленных данных им будет поступление ответа от сервера.

Добавьте в метод $.post анонимную функцию в качестве третьего аргумента. Эта анонимная функция ожидает на входе аргумент, который мы пометим названием serverResponse. Выведите serverResponse в консоль с помощью вызова функции console.log:

```
...
  RemoteDataStore.prototype.add = function (key, val) {
    $.post(this.serverUrl, val, function (serverResponse) {
      console.log(serverResponse);
    });
  };

  App.RemoteDataStore = RemoteDataStore;
  window.App = App;
...
```

Теперь методу $.post известны три вещи: с кем ему связываться, что передавать и что делать с полученной в ответ информацией.

После сохранения этих изменений в файле remotedatastore.js запустите утилиту browser-sync и откройте консоль в браузере. Создайте экземпляр RemoteDataStore с приведенным ниже URL, представляющим собой адрес тестового сервера, созданного специально для этой книги (еще раз повторим, что эта строка разорвана, чтобы поместиться на странице, поэтому вводите ее одной строкой):

```
var remoteDS = new App.RemoteDataStore
    ("http://coffeerun-v2-rest-api.herokuapp.com/api/coffeeorders");
```

Теперь вызовите метод add, передав ему какие-нибудь тестовые данные:

```
remoteDS.add('a@b.com', {emailAddress: 'a@b.com', coffee: 'espresso'});
```

Посмотрите в консоли, что было выведено нашим оператором console.log (рис. 13.4).

```
> var remoteDS = new App.RemoteDataStore("http://coffeerun-v2-rest-api.herokuapp.com/api/coffeeorders");
< undefined
> remoteDS.add('a@b.com', {emailAddress: 'a@b.com', coffee: 'espresso'});
< undefined
                                                                    remotedatastore.js:17
  Object {__v: 0, coffee: "espresso", emailAddress: "a@b.com", _id: "5712c496e36db403007d087c"}
>
```

Рис. 13.4. Консоль отображает результаты вызова метода RemoteDataStore.add

Объект в операторе вывода в консоль содержит информацию, отправленную сервером в его ответе: данные `coffee` и `emailAddress`, помимо некоторых технических деталей, варьирующихся от сервера к серверу.

Исследуем запрос и ответ Ajax

Откройте в DevTools панель сети, выбрав команду Network (Сеть) в меню вверху (между Sources и Timeline). Эта панель отображает список запросов, выполненных вашим браузером, и дает вам возможность изучить каждый из них более подробно (рис. 13.5).

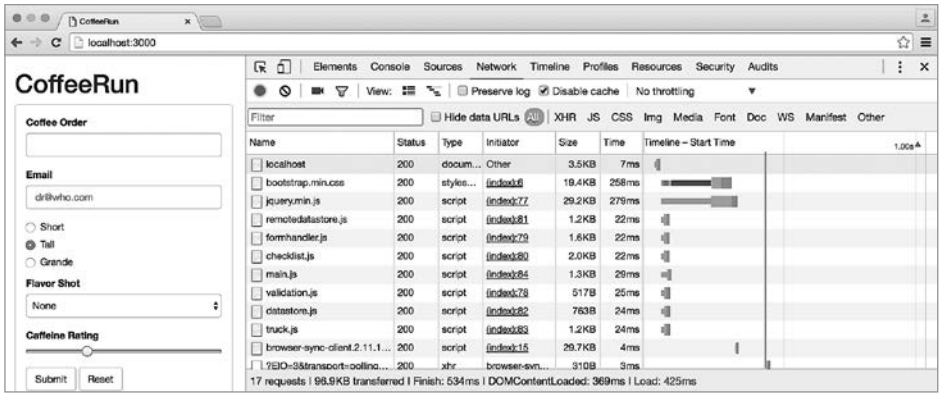


Рис. 13.5. Просмотр запросов Ajax на панели сети

Вероятно, на вашей панели сети будет множество сетевых запросов в списке. Очистите его, щелкнув на значке слева вверху DevTools. После этого активизируйте всплывающую панель внизу, чтобы видеть и консоль, и панель сети. Это можно сделать, нажав клавишу Esc или щелкнув на значке ⊘ справа вверху. При этом откроется меню с пунктом Show console (Отобразить консоль).

Выдвижная панель консоли появится внизу DevTools (рис. 13.6).

Введите в консоли следующий код:

```
var remoteDS = new App.RemoteDataStore
    ("http://coffeerun-v2-rest-api.herokuapp.com/api/coffeeorders");
remoteDS.add('a@b.com', {emailAddress: 'a@b.com', coffee: 'espresso'});
```

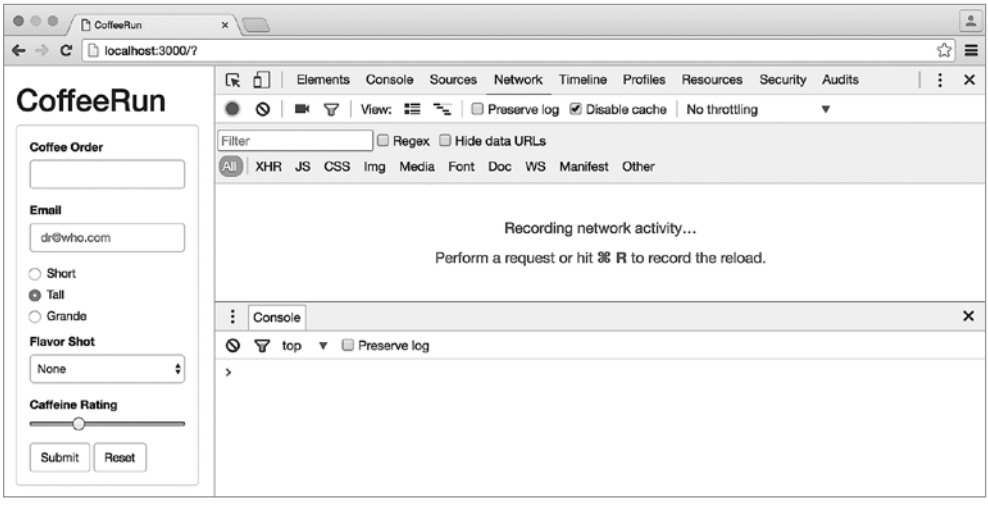


Рис. 13.6. Внизу от панели сети открывается всплывающая панель консоли

Вы увидите новую запись на панели сети (рис. 13.7).

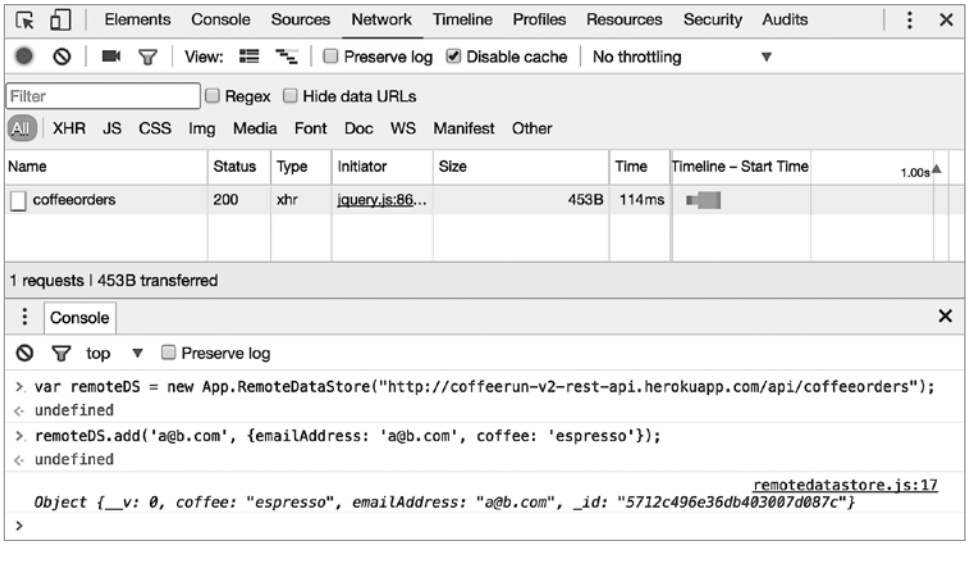


Рис. 13.7. Запрос Ajax на панели сети

Чтобы узнать больше подробностей о запросе, щелкните на соответствующей записи (рис. 13.8). Возможно, вам будет удобнее просматривать их, если вы спрячете всплывающую панель консоли.

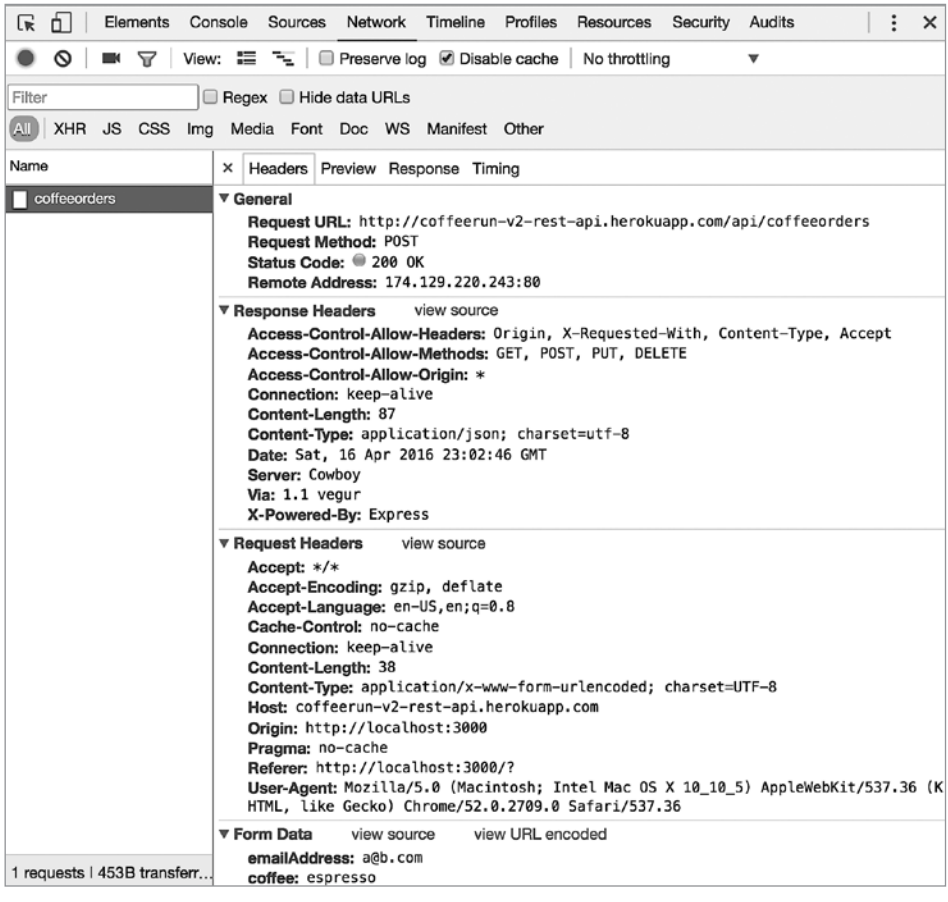


Рис. 13.8. Подробности запроса

Подробности включают определенную общую информацию о запросе вверху и данные формы внизу. В середине находятся субпанели, отображающие информацию о *заголовках* запроса и ответа. Заголовки представляют собой метаданные и заданные для запроса и ответа параметры.

Из всех этих данных наиболее полезными для разработки и отладки запросов Ajax являются код состояния (на подпанели General) и подпанель Form Data.

Извлечение данных с сервера

Наш модуль `RemoteDataStore` может сохранять отдельные заказы кофе на сервере. Далее нам нужно добавить метод прототипа `getAll` для извлечения всех заказов с сервера. Начнем работу в файле `remotedatastore.js`:

```
...
  RemoteDataStore.prototype.add = function (key, val) {
    ...
  };

  RemoteDataStore.prototype.getAll = function () {
    // Здесь будет находиться код
  };

  App.RemoteDataStore = RemoteDataStore;
  window.App = App;
...
```

Далее мы добавим вызов метода `$.get` библиотеки jQuery. Аналогично `$.post` мы передадим в него URL сервера, но не будем отправлять никаких данных, поскольку мы извлекаем, а не сохраняем информацию. Нам придется передать ему функциональный аргумент, чтобы он знал, что делать с данными, полученными с сервера.

Вызовите метод `$.get` в `RemoteDataStore.prototype.getAll`:

```
...
  RemoteDataStore.prototype.getAll = function () {
    // Здесь будет находиться код
    $.get(this.serverUrl, function (serverResponse) {
      console.log(serverResponse);
    });
  };

  App.RemoteDataStore = RemoteDataStore;
  window.App = App;
...
```

Сохраните и вернитесь к DevTools в браузере.

Изучаем данные ответа

Создайте в консоли экземпляр `RemoteDataStore` с тем же URL, что и ранее (продвинутая рекомендация: вместо того чтобы набирать очень длинную строку с URL, можно использовать стрелки «вверх» и «вниз» для перехода по введенным ранее в консоли операторам). После этого вызовите его метод `getAll`:

```
var remoteDS = new App.RemoteDataStore
    ("http://coffeerun-v2-rest-api.herokuapp.com/api/coffeeorders");
remoteDS.getAll();
```

Вы увидите на панели сети DevTools, что запрос `GET` был отправлен. Ответ должен прийти через несколько десятков миллисекунд. В консоли появятся (рис. 13.9) данные каких-то заказов кофе (из заранее загруженных на сервер).

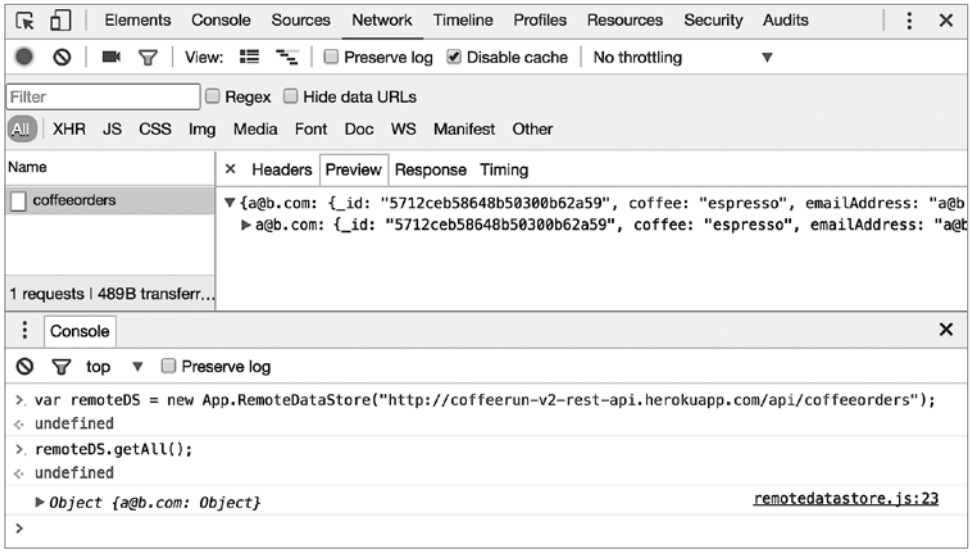


Рис. 13.9. Изучаем ответ, полученный из метода getAll

Результаты, которые увидите вы, могут слегка отличаться (в зависимости от того, что было добавлено на сервер). Однако получение любого результата подтверждает успешное извлечение данных с сервера.

Добавляем аргумент-обратный вызов

Мы можем извлекать данные с сервера, но не можем вернуть их из метода `getAll` (потому что `getAll` только выполняет первоначальный запрос Ajax, но не обрабатывает ответ). Вместо этого передадим в метод `$.get` обратный вызов для обработки ответа. Этот вызов будет функционировать аналогично созданным нами ранее обратным вызовам обработки событий (в обоих случаях обратный вызов должен ожидать на входе аргумент). Это значит, что данные ответа доступны только внутри тела обратного вызова. Как же получить к ним доступ вне обратного вызова?

Если мы передадим `getAll` функциональный аргумент, то сможем вызвать эту функцию *внутри* обратного вызова `$.get`. Там у нас есть доступ как к функциональному аргументу, *так и* к ответу сервера.

Добавьте функциональный аргумент и вызов в файле `remotedatastore.js`:

```
...
  RemoteDataStore.prototype.getAll = function (cb) {
    $.get(this.serverUrl, function (serverResponse) {
      console.log(serverResponse);
```

```
      cb(serverResponse);
    });
  };

  App.RemoteDataStore = RemoteDataStore;
  window.App = App;
...
```

Метод `getAll` извлекает все заказы кофе, имеющиеся на удаленном сервере, и отправляет их переданной в него функции обратного вызова `cb`.

Нам также нужно реализовать метод `get`, извлекающий отдельный заказ кофе по адресу электронной почты посетителя. Аналогично `getAll` он принимает на входе функциональный аргумент, который затем вызывает, передавая ему извлеченный заказ.

Добавьте эту реализацию `get` в файл `remotedatastore.js`:

```
...
  RemoteDataStore.prototype.getAll = function (cb) {
    ...
  };

  RemoteDataStore.prototype.get = function (key, cb) {
    $.get(this.serverUrl + '/' + key, function (serverResponse) {
      console.log(serverResponse);
      cb(serverResponse);
    });
  };

  App.RemoteDataStore = RemoteDataStore;
  window.App = App;
...
```

Сохраните изменения в файле `remotedatastore.js`. Введите следующий код в консоли, передавая методу `remoteDS.get` пустую анонимную функцию (он ожидает на входе функциональный аргумент, но мы хотим всего лишь протестировать его по-быстрому):

```
var remoteDS = new App.RemoteDataStore
    ("http://coffeerun-v2-rest-api.herokuapp.com/api/coffeeorders");
remoteDS.get('a@b.com', function () {});
```

Ваша консоль при этом будет выглядеть примерно как на рис. 13.10.

Удаление данных с сервера

Благодаря Ajax мы теперь можем сохранять заказы на сервере и извлекать их оттуда. Последнее, что нам осталось сделать, — удалять заказы с сервера после их выдачи.

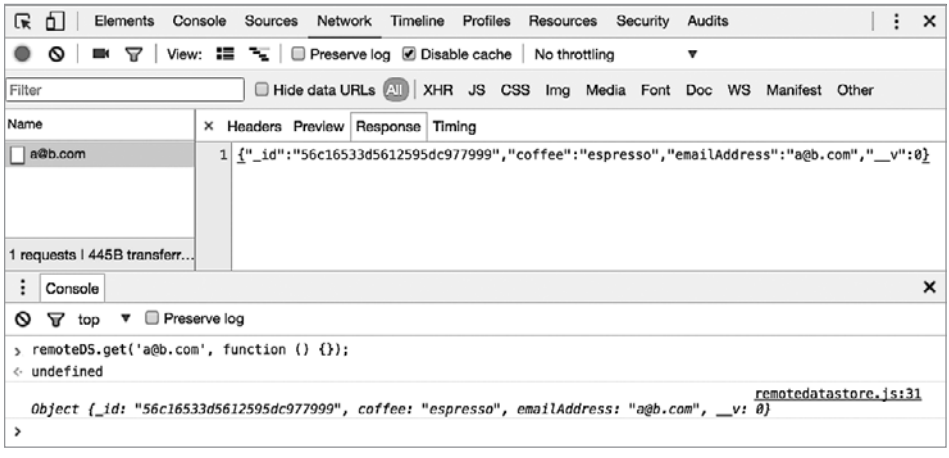


Рис. 13.10. Проверка RemoteDataStore.prototype.get

Для этого мы будем отправлять HTTP-запрос по URL конкретного заказа. Как и в случае с методом `RemoteDataStore.prototype.get`, мы будем использовать URL сервера, но добавим в его конец косую черту и адрес электронной почты посетителя.

Отправлять на сервер мы будем запрос `DELETE`. `DELETE` — один из глаголов HTTP. Сервер будет знать при получении запроса `DELETE` по этому URL, что нужно удалить данные, связанные с соответствующим адресом электронной почты посетителя.

Использование метода $.ajax библиотеки jQuery. Для удобства разработчиков jQuery предоставляет методы `$.get` и `$.post` (поскольку это два чаще всего применяемых HTTP-глагола). Например, запрос `GET` используется всякий раз, когда браузер запрашивает файл HTML, CSS, JavaScript или файл изображения (помимо всего прочего). Запрос `POST` применяется чаще всего при отправке форм.

jQuery не предоставляет удобного метода для отправки с помощью Ajax запросов `DELETE`. Вместо него приходится использовать метод `$.ajax` (фактически методы `$.get` и `$.post` тоже вызывают метод `$.ajax`, указывая в качестве HTTP-глаголов `GET` и `POST`).

Добавьте в файл `remotedatastore.js` метод прототипа `remove`. В нем вызовите метод `$.ajax`, передав ему два аргумента. Первый — URL отдельного заказа кофе, сформированный из URL сервера, прямой косой черты и `key` (адрес электронной почты посетителя). Второй — объект, содержащий параметры (настройки) для запроса Ajax. Единственный параметр, который вам нужно указать для метода `remove`, — тип запроса `DELETE`.

```
...
  RemoteDataStore.prototype.get = function (key, cb) {
    ...
  };
```

```
  RemoteDataStore.prototype.remove = function (key) {
    $.ajax(this.serverUrl + '/' + key, {
      type: 'DELETE'
    });
  };

  App.RemoteDataStore = RemoteDataStore;
  window.App = App;
...
```

Существует множество параметров запросов Ajax, о которых можно прочитать по адресу api.jquery.com/jquery.ajax/.

Сохраните изменения и вернитесь в консоль. Создайте новый экземпляр `RemoteDataStore` и вызовите метод `remove`. Передайте ему адрес электронной почты для созданного вами тестового заказа. Наконец, вызовите метод `getAll`, чтобы убедиться, что этот заказ более не включается в возвращаемые с сервера заказы.

```
var remoteDS = new App.RemoteDataStore
    ("http://coffeerun-v2-rest-api.herokuapp.com/api/coffeeorders");
remoteDS.remove('a@b.com');
remoteDS.getAll(function (data) { console.log(data); });
```

Если вы внимательно изучите ответ сервера для запроса `DELETE`, то увидите, что сервер возвращает информацию о том, что он сделал (рис. 13.11). Как мы уже упоминали, другие серверы могут возвращать в ответе другую информацию.

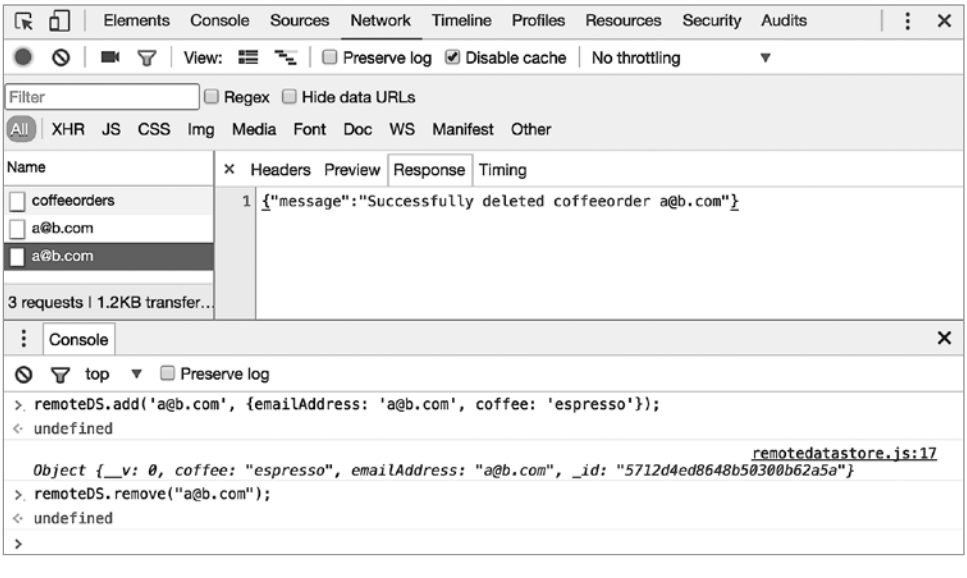


Рис. 13.11. Изучаем ответ для метода DELETE

Заменяем DataStore на RemoteDataStore

Мы закончили разработку модуля `RemoteDataStore`. Пришло время заменить экземпляр `DataStore` на экземпляр `RemoteDataStore`.

Откройте файл `main.js`. Начните с импорта `RemoteDataStore` из пространства имен `App`:

```
(function (window) {
  'use strict';
  var FORM_SELECTOR = '[data-coffee-order="form"]';
  var CHECKLIST_SELECTOR = '[data-coffee-order="checklist"]';
  var App = window.App;
  var Truck = App.Truck;
  var DataStore = App.DataStore;
  var RemoteDataStore = App.RemoteDataStore;
  var FormHandler = App.FormHandler;
...
```

Кроме того, добавьте новую переменную `SERVER_URL` и присвойте ей строку с URL тестового сервера приложения CoffeeRun.

```
(function (window) {
  'use strict';
  var FORM_SELECTOR = '[data-coffee-order="form"]';
  var CHECKLIST_SELECTOR = '[data-coffee-order="checklist"]';
  var SERVER_URL = 'http://coffeerun-v2-rest-api.herokuapp.com/api/
                    coffeeorders';
  var App = window.App;
  var Truck = App.Truck;
  var DataStore = App.DataStore;
  var RemoteDataStore = App.RemoteDataStore;
...
```

Далее создайте новый экземпляр `RemoteDataStore`, передав ему переменную `SERVER_URL`.

```
...
  var RemoteDataStore = App.RemoteDataStore;
  var FormHandler = App.FormHandler;
  var Validation = App.Validation;
  var CheckList = App.CheckList;
  var remoteDS = new RemoteDataStore(SERVER_URL);
  var myTruck = new Truck('ncc-1701', new DataStore());
  window.myTruck = myTruck;
...
```

И наконец, вместо отправки конструктору `Truck` нового экземпляра `DataStore` передайте ему переменную `remoteDS`. Поскольку имена методов `DataStore` и `RemoteDataStore` и большинство принимаемых ими на входе аргументов совпадают, это изменение пройдет незаметно.

```
...
  var RemoteDataStore = App.RemoteDataStore;
  var FormHandler = App.FormHandler;
  var Validation = App.Validation;
  var CheckList = App.CheckList;
  var remoteDS = new RemoteDataStore(SERVER_URL);
  var myTruck = new Truck('ncc-1701', new DataStore()); remoteDS);
  window.myTruck = myTruck;
...
```

Сохраните изменения и вернитесь в браузер. Введите какую-нибудь информацию для заказа кофе и подтвердите отправку формы. Вы должны увидеть сетевые транзакции для каждого добавленного вами через форму заказа кофе или помеченного в перечне, что был выдан (рис. 13.12).

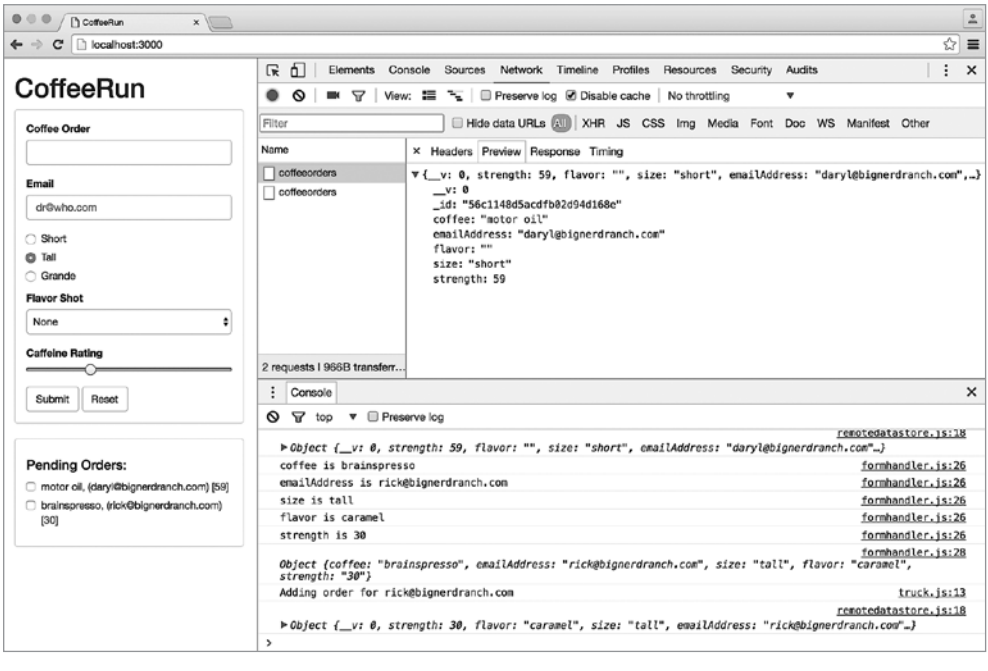


Рис. 13.12. Сохранение заказов кофе на удаленном сервере

Поздравляем! Приложение CoffeeRun полнофункционально и интегрировано с удаленным веб-сервером.

Следующая глава — последняя касающаяся CoffeeRun. В ней в приложение CoffeeRun не добавляются новые возможности. Она концентрируется на переделке уже существующего кода, чтобы предоставить вам возможность изучить новый паттерн по работе с асинхронным кодом.

Серебряное упражнение: сверка с удаленным сервером

Наш код проверки сейчас выполняет проверку домена. Исправьте код, чтобы он, помимо этого, проверял, был ли данный адрес электронной почты уже использован для имеющегося на сервере заказа. Блокируйте отправку формы, если адрес уже был использован, и выведите соответствующее предупреждение о неудаче при проверке.

Откройте второе окно браузера для приложения CoffeeRun и введите различные заказы кофе в два окна.

Обратите внимание на частоту отправки запросов на сервер при выполнении данной проверки допустимости (это можно посмотреть на панели сети DevTools). Нельзя ли отыскать приемлемый способ минимизации количества запросов?

Для самых любознательных: Postman

Один из лучших инструментов для отправки тестовых запросов на сервер — Postman, бесплатный плагин для Chrome. Он позволяет формировать HTTP-запросы, задавая HTTP-глаголы, данные формы, заголовки и учетные данные пользователя (рис. 13.13).

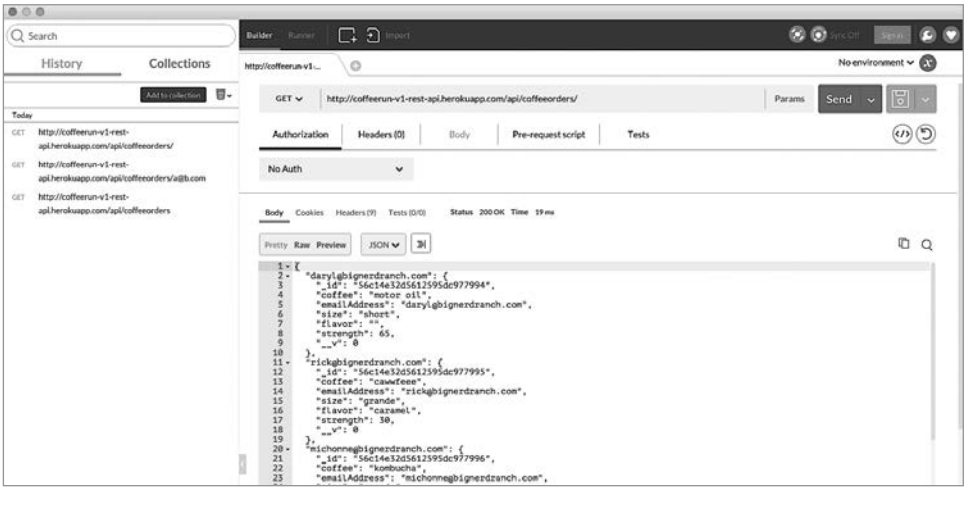


Рис. 13.13. Postman

Postman — незаменимый инструмент, если необходимо исследовать API до написания кода взаимодействия с сервером. Скачайте его в интернет-магазине Chrome: chrome.google.com/webstore/ (чтобы найти его, выполните поиск по ключевому слову `Postman`).

14 Объекты Deferred и Promise

Модульный код в CoffeeRun позволил нам избежать спагетти-кода — вполне возможного исхода при смешении кода обработки событий (UI) с внутренней логикой приложения.

Наши модули взаимодействуют с помощью функциональных аргументов, известных также как обратные вызовы. Обратные вызовы — отличное решение при наличии кода, зависящего только от одного асинхронного шага. Рисунок 14.1 демонстрирует упрощенную версию одного асинхронного потока выполнения из приложения CoffeeRun.

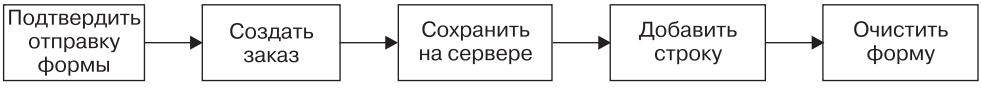


Рис. 14.1. Асинхронный поток выполнения для добавления заказа кофе

```
formHandler.addSubmitHandler(function (data) {
  try {
    myTruck.createOrder(function (error) {
      if (error) {
        throw new Exception(error)
      } else {
        try {
          saveOnServer(function (error) {
            if (error) {
              throw new Exception({message: 'server error'});
            } else {
              try {
                checkList.addRow();
              } catch (e2) {
```

```
                handleDomError(e2);
              }
            }
          })
        } catch (e) {
          handleServerError(e, function () {
            // Снова пытаемся добавить строку
            try {
              checkList.addRow();
            } catch (e3) {
              handleDomError(e3);
            }
          });
        }
      }
    });
  } catch (e) {
    alert('Something bad happened.');
  }
});
```

Объекты `Promise`, с которыми мы познакомимся в этой главе, — намного более удачное решение. Ту же самую последовательность шагов можно выразить в виде такой *цепочки* объектов `Promise`:

```
formHandler.addSubmitHandler()
  .then(myTruck.createOrder)
  .then(saveOnServer)
  .catch(handleServerError)
  .then(checkList.addRow)
  .catch(handleDomError);
```

Объекты `Promise` помогают проектировать очень сложный асинхронный код, и в данной главе мы воспользуемся ими для упрощения архитектуры приложения CoffeeRun. Объекты `Promise` — относительно новая возможность, но их поддерживают все современные браузеры, включая Chrome.

В CoffeeRun нам в основном нужно выполнять следующий шаг тогда, когда текущий выполнился без ошибок. Объекты `Promise` делают это элементарным. Вместо того чтобы использовать аргументы — обратные вызовы, мы будем возвращать объекты `Promise`, что позволит еще больше расцепить наши модули между собой.

Объекты Promise и Deferred

Объекты `Promise` всегда находятся в одном из трех состояний: *ожидание выполнения*, *выполнено успешно* или *отказано в выполнении*.

У каждого объекта `Promise` имеется метод `then`, срабатывающий при его переходе в состояние «выполнено успешно». Можно вызвать `then` и передать в него об-

ратный вызов; когда Promise будет (успешно) выполнен, произойдет обращение к обратному вызову с передачей ему полученного при асинхронном выполнении объекта Promise значения.

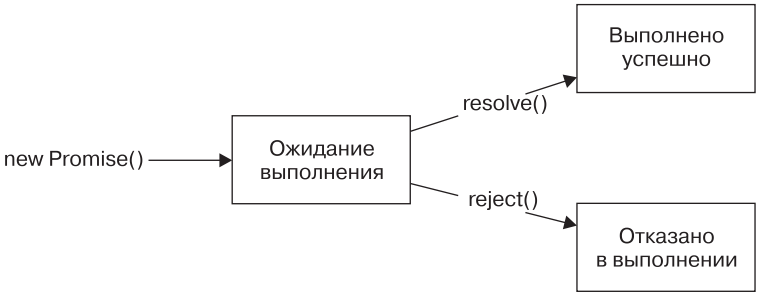


Рис. 14.2. Три состояния объекта Promise

Можно также связать в цепочку несколько вызовов then. Вместо того чтобы писать функции, принимающие и затем выполняющие обратные вызовы, лучше возвращать объекты Promise, чтобы вызывающая сторона привязывала цепочкой вызов метода then для этого Promise.

Мы начнем с объекта Deferred библиотеки jQuery, функционирующего при простых сценариях использования аналогично Promise.

Методы $.ajax библиотеки jQuery (включая $.post и $.get) возвращают Deferred. У объектов Deferred имеются методы, позволяющие регистрировать обратные вызовы для двух состояний: *выполнено успешно* и *отказано в выполнении*. Мы начнем с изменения модуля RemoteDataStore, чтобы он возвращал создаваемые методами Ajax библиотеки jQuery объекты Deferred. Далее мы изменим другие наши модули для регистрации обратных вызовов с помощью Deferred.

Возвращаем Deferred

Воспользуемся объектами Deferred, возвращаемыми методами $.ajax библиотеки jQuery. Измените методы прототипа в файле remotedatastore.js, чтобы они возвращали результаты вызовов методов $.get, $.post и $.ajax:

```
...
  RemoteDataStore.prototype.add = function (key, val) {
    return $.post(this.serverUrl, val, function (serverResponse) {
      console.log(serverResponse);
    });
  };

  RemoteDataStore.prototype.getAll = function (cb) {
    return $.get(this.serverUrl, function (serverResponse) {
```

```
      console.log(serverResponse);
      cb(serverResponse);
    });
  };

  RemoteDataStore.prototype.get = function (key, cb) {
    return $.get(this.serverUrl + '/' + key, function (serverResponse) {
      console.log(serverResponse);
      cb(serverResponse);
    });
  };

  RemoteDataStore.prototype.remove = function (key) {
    return $.ajax(this.serverUrl + '/' + key, {
      type: 'DELETE'
    });
  };
...
```

Поскольку теперь они возвращают создаваемый методами Ajax библиотеки jQuery объект `Deferred`, отсутствует необходимость в том, чтобы методы `get` и `getAll` принимали обратные вызовы в качестве параметров. Дабы учесть возможность отсутствия обратного вызова, вставим оператор `if` для проверки (до вызова), был ли передан аргумент `cb`:

```
...
  RemoteDataStore.prototype.getAll = function (cb) {
    return $.get(this.serverUrl, function (serverResponse) {
      if (cb) {
        console.log(serverResponse);
        cb(serverResponse);
      }
    });
  };

  RemoteDataStore.prototype.get = function (key, cb) {
    return $.get(this.serverUrl + '/' + key, function (serverResponse) {
      if (cb) {
        console.log(serverResponse);
        cb(serverResponse);
      }
    });
  };
...
```

Сохраните файл `remotedatastore.js`. Поскольку методы модуля `RemoteDataStore` теперь возвращают объекты `Deferred`, необходимо поменять методы модуля `Truck`, чтобы они делали то же самое. Пока что сконцентрируемся на методах `createOrder` и `deliverOrder`.

Откройте файл `truck.js` и добавьте `return` к этим двум методам — там, где они вызываются для объекта `this.db`.

```
...
  Truck.prototype.createOrder = function (order) {
    console.log('Adding order for ' + order.emailAddress);
    return this.db.add(order.emailAddress, order);
  };

  Truck.prototype.deliverOrder = function (customerId) {
    console.log('Delivering order for ' + customerId);
    return this.db.remove(customerId);
  };
...
```

Сохраните файл `truck.js`. `Truck` теперь возвращает объекты `Deferred`, порождаемые `RemoteDataStore`. При использовании объектов `Promise` и `Deferred` рекомендуется возвращать их из ваших функций. Их возврат даст возможность всем вызывающим методы `createOrder` и `deliverOrder` объектам регистрировать обратные вызовы, которые будут срабатывать после завершения асинхронных операций.

В следующем разделе мы как раз этим и займемся.

Регистрация обратных вызовов с помощью then

Метод `$.ajax` возвращает объект `Deferred`, у которого имеется метод `then`. Метод `then` регистрирует обратный вызов, запускаемый при разрешении `Deferred`. При вызове обратного вызова в него передается значение, присланное в ответе сервера (рис. 14.3).

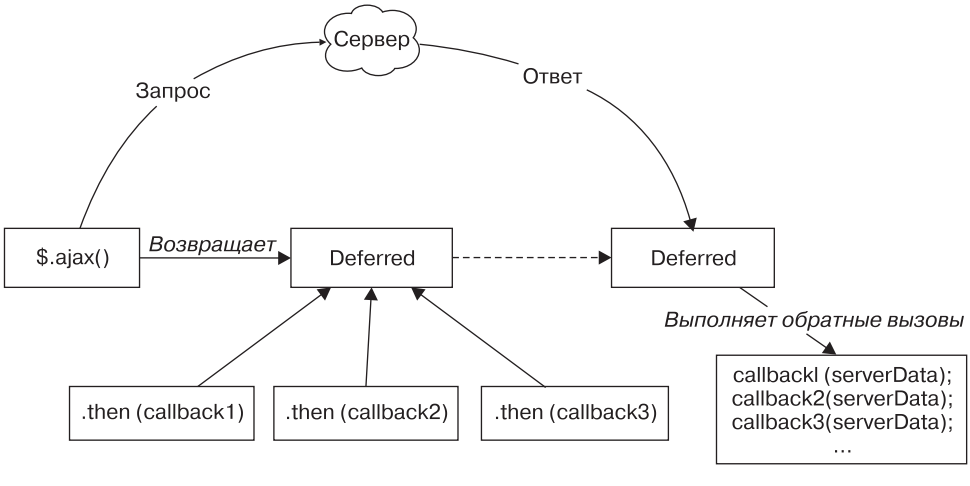


Рис. 14.3. Объект Deferred выполняет обратные вызовы, зарегистрированные с помощью then

Начнем с простого варианта использования метода `then`. В файле `main.js` наш обработчик подтверждения отправки формы вызывает методы `createOrder` и `addRow`.

Измените это таким образом, чтобы регистрировать метод `addRow` в качестве обратного вызова для метода `createOrder`.

Откройте файл `main.js` и измените вызов метода `formHandler.addSubmitHandler`. Привяжите `.then` цепочкой к вызову метода `createOrder`. Передайте ему обратный вызов, выполняющий метод `checkList.addRow`.

```
...
  formHandler.addSubmitHandler(function (data) {
    myTruck.createOrder.call(myTruck, data)
      .then(function () {
        checkList.addRow.call(checkList, data);
      });
  });
...
```

Вместо того чтобы вызывать метод `addRow` сразу же после метода `createOrder`, мы ставим `addRow` в зависимость от завершения (без ошибок и исключений) выполнения `createOrder`.

Обработка сбоев с помощью then

Метод `then` принимает на входе второй аргумент, вызываемый в случае перехода объекта `Deferred` в состояние «отказано в выполнении». Чтобы увидеть это на деле, добавьте второй функциональный аргумент (не забыв поставить между двумя функциональными аргументами запятую) в методе `formHandler.addSubmitHandler` в файле `main.js`. В этой функции выведите предупреждение с простым сообщением об ошибке:

```
...
  formHandler.addSubmitHandler(function (data) {
    myTruck.createOrder.call(myTruck, data)
      .then(function () {
        checkList.addRow.call(checkList, data);
      },
      function () {
        alert('Server unreachable. Try again later.');
      }
      );
  });
...
```

Поменяйте имя сервера вверху файла `main.js` таким образом, чтобы при запросах Ajax происходил сбой (это изменение всего лишь временное, так что можете просто вырезать часть URL, чтобы затем вставить ее обратно):

```
(function (window) {
  'use strict';
  var FORM_SELECTOR = '[data-coffee-order="form"]';
  var CHECKLIST_SELECTOR = '[data-coffee-order="checklist"]';
```

```
  var SERVER_URL = 'http://coffeerun-v2-rest-api.herokuapp.com/api/
                   coffeeorders/';
  var App = window.App;
...
```

Сохраните изменения, убедитесь, что утилита browser-sync запущена, и откройте приложение CoffeeRun в браузере. Заполните форму. Вы увидите предупреждение, всплывающее при подтверждении ее отправки (рис. 14.4).

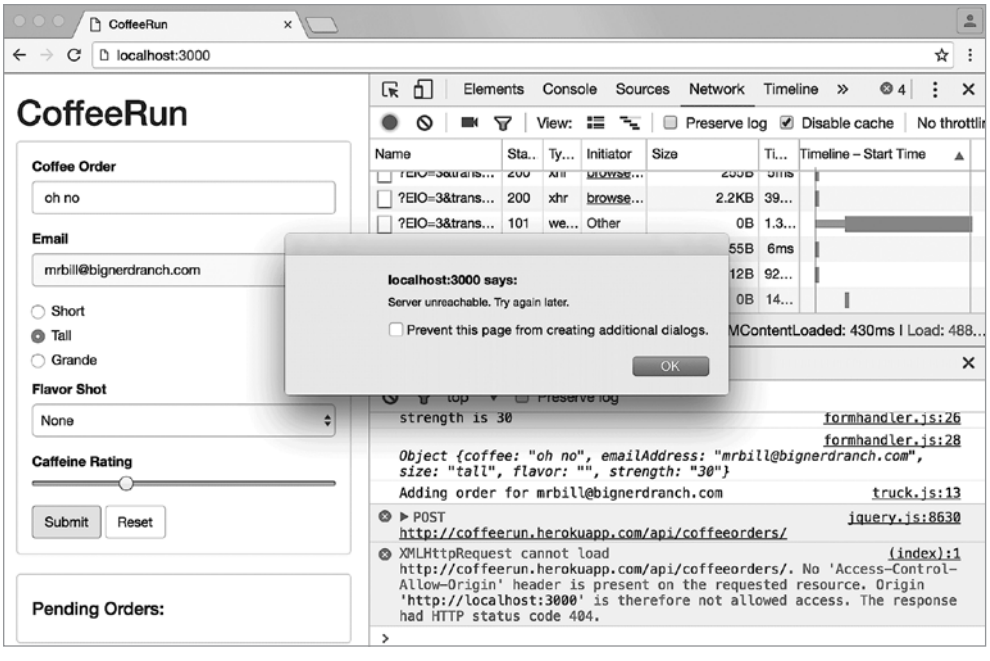


Рис. 14.4. Предупреждение, отображаемое при сбое вызова Ajax

Восстановите прежнее значение переменной `SERVER_URL` на http://coffeerun-v2-rest-api.herokuapp.com/api/coffeeorders/. Можете также удалить выводящий предупреждение (`alert`) функциональный аргумент.

```
(function (window) {
  'use strict';
  var FORM_SELECTOR = '[data-coffee-order="form"]';
  var CHECKLIST_SELECTOR = '[data-coffee-order="checklist"]';
  var SERVER_URL = 'http://coffeerun-v2-rest-api.herokuapp.com/api/
                   coffeeorders/';

  ...

  formHandler.addSubmitHandler(function (data) {
    myTruck.createOrder.call(myTruck, data)
```

```
      .then(function () {
        checkList.addRow.call(checkList, data);
      },
      function () {
        alert('Server unreachable. Try again later.');
      }
      );
  });
...
```

Использование метода `then` для регистрации обратных вызовов хорошо вписывается в образ работы `Promise`. Если объект `Promise` меняет состояние на «выполнено успешно», запускается один набор обратных вызовов; если `Promise` меняет состояние на «отказано в выполнении», то другой.

Использование объектов Deferred с API, основанными на использовании обратных вызовов

Иногда необходимо согласовывать основанный на объекте `Deferred` код с API, основанными на использовании обратных вызовов, такими как прослушиватели событий.

В настоящий момент метод `formHandler.addSubmitHandler` сбрасывает состояние формы и помещает фокус на первый элемент независимо от того, что происходит с запросом Ajax. Однако нам хотелось бы, чтобы это происходило только в случае успешного выполнения запроса Ajax. Другими словами, чтобы это происходило, только если `Deferred` выполнен успешно.

Как узнать, что объект `Deferred` выполнен успешно? Наш функциональный аргумент для метода `addSubmitHandler` может вернуть объект `Deferred`, а внутри метода `addSubmitHandler` можно привязать вызов метода `.then` цепочкой к объекту `Deferred`.

Добавьте в файле `main.js` в обратный вызов ключевое слово `return`:

```
...
  formHandler.addSubmitHandler(function (data) {
    return myTruck.createOrder.call(myTruck, data)
      .then(function () {
        checkList.addRow.call(checkList, data);
      });
  });
...
```

Сохраните файл `main.js`. Далее откройте файл `formhandler.js`, найдите метод `addSubmitHandler` и вызов анонимной функции `fn`. Поскольку эта анонимная функция теперь возвращает объект `Deferred`, можно привязать к ней цепочкой вызов

метода `.then`. Воспользуйтесь `.then` для регистрации обратного вызова, который бы очищал форму и помещал фокус на первый элемент:

```
...
  FormHandler.prototype.addSubmitHandler = function (fn) {
    console.log('Setting submit handler for form');
    this.$formElement.on('submit', function (event) {
      event.preventDefault();

      var data = {};
      $(this).serializeArray().forEach(function (item) {
        data[item.name] = item.value;
        console.log(item.name + ' is ' + item.value);
      });
      console.log(data);
      fn(data);
      .then(function () {
        this.reset();
        this.elements[0].focus();
      });
    });
  };
...
```

Ранее у нас было три последовательных оператора: выполнение обратного вызова, сброс формы и помещение фокуса на первый элемент. Теперь же мы получили операторы, зависящие от результатов предыдущих. Мы выполняем обратный вызов и — *если* выполнение завершается успешно без генерации исключений — очищаем форму и помещаем фокус на первый элемент.

Остался только один нюанс. Когда мы регистрируем функцию обратного вызова с помощью метода `.then`, она получает новую область видимости. Необходимо выполнить `.bind` этой анонимной функции, чтобы переменная `this` указывала на экземпляр `FormHandler`.

Выполните это изменение в файле `formhandler.js`:

```
...
  FormHandler.prototype.addSubmitHandler = function (fn) {
    ...
      fn(data)
        .then(function () {
          this.reset();
          this.elements[0].focus();
        }.bind(this));
    });
  };
...
```

Сохраните файл `formhandler.js`.

Аналогично удалять пункт перечня необходимо только в случае успеха обращения к методу `Truck.prototype.deliverOrder`.

Привяжите цепочкой вызов `.then` к функции, передаваемой методу `addClickHandler` в файле `checklist.js`. Не забудьте выполнить `.bind` переменной `this` для этой анонимной функции.

```
...
  CheckList.prototype.addClickHandler = function (fn) {
    this.$element.on('click', 'input', function (event) {
      var email = event.target.value;
      this.removeRow(email);
      fn(email);
        .then(function () {
          this.removeRow(email);
        }.bind(this));
    }.bind(this));
  };
...
```

Сохраните файл `checklist.js`. Лишний раз напомним, что мы вызываем метод `addClickHandler` в файле `main.js`:

```
checkList.addClickHandler(myTruck.deliverOrder.bind(myTruck);
```

Менять что-либо в этом вызове метода необходимости нет. Метод `addClickHandler` будет работать как следует, поскольку метод `Truck.prototype.deliverOrder` возвращает объект `Deferred`.

Все наши данные расположены удаленно, а значит, нам необходимо загрузить их и отрисовать пункты перечня для всех заказов кофе. Для этого можно воспользоваться методами `Truck.prototype.printOrders` и `CheckList.prototype.addRow`.

Нам нужно внести два изменения в метод `printOrders`. Во-первых, изменить `printOrders` для работы с объектами `Deferred`. Далее добавить в `printOrders` функциональный аргумент, который он будет вызывать при проходе в цикле по выводимым данным.

В настоящий момент наш код для метода `Truck.prototype.printOrders` в файле `truck.js` выглядит следующим образом:

```
...
  Truck.prototype.printOrders = function () {
    var customerIdArray = Object.keys(this.db.getAll());

    console.log('Truck #' + this.truckId + ' has pending orders:');
    customerIdArray.forEach(function (id) {
      console.log(this.db.get(id));
    }.bind(this));
  };
...
```

Исправьте эту реализацию для вызова и возврата метода `this.db.getAll`, привязав к нему цепочкой `.then`. Передайте в `.then` анонимную функцию и задайте значение его ключевого слова `this` с помощью `.bind`:

```
...
  Truck.prototype.printOrders = function () {
    return this.db.getAll()
      .then(function (orders) {
        var customerIdArray = Object.keys(this.db.getAll());

        console.log('Truck #' + this.truckId + ' has pending orders:');
        customerIdArray.forEach(function (id) {
          console.log(this.db.get(id));
        }.bind(this));
      }.bind(this));
  };
...
```

Наша анонимная функция ожидает на входе объект, содержащий все полученные с сервера данные о заказах кофе. Извлеките из этого объекта `keys`[1] и присвойте их переменной `customerIdArray`:

```
...
  Truck.prototype.printOrders = function () {
    return this.db.getAll()
      .then(function (orders) {
        var customerIdArray = Object.keys(~~this.db.getAll());~~ orders);

        console.log('Truck #' + this.truckId + ' has pending orders:');
        customerIdArray.forEach(function (id) {
          console.log(this.db.get(id));
        }.bind(this));
      }.bind(this));
  };
...
```

Затем аналогично поменяйте оператор `console.log`, чтобы он не вызывал метод `this.db.get(id)`. Он должен использовать объект `allData`, в котором уже содержатся все заказы кофе. Не следует выполнять лишний вызов Ajax для каждого элемента, который нужно вывести.

```
...
  Truck.prototype.printOrders = function () {
    return this.db.getAll()
      .then(function (orders) {
        var customerIdArray = Object.keys(orders);

        console.log('Truck #' + this.truckId + ' has pending orders:');
        customerIdArray.forEach(function (id) {
          console.log(~~this.db.get(id));~~ orders[id]);
        }.bind(this));
      }.bind(this));
  };
...
```

[1] Адреса электронной почты. — *Примеч. пер.*

Метод `printOrders` должен принимать на входе необязательный функциональный аргумент. Нам придется проверять, передан ли он, и если да — вызывать его. При вызове мы будем передавать ему текущий заказ кофе: `allData[id]`.

```
...
  Truck.prototype.printOrders = function (printFn) {
    return this.db.getAll()
      .then(function (orders) {
        var customerIdArray = Object.keys(orders);

        console.log('Truck #' + this.truckId + ' has pending orders:');
        customerIdArray.forEach(function (id) {
          console.log(orders[id]);
          if (printFn) {
            printFn(orders[id]);
          }
        }.bind(this));
      }.bind(this));
  };
...
```

Сохраните файл `truck.js`. Вызовите в файле `main.js` метод `printOrders` и передайте ему вызов метода `checkList.addRow`. Не забудьте привязать метод `addRow` к экземпляру `CheckList`.

```
...
  formHandler.addInputHandler(Validation.isCompanyEmail);

  myTruck.printOrders(checkList.addRow.bind(checkList));

})(window);
```

Сохраните изменения и вернитесь в браузер. Приложение CoffeeRun должно отображать в перечне имеющиеся заказы кофе. Вручную обновите страницу, чтобы убедиться, что перечень каждый раз заполняется заново. Загляните на панель сети и убедитесь, что запросы Ajax действительно выполняются (рис. 14.5).

Объекты Promise в DataStore

Благодаря возврату объектов `Deferred` из модуля `RemoteDataStore` мы получили возможность гибко использовать присланные с сервера данные.

Но вы могли заметить, что методы `RemoteDataStore` далеко ушли от методов `DataStore`. Если бы нам нужно было вернуть обратно обычный экземпляр `DataStore`, то мы бы обнаружили, что он больше не может работать с нашим приложением (рис. 14.6).

На рис. 14.6 вы можете видеть, что создание экземпляра `Truck` с помощью обычного `DataStore` приводит к генерации ошибки. Этот вариант не работает с UI как

надо. Чтобы функционировать без перебоев, приложению CoffeeRun требуется `DataStore`, основанный на объектах `Deferred`.

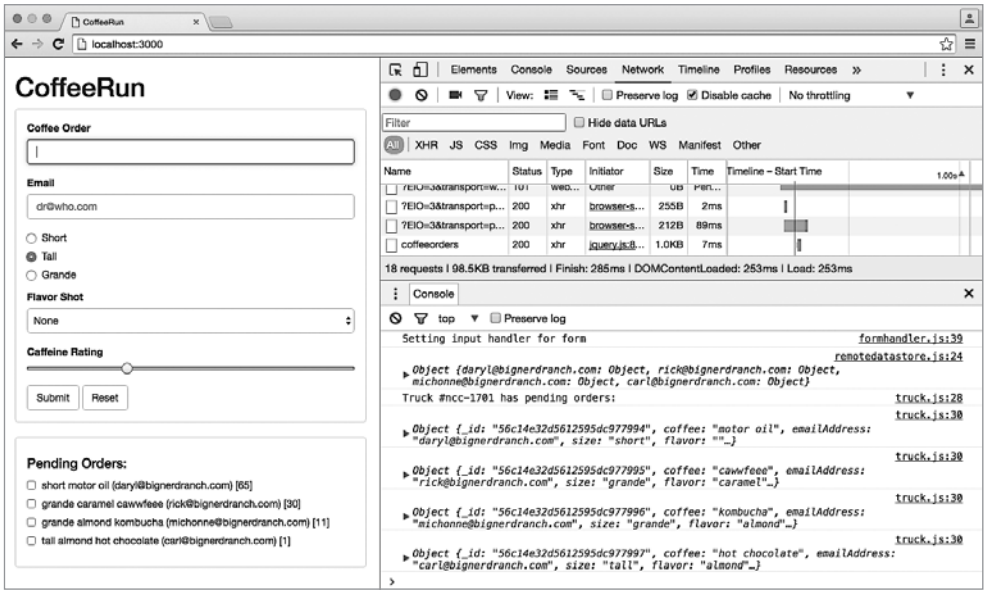


Рис. 14.5. Отрисовка заказов при загрузке страницы

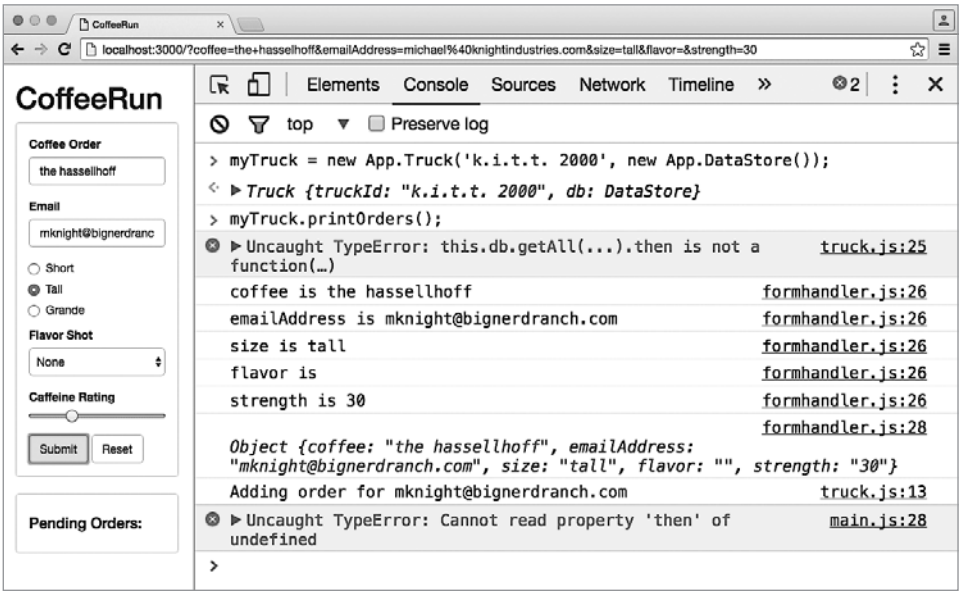


Рис. 14.6. Упс. DataStore теперь несовместим с приложением

Дабы исправить эту ситуацию, поменяем четыре метода модуля `DataStore`, чтобы они возвращали объекты `Promise`.

Объекты `Deferred` библиотеки jQuery неплохо послужили нам. Однако раз `DataStore` не использует методы `$.ajax` библиотеки jQuery для доступа к объектам `Deferred`, нам понадобится использовать нативный конструктор `Promise` для создания и возврата объектов `Promise`.

Создание и возврат объектов Promise

Нам понадобится изменить в файле `datastore.js` метод `add`. Но сначала создадим переменную типа `Promise` и присвоим ей значение `window.Promise`. Хотя это не обязательно, но будет неплохой идеей продолжить воплощать паттерн импортирования из глобальной области видимости всего, что нам понадобится в нашем модуле.

В методе `add` создайте новую переменную под названием `promise`. Присвойте ей новый экземпляр объекта `Promise`. Обязательно выполните возврат переменной `promise` в конце метода `add`:

```
(function (window) {
  'use strict';
  var App = window.App || {};
  var Promise = window.Promise;

  function DataStore() {
    this.data = {};
  }

  DataStore.prototype.add = function (key, val) {
    this.data[key] = val;
    var promise = new Promise();

    return promise;
  };
...
```

Конструктору `Promise` необходим функциональный аргумент. Передайте ему анонимную функцию, принимающую на входе два функциональных аргумента — `resolve` и `reject`:

```
...
  DataStore.prototype.add = function (key, val) {
    this.data[key] = val;
    var promise = new Promise(function (resolve, reject) {
    });

    return promise;
  };
...
```

В ходе работы объект `Promise` вызовет анонимную функцию и передаст ей два значения: `resolve` и `reject`. Функция `resolve` вызывается для изменения состояния объекта `Promise` на «выполнен успешно». Функция `reject` вызывается для изменения состояния объекта `Promise` на «отказано в выполнении».

Далее переместите строку сохранения данных (`this.data[key] = val;`) вниз, в тело анонимной функции. Чтобы свойство `this.data` гарантированно указывало на переменную экземпляра `data` объекта `DataStore`, выполните `bind` этой анонимной функции к переменной `this`.

```
...
  DataStore.prototype.add = function (key, val) {
    this.data[key] = val;
    var promise = new Promise(function (resolve, reject) {
      this.data[key] = val;
    }.bind(this));

    return promise;
  };
...
```

Выполнение resolve объекта Promise

Вызовите в самом конце анонимной функции `resolve` без аргументов:

```
...
  DataStore.prototype.add = function (key, val) {
    var promise = new Promise(function (resolve, reject) {
      this.data[key] = val;
      resolve(null);
    }.bind(this));

    return promise;
  };
...
```

Почему необходимо использовать `null` в качестве аргумента? Вызов метода `add` для добавления значения в объект `DataStore` не порождает значения, на которое мы могли бы сослаться для выполнения метода `resolve`. Если необходимо явным образом вернуть указание на отсутствие значения, следует предпочесть `null`. (Можно было также использовать `resolve(val)`, чтобы предоставить следующей функции в цепочке доступ к только что сохраненному значению. Для приложения CoffeeRun этого не требуется, потому и не сделано в данном примере.)

Обеспечиваем возврат объектов Promise остальными методами DataStore

Вы можете вручную изменить остальные три метода, используя тот же паттерн. Но вместо повторного ввода всего этого кода напишите вспомогательную функцию

с именем promiseResolvedWith для создания объекта Promise, разрешите ее и верните. Поменяйте код метода DataStore.prototype.add, чтобы использовать эту вспомогательную функцию.

```
...
  function DataStore() {
    this.data = {};
  }

  function promiseResolvedWith(value) {
    var promise = new Promise(function (resolve, reject) {
      resolve(value);
    });
    return promise;
  }

  DataStore.prototype.add = function (key, val) {
    var promise = new Promise(function (resolve, reject) {
      this.data[key] = val;
      resolve(null);
    }.bind(this));

    return promise;
    return promiseResolvedWith(null);
  };
...
```

Функция promiseResolvedWith представляет собой допускающую повторное использование версию кода Promise, написанного нами в методе add. Она принимает на входе параметр value, создает новую переменную promise и присваивает ей новый экземпляр Promise. Она также передает конструктору Promise анонимную функцию, принимающую на входе два аргумента: resolve и reject. Внутри этой анонимной функции вызывается resolve, которому передается аргумент value.

Необходимости связывать (bind) функциональный аргумент и переменную this в функции promiseResolvedWith нет, поскольку нет ссылок на this.

Измените остальные методы, чтобы они применяли функцию promiseResolvedWith. Передайте методам get и getAll значение, которое мы возвращали в версии, не использующей Promise. Передайте в метод remove null.

```
 ...
  DataStore.prototype.get = function (key) {
    return this.data[key];
    return promiseResolvedWith(this.data[key]);
  };

  DataStore.prototype.getAll = function () {
    return this.data;
    return promiseResolvedWith(this.data);
  };
```

```
  DataStore.prototype.remove = function (key) {
    delete this.data[key];
    return promiseResolvedWith(null);
  };
...
```

Наконец, измените файл `main.js`, чтобы использовать экземпляр `DataStore` вместо `RemoteDataStore`:

```
...
  var remoteDS = new RemoteDataStore(SERVER_URL);
  var myTruck = new Truck('ncc-1701', remoteDS); new DataStore());
  window.myTruck = myTruck;
...
```

После выполнения всех этих изменений сохраните код и еще раз проверьте работу приложения CoffeeRun. Вы должны увидеть, что при использовании экземпляра `DataStore` оно работает правильно, но не выполняет никаких запросов Ajax (рис. 14.7).

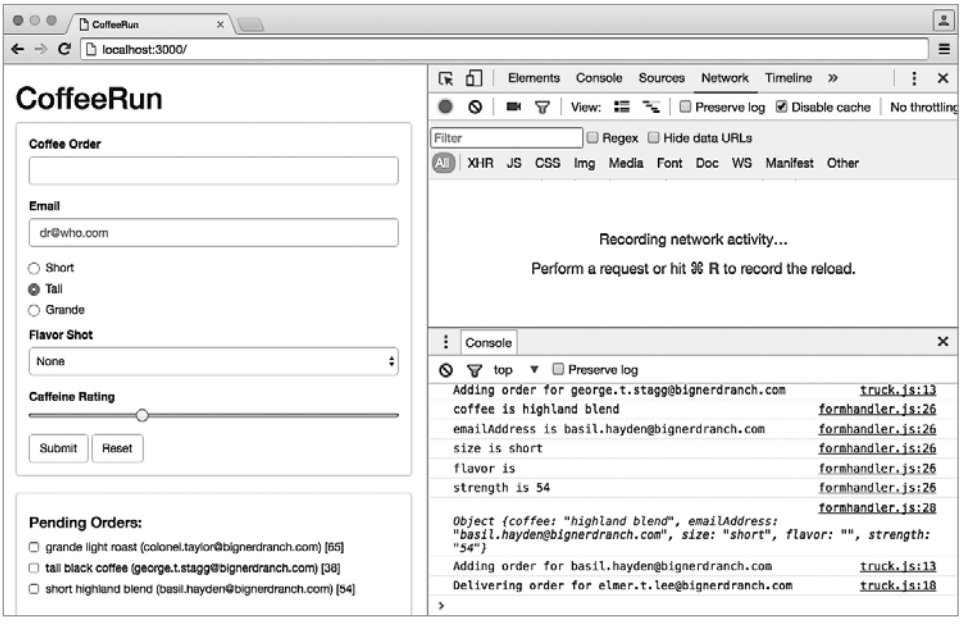


Рис. 14.7. CoffeeRun готов!

Мы прошли с CoffeeRun долгий путь! Мы написали достаточно серьезный JavaScript-код с использованием множества IIFE, обратных вызовов и объектов `Promise`. Мы также познакомились с библиотекой jQuery, которую использовали для работы с элементами DOM и взаимодействия с воплощающим REST веб-сервисом.

Пришло время попрощаться с CoffeeRun и двигаться дальше. Следующее приложение Chattrbox представляет собой *fullstack*-приложение для чата. Мы не только создадим код клиентской части, но и напишем сервер. Не волнуйтесь, даже если это ваше первое серверное приложение. Мы по-прежнему будем использовать JavaScript, просто не для браузера. Приготовьтесь поработать с платформой Node.js!

Серебряное упражнение: автоматическое переключение на DataStore

Если вы везунчик, то ваше сетевое соединение всегда будет надежным и устойчивым. Но лучше заранее быть готовым к разрывам соединения.

Исправьте приложение CoffeeRun, чтобы использовать `DataStore` в случае, когда Ajax-запросы не могут добраться до сервера.

Чтобы проверить работоспособность этого решения, отключите сетевое соединение компьютера при загрузке и сохранении заказов кофе.

Часть III. Данные, поступающие в режиме реального времени

15 Введение в Node.js

Платформа Node.js — проект с открытым исходным кодом, предоставляющий возможность написания кода на языке JavaScript, выполняемого вне браузера.

При написании JavaScript для браузера код получает доступ к глобальным объектам, таким как `document` и `window`, наряду с другими API и библиотеками. С помощью Node код может обращаться к жесткому диску, базам данных и сети (рис. 15.1).

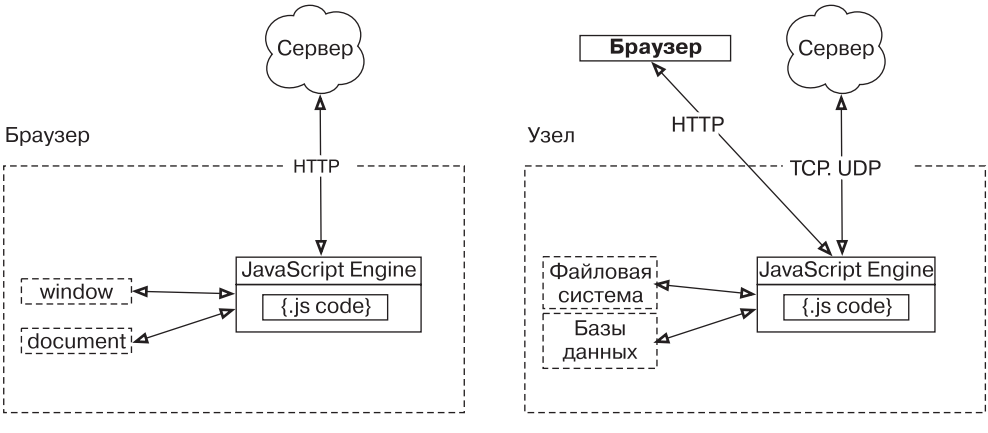


Рис. 15.1. Выполнение кода JavaScript в браузере по сравнению с выполнением на платформе Node

При использовании платформы Node можно создать все что угодно, начиная от утилит командной строки и заканчивая веб-серверами. На протяжении следующих четырех глав мы будем использовать Node для написания работающего в режиме реального времени приложения для чата под названием Chattrbox (рис. 15.2).

Рис. 15.2. Приложение Chattrbox: только для важных разговоров

Приложение Chattrbox будет состоять из двух частей: Node.js-сервера и JavaScript-приложения, работающего в браузере. Браузер будет соединяться с Node-сервером и получать HTML-, CSS- и JavaScript-файлы. Пока что приложение JavaScript в браузере начнет с обработки взаимодействия в режиме реального времени посредством протокола WebSockets. Этот процесс схематично показан на рис. 15.3.

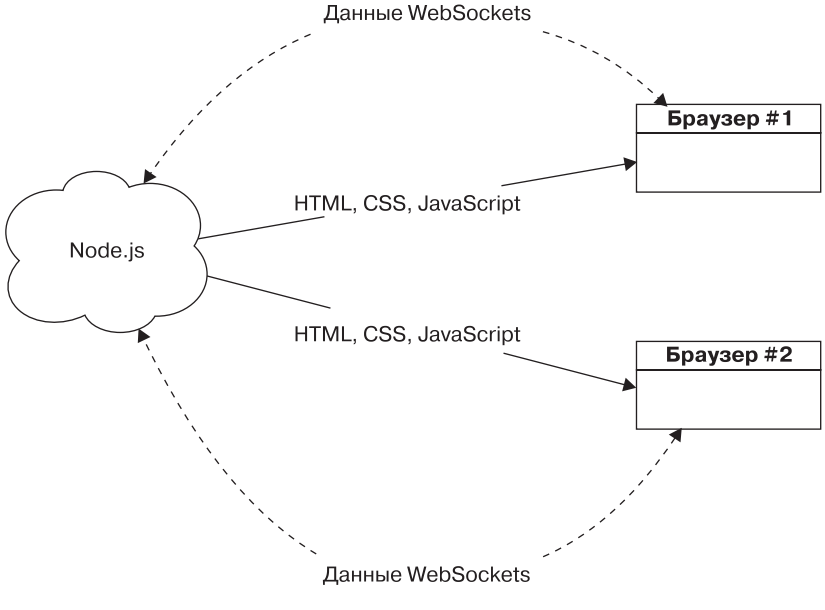


Рис. 15.3. Сетевой график приложения Chattrbox

Мы познакомимся с протоколом WebSockets в следующей главе. А данная глава посвящена платформе Node.

Утилиты node и npm

Когда мы установили Node.js в главе 1, мы получили доступ к двум программам командной строки: node и системе управления пакетами Node — npm. Как вы помните, npm предоставляет возможность установки свободно распространяемых инструментов разработки, таких как утилита browser-sync. Программа node выполняет запуск написанных на JavaScript программ.

По большей части то, чем мы будем заниматься в данной главе, будет связано с npm. Утилита командной строки npm может решать множество задач, например установить сторонний код, включаемый в наш проект, управлять потоком выполнения проекта и внешними зависимостями. В этой главе мы будем использовать npm:

- для создания файла `package.json` с помощью команды `npm init`;
- для добавления сторонних модулей с помощью команды `npm install --save`;
- для запуска часто используемых инструментов, занесенных в раздел `scripts` файла `package.json`.

Платформа Node — это намного больше, чем просто команды node и npm. Она содержит множество полезных модулей, предоставляющих конструкторы, помогающие при работе с файлами и каталогами, взаимодействии по сети и обработке событий. При написании кода JavaScript для платформы Node мы получаем доступ к вспомогательным функциям, облегчающим взаимодействие JavaScript с экосистемой модулей Node. Например, Node предоставляет гораздо более простой паттерн модуля, чем IIFE, которые мы использовали для приложения CoffeeRun.

Упомянутый выше файл `package.json` — файл-описание проекта Node. Он содержит название проекта, номер версии, описание и другую информацию. Что важнее, именно в нем можно хранить конфигурационные настройки и команды, используемые npm при тестировании и компоновке приложения.

Создать этот файл можно вручную, но гораздо удобнее делегировать выполнение этой задачи npm.

Команда npm init

Создайте в каталоге проектов подкаталог `chattrbox`. Перейдите в него в терминале и выполните команду `npm init` для создания файла `package.json`.

Утилита npm запросит информацию о проекте, а также предложит ответы по умолчанию, которые пока нас вполне устроят. Нажмите клавишу Enter, чтобы согласиться со значениями по умолчанию (рис. 15.4).

Рис. 15.4. Выполнение команды npm init

Откройте каталог `chattrbox` в Atom. Вы увидите, что файл `package.json` действительно был создан (рис. 15.5).

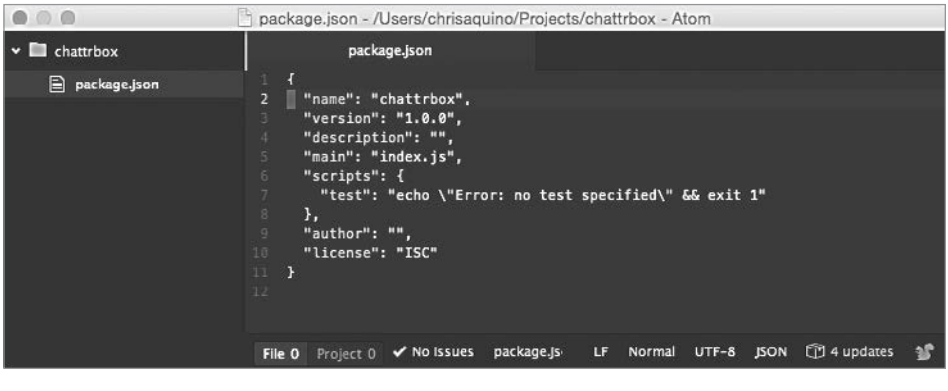


Рис. 15.5. Содержимое файла package.json после выполнения npm init

Сценарии npm

Обратите внимание на раздел `"scripts"` в файле `package.json`. Он предназначен для команд, которые при работе над проектом иногда понадобится запускать многократно.

Мы будем добавлять новые сценарии в раздел `"scripts"` файла `package.json`, чтобы сделать процесс разработки более эффективным. Создайте свой первый сценарий npm, добавив туда `"start"` (обратите внимание на необходимость поставить запятую в конце строки `"test"`):

```
...
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1",
    "start": "node index.js"
  },
...
```

Благодаря этому сценарию у нас появится возможность запускать сервер Node путем выполнения `npm start` из командной строки.

Hello, World

Чтобы познакомиться с написанием JavaScript вне браузера, начнем с классической программы `Hello, World`. Создайте в каталоге `chattrbox` новый файл `index.js` и введите следующий код, который мы поясним после того, как вы его введете:

```
var http = require('http');

var server = http.createServer(function (req, res) {
  console.log('Responding to a request.');
  res.end('<h1>Hello, World</h1>');
});
server.listen(3000);
```

В первой строке мы воспользовались встроенной функцией `require` платформы Node для обращения к прилагаемому к Node модулю `http`. Этот модуль предоставляет множество утилит для работы с HTTP-запросами и ответами, такими как функция `http.createServer`.

`http.createServer` принимает на входе один аргумент — функцию, вызываемую для каждого HTTP-запроса. Это может напомнить вам паттерн обратных вызовов, который мы использовали для событий браузера, за исключением того, что в данном случае обратный вызов срабатывает при событии на стороне сервера (получении HTTP-запроса).

В нашем обратном вызове мы выводим сообщение в консоль и пишем какой-либо HTML-текст в качестве ответа. В платформе Node принято использовать `req` и `res` в качестве имен переменных для объектов HTTP-запроса и ответа.

Наконец, мы говорим браузеру прослушивать на порте 3000 с помощью вызова функции `server.listen`. Это обычно называют привязкой к порту (binding to a port).

Сохраните файлы. Чтобы посмотреть на сервер Node в действии, выполните команду `npm start`. Выведенные в терминал результаты показаны на рис. 15.6.

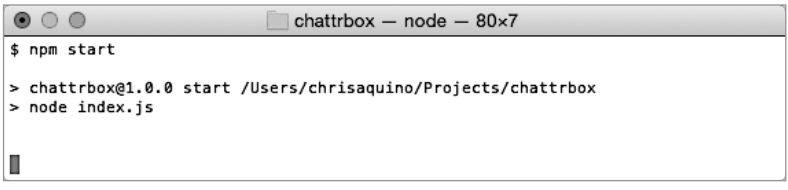


Рис. 15.6. Запуск index.js с помощью npm start

Далее откройте в браузере адрес http://localhost:3000. Результат будет выглядеть так, как показано на рис. 15.7 (обратите внимание, что в некоторых других браузерах (не в Chrome) HTML может отображаться в виде простого текста. Эти браузеры ожидают встретить в ответе или doctype, или дополнительный элемент метаданных, указывающий, что данный ответ необходимо интерпретировать как HTML. Вам будет предложено решить эту проблему в одном из упражнений в конце данной главы).

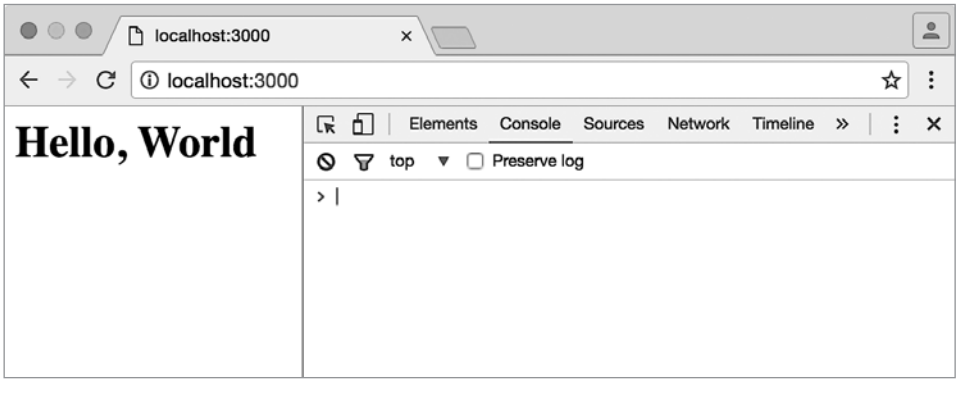


Рис. 15.7. Обращение к серверу Node через браузер

В отличие от запуска приложений Ottergram и CoffeeRun в браузере не выполняется никакого JavaScript. К тому времени как вы увидите страницу, код JavaScript уже выполнил свою работу на сервере.

Вернитесь в терминал. Вы должны увидеть, что при получении запроса оператор `console.log` вывел туда выражение `Responding to a request` (рис. 15.8).

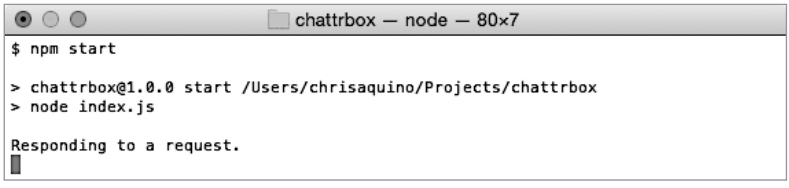


Рис. 15.8. console.log при поступлении запроса

Добавление сценария npm

Помимо предоставления возможности написания программ JavaScript для командной строки платформа Node обеспечивает способ организации процесса разработки этих программ. Это великолепная возможность, которой не помешает воспользоваться. Чтобы посмотреть, как она работает, мы добавим в наш проект толику автоматизации.

Возьмем запуск сервера, например. Каждый раз, когда нам хочется попробовать какой-нибудь новый код, приходится повторять несколько шагов:

- внести в редакторе изменения в код;
- переключиться на терминал;
- нажать Control+C для остановки программы;
- выполнить команду `npm start`, чтобы запустить программу снова.

Можно написать программу для автоматизации перезапуска сервиса. Вам повезло, поскольку кто-то уже написал ее для нас в модуле nodemon. Если заранее внедрить nodemon в наш технологический процесс, процесс написания программы станет гораздо приятнее.

В терминале остановите программу и выполните следующую команду для установки модуля nodemon:

```
npm install --save-dev nodemon
```

Вы увидите приведенные ниже строки, в которых утилита npm предупреждает вас о незаполненных полях в файле `package.json`. Не волнуйтесь — просто отметьте для себя, что npm педантична в деталях.

```
npm WARN chattrbox@1.0.0 No description
npm WARN chattrbox@1.0.0 No repository field.
```

Обратите внимание на параметр `--save-dev`, использованный нами в команде `npm install`. Он указывает npm на необходимость помочь нам в хранении списка всех сторонних модулей, от которых зависит приложение. Этот список хранится в файле `package.json`. При необходимости все зависимости из этого списка можно установить с помощью команды `npm install` (без аргументов).

Это значит, что при распространении своего кода не обязательно включать еще и все сторонние модули.

Если вы заглянете в файл `package.json`, то увидите, что npm создала для нас новый раздел `"devDependencies"`, содержащий запись для утилиты nodemon.

```
...
  "author": "",
  "license": "ISC",
  "devDependencies": {
    "nodemon": "^1.9.1"
  }
```

Теперь измените файл `package.json`, добавив в него еще один элемент в раздел `"scripts"`:

```
...
"scripts": {
  "test": "echo \"Error: no test specified\" && exit 1",
  "start": "node index.js",
  "dev": "nodemon index.js"
},
...
```

Перезапустите в терминале программу node с помощью нашего нового сценария, воспользовавшись командой `npm run dev`. Обратите внимание, что это не просто команда `npm dev`. Отличие от `npm start` состоит в том, что npm предполагает, будто некоторые команды (например, `start`) заведомо существуют. Что же касается пользовательских сценариев npm, необходимо указать, что мы хотим их выполнить.

Вы увидите, что теперь утилита nodemon взяла на себя управление нашей программой node (рис. 15.9).

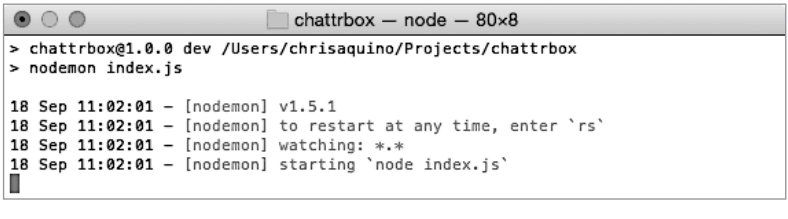


Рис. 15.9. Запуск с помощью npm run dev

Поменяйте в файле `index.js` `"Hello, World"` на `"Hello, World!!"` и сохраните изменения. Утилита nodemon обнаружит это и автоматически перезапустит программу node (рис. 15.10).

При работе с node и npm в следующих главах мы периодически будем включать новые модули, чтобы справиться с какими-либо трудностями.

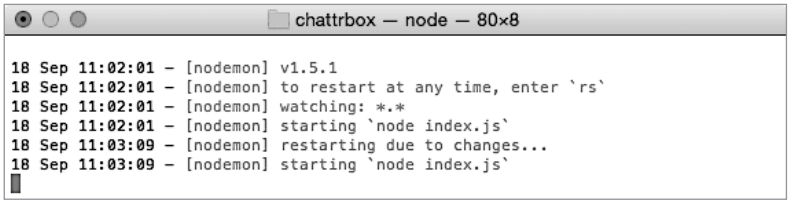


Рис. 15.10. Утилита nodemon перезапускает программу при изменении кода

Выдача контента из файлов

Уметь писать и запускать серверный код JavaScript нужно каждому программисту. Большинству серверов понадобится также выдавать и обрабатывать находящийся в файлах контент. Наша следующая задача — научить наш сервер читать файлы из подкаталогов и отправлять их в ответе браузеру. Это аналогично тому, что делал для нас browser-sync в предыдущих главах.

Создайте новый каталог `app` в папке проекта `chattrbox`. Создайте в `app` файл `index.html` со следующим текстом:

```
Hello, File!
```

Нам не понадобится в этом файле никакой HTML-код — в нем достаточно контента, который можно было бы прочитать. Каталог проекта должен выглядеть так, как показано на рис. 15.11.

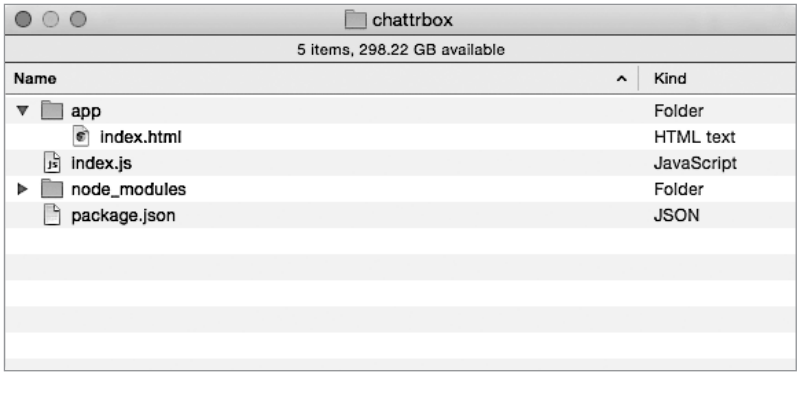


Рис. 15.11. Макет проекта Chattrbox

Читаем файл с помощью модуля fs

В файле `index.js` импортируйте файл `fs` системного модуля платформы Node.js и вызовите его метод `readFile`.

```
var http = require('http');
var fs = require('fs');

var server = http.createServer(function (req, res) {
  console.log('Responding to a request.');
  res.end('<h1>Hello, World</h1>');
  fs.readFile('app/index.html', function (err, data) {
    res.end(data);
  });
});
server.listen(3000);
```

Метод `readFile` принимает на входе имя файла и обратный вызов. Внутри обратного вызова мы вместо текста HTML отправляем с помощью функции `res.end` содержимое файла.

Обратите внимание, что наш обратный вызов принимает на входе и аргумент `err`, помимо прочитанных из файла данных. Таковы соглашения по разработке программ платформы Node.js, которые мы обсудим далее в этой главе.

Утилита nodemon должна была перезапустить нашу программу, так что можете сразу перейти в браузер и перезагрузить страницу. В браузере вы должны увидеть в точности то, что было написано в файле `index.html`:

```
Hello, File!
```

Для начала неплохо, но нашему приложению для чата придется делать гораздо больше простой выдачи отдельного HTML-файла. Этот HTML-файл может запрашивать другие файлы CSS или JavaScript. Для выполнения этих запросов программа node должна понимать, какой файл был запрошен и где его искать. Этим мы и займемся.

Работаем с URL запроса

Во-первых, нам понадобится получить путь URL из объекта запроса. Если путь представляет собой всего лишь `'/'`, лучше вернуть файл `index.html`. Это распространенное соглашение, принятое на заре Интернета.

В противном случае желательно попытаться вернуть тот файл, который требуется объекту запроса.

Измените в файле `index.js` обратный вызов, чтобы проверить, какой файл запрашивает браузер:

```
var http = require('http');
var fs = require('fs');

var server = http.createServer(function (req, res) {
  console.log('Responding to a request.');
```

```
  var url = req.url;

  var fileName = 'index.html';
  if (url.length > 1) {
    fileName = url.substring(1);
  }
  console.log(fileName);
  fs.readFile('app/index.html', function (err, data) {
    res.end(data);
  });
});
server.listen(3000);
```

Воспользовавшись свойством `url` объекта запроса, можно определить, нужен ли браузеру файл по умолчанию (`index.html`) или другой файл. Если требуется другой файл, вызовите `url.substring(1)`, чтобы удалить первый символ, который будет равен `'/'`.

Пока что мы только выводим имя файла в консоль.

После того как nodemon перезапустит нашу программу, попробуйте перейти по адресу `http://localhost:3000/woohoo` (или какому-нибудь другому пути, включая путь по умолчанию `'/'`). Результаты, которые появятся в терминале, будут соответствовать рис. 15.12.

Рис. 15.12. Выводим в консоль пути запрошенных файлов

(Как вы помните из главы 2, браузеры автоматически запрашивают файл `favicon.ico`, так что запрос для него тоже выведен в консоль.)

Пришло время воспользоваться этой информацией о пути.

Использование модуля path

Вы можете просто передать переменную `fileName` в метод `fs.readFile`, но лучше воспользоваться модулем `path`, в котором есть инструменты для обработки и преобразования путей файлов. Одна небольшая, но важная причина выбора этого модуля: некоторые файловые системы используют прямую косую черту, а некоторые — обратную. Модуль `path` с легкостью управляется с этими различиями.

Измените файл `index.js`, импортировав модуль `path` и воспользовавшись им для поиска запрошенного файла:

```
var http = require('http');
var fs = require('fs');
var path = require('path');

var server = http.createServer(function (req, res) {
  console.log('Responding to a request.');
  var url = req.url;

  var fileName = 'index.html';
  if (url.length > 1) {
    fileName = url.substring(1);
  }
  console.log(fileName);
  var filePath = path.resolve(__dirname, 'app', fileName);
  fs.readFile('app/index.html' filePath, function (err, data) {
    res.end(data);
  });
});

server.listen(3000);
```

Попробуйте ввести разные пути файлов в браузере, чтобы убедиться, что приложение работает так же, как и раньше.

Для пути по умолчанию должен возвращаться файл `index.html`, а для несуществующих путей (таких как `'/woohoo/'`) не должно отображаться ничего (только выводятся в консоль их имена файлов).

Далее создайте в каталоге `app` файл `test.html` и напишите в нем:

```
Hola, Node!
```

Попробуйте получить к нему доступ в браузере. Программа node должна вернуть его без всяких проблем (рис. 15.13).

Мы добавили в приложение код, успешно выдающий конкретный файл по его пути URL. Следующее, что нужно сделать, — выделить эту функциональность в отдельный модуль.

Рис. 15.13. Извлечение файла test.html

Создание пользовательского модуля

У нашего обратного вызова есть как минимум две задачи. Он выясняет, какие файлы были запрошены, и выполняет чтение этих файлов для отправки в ответе. Чтобы сделать код модульным и более удобным для сопровождения, одну из этих обязанностей можно переместить в отдельный модуль.

В приложении CoffeeRun мы объявляли модули в IIFE, присваивавшем значение свойству глобального пространства имен. Модули в программах node работают иначе. Код модулей по-прежнему пишется в отдельных файлах, но IIFE не требуется.

Создайте новый файл `extract.js` в том же каталоге, что и файл `index.js` (не в каталоге `app`). Добавьте функцию `extractFilePath`, которая будет искать соответствующий файл (этот код очень похож на уже написанный нами в `index.js`).

```
var path = require('path');

var extractFilePath = function (url) {
  var filePath;
  var fileName = 'index.html';

  if (url.length > 1) {
    fileName = url.substring(1);
  }
  console.log('The fileName is: ' + fileName);

  filePath = path.resolve(__dirname, 'app', fileName);
  return filePath;
};
```

Мы взяли большой фрагмент кода из файла `index.js` и вынесли в отдельную функцию с названием `extractFilePath`. Далее сделайте функцию `extractFilePath` доступной, чтобы другие модули могли импортировать ее с помощью вызова функции `require`. Для этого присвойте `extractFilePath` глобальной переменной `module.exports`. Это

особая переменная, предоставляемая платформой Node. Любые присваиваемые ей значения могут импортироваться другими модулями. Все остальные переменные и функции не будут видимы другим модулям.

```
...
  filePath = path.resolve(__dirname, 'app', fileName);
  return filePath;
};

module.exports = extractFilePath;
```

Эта новая строка сообщает Node, что при импорте модуля `extract` путем вызова `require('./extract')` возвращаемое значение представляет собой функцию `extractFilePath`. Внесите сейчас эту правку в файл `index.js`.

Используем наш пользовательский модуль

Измените файл `index.js`, чтобы он использовал наш новый модуль `extract`:

```
var http = require('http');
var fs = require('fs');
~~var path = require('path');~~
var extract = require('./extract');

var server = http.createServer(function (req, res) {
  console.log('Responding to a request.');
  ~~var url = req.url;~~

  ~~var fileName = 'index.html';~~
  ~~if (url.length > 1) {~~
    ~~fileName = url.substring(1);~~
  ~~}~~
  ~~console.log(fileName);~~
  ~~var filePath = path.resolve(__dirname, 'app', fileName);~~
  var filePath = extract(req.url);
  fs.readFile(filePath, function (err, data) {
    res.end(data);
  });
});
server.listen(3000);
```

Вы импортировали наш пользовательский модуль с помощью функции `require` и присвоили значение модуля новой переменной `extract`. Теперь можно использовать функцию `extract` точно так же, как мы использовали бы функцию `extractFilePath`.

После того как утилита nodemon перезагрузит код, протестируйте несколько путей URL и убедитесь, что файлы `index.html` (по умолчанию) и `test.html` по-прежнему загружаются. Удостоверьтесь также, что несуществующие пути отображаются в виде пустой страницы и без вывода ошибки.

Обработка ошибок

И последнее. Если найти файл не получается, лучше вернуть код ошибки, чем притворяться, что все нормально. Для этого нужно уметь обнаруживать, когда функция `fs.readFile` возвращает ошибку вместо файла.

В языке программирования JavaScript передача обратных вызовов методам API — распространенная практика. То же самое справедливо для платформы Node.js, и обычно первым аргументом обратных вызовов является ошибка. Поскольку ошибка стоит до результата, вам придется по крайней мере увидеть ее (независимо от того, будете ли вы ее обрабатывать).

Добавьте в файл `index.html` проверку на ошибку при работе с файлом и отправьте код ошибки `404`, если она будет обнаружена:

```
var http = require('http');
var fs = require('fs');
var extract = require('./extract');

var handleError = function (err, res) {
  res.writeHead(404);
  res.end();
};

var server = http.createServer(function (req, res) {
  console.log('Responding to a request.');
  var filePath = extract(req.url);
  fs.readFile(filePath, function (err, data) {
    if (err) {
      handleError(err, res);
      return;
    } else {
      res.end(data);
    }
  });
});
server.listen(3000);
```

Сохраните изменения. После перезапуска nodemon перейдите по какому-нибудь несуществующему пути, например http://localhost:3000/woohoo. Откройте панель сети в DevTools — и вы увидите код ошибки, как показано на рис. 15.14.

Самое первое, что мы делаем в нашем обратном вызове, — проверяем, отличается ли значение аргумента `err` от `null` или `undefined` и выполняем с ними какие-то действия. В данном примере мы передали информацию об ошибке функции `handleError`, после чего выполнили `return` для выхода из анонимного обратного вызова.

Не следует никогда молча игнорировать ошибки. Простого `404` пока вполне достаточно, что и делает функция `handleError`.

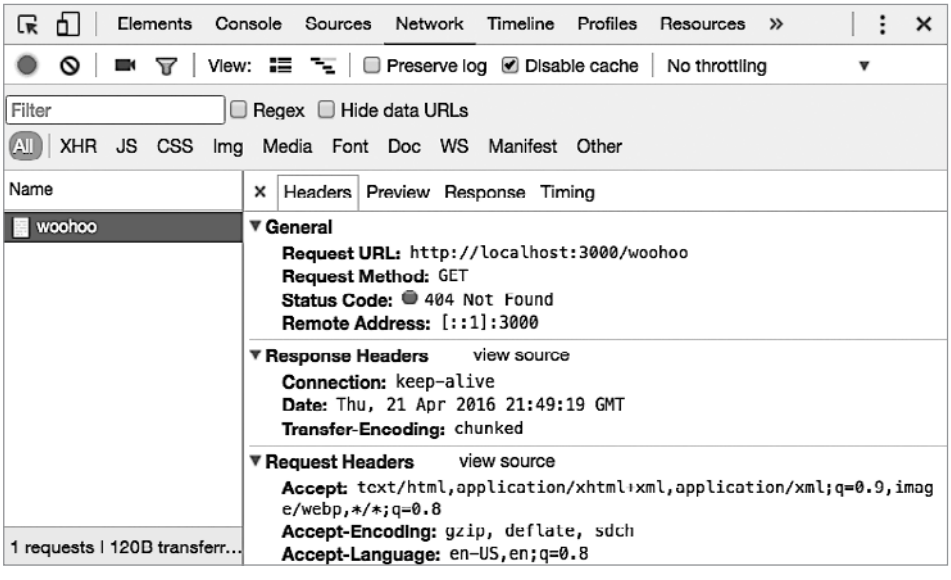


Рис. 15.14. Код состояния 404 на панели сети

Этот паттерн — «в первую очередь обработка ошибок с ранним возвратом» — одна из рекомендуемых в экосистеме Node практик. Все поставляемые вместе с Node модули следуют этому паттерну, как и вообще большинство свободно поставляемых модулей.

Мы создали работающий веб-сервер всего лишь с помощью нескольких десятков строк кода JavaScript, используя уже знакомые нам паттерны (такие как обратные вызовы).

Платформа Node предоставляет богатый набор модулей для работы с сетями и файлами, такие как используемые в нашем проекте модули `http` и `fs`. Благодаря таким ключевым словам Node, как `require` и `module.exports`, мы с легкостью смогли сделать наш код модульным.

В следующих трех главах мы продолжим создавать сервер Chattrbox и работать над его клиентской частью.

Для самых любознательных: реестр модулей npm

Существует множество модулей, доступных для установки с помощью системы управления пакетами npm. Найти нужный модуль или просмотреть все имеющиеся модули можно в реестре модулей на сайте www.npmjs.com.

Не забудьте заглянуть в документацию npm на сайте docs.npmjs.com. Вас может также заинтересовать вопрос создания собственных модулей, которыми могли бы пользоваться другие разработчики. В этом случае посмотрите страницу docs.npmjs.com/getting-started/creating-node-modules.

Бронзовое упражнение: создание пользовательской страницы ошибки

При переходе по пути несуществующего файла в настоящий момент выдается пустая страница браузера и код состояния 404.

В данном упражнении создайте специальную страницу ошибки для отображения ее пользователю вместо того, чтобы возвращать ошибку в виде кода состояния.

Для самых любознательных: типы MIME

Интересовало ли вас когда-нибудь, откуда компьютер знает, что нужно открывать файл фильма с помощью видеопроигрывателя, а файл PDF — с помощью программы просмотра документов? Дело в том, что компьютер поддерживает таблицу типов файлов и ассоциированных с ними программ. Вывод о типе файла он делает исходя из расширения имени этого файла (например, `.html` или `.pdf`).

Браузеру необходимы аналогичные ассоциации, чтобы знать, визуализировать ли ответ как HTML, использовать плагин для проигрывания музыки или скачивать файл на жесткий диск. Но HTTP-ответы не содержат расширения имен файлов. Вместо этого о том, какой тип информации содержится в ответе, браузеру должен сообщить сервер.

Это выполняется путем задания *типа MIME* или *типа файла мультимедиа* в заголовке `Content-Type` ответа. Например, на рис. 15.15 показано, что вы увидели бы на панели сети DevTools, если бы посмотрели ответ для сайта www.bignerdranch.com.

Значение заголовка `Content-Type` задано равным `text/html` — тип MIME для HTML. Вы можете задавать значение этого заголовка в своих проектах. Вот как это выглядело бы для приложения Chattrbox:

```
...
var server = http.createServer(function (req, res) {
  console.log('Responding to a request.');
  var filePath = extract(req.url);
  fs.readFile(filePath, function (err, data) {
    if (err) {
      handleError(err, res);
```

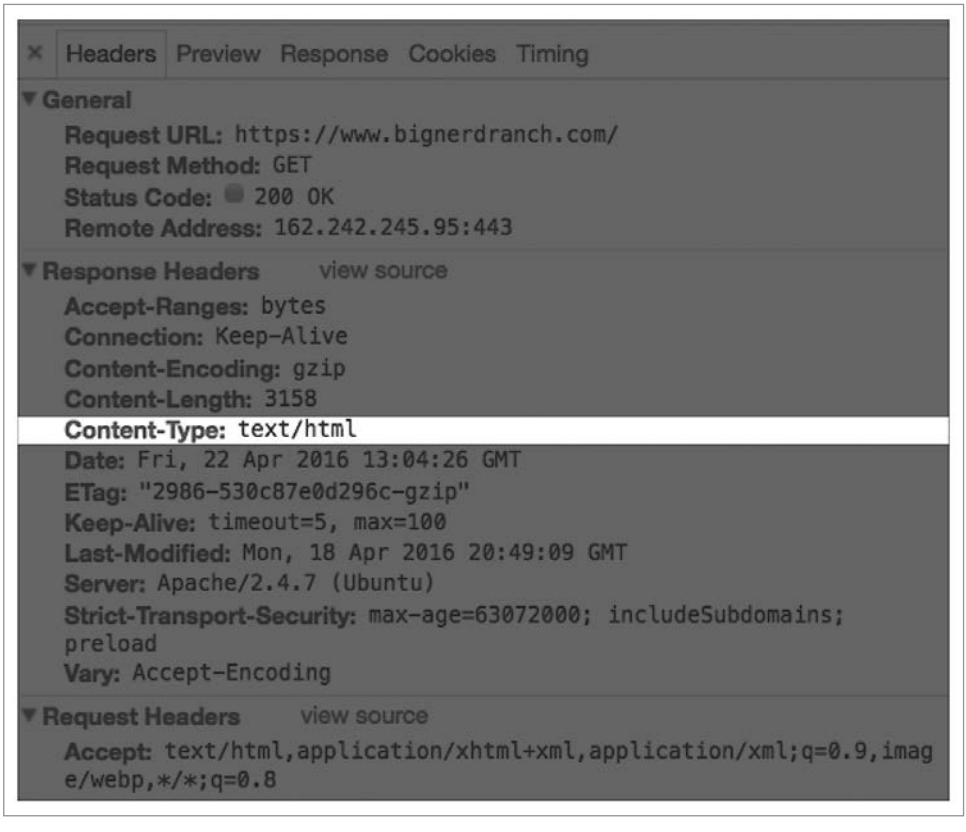


Рис. 15.15. Внимательно изучаем заголовок Content-Type на сайте www.bignerdranch.com

```
      return;
    } else {
      res.setHeader('Content-Type', 'text/html');
      res.end(data);
    }
  });
});
server.listen(3000);
```

Обратите внимание, что задать значение заголовка необходимо до того, как завершить ответ.

Чтобы узнать больше о типах MIME, загляните по адресу en.wikipedia.org/wiki/Media_type[1]. Для дополнительной информации о задании значений заголовков в программах Node зайдите на nodejs.org/api/http.html#http_response_setheader_name_value.

[1] Или ru.wikipedia.org/wiki/Список_MIME-типов. — *Примеч. пер.*

Серебряное упражнение: динамическое задание типа MIME

Обеспечьте динамическое задание типа MIME для ответов в зависимости от типа файла. Для этого полезно установить с помощью утилиты npm модуль `mime`. Информация и документация о модуле `mime` доступна по адресу github.com/broofa/node-mime.

Добавьте различные файлы в каталог приложения, включая текстовые файлы, PDF, аудиофайлы и фильмы. Убедитесь, что ваш браузер правильно отображает все типы.

Золотое упражнение: перенесите обработку ошибок в отдельный модуль

Перенесите в отдельный модуль код, отвечающий за чтение файлов и обработку ошибок.

Кроме того, сделайте этот модуль настраиваемым, чтобы при его импорте можно было указать, в каком каталоге находятся статические файлы HTML, CSS и JavaScript.

16 Обмен данными в режиме реального времени с помощью протокола WebSockets

В случае обычных запросов `GET` и `POST` браузеру приходится при каждом обмене данными с сервером выполнять новый запрос и ждать ответа. Как вы уже знаете из предыдущих глав, то же самое справедливо и для Ajax-запросов. Хотя Ajax-запрос не приводит к обновлению страницы, выполнение и обработка каждого запроса и ответа влекут за собой определенные (хотя и небольшие) расходы.

Технология WebSockets, напротив, предоставляет двусторонний протокол связи по HTTP. WebSockets создает отдельное соединение и поддерживает его в открытом состоянии для обмена данными в режиме реального времени (рис. 16.1).

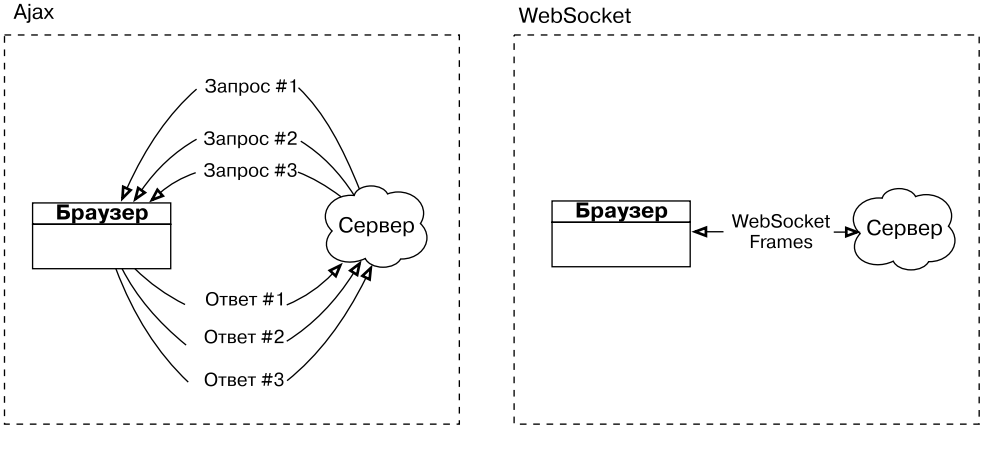


Рис. 16.1. Многочисленные Ajax-запросы по сравнению с отдельным соединением WebSockets

С помощью протокола WebSockets веб-приложения получают возможность не только загружать и сохранять удаленные данные. Push-уведомления, совместное редактирование документов, а также чат в режиме реального времени — только первый шаг. WebSockets делает возможным обработку серверами данных, поступающих по Интернету от различных объектов (например, умных светильников, умных замков, умных автомобилей). Традиционные технологии, такие как Ajax-опросы, оказываются неэффективными при работе с таким интенсивным трафиком.

В данной главе мы создадим клиента и сервер для чата. Если бы использовали для этого Ajax, нам пришлось бы поддерживать по крайней мере два соединения — одно для опроса на предмет новых сообщений, а второе — для отправки сообщений. Применяя протокол WebSockets, можно решить эту же задачу с помощью только одного соединения.

К концу данной главы приложение Chattrbox сможет одновременно работать с несколькими клиентами чата, отправляя новые сообщения каждому из них (рис. 16.2).

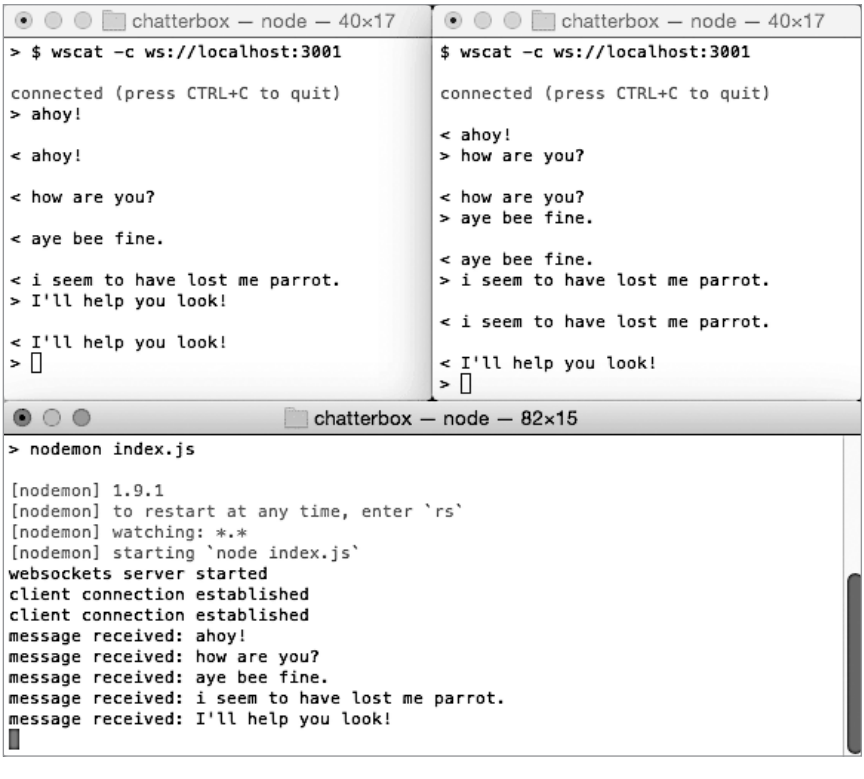


Рис. 16.2. Пиратский чат

Настройка WebSockets

Для настройки работы приложения Chattrbox с WebSockets необходимо сделать следующее.

1. Установить модуль ws.
2. Создать сервер WebSockets.
3. Добавить в сервер функциональность чата.
4. Предоставить новым пользователям историю сообщений путем отправки ее с сервера.

Начнем с самого начала.

Прилагаемый к платформе Node.js модуль `http` предоставляет нам удобный способ запустить HTTP-сервер, чтобы браузеры могли с ним взаимодействовать.

Аналогично модуль `ws` предоставляет программам Node.js удобный способ обмена данными через WebSockets. Существует несколько модулей, реализующих протокол WebSockets для платформы Node.js, но `ws` — стандартная реализация, работающая вполне удовлетворительно.

Начнем с установки модуля `ws` в каталог приложения Chattrbox (не беспокойтесь, если увидите предупреждения от системы управления пакетами npm об отсутствующем описании или поле репозитория для нашего проекта).

```
npm install --save ws
```

Создайте файл `websockets-server.js` в корневом каталоге приложения Chattrbox. Вставьте в него следующий код, чтобы импортировать модуль `ws` и начать прослушивание:

```
var WebSocket = require('ws');
var WebSocketServer = WebSocket.Server;
var port = 3001;
var ws = new WebSocketServer({
    port: port
});

console.log('websockets server started');
```

Мы импортировали модуль `ws` с помощью оператора `require`. Этот модуль содержит свойство `Server`, которое понадобится для создания работающего сервера WebSockets.

Именно это и делает код `ws = new WebSocketServer(/*...*/);`. Во время его выполнения сервер WebSockets создается и привязывается к заданному порту (в нашем случае 3001).

В отличие от созданного нами в файле `extract.js` модуля, нам не понадобится присваивание `module.exports`. Код из файла `websockets-server.js` будет выполняться при импорте. Он задаст все исходные значения и выполнит обработку всех связанных с WebSockets событий.

Теперь пора заняться соединениями. Создайте в файле `websockets-server.js` обратный вызов для всех событий `connection` для нашего сервера WebSockets:

```
var WebSocket = require('ws');
var WebSocketServer = WebSocket.Server;
var port = 3001;
var ws = new WebSocketServer({
    port: port
});

console.log('websockets server started');

ws.on('connection', function (socket) {
  console.log('client connection established');
});
```

Синтаксис обработки событий аналогичен таковому для библиотеки jQuery. Мы еще увидим, что многие библиотеки JavaScript (в Node и браузере) используют такой же стиль.

Наш обрабатывающий события обратный вызов принимает на входе один аргумент — объект `socket`. При соединении клиента с нашим сервером WebSockets мы можем получить доступ к этому соединению через объект `socket`.

Прежде чем написать код сервера чата, настроим сервер на повтор любых отправленных ему сообщений. Это называется *эхо-сервером*.

Добавьте эхо-функциональность в файл `websockets-server.js` с помощью регистрации обратного вызова для всех генерируемых клиентским соединением событий `message`:

```
...
console.log('websockets server started');
ws.on('connection', function (socket) {
  console.log('client connection established');

  socket.on('message', function (data) {
    console.log('message received: ' + data);
    socket.send(data);
  });
});
```

Мы зарегистрировали обработчик события непосредственно для объекта `socket`. Наш обратный вызов `message` передает всю отправляемую клиентом информацию. Пока что мы будем просто передавать ее обратно тому же соединению `socket`.

Через минуту мы увидим все это в действии.

Можно запустить наш сервер WebSockets отдельно с помощью команды `node websockets-server.js`, но ничуть не сложнее подключить его в файле `index.js`. Преимущество состоит в том, что программа nodemon автоматически перечитает ваш код при внесении изменений в файл `websockets-server.js` или `index.js`.

Добавьте оператор `require` вверху файла `index.js` для импорта модуля `websockets-server`:

```
var http = require('http');
var fs = require('fs');
var extract = require('./extract');
var wss = require('./websockets-server');
...
```

Сохраните файл — и nodemon перечитает код, предоставляя вам возможность опробовать его в деле.

Тестирование нашего сервера WebSockets

Один из способов протестировать наш сервер — воспользоваться модулем npm под названием `wscat` (это утилита для подключения к серверу WebSockets и обмена с ним данными). Модуль предоставляет программу командной строки, которую мы будем использовать в качестве клиента чата.

Откройте второе окно терминала и установите `wscat` глобально. Возможно, вам придется выполнить это с правами администратора (если вам требуется освежить эту тему в памяти, загляните в главу 1).

```
npm install -g wscat
```

Установив `wscat`, мы готовы подключиться к серверу WebSockets.

Выполните во втором окне терминала команду `wscat -c ws://localhost:3001`. Вы должны получить сообщение `connected (press CTRL+C to quit)` (подключено (нажмите Ctrl+C для отключения)) во втором окне терминала и сообщение `'client connection established'` (установлено соединение с клиентом) в первом.

Введите во втором окне терминала какой-нибудь текст в ответ на приглашение командной строки. Каждый раз, когда вы набираете какой-нибудь текст и нажимаете Enter, он дублируется сервером WebSockets, как показано на рис. 16.3.

Теперь, когда мы убедились, что можем связываться с сервером по протоколу WebSockets, пришло время добавить настоящую функциональность, которая обеспечит работу чат-системы приложения Chattrbox.

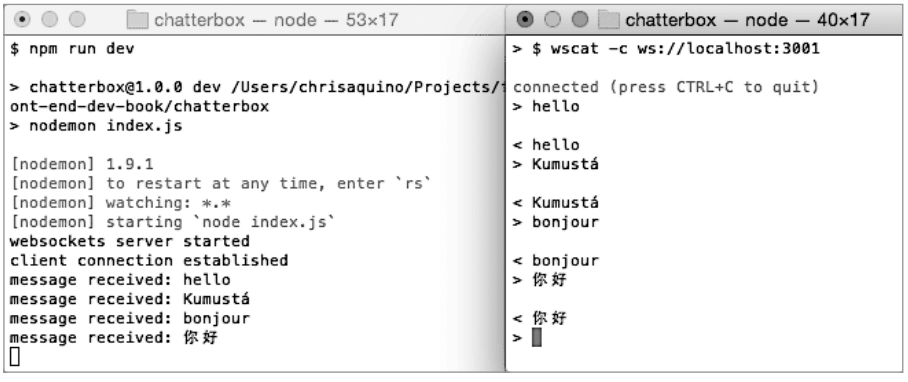


Рис. 16.3. Тестирование сервера с помощью команды wscat

Создаем функциональность сервера чата

Когда сервер WebSockets настроен и запущен, мы можем приступить к созданию сервера чата. Он должен выполнять несколько дел:

- хранить журнал отправленных на сервер сообщений;
- рассылать старые сообщения присоединившимся к чату новым пользователям;
- рассылать новые сообщения всем клиентам.

Сохранение журнала сообщений по мере отправки их пользователями необходимо для рассылки истории сообщений новым пользователям, так что мы займемся этим в первую очередь.

Создайте в файле `websockets-server.js` массив для хранения сообщений:

```
var WebSocket = require('ws');
var WebSocketServer = WebSocket.Server;
var port = 3001;
var ws = new WebSocketServer({
    port: port
});
var messages = [];

console.log('websockets server started');
...
```

Если бы нам нужна была более устойчивая к ошибкам чат-система, можно было бы хранить сообщения в базе данных. Но пока что простого массива вполне достаточно.

Далее вызовите метод `messages.push(data)` для добавления новых сообщений в массив по мере их поступления.

```
...
ws.on('connection', function (socket) {
  console.log('client connection established');
  socket.on('message', function (data) {
    console.log('message received: ' + data);
    messages.push(data);
    socket.send(data);
  });
});
```

Вот так мы организовали массив всех сообщений, полученных сервером нашего чата.

Следующий этап — предоставить новым пользователям возможность видеть все предыдущие сообщения. Измените обработчик события подключения в файле `websockets-server.js`, чтобы он отправлял все старые сообщения каждому новому соединению:

```
...
ws.on('connection', function (socket) {
  console.log('client connection established');

  messages.forEach(function (msg) {
    socket.send(msg);
  });

  socket.on('message', function (data) {
    console.log('message received: ' + data);
    messages.push(data);
    socket.send(data);
  });
});
```

Как только соединение установлено, сервер проходит в цикле по сообщениям и отправляет их новому соединению.

Последнее, что осталось сделать, — организовать отправку новых сообщений по мере их поступления всем пользователям. WebSockets отслеживает подключенных пользователей. Воспользуемся этим механизмом в файле `websockets-server.js` для ретрансляции полученных сообщений:

```
...
ws.on('connection', function (socket) {
  console.log('client connection established');

  messages.forEach(function (msg) {
    socket.send(msg);
  });

  socket.on('message', function (data) {
    console.log('message received: ' + data);
```

```
    messages.push(data);
    ws.clients.forEach(function (clientSocket) {
      clientSocket.send(data);
    });

    socket.send(data);
  });
});
```

Объект `ws` отслеживает все соединения с помощью своего свойства `clients`. Оно представляет собой массив, по которому можно пройти в цикле. Вам нужно только отправить (`send`) данные сообщений в обратном вызове итератора.

Наконец, поскольку вы заканчиваете отправкой сообщения своему собственному сокету при итерации по всем сокетам, необходимости в вызове метода `socket.send(data)` больше нет. Удалите его.

Наш первый чат!

Протестируем новую функциональность. Убедитесь, что утилита nodemon перечитала наш код (при необходимости можно вручную завершить работу nodemon нажатием Control+C и запустить ее снова, используя команду `npm run dev`).

Откройте третье окно терминала и выполните команду `wscat -c http://localhost:3001` (у вас должен оказаться один терминал с запущенной командой `nodemon` и два — с `wscat`). Введите несколько сообщений чата в два подключенных к серверу окна.

Пообщавшись немного с самим собой, откройте четвертый терминал и выполните команду `wscat -c http://localhost:3001`. Этому клиенту чата будут отправлены все предыдущие сообщения.

Если все пройдет нормально, вы должны увидеть что-то вроде того, что изображено на рис. 16.4.

Поздравляем! Вы создали полнофункциональный сервер чата, используя протокол WebSockets. И это потребовало менее трех десятков строк кода JavaScript.

Для самых любознательных: библиотека socket.io для WebSockets

Модуль npm `ws` — замечательная реализация протокола WebSockets. Но следует признать, что она неидеальна. Например, подключения WebSockets иногда обрываются, но `ws` не обеспечивает автоматического переподключения.

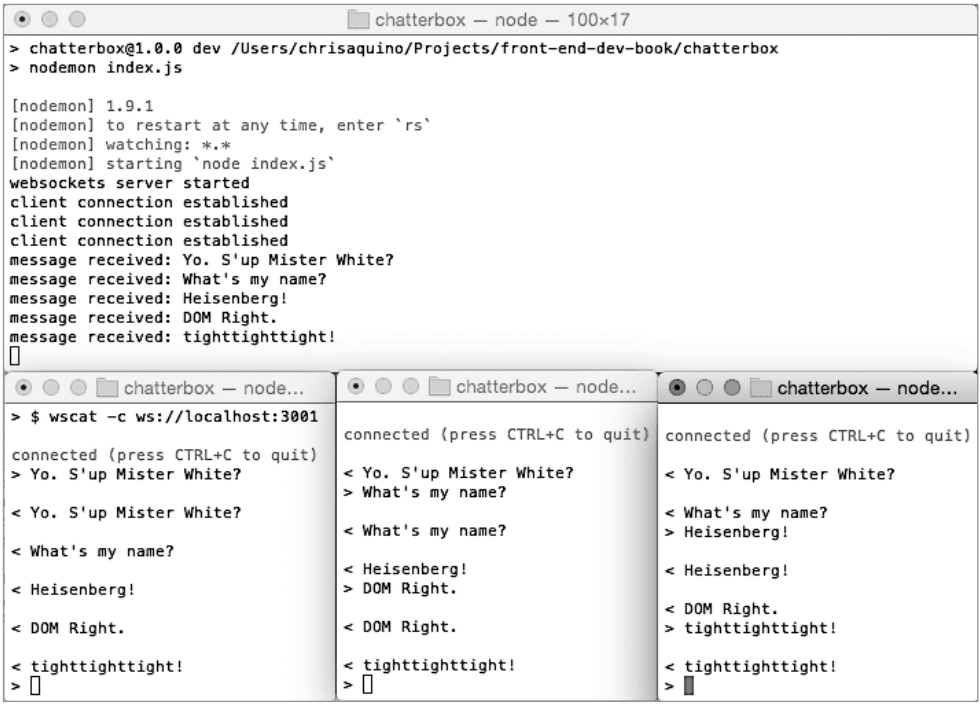


Рис. 16.4. Общение с друзьями

Другая проблема — `ws` располагается на платформе Node.js. Это значит, что он доступен только на сервере. Вам придется изучить совершенно другую библиотеку для выполнения тех же задач в коде JavaScript на стороне клиента.

В довершение всего на стороне клиента вас будут ожидать дополнительные трудности: что если ваш браузер устарел и не поддерживает WebSockets? Придется предусмотреть какой-то механизм автоматического переключения.

Библиотека `socket.io` позволяет решить эти проблемы. Браузерам она обеспечивает обратно совместимые механизмы автоматического переключения, включая реализацию Flash. Кроме того, она была перенесена на множество других платформ, включая iOS и Android.

Для самых любознательных: WebSockets как сервис

Если вас интересует платформа реального времени как сервис, рекомендуем обратить внимание на firebase (www.firebase.com). В то время как библиотека `socket.io` пытается облегчить написание вами сервера, firebase идет дальше: предоставляет сам сервер,

включая механизмы, позволяющие клиентам совместно использовать и синхронизировать данные; firebase предлагает решение для браузеров и решения для систем iOS и Android.

Бронзовое упражнение: не повторяюсь ли я?

Измените обработчик `message` таким образом, чтобы каждое полученное сообщение отправлялось каждому пользователю дважды.

Протестируйте его с помощью команды `wscat` и убедитесь, что все сообщения повторяются.

Чтобы добиться по-настоящему интересного эффекта, увеличивайте количество повторений на единицу всякий раз, когда отправляется новое сообщение.

Серебряное упражнение: «тихий» бар

В 20-е годы XX века производство и продажа спиртного в США были запрещены. В ответ начали появляться так называемые тихие бары: заведения, продававшие алкогольные напитки и требовавшие от клиента назвать пароль, прежде чем он переступит порог.

Создайте «тихую» версию программы чата, но без спиртного. Скрывайте все сообщения от пользователя до ввода секретного пароля (хороший вариант пароля — «меч-рыба»[1] — по историческим причинам).

После ввода пользователем пароля отправьте ему все предыдущие сообщения и разрешите видеть новые.

Золотое упражнение: чат-бот

Мы воспользовались свойством `WebSocket.Server` для создания сервера чата. Можно также программным образом создать клиента для чата, используя `WebSocket` в качестве конструктора.

В следующей строке приведен пример, как это сделать:

```
var chatClient = new WebSocket('http://localhost:3001');
```

[1] Этот пароль, часто используемый в кино, впервые прозвучал в фильме 1932 года «Лошадиные перья», в сцене, когда один из персонажей пытается пройти в «тихий» бар. — *Примеч. пер.*

В документации на сайте github.com/websockets/ws имеется пример отправки и получения текстовых данных.

Создайте чат-бот, который бы автоматически подключался к серверу чата. Он должен здороваться с каждым новым пользователем, но в остальное время молчать, за исключением случаев, когда обращаются непосредственно к нему. Например, если ваш чат-бот откликается на имя Джинкс, можно набрать: «Джинкс, отправь Макса в космос», и ваш чат-бот ответит соответствующим образом (что считать соответствующим — на ваше усмотрение).

Сделайте так, чтобы код вашего чат-бота находился в отдельном модуле, а не был встроен в код сервера чата.

17 Используем ES6 с помощью компилятора Babel

Язык программирования JavaScript был создан в 1994 году, изменен в 1999-м, после чего не менялся вплоть до 2009-го. Небольшая подборка изменений была предложена в 2009-м, что привело к созданию версии JavaScript, известной под названием ES5 (пятая версия стандарта).

В 2015 году в шестую версию стандарта языка было внесено немало усовершенствований. Многие из новых возможностей языка были добавлены под влиянием таких языков программирования, как Ruby и Python. Формально шестая версия называется ES2015, но более известна как ES6.

ES6 отлично поддерживается браузерами Google Chrome, Mozilla Firefox и Microsoft Edge. Это так называемые *всегда актуальные* браузеры, в том смысле, что они обновляются самостоятельно, пользователю не нужно вручную скачивать и устанавливать свежую версию. Компании Google, Mozilla и Microsoft по мере роста совместимости своих браузеров с ES6 могли быстро распространять эти усовершенствования среди пользователей.

Однако те браузеры, которые не являются всегда актуальными, а также большинство мобильных браузеров плохо поддерживают ES6. Рисунок 17.1 демонстрирует процент поддерживаемых возможностей ES6 последними версиями браузеров для настольных компьютеров и мобильных браузеров (на рисунке IE = Internet Explorer, FF = Mozilla Firefox, CH = Google Chrome, SF = Safari, KQ = Konqueror и AN = Android).

Если вы хотите узнать больше подробностей или получить более актуальную информацию по поддержке браузерами ES6, зайдите на сайт kangax.github.io/compat-table/es6/, чтобы увидеть самые свежие данные. Создатель этой таблицы Юрий Зайцев регулярно обновляет информацию.

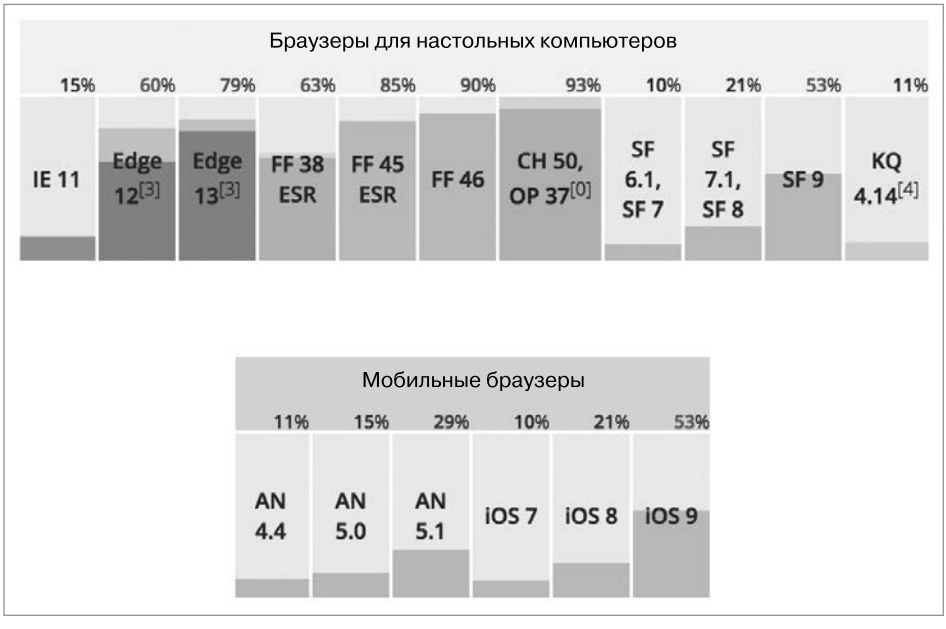


Рис. 17.1. Поддержка возможностей ES6 по состоянию на весну 2016 года

Хотя поддержка ES6 старыми браузерами и неоднородна, мы очень, очень, очень любим ES6. Это чудесная вещь, и стоит начать ее использовать как можно скорее, а не ждать поддержки всеми браузерами.

В данной главе мы начнем работать над интерфейсной частью приложения Chattrbox, при разработке которой будем использовать возможности ES6. Чтобы гарантировать работу нашего приложения на всех браузерах, воспользуемся свободно распространяемым инструментом Babel для обеспечения совместимости.

Прежде чем начать, позаботимся об одном техническом нюансе. Чтобы вы могли не отвлекаться и сосредоточить внимание на изучении ES6 и использовании Babel, можете взять уже готовые файлы `index.html` и `stylesheets/styles.css` по адресу www.bignerdranch.com/downloads/front-end-dev-resources.zip. Скачайте файл `.zip` и извлеките его содержимое (включая весь каталог `stylesheets/`) в каталог `chattrbox/app` (файл `index.html` при этом должен заменить существующую копию `index.html`).

Кроме того, еще одно замечание: при работе над кодом в этой главе вы можете столкнуться в консоли с предупреждением относительно типа MIME ваших файлов CSS. Это предупреждение можно спокойно проигнорировать.

Вперед и вверх! К концу этой главы приложение Chattrbox научится взаимодействовать с сервером чата посредством протокола WebSockets (рис. 17.2).

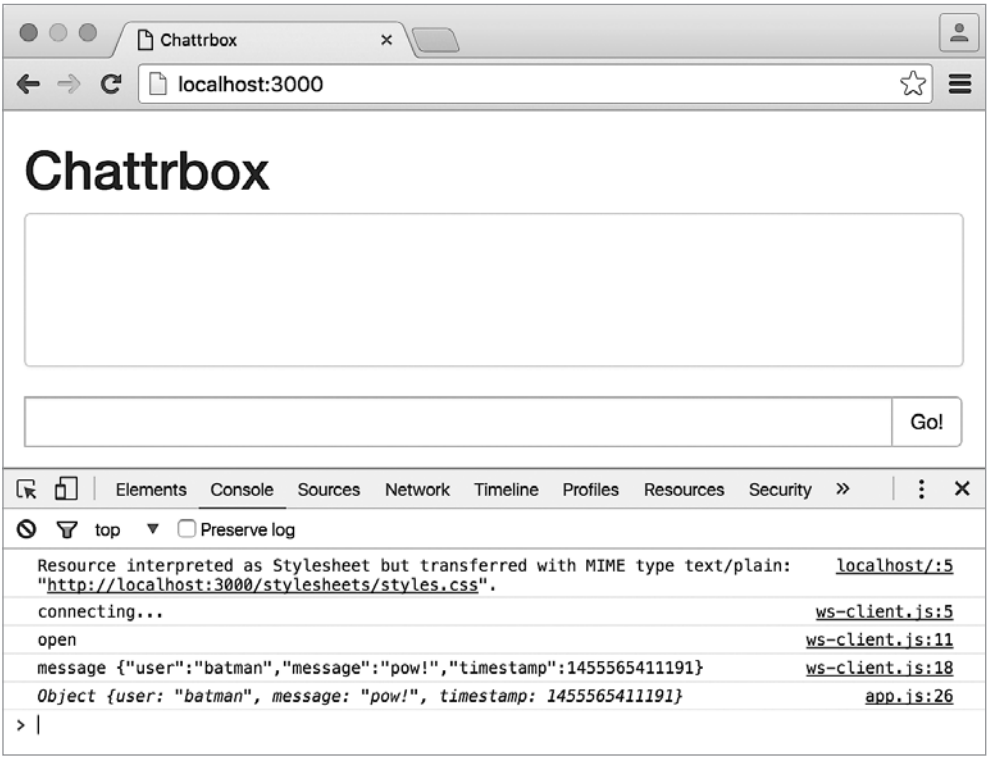


Рис. 17.2. Приложение Chattrbox по состоянию на конец данной главы

Инструменты для компиляции JavaScript

Babel — это компилятор, и его задача заключается в трансляции синтаксиса ES6 в эквивалентный код ES5, выполняемый JavaScript-движком браузера (рис. 17.3).

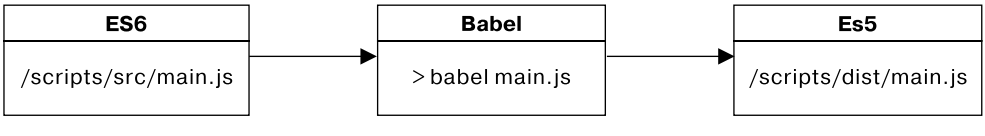


Рис. 17.3. Формирование кода ES5 из файлов ES6

Чтобы эффективно использовать компилятор Babel, нам понадобится установить несколько модулей npm для автоматизации процесса сборки. Мы будем применять Babel для компиляции кода ES6 в ES5, модуль `Browserify` — для компоновки модулей в один файл, `Babelify` — для обеспечения совместной работы двух предыдущих утилит. Кроме того, мы задействуем модуль `Watchify` с целью запуска процесса сборки всякий раз, когда сохраняются изменения в коде (рис. 17.4).

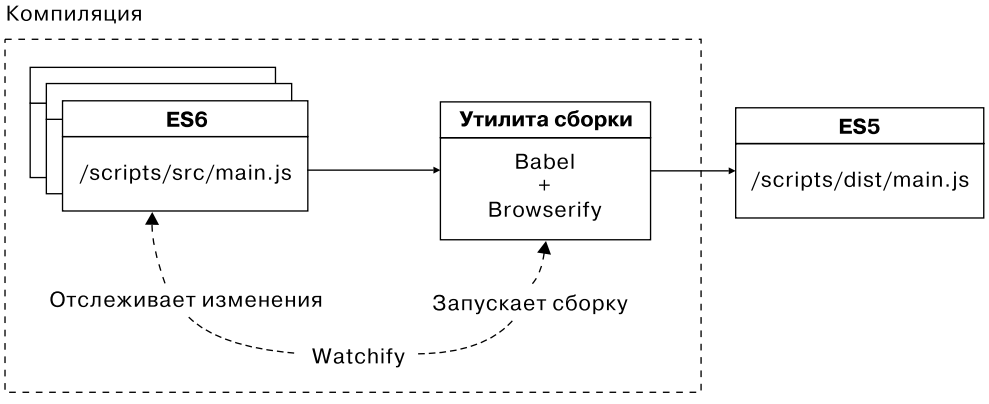


Рис. 17.4. Поток выполнения компиляции

Во-первых, нам нужно установить компилятор Babel. Он состоит из нескольких частей, которые можно подключать в зависимости от наших потребностей. В нашем случае понадобится возможность компилировать двумя способами: из командной строки и программными средствами. Эти задачи решают соответственно утилиты babel-cli и babel-core. Нам также понадобится установить конфигурацию Babel, подходящую для компиляции кода, соответствующего стандарту ES6 — babel-preset-es2015.

Выполните следующие команды npm в каталоге `chattrbox` для установки соответствующих инструментов Babel (если вы забыли, как выполнять `npm install -g` с правами администратора, обратитесь к главе 1):

```
npm install -g babel-cli
npm install --save-dev babel-core
npm install --save-dev babel-preset-es2015
```

Теперь необходимо сконфигурировать Babel для использования установленного нами заранее заданного набора настроек `es2015`. Создайте файл `.babelrc` в корневом каталоге `chattrbox` и добавьте в него следующую конфигурационную информацию:

```
{
  "presets": [
    "es2015"
  ],
  "plugins": []
}
```

Наконец, установите модули `Babelify`, `Browserify` и `Watchify` в каталог `chattrbox/node_modules/`:

```
npm install --save-dev browserify babelify watchify
```

Мы воспользуемся этими тремя утилитами далее в главе, после того как настроим и запустим Babel.

Клиентское приложение Chattrbox

Мы уже создали сервер Chattrbox, выдающий статические файлы и управляющий обменом данными через протокол WebSockets. Клиентское приложение будет отправлять на сервер и получать с него сообщения через WebSockets, а также задавать формат отдельных сообщений. Пользователь сможет просматривать сообщения в списке, а также создавать новые сообщения путем ввода текста в форму.

За выполнение этих задач будут отвечать три модуля:

- модуль `ws-client` будет управлять обменом данными через WebSockets для клиента;
- модуль `dom` будет отображать данные в пользовательском интерфейсе и обрабатывать подтверждения отправки форм;
- модуль `app` будет задавать структуру сообщений и передавать сообщения между `ws-client` и `dom`.

Рисунок 17.5 схематично изображает взаимодействие этих трех модулей.

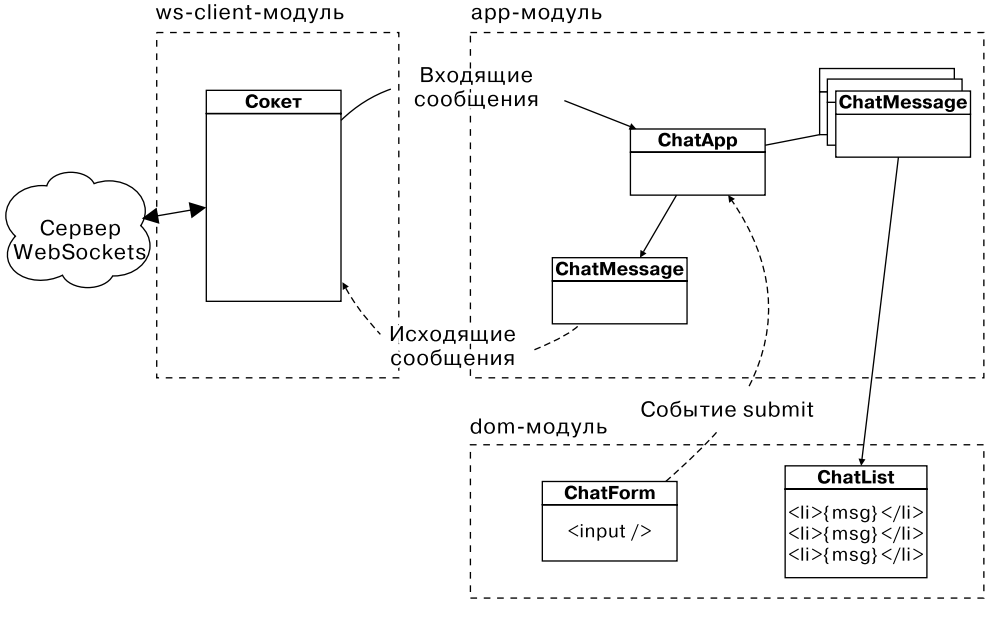


Рис. 17.5. Модули приложения Chattrbox

Создайте в каталоге `chattrbox/app` подкаталоги `scripts`, `scripts/dist` и `scripts/src`, как показано на рис. 17.6.

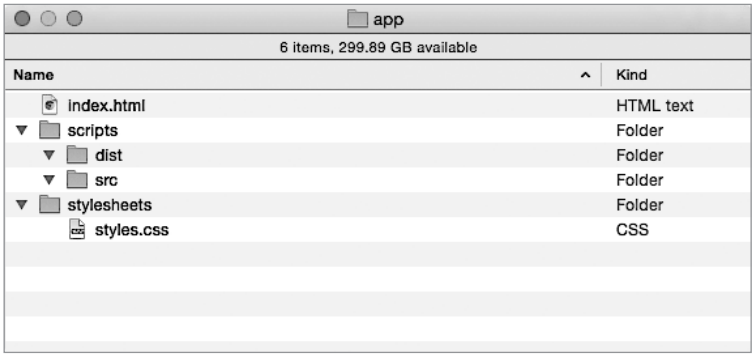


Рис. 17.6. Структура каталога chattrbox/app

Теперь создайте четыре JavaScript-файла в `scripts/src`:

- `app.js`;
- `dom.js`;
- `main.js`;
- `ws-client.js`.

Ваша структура файлов должна выглядеть так, как показано на рис. 17.7.

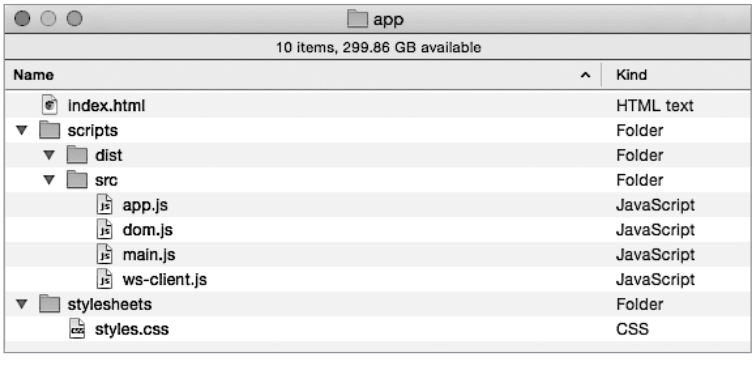


Рис. 17.7. chattrbox/app

Файлы `app.js`, `dom.js` и `ws-client.js` соответствуют показанным на рис. 17.5 модулям. Файл `main.js` будет содержать код для инициализации нашего приложения.

Начинаем работу с Babel

Теперь, установив утилиты и создав файлы, можно приступать к работе с ES6. Пока что мы будем пользоваться компилятором Babel из командной строки. Позднее

внесем его в наши сценарии npm, чтобы компиляция происходила автоматически. Работая с новыми возможностями ES6, вы сможете сосредоточить внимание на синтаксисе и не отвлекаться на выполнение лишних команд в терминале.

Синтаксис класса. Первой возможностью ES6, которой мы воспользуемся для создания клиента приложения Chattrbox, будет ключевое слово `class`. Важно помнить, что ключевое слово `class` в компиляторе Babel работает не совсем так, как классы в других языках программирования. Вместо обычных классов ES6 предоставляет сокращенный синтаксис для функций-конструкторов и методов `prototype`.

Откройте файл `app.js` и опишите там новый класс `ChatApp`:

```
class ChatApp {
}
```

В данной главе `ChatApp` ничего особенного делать не будет. В конечном счете, однако, `ChatApp` будет отвечать за бо́льшую часть логики нашего приложения.

Описание класса пока что пусто. Добавьте в него метод `constructor` с оператором `console.log`:

```
class ChatApp {
  constructor() {
    console.log('Hello ES6!');
  }
}
```

`constructor` — метод, выполняющийся при каждом создании нового экземпляра класса. Обычно конструктор задает значения свойств экземпляра.

Далее создайте экземпляр класса `ChatApp` в файле `app.js` сразу после объявления класса:

```
class ChatApp {
  constructor() {
    console.log('Hello ES6!');
  }
}
new ChatApp();
```

Опробуем наш код в деле. Откройте второе окно терминала и перейдите в корневой каталог приложения Chattrbox, где располагаются файлы `package.json`, `index.js` и каталог `app/`. Мы будем использовать это окно для выполнения команд сборки, а второе оставим открытым для работы сервера.

Чтобы протестировать наш код, воспользуемся Babel для компиляции файла `app/scripts/src/app.js` и вывода результата в файл `app/scripts/dist/main.js`:

```
babel app/scripts/src/app.js -o app/scripts/dist/main.js
```

Если на первый взгляд ничего в терминале не произошло — это хорошая новость. Компилятор Babel ни о чем не уведомляет в командной строке, если не произошла какая-нибудь ошибка (рис. 17.8).

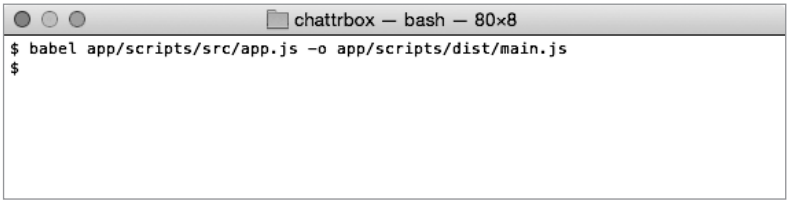


Рис. 17.8. Babel работает молча

Убедитесь, что сервер Node запущен во втором терминале (с помощью команды `npm run dev`), и перейдите в браузере по адресу http://localhost:3000. Теперь вы увидите результаты (рис. 17.9).

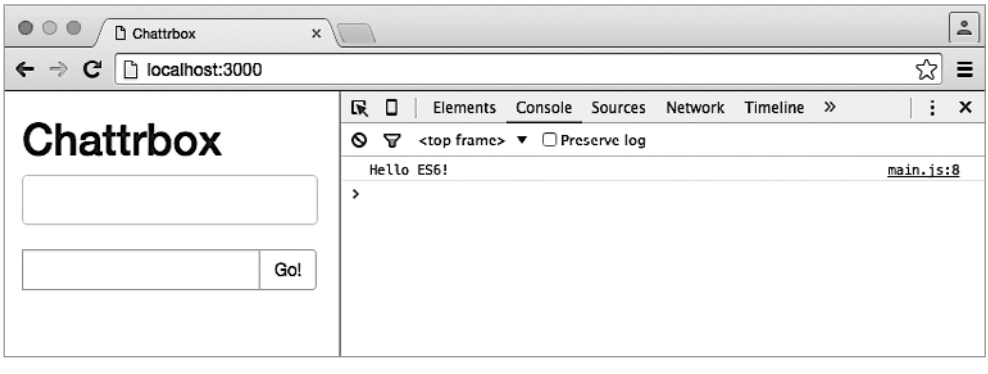


Рис. 17.9. Здравствуй, ES6!

Файл `app/index.html` будет включать сгенерированный из `app.js` файл `main.js`. И поскольку в файле `app.js` создается новый экземпляр `ChatApp`, выполняется код в конструкторе класса `ChatApp`, выводя `Hello, ES6!`.

Теперь, когда мы убедились, что компилятор Babel работает для отдельного JavaScript-файла, настало время приступить к работе с несколькими модулями.

Используем Browserify для компоновки модулей

В ES5 нет встроенной системы модулей. При создании приложения CoffeeRun мы использовали обходной путь, который дал нам возможность писать модульный код, но зависел от изменения глобальной переменной.

ES6 предоставляет нам настоящие модули (как во многих других языках программирования). Компилятор Babel понимает синтаксис модулей ES6, но эквивалентного кода ES5, в который он мог бы преобразовать его, не существует. Именно поэтому нам необходима утилита Browserify.

Рисунок 17.10 показывает совместную работу Babel и Browserify.

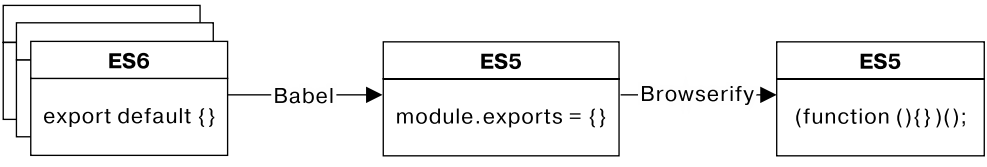


Рис. 17.10. Преобразование из модулей ES6 в модули ES5 с помощью Babel и Browserify

По умолчанию Babel преобразует синтаксис модулей ES6 в эквивалентный синтаксис `require` и `module.exports` в стиле платформы Node.js. После этого утилита Browserify преобразует модульный код Node.js в подходящие для ES5 функции.

Откройте файл `package.json` и добавьте в него раздел конфигурации для пакета Browserify:

```
...
  "scripts": {
    "test": "echo \"Error: no test specified\" &&
     exit 1",
    "start": "node index.js",
    "dev": "nodemon index.js",
  },
  "browserify": {
    "transform": [
      ["babelify", {"presets": ["es2015"], "sourceMap": true}]
    ]
  },
...
```

Этот код говорит пакету Browserify использовать в качестве плагина Babelify. Он передает Babelify два параметра. Во-первых, делает активной опцию компилятора `ES2015`, во-вторых, включает опцию `sourceMap`, помогающую при отладке. Во время создания оставшейся части приложения Chattrbox вы научитесь выполнять отладку с помощью карт кода.

Хорошо было бы также написать сценарии для некоторых распространенных задач Browserify аналогично тому, как мы сделали для программы nodemon. Выполните это в разделе `"scripts"` в файле `package.json` (не забудьте добавить запятую в конце `"dev": "nodemon index.js"`):

```
...
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1",
```

```
    "start": "node index.js",
    "dev": "nodemon index.js",
    "build": "browserify -d app/scripts/src/main.js -o app/scripts/dist/main.js",
    "watch": "watchify -v -d app/scripts/src/main.js -o app/scripts/dist/main.js"
  },
  "browserify": {
    "transform": [
      ["babelify", {"presets": ["es2015"], "sourceMap": true}]
    ]
  },
...
```

Первый сценарий `build` использует непосредственно команду `browserify`. Второй сценарий `watch` — утилиту Watchify для перезапуска Browserify при изменении кода (аналогично nodemon).

Теперь дадим некоторые пояснения насчет использования системы модулей ES6. В модулях ES6 необходимо явным образом экспортировать те части модулей, которые будут использоваться кем-то еще. Исправьте файл `app.js`, чтобы экспортировать класс `ChatApp` вместо простого создания экземпляра:

```
class ChatApp {
  constructor() {
    console.log('Hello ES6!');
  }
}
~~new ChatApp();~~
export default ChatApp;
```

Мы определили `ChatApp` в качестве значения по умолчанию, доступного из модуля `app`. Некоторые другие наши модули будут экспортировать несколько значений. Когда необходимо экспортировать только одно значение, лучше использовать `export default`.

Импортируйте в файле `main.js` класс `ChatApp` и создайте новый его экземпляр:

```
import ChatApp from './app';
new ChatApp();
```

Файл `main.js` импортирует класс `ChatApp`, экспортируемый в файле `app.js`. После импорта мы создали новый экземпляр класса `ChatApp`.

Важное примечание: имя `ChatApp` несущественно в файле `main.js`. Поскольку `ChatApp` экспортируется по умолчанию из файла `app.js`, можно было бы, написав, например, `import MyChatApp from './app'`, привести экспортируемое по умолчанию значение в соответствие локальному имени `MyChatApp`. Однако мы рекомендуем назвать его `ChatApp`, поскольку таково его имя в файле `app.js`.

Запуск процесса сборки. Перейдите в терминал и запустите сценарий сборки:

```
npm run build
```

npm запустит команду `build`, которая, в свою очередь, выполнит команду `browserify`. По мере выполнения каждой из команд будет выводиться информация о совершаемых действиях. Впрочем, сама по себе утилита Browserify работает молча, за исключением случая возникновения ошибки (рис. 17.11).

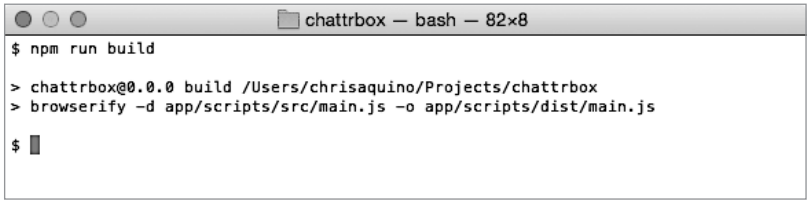


Рис. 17.11. Запуск утилиты Browserify посредством команды npm run build

В случае успешной работы утилита Browserify скомпонует преобразованный компилятором Babel файл `main.js` в каталог `app/dist/`, точно так же, как мы ранее делали вручную.

Теперь обновите страницу в браузере и оцените результат. Мы не добавили никакой новой функциональности, разве что поменяли место расположения вызова конструктора класса `ChatApp`. Так что вы должны увидеть в консоли то же самое сообщение, что и ранее (рис. 17.12).

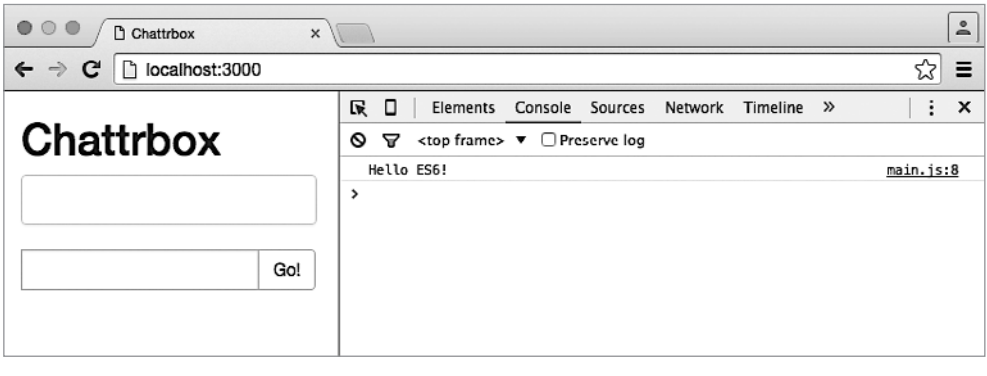


Рис. 17.12. И снова здравствуйте!

Следующий инструмент, который нам нужно подключить, — Watchify. Утилита Watchify сделает то же самое для запуска сборки с использованием Browserify, что программа nodemon делала при работе сервера Node.js. Она будет автоматически запускать повторную сборку при каждом изменении файлов исходного кода.

Запустите утилиту Watchify, чтобы процесс сборки начинался каждый раз, когда вы будете менять что-либо в коде:

```
npm run watch
```

`Watchify` подтвердит, что она запущена (рис. 17.13).

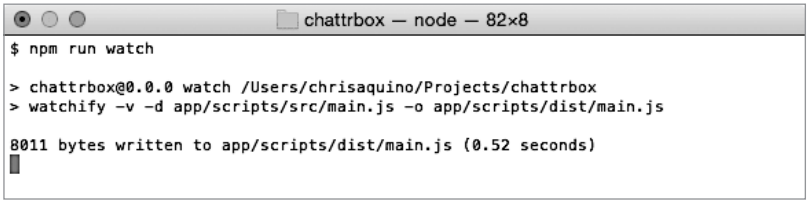


Рис. 17.13. Запуск утилиты Watchify с помощью команды npm run watch

Утилита Watchify более «общительна», чем Browserify. Каждый раз, когда она запускает Browserify, она сообщает вам, сколько байт записано в файл. Это не очень интересно, но вы сможете увидеть, если вывод изменится. Оставьте утилиту Watchify запущенной в одном из ваших терминалов, пока продолжаете работать над приложением Chattrbox (сервер должен по-прежнему быть запущен в другом терминале).

Добавление класса ChatMessage

Общение между двумя терминалами — вещь забавная, но пришло время усовершенствовать приложение для отправки сообщений от браузера к браузеру. Сейчас вы напишете вспомогательный класс, который будет отвечать за построение и форматирование данных сообщения.

Существует три элемента информации, которые нам хотелось бы отслеживать для каждого сообщения: текст сообщения, кто его отправил и когда.

Нотация объектов JavaScript (JavaScript Object Notation) — более известная как JSON (произносится «джейсон») и разработанная Дугласом Крокфордом — облегченный формат обмена данными. Мы уже использовали JSON для файла `package.json`. Она легко читается, может применяться практически с любым языком программирования и идеальна для отправки и получения данных того типа, которые будут задействованы для обмена в Chattrbox.

Вот пример сообщения в формате JSON:

```
{
  "message": "I'm Batman",
  "user": "batman",
  "timestamp": 614653200000
}
```

Данные сообщений приложения Chattrbox будут поступать из двух различных источников. Один источник — клиентское приложение (при заполнении пользователем формы), второй — сервер (при отправке сообщения через соединение по протоколу WebSockets другим клиентам).

При поступлении данных сообщения из формы необходимо добавить имя пользователя и метку даты/времени, прежде чем отправлять его на сервер. При поступлении данных с сервера должны быть включены все три части информации. Что делать с этим несоответствием? Есть несколько вариантов. Вкратце рассмотрим некоторые из них, включая те, что используют определенные возможности ES6.

Создайте в файле `app.js` класс, который будет представлять отдельные сообщения чата:

```
class ChatApp {
  constructor() {
    console.log('Hello ES6!');
  }
}

class ChatMessage {
  constructor(data) {
  }
}

export default ChatApp;
```

Первый способ решения описанной выше проблемы — простой конструктор, принимающий на входе текст сообщения, имя пользователя и метку даты/времени (не вносите эти изменения в ваш файл, это всего лишь пример):

```
...
class ChatMessage {
  constructor(message, user, timestamp) {
    this.message = message;
    this.user = user || 'batman';
    this.timestamp = timestamp || (new Date()).getTime();
  }
}
...
```

Мы уже встречались с этим паттерном неоднократно. Мы присвоили значения параметров свойствам экземпляра, обеспечив запасные значения (на случай отсутствия значений основных) для `username` и `timestamp` с помощью оператора `||`.

Это верное решение, но ES6 позволяет записать тот же самый паттерн компактнее, воспользовавшись *аргументами по умолчанию*:

```
...
class ChatMessage {
  constructor(message, user='batman', timestamp=(new Date()).getTime()) {
    this.message = message;
    this.user = user;
    this.timestamp = timestamp;
  }
}
...
```

Этот синтаксис демонстрирует, какие значения обязательные, а какие — нет. Как вы можете видеть, здесь обязателен только аргумент `message`. Для остальных указаны значения по умолчанию.

Данная версия конструктора может обрабатывать как полученные от сервера, так и созданные формой сообщения. Однако нужно, чтобы вызывающая сторона знала порядок аргументов, что может оказаться обременительным для функций и методов с тремя и более аргументами.

Альтернатива — получение конструктором в качестве аргумента одного объекта, содержащего пары «ключ/значение», задающие значения для свойств `message`, `user` и `timestamp`. Для этого можно использовать *синтаксис деструктурирующего присваивания*.

```
...
class ChatMessage {
  constructor({message: m, user: u, timestamp: t}) {
    this.message = m;
    this.user = u;
    this.timestamp = t;
  }
}
...
```

Деструктуризация может выглядеть непривычно, но посмотрим, как она работает. Конструктор вызывается следующим образом:

```
new ChatMessage({message: 'hello from the outside',
                 user: 'adele25@bignerdranch.com', timestamp: 1462399523859});
```

При использовании синтаксиса деструктуризации выполняется поиск в аргументе ключевого слова `message`. Находится значение `'hello from the outside'`, которое и присваивается новой локальной переменной `m`. Эту переменную далее можно применять внутри тела конструктора. То же самое со свойствами `user` и `timestamp`.

Но при таком синтаксисе снижается удобство использования параметров по умолчанию. К счастью, можно совместить оба метода. Вот итоговая версия конструктора, которую вам необходимо вставить в файл `app.js`:

```
...
class ChatMessage {
  constructor(~~data~~){
    message: m,
    user: u='batman',
    timestamp: t=(new Date()).getTime()
}) {
    this.message = m;
    this.user = u;
    this.timestamp = t;
  }
}
...
```

В этой версии мы извлекаем значения из передаваемого конструктору объекта. Для любых отсутствующих значений предоставляются значения по умолчанию.

Хотя аргументы по умолчанию могут существовать только в виде части описания функции (или конструктора), деструктуризация может использоваться при присваивании. Можно также написать конструктор следующим образом:

```
...
class ChatMessage {
  constructor(data) {
    var {message: m, user: u='batman', timestamp: t=(new Date()).getTime()} = data;
    this.message = m;
    this.user = u;
    this.timestamp = t;
  }
}
...
```

А сейчас пора вернуться к Chattrbox!

Наш класс `ChatMessage` содержит всю важную информацию в виде свойств, но его экземпляры также наследуют методы `ChatMessage` и другие данные. Это делает экземпляры `ChatMessage` неподходящими для отправки через протокол WebSockets. Необходима урезать эту информацию до минимума.

Напишите в файле `app.js` метод `serialize`, который будет представлять данные из свойств `ChatMessage`, в виде обычного JavaScript-объекта.

```
...
class ChatMessage {
  constructor({
    message: m,
    user: u='batman',
    timestamp: t=(new Date()).getTime()
  }) {
    this.message = m;
    this.user = u;
    this.timestamp = t;
  }
  serialize() {
    return {
      user: this.user,
      message: this.message,
      timestamp: this.timestamp
    };
  }
}

export default ChatApp;
```

Теперь наш класс `ChatMessage` готов к использованию. Пришло время перейти к следующему модулю для приложения Chattrbox.

Создание модуля ws-client

Модуль `ws-client` обеспечивает обмен данными с сервером Node WebSocket.

У него четыре обязанности:

- подключение к серверу;
- выполнение начальных настроек при первом открытии соединения;
- пересылка входящих сообщений их обработчикам;
- отправка исходящих сообщений.

Рассмотрим, как эти задачи соотносятся с другими нашими компонентами (рис. 17.14).

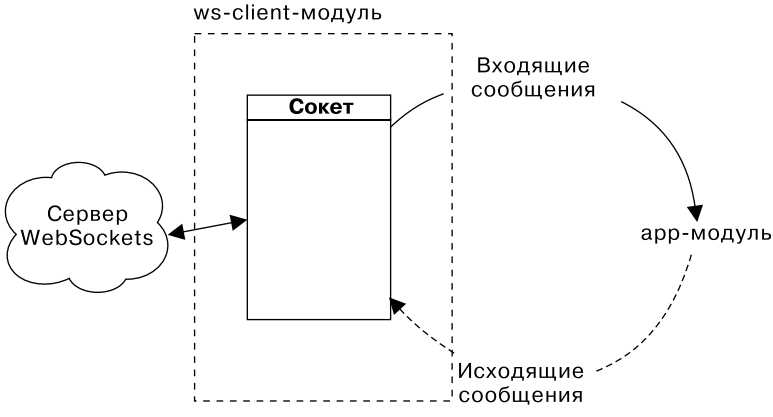


Рис. 17.14. Интерфейсы модуля ws-client

Организация обмена данными

Прежде всего организуем подключение. Откроем файл `ws-client.js` и объявим переменную для соединения WebSockets:

```
let socket;
```

Это объявление использует новый для нас способ объявления переменных в ES6, называемый *управлением областью видимости* переменных. При его применении для объявления переменной (при использовании ключевого слова `let` вместо `var`) переменная не будет *поднята*.

Поднятие переменных означает, что объявления переменных перемещаются в начало области видимости функции, в которой они созданы. Эту работу незаметно для нас делает интерпретатор языка JavaScript. К сожалению, поднятие переменных может приводить к трудно обнаруживаемым ошибкам.

Прочитать о поднятии переменных подробнее вы можете в конце данной главы. Пока что достаточно знать, что `let` — безопасный способ объявления переменных в выражениях `if/else` и в теле цикла.

Теперь добавьте в файл `ws-client.js` метод для инициализации нашего подключения.

```
let socket;

function init(url)  {
  socket = new WebSocket(url);
  console.log('connecting...');
}
```

Функция `init` подключается к серверу WebSockets. Далее нам нужно связать модуль `ws-client` с `ChatApp` в файле `app.js`.

Чтобы оставаться действующим, модуль `ws-client` должен четко определять, что он экспортирует. Нам нужно экспортировать одно значение: код объекта с экспортируемыми функциями в качестве его свойств. Мы воспользуемся тем же синтаксисом `export default`, который использовали в начале данной главы, плюс немного удобств от ES6.

Добавьте экспорт в конец файла `ws-client.js`:

```
...
function init(url)  {
  socket = new WebSocket(url);
  console.log('connecting...');
}

export default {
  init,
}
```

Обратите внимание, что нам не пришлось указывать имена свойств. Эта сокращенная запись эквивалентна следующему:

```
export default {
  init: init
}
```

Если имена у ключа и значения одинаковые, можно опускать двоеточие и значение. Ключ автоматически становится именем переменной, а значение — связанным с этим именем. Эта возможность ES6 называется *расширенным синтаксисом литеральных объектов*.

После окончания работ по настройке модуля `ws-client` настало время импортировать предоставляемые им значения в `app.js`. Начните с добавления оператора импорта вверху файла `app.js`:

```
import socket from './ws-client';

class ChatApp {
  constructor() {
    console.log('Hello ES6!');
  }
}
...
```

`socket` — объект, который мы экспортировали из модуля `ws-client`.

Далее в конструкторе класса `ChatApp` вызовите метод `socket.init`, передав ему URL нашего сервера WebSockets:

```
import socket from './ws-client';

class ChatApp {
  constructor() {
    console.log('Hello ES6!');
    socket.init('ws://localhost:3001');
  }
}
...
```

Наш сценарий npm должен пересобрать код (возможно, потребуется перезапустить `npm run watch` и `npm run dev` в отдельных окнах, если вы завершили работу кого-либо из них). После обновления страницы браузера вы должны увидеть выведенное в консоль сообщение connecting... (подключаемся...), как показано на рис. 17.15.

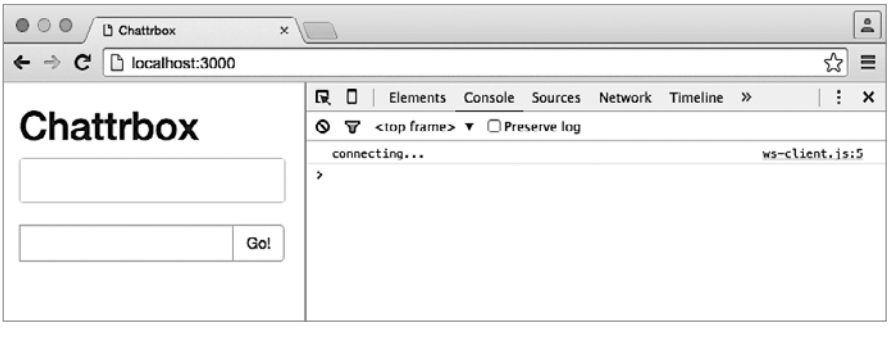


Рис. 17.15. Выведенное в консоль при инициализации WebSockets сообщение

Теперь каркас нашего приложения готов и запущен.

Обработка событий и отправка сообщений

При вызове нашим модулем `App` метода `init` создается новый экземпляр `WebSocket` и производится подключение к серверу. Но нашему модулю `App` необходимо знать, когда этот процесс завершится, чтобы выполнять дальнейшее соединение.

У объекта `WebSocket` имеется набор особых свойств для обработки событий, например `onopen`. Любая присвоенная этому свойству функция будет вызываться при установлении подключения к серверу WebSockets. Внутри этой функции можно выполнять любые требующиеся после подключения действия.

Для сохранения гибкости модуля `ws-client` и обеспечения возможности его многократного использования мы не будем «жестко зашивать» те действия, которые должен выполнять модуль `App` после соединения. Вместо этого воспользуемся тем же паттерном, который использовали при регистрации обработчиков щелчков кнопкой мыши и подтверждения отправки в приложении CoffeeRun.

Добавьте в файл `ws-client.js` функцию `registerOpenHandler`. Она будет принимать на входе обратный вызов, присваивать функцию свойству `onopen`, после чего выполнять обратный вызов внутри функции `onopen`.

```
let socket;

function init(url)  {
  socket = new WebSocket(url);
  console.log('connecting...');
}

function registerOpenHandler(handlerFunction) {
  socket.onopen = () => {
    console.log('open');
    handlerFunction();
  };
}
...
```

Такое описание функции отличается от того, что мы писали ранее. Это новый синтаксис ES6, называемый *стрелочными функциями*. Стрелочные функции — сокращенная форма записи для анонимных функций. Не считая более удобной формы записи, они работают точно так же, как и анонимные.

Метод `registerOpenHandler` принимает на входе функциональный аргумент (`handlerFunction`) и присваивает анонимную функцию свойству `onopen` соединения с сокетом. В этой анонимной функции мы вызываем переданную в нее функцию `handlerFunction`.

Надо отметить, что использовать анонимную функцию сложнее, чем просто написать `socket.onopen = handlerFunction`. Этот паттерн полезен в случаях, когда необходимо отреагировать на какое-либо событие, но сначала выполнить какие-то промежуточные шаги, например записать сообщение в журнал, как мы уже делали ранее.

Далее нам нужно написать интерфейс для обработки сообщений по мере их поступления через соединение WebSockets. Напишите в модуле `ws-client` новый метод `registerMessageHandler`. Присвойте стрелочную функцию свойству `onmessage` сокета (она должна ожидать на входе аргумент в виде события).

```
...
function registerOpenHandler(handlerFunction) {
  socket.onopen = () => {
    console.log('open');
    handlerFunction();
  };
}

function registerMessageHandler(handlerFunction) {
  socket.onmessage = (e) => {
    console.log('message', e.data);
    let data = JSON.parse(e.data);
    handlerFunction(data);
  };
}
...
```

Параметры стрелочных функций помещаются в скобки (как и у обычных функций).

Клиент приложения Chattrbox получает от сервера объект в обратном вызове `onmessage` внутри функции `registerMessageHandler`. Этот объект представляет событие, у него имеется свойство `data`, содержащее строку JSON с сервера. Каждый раз при получении строки мы преобразовываем ее в объект JavaScript, после чего пересылаем функции `handlerFunction`.

Последнее, что нам осталось, — создать фрагмент кода, отправляющий сообщение нашему WebSockets. Мы напишем его в файле `ws-client.js` в виде функции `sendMessage` и разобьем на две части. Во-первых, мы преобразуем содержимое сообщения (включающее сообщение, имя пользователя и метку даты/времени) в строку JSON, а затем отправим ее серверу WebSockets;

```
...
function registerMessageHandler(handlerFunction) {
  socket.onmessage = (e) => {
    console.log('message', e.data);
    let data = JSON.parse(e.data);
    handlerFunction(data);
  };
}

function sendMessage(payload) {
  socket.send(JSON.stringify(payload));
}
...
```

Наконец, добавим экспорты для наших новых методов с помощью расширенного синтаксиса литеральных объектов:

```
...
function sendMessage(payload) {
  socket.send(JSON.stringify(payload));
}
```

```
export default {
  init,
  registerOpenHandler,
  registerMessageHandler,
  sendMessage
}
```

После этого в модуле `ws-client` появится все необходимое для двустороннего обмена данными с сервером. Наша последняя задача, касающаяся `ws-client`, будет заключаться в тестировании его путем отправки сообщения.

Отправка и эхо-контроль сообщения

Измените конструктор `ChatApp` в файле `app.js`. После вызова `socket.init` вызовите методы `registerOpenHandler` и `registerMessageHandler`, передав им в качестве аргументов стрелочные функции:

```
import socket from './ws-client';
class ChatApp {
  constructor() {
    socket.init('ws://localhost:3001');

    socket.registerOpenHandler(() => {
      let message = new ChatMessage({ message: 'pow!' });
      socket.sendMessage(message.serialize());
    });

    socket.registerMessageHandler((data) => {
      console.log(data);
    });
  }
}
...
```

После открытия соединения нужно сразу же отправить фиктивное сообщение. А после получения вывести его в консоль.

Сохраните код и обновите страницу в браузере после завершения процесса сборки. Вы должны увидеть, что сообщение отправлено и на него получен ответ (рис. 17.16).

Отлично! У нас уже работают два из трех основных модулей приложения Chattrbox. Мы доделаем Chattrbox в следующей главе, создав модуль, связывающий существующие модули с пользовательским интерфейсом. Этот модуль будет отображать новые сообщения в списке сообщений и отправлять сообщения при подтверждении отправки формы.

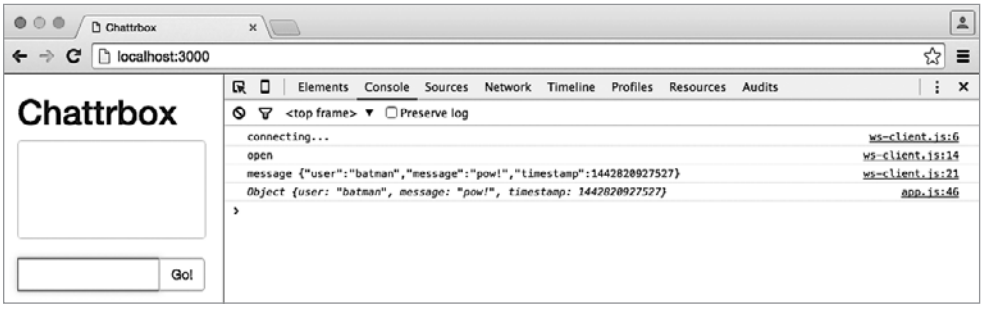


Рис. 17.16. Вызов и ответ с помощью WebSockets

Для самых любознательных: компиляция в JavaScript из других языков программирования

Существует несколько языков программирования, чей код можно скомпилировать в код на языке JavaScript. Вот краткий список этих языков:

- CoffeeScript: coffeescript.org;
- TypeScript: www.typescriptlang.org;
- C/C++: kripken.github.io/emscripten-site.

Одним из важнейших можно назвать язык программирования CoffeeScript, предоставляющий сокращенную форму записи для некоторых наиболее распространенных паттернов (например, стрелочного синтаксиса для анонимных функций). Можно даже сказать, что CoffeeScript оказал серьезное влияние на ES6.

Google, Microsoft, Mozilla и другие компании совместно работали над проектом по стандартизации языка ассемблера для движков JavaScript под названием WebAssembly. Целью проекта было создание высокопроизводительного, низкоуровневого языка, в который можно выполнять компиляцию из других языков.

Цель WebAssembly — дополнить язык JavaScript, опираясь на сильные стороны других языков. JavaScript хорошо подходит для создания браузерных приложений (но не для визуализации, требующей большого объема вычислений игровой графики). Языки программирования C и C++ отлично подходят для визуализации игрового кода. Вместо того чтобы переносить код C++ на JavaScript, потенциально создавая ошибки, можно просто скомпилировать его в WebAssembly.

Проект WebAssembly берет начало в более раннем проекте asm.js, задававшем подмножество языка JavaScript для написания высокопроизводительного кода.

Чтобы узнать больше о WebAssembly и asm.js, посмотрите следующее сообщение в блоге создателя JavaScript Брендана Айка: brendaneich.com/2015/06/from-asm-js-to-webassembly.

Бронзовое упражнение: имя по умолчанию для импорта

Наш оператор импорта в файле `main.js` создает локальную переменную `ChatApp`. Что произойдет, если вы поменяете ее имя на `ApplicationForChatting`?

Попробуйте сделать это (не забудьте поменять также оператор `new` на следующей строке) и выяснить, по-прежнему ли все работает. Если да, то почему? Если нет, то почему?

Серебряное упражнение: предупреждение о закрытии соединения

Добавьте в модуль `ws-client` еще одну функцию — `registerCloseHandler`. Она потребует обратного вызова, который будет выполняться при срабатывании на сокете события `close`.

Воспользуйтесь функцией `registerCloseHandler` в файле `main.js`, чтобы предупредить пользователя о закрытии соединения. И проверьте, что все работает.

Как можно это проверить? Конечно, вы не можете закрыть окно браузера. Вам придется закрыть соединение на другом конце.

В качестве дополнительного упражнения напишите функцию, которая бы выполняла попытку подключиться заново. Можете или воспользоваться методом `setTimeout`, или запросить у пользователя подтверждение (поищите подробности в MDN).

Для самых любознательных: поднятие переменных

Язык программирования JavaScript позволяет даже программистам-любителям создавать программы хотя бы с минимальной интерактивностью. Хотя JavaScript содержит возможности, предназначенные для обеспечения устойчивости кода к ошибкам, некоторые из них на практике приводят к появлению ошибок. Одна из таких возможностей — поднятие переменных.

При интерпретации вашего кода движком JavaScript он находит все переменные и объявления функций и перемещает их в начало функции, в которой они распола-

гаются (если они располагаются не в функции, их значения вычисляются раньше остальной части кода).

Лучше всего проиллюстрировать это на примере. Если вы напишете следующий код:

```
function logSomeValues () {
  console.log(myVal);
  var myVal = 5;
  console.log(myVal);
}
```

он будет интерпретироваться так, как если бы вы написали:

```
function logSomeValues () {
  var myVal;
  console.log(myVal);
  myVal = 5;
  console.log(myVal);
}
```

Если вы вызовете функцию `logSomeValues` в консоли, то увидите следующее:

```
> logSomeValues();
undefined
5
```

Обратите внимание, что поднимается только *объявление*. Присваивание остается на своем месте. Безусловно, это может привести к путанице, особенно если вы пытались объявить переменные в операторе `if` или внутри цикла. В других языках программирования фигурные скобки определяют *блок* со своей областью видимости. В языке JavaScript фигурные скобки блоков не создают области видимости. Область видимости создают только функции.

Взгляните еще на один пример:

```
var myVal =  11;
function doNotWriteCodeLikeThis() {
  if (myVal > 10) {
    var myVal = 0;
    console.log('myVal превышал 10; сбрасываем в 0');
  } else {
    console.log('необходимости сбрасывать значение нет.');
  }
  return myVal;
}
```

Вы могли бы ожидать, что в консоль будет выведено сообщение `myVal превышал 10; сбрасываем в 0` и возвращено значение `0`. Вместо этого будет выведено следующее:

```
> doNotWriteCodeLikeThis();
нет необходимости сбрасывать значение.
Undefined
```

Объявление `var myVal` было перемещено наверх функции, так что до вычисления значения выражения `if` значение переменной `myVal` равнялось `undefined`. Присваивание остается в блоке `if`.

Объявления функций тоже поднимаются, но целиком. Это значит, что следующий код работает без всяких проблем:

```
boo();
// Объявляем после вызова:
function boo() {
  console.log('BOO!!');
}
```

JavaScript перемещает весь блок объявления функции вверх:

```
> boo();
BOO!!
```

Операторы `let` не подлежат поднятию. Операторы `const`, предоставляющие возможность объявления переменных, значение которых нельзя поменять, — тоже.

Для самых любознательных: стрелочные функции

Мы слукавили: стрелочные функции не работают в точности так же, как анонимные. В некоторых случаях они работают *лучше*.

Помимо более краткого синтаксиса, стрелочные функции:

- работают так, как если бы вы написали `function () {}.bind(this)`, благодаря чему вы получаете правильную ссылку `this` в теле стрелочной функции;
- предоставляют вам возможность опускать фигурные скобки при наличии только одного оператора;
- возвращают результат единственного оператора в случае опущенных фигурных скобок.

Например, вот метод `CheckList.prototype.addClickHandler` приложения CoffeeRun:

```
CheckList.prototype.addClickHandler = function(fn) {
  this.$element.on('click', 'input', function (event) {
    var email = event.target.value;
    fn(email)
      .then( function () {
        this.removeRow(email);
      }.bind(this));
  }.bind(this));
};
```

Замена анонимной функции на стрелочную функцию делает этот код понятнее:

```
CheckList.prototype.addClickHandler = (fn) => {
  this.$element.on('click', 'input', (event) => {
    let email = event.target.value;
    fn(email)
      .then(() => this.removeRow(email));
  });
};
```

Выполняемые методом `addClickHandler` действия выглядят намного понятнее без лишнего мусора в виде `function` и `.bind(this)`.

18 ES6. Приключения продолжаются

Chattrbox уже работает, но в настоящий момент большая часть его кода посвящена бизнес-логике. Приложение подключается к серверу WebSockets, задает формат сообщений и умеет отправлять и получать сообщения.

В этой главе мы завершим работу над приложением Chattrbox, подключив слой UI. Мы продолжим использовать программы Node и npm для управления процессом сборки и работы в качестве сервера, и к концу главы у нас будет полнофункциональное веб-приложение для чата (рис. 18.1).

Когда мы работали над приложением CoffeeRun, мы создали модули `FormHandler` и `CheckList`, соответствующие форме и области перечня. Мы будем следовать тому же паттерну и в Chattrbox, создавая модули `ChatForm` и `ChatList`.

Мы также создадим модуль `UserStore` для хранения информации о текущем пользователе чата. Это сделает приложение Chattrbox более устойчивым к ошибкам, а его основные модули — пригодными для повторного использования.

Установка библиотеки jQuery в качестве модуля Node

Chattrbox будет использовать библиотеку jQuery для работы с DOM. Но мы не станем ни загружать jQuery с `cdnjs.com` (как делали для приложения CoffeeRun), ни использовать тег `<script>` в HTML аналогично тому, как интегрировали зависимости на стороне клиента.

При наличии модуля Browserify этого больше не требуется. Browserify автоматически компонует зависимости JavaScript в пакет приложения для дальнейшего использования в браузере. Так что все, что требуется для интеграции jQuery, — включить ее с помощью оператора `import`, а Browserify сделает все остальное.

Рис. 18.1. Готовое приложение Chattrbox

Начнем с установки библиотеки jQuery в каталог `node_modules`:

```
npm install --save-dev jquery
```

Откройте файл `dom.js`, чтобы начать создание модуля. Модуль `dom` будет использовать библиотеку jQuery, так что вставьте оператор `import`, чтобы включить ее:

```
import $ from 'jquery';
```

Далее в этой главе мы установим и будем использовать другие сторонние библиотеки. При этом пройдем те же этапы установки и импорта.

Создание класса ChatForm

Как и в приложении CoffeeRun, мы создадим объект для управления элементами формы в DOM. Им станет класс `ChatForm`. Использование классов ES6 сделает код более читабельным, чем код CoffeeRun.

Создание экземпляра `ChatForm` и инициализация его обработчиков событий будет происходить в два этапа, поскольку задача конструктора должна заключаться только в установке значений свойств экземпляра. Все остальное (например, подключение обработчиков событий) должно выполняться в других методах.

Опишите `ChatForm` в файле `dom.js` с конструктором, принимающим на входе селекторы. Добавьте в конструкторе свойства для элементов, которые должен будет отслеживать экземпляр:

```
import $ from 'jquery';

class ChatForm {
  constructor(formSel, inputSel) {
    this.$form = $(formSel);
    this.$input = $(inputSel);
  }
}
```

Далее добавьте метод `init`, чтобы связать обратный вызов с событием `submit` формы:

```
...
class ChatForm {
  constructor(formSel, inputSel) {
    this.$form = $(formSel);
    this.$input = $(inputSel);
  }

  init(submitCallback) {
    this.$form.submit((event) => {
      event.preventDefault();
      let val = this.$input.val();
      submitCallback(val);
      this.$input.val('');
    });

    this.$form.find('button').on('click', () => this.$form.submit());
  }
}
```

В методе `init` мы использовали стрелочную функцию для обработки события подтверждения отправки формы. Внутри стрелочной функции мы помешали поведению формы по умолчанию, извлекли значение из поля ввода, после чего передали это значение в `submitCallback`. Наконец, мы сбросили значение поля ввода.

Чтобы гарантировать отправку формы при нажатии кнопки, мы добавили обработчик, указывающий форме сгенерировать событие `submit`. Мы достигли этого путем получения элемента формы с помощью библиотеки jQuery (с последующим вызовом ее метода `submit`). Мы воспользовались для этого версией стрелочной функции (в виде отдельного выражения), позволяющей опустить фигурные скобки.

Чтобы можно было использовать этот модуль, необходимо экспортировать класс `ChatForm`. В предыдущей главе мы применили для этого `export default`, что дало нам возможность экспортировать для модуля одно значение. В некоторых случаях мы использовали простые JavaScript-объекты для группировки нескольких значений в одно значение по умолчанию.

В данной главе мы выберем *поименованные экспорты*, чтобы экспортировать несколько поименованных значений вместо одного значения по умолчанию.

Экспортируйте класс `ChatForm` в виде поименованного значения для пользователей данного модуля, добавив ключевое слово `export` прямо перед объявлением `class`:

```
...
export class ChatForm {
  constructor(formSel, inputSel) {
    this.$form = $(formSel);
    this.$input = $(inputSel);
  }
  ...
```

Все достаточно просто. Теперь перейдем к импорту класса `ChatForm` в файле `app.js`.

В приложениях Ottergram и CoffeeRun мы использовали ключевое слово `var` для строк селекторов. В ES6 можно объявлять для этой цели константы, поскольку значения строк не будут меняться. Подобно `let`, область видимости переменной `const` равна блоку, то есть она видна любому коду внутри тех же фигурных скобок. Если же она находится вне любых фигурных скобок (как раз наш случай), то будет видна любому коду из соответствующего файла.

В файле `app.js` импортируйте класс `ChatForm` и создайте константы для селектора формы и селектора поля ввода сообщения. Создайте также экземпляр класса `ChatForm` в функции-конструкторе `ChatApp`:

```
import socket from './ws-client';
import {ChatForm} from './dom';

const FORM_SELECTOR = '[data-chat="chat-form"]';
const INPUT_SELECTOR = '[data-chat="message-input"]';

class ChatApp {
  constructor() {
    this.chatForm = new ChatForm(FORM_SELECTOR, INPUT_SELECTOR);

    socket.init('ws://localhost:3001');
    socket.registerOpenHandler(() => {
      let message = new ChatMessage('pow!');
      socket.sendMessage(message.serialize());
    });
    socket.registerMessageHandler((data) => {
```

```
      console.log(data);
    });
  }
}
...
```

При импорте `ChatForm` мы поместили его в фигурные скобки: `{ChatForm}`. Такой синтаксис обозначает *поименованный импорт*. В результате поименованного импорта для класса `ChatForm` объявляется локальная переменная с именем `ChatForm` (она привязывается к значению из модуля `dom` с тем же именем).

Подключение ChatForm к сокету. В предыдущей главе мы отправляли фиктивное сообщение: `"pow!"`. Теперь мы готовы отправлять из `ChatForm` настоящие данные формы.

Внутри обратного вызова метода `socket.registerOpenHandler` зададим начальное значение экземпляра `ChatForm`. Важно сделать это *после* открытия сокета, а не выполнять инициализацию сразу после создания экземпляра. Подождав немного, мы предотвратим возможный ввод пользователем сообщений чата до того, как их можно будет действительно отправить на сервер.

Как вы помните, метод `init` класса `ChatForm` принимает в качестве аргумента обратный вызов. Мы воспользуемся этим обратным вызовом для выполнения отправок данных формы.

Удалите из файла `app.js` код для фиктивного сообщения и замените его вызовом метода `ChatForm.init`, передав последнему обратный вызов, отправляющий данные сообщений, поступающие в наш сокет от `ChatForm`:

```
...
class ChatApp {
  constructor() {
    this.chatForm = new ChatForm(FORM_SELECTOR, INPUT_SELECTOR);

    socket.init('ws://localhost:3001');
    socket.registerOpenHandler(() => {
      let message = new ChatMessage('pow!');
      socket.sendMessage(message.serialize());
      this.chatForm.init((data) => {
        let message = new ChatMessage({message: data});
        socket.sendMessage(message.serialize());
      });
    });
    socket.registerMessageHandler((data) => {
      console.log(data);
    });
  }
}
...
```

Посмотрим, что теперь происходит в классе `ChatApp`. Во-первых, он открывает соединение сокета к серверу. После открытия соединения `ChatApp` инициализирует

наш экземпляр `ChatForm` с помощью обратного вызова подтверждения отправки формы.

Теперь, когда пользователь подтверждает отправку сообщения в форме, экземпляр `ChatForm` передает эти данные в обратный вызов `ChatApp`, после чего обратный вызов компонует их в экземпляр `ChatMessage` и отправляет серверу WebSockets.

Создание класса ChatList

Мы обеспечили отправку исходящих сообщений чата. Следующая задача — обеспечить отображение новых сообщений от сервера по мере их поступления. Для этого мы создадим в файле `dom.js` еще один класс, олицетворяющий список сообщений чата, которые видит пользователь.

Модуль `ChatList` будет создавать для каждого сообщения элементы DOM, отображающие имя пользователя, отправившего сообщение, и текст сообщения. Для этой цели создайте и экспортируйте в файле `dom.js` описание нового класса `ChatList`:

```
import $ from 'jquery';

export class ChatForm {
  ...
}

export class ChatList {
  constructor(listSel, username) {
    this.$list = $(listSel);
    this.username = username;
  }
}
```

Конструктор `ChatList` принимает в качестве аргументов селектор атрибута и имя пользователя. Селектор атрибута необходим ему, чтобы знать, куда прикреплять создаваемым им элементы списка сообщений. Имя пользователя нужно, чтобы знать, какие сообщения были отправлены вами, а какие — остальными пользователями (ваши сообщения будут отображаться не так, как отправленные другими пользователями).

Теперь, когда у класса `ChatList` есть конструктор, ему потребуется создавать элементы DOM для сообщений.

Добавьте в класс `ChatList` метод `drawMessage`. Он будет ожидать на входе объектный аргумент, который затем начнет деструктурировать на локальные переменные для имени пользователя, метки даты/времени и относящегося к сообщению текста (чтобы пояснить, что делает деструктурирующее присваивание, используются однобуквенные локальные переменные).

```
...
export class ChatList {
  constructor(listSel, username) {
    this.$list = $(listSel);
    this.username = username;
  }

  drawMessage({user: u, timestamp: t, message: m}) {
    let $messageRow = $('<li>', {
      'class': 'message-row'
    });

    if (this.username === u) {
      $messageRow.addClass('me');
    }

    let $message = $('<p>');

    $message.append($('<span>', {
      'class': 'message-username',
      text: u     }));

    $message.append($('<span>', {
      'class': 'timestamp',
      'data-time': t,
      text: (new Date(t)).getTime()
    }));

    $message.append($('<span>', {
      'class': 'message-message',
      text: m     }));

    $messageRow.append($message);
    this.$list.append($messageRow);
    $messageRow.get(0).scrollIntoView();
  }
}
```

Метод `drawMessage` создает для сообщения строку с отображением имени пользователя, метки даты/времени и самого сообщения. Если отправитель сообщения вы, добавляется дополнительный класс CSS для стилизации. Затем он присоединяет соответствующую сообщению строку к элементу списка `ChatList` и прокручивает строку нового сообщения в видимую область.

Теперь класс `ChatList` готов к работе. Время интегрировать его в класс `ChatApp`.

Измените в файле `app.js` оператор импорта из модуля `dom`, чтобы он импортировал также и класс `ChatList`. Добавьте константу для селектора списка, после чего создайте новый экземпляр `ChatList` в конструкторе:

```
import socket from './ws-client';
import {ChatForm, ChatList} from './dom';
```

```
const FORM_SELECTOR = '[data-chat="chat-form"]';
const INPUT_SELECTOR = '[data-chat="message-input"]';
const LIST_SELECTOR = '[data-chat="message-list"]';

class ChatApp {
  constructor() {
    this.chatForm = new ChatForm(FORM_SELECTOR, INPUT_SELECTOR);
    this.chatList = new ChatList(LIST_SELECTOR, 'wonderwoman');

    socket.init('ws://localhost:3001');
    ...
```

Почти все готово. Последнее, что осталось сделать для реализации базовой функциональности чата, — отрисовывать новые сообщения по мере их поступления путем вызова метода `chatList.drawMessage`. Выполните это в функции `registerMessageHandler` в файле `app.js`:

```
...
class ChatApp {
  ...
    socket.registerMessageHandler((data) => {
      console.log(data);
      let message = new ChatMessage(data);
      this.chatList.drawMessage(message.serialize());
    });
  }
}
...
```

Мы создали новый экземпляр `ChatMessage` на основе поступивших данных, после чего сериализовали `message`. Эта мера предосторожности предназначена для удаления лишних метаданных, которые могли быть добавлены к данным. Новый экземпляр `ChatMessage`, созданный из данных сокета, выдает нам наше сообщение, а метод `this.chatList.drawMessage` отрисовывает сериализованное сообщение в браузере.

Пришло время опробовать все в деле. Запустите, если вы еще этого не сделали, утилиты Watchify (с помощью команды `npm run watch`) и nodemon (с помощью команды `npm run dev`). Откройте браузер или обновите в нем страницу, после чего введите сообщение (рис. 18.2).

Ура! Наше приложение чата работает. Осталось несколько штрихов внешнего оформления.

Использование граватаров

Gravatar — бесплатный сервис, позволяющий связывать изображение для своего профиля с адресом электронной почты. Граватары делают изображение профиля для каждого пользователя доступным по специально отформатированному URL. Например, рис. 18.3 демонстрирует граватар одной из наших тестовых учетных записей.

Рис. 18.2. Просмотр своего сообщения чата

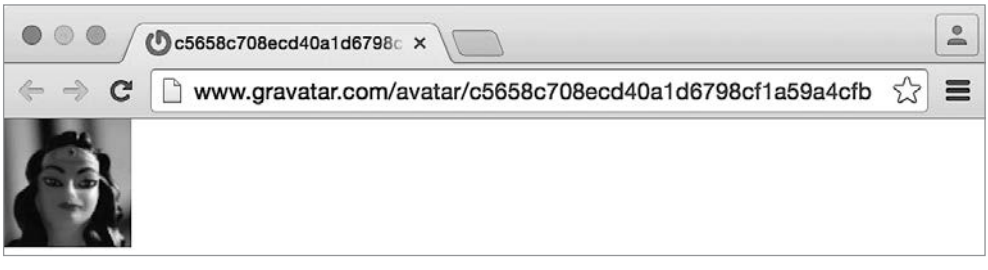


Рис. 18.3. Пример граватара

Обратили внимание на концовку URL? Это уникальный идентификатор, сгенерированный из адреса электронной почты пользователя. Этот идентификатор, называемый *хешем*, можно легко сгенерировать с помощью сторонней библиотеки `crypto-js`.

Добавьте библиотеку `crypto-js` в наш проект с помощью системы управления пакетами `npm`:

```
npm install --save-dev crypto-js
```

Библиотека `crypto-js` теперь установлена в локальный каталог `node_modules` проекта и готова к использованию.

При создании строк в JavaScript часто требуется конкатенировать строку с каким-то другим значением. ES6 предоставляет более удобный способ создания строк, включающих значения из переменных и выражений, называемый *шаблонными строками*. Мы воспользуемся ими для создания URL, чтобы получить доступ к изображениям Gravatar.

Добавьте в файле `dom.js` еще один оператор `import` для подмодуля `md5` библиотеки `crypto-js`, используя `/` для отделения имени главного модуля от имени подмодуля. После этого напишите функцию `createGravatarUrl`, принимающую на входе имя пользователя, генерирующую MD5-хеш и возвращающую URL граватара:

```
import $ from 'jquery';
import md5 from 'crypto-js/md5';

function createGravatarUrl(username) {
  let userhash = md5(username);
  return `http://www.gravatar.com/avatar/${userhash.toString()}`;
}
...
```

Обратите внимание: это не одинарные кавычки в строке `return `http://www.gravatar.com/avatar/${userhash.toString()}`;`. Это обратные апострофы, располагающиеся в большинстве американских раскладок клавиатур под клавишей Esc.

Внутри обратных апострофов можно использовать синтаксис `${userhash.toString()}` для включения значений JavaScript-выражений прямо в строку. В данном случае мы ссылаемся на переменную `userhash` и вызываем ее метод `toString`, но внутри фигурных скобок будет корректно любое выражение.

Воспользуемся этой функцией для отображения граватаров в новых сообщениях. Внизу метода `drawMessage` модуля `chatList` (по-прежнему в файле `dom.js`) создайте новый элемент для изображения и установите значение его атрибута `src` равным граватару пользователя:

```
...
    $message.append($('<span>', {
      class: 'message-message',
      text: m     }));

    let $img = $('<img>', {
      src: createGravatarUrl(u),
      title: u     });

    $messageRow.append($img);
    $messageRow.append($message);
    $(this.listId).append($messageRow);
    $messageRow.get(0).scrollIntoView();
...
```

Запустите приложение чата — и вы увидите, что теперь на экране высвечивается граватар (рис. 18.4).

К сожалению, для пользователя с именем `wonderwoman` нет соответствующего граватара. В результате вы видите унылый граватар по умолчанию.

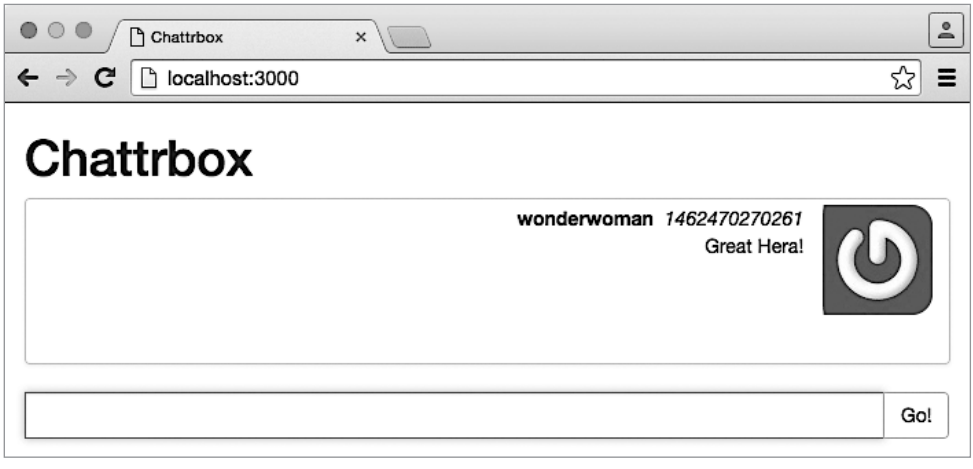


Рис. 18.4. Отображение граватара

Приглашение ввести имя пользователя

Быть Чудо-Женщиной[1] действительно круто. Но быть использующим приложение Chattrbox JavaScript-разработчиком — еще круче (особенно с учетом того, что у настоящих его пользователей уже есть граватары). Чтобы знать, кто работает с Chattrbox, необходимо запрашивать у пользователей их имена.

Взаимодействовать с пользовательским интерфейсом — обязанность модуля `dom`, так что создайте функцию `promptForUsername` в файле `dom.js`. Вместо того чтобы делать ее частью классов `ChatForm` или `ChatList`, добавьте ее в `exports`.

```
...
function createGravatarUrl(username) {
  let userhash = md5(username);
  return `http://www.gravatar.com/avatar/${userhash.toString()}`;
}

export function promptForUsername() {
  let username = prompt('Enter a username');
  return username.toLowerCase();
}
...
```

Создайте в функции `promptForUsername` переменную `let` для хранения введенного пользователем текста (функция `prompt` встроена в браузер и возвращает строку). В дальнейшем мы будем возвращать преобразованную в нижний регистр версию этого текста.

[1] Вымышленный персонаж комиксов DC Comics. — *Примеч. пер.*

Чтобы использовать эту новую функцию, вам нужно изменить файл `app.js`. Исправьте оператор `import` для модуля `dom` и вызовите функцию `promptForUsername` для получения значения переменной `username`:

```
import socket from './ws-client';
import {ChatForm, ChatList, promptForUsername} from './dom';

const FORM_SELECTOR = '[data-chat="chat-form"]';
const INPUT_SELECTOR = '[data-chat="message-input"]';
const LIST_SELECTOR = '[data-chat="message-list"]';

let username = '';
username = promptForUsername();

class ChatApp {
  ...
```

Теперь измените класс `ChatMessage`, чтобы использовать это имя пользователя в качестве значения по умолчанию. Помните: только у полученных от сервера сообщений имеется значение `data.user`.

```
...
class ChatMessage {
  constructor({
    message: m,
    user: u='batman', username,
    timestamp: t=(new Date()).getTime()
...
```

Наконец, передайте имя пользователя конструктору класса `ChatList`:

```
...
class ChatApp {
  constructor() {
    this.chatForm = new ChatForm(FORM_SELECTOR, INPUT_SELECTOR);
    this.chatList = new ChatList(LIST_SELECTOR, 'wonderwoman' username);
    ...
```

После окончания процесса сборки обновите страницу в браузере и введите имя пользователя в ответ на приглашение (рис. 18.5).

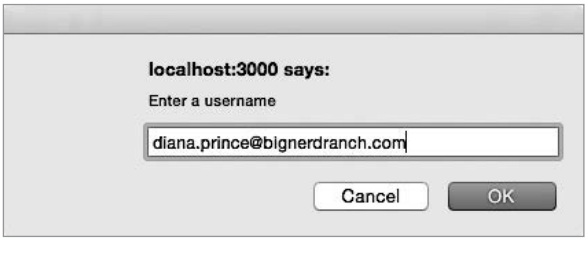


Рис. 18.5. Приглашение ввести имя пользователя

Теперь попробуйте отправить несколько сообщений. Вы должны увидеть, что выбранное вами имя пользователя сразу же отображается на экране, как и связанный с этим именем граватар (рис. 18.6).

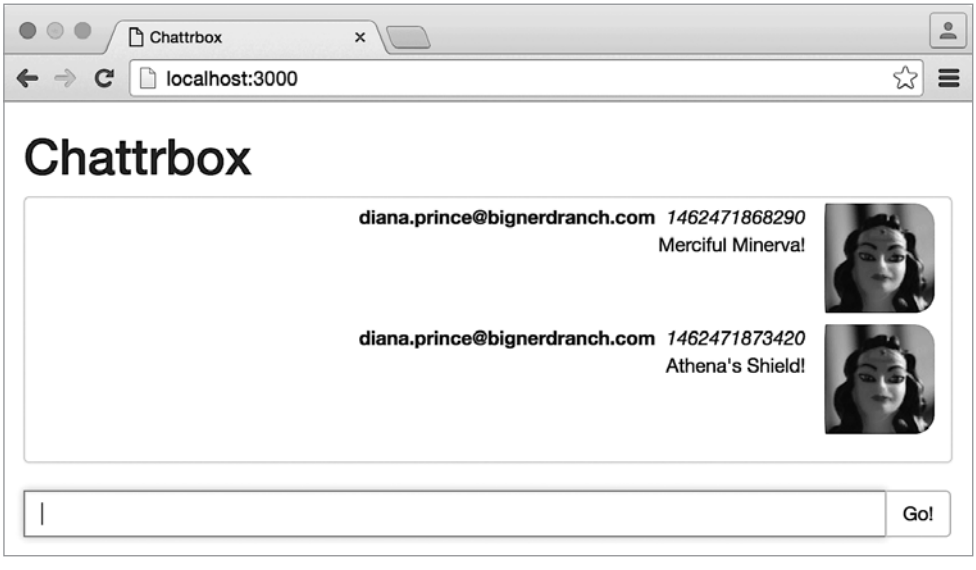


Рис. 18.6. Ваше имя пользователя

Граватары присваиваются на основе адресов электронной почты. Если у вас нет связанного с вашей электронной почтой граватара, воспользуйтесь diana.prince@bignerdranch.com или clark.kent@bignerdranch.com.

Сеансовое хранилище пользователя

Вводить имя пользователя при каждом обновлении страницы быстро надоедает. Лучше было бы хранить это имя пользователя в браузере. Для создания простого хранилища браузер предоставляет два API, предназначенных для хранения пар «ключ/значение» (с одним ограничением: значение должно быть строчным), — `localStorage` и `sessionStorage`. Данные, содержащиеся в `localStorage` и `sessionStorage`, связываются с адресом сервера веб-приложения. У кода с различных сайтов нет доступа к данным друг друга.

Мы могли бы воспользоваться `localStorage`, но нам может понадобиться хранить имя пользователя только до момента закрытия вкладки браузера или окна, так что выберем `sessionStorage`, который полностью аналогичен `localStorage`, но данные в нем удаляются при завершении сеанса (путем закрытия вкладки браузера или окна).

Сформируем новый набор классов для работы с нашей информацией из API `sessionStorage`.

Создайте в каталоге `app/scripts/src` новый файл `storage.js` и вставьте в него описание класса:

```
class Store {
  constructor(storageApi) {
    this.api = storageApi;
  }
  get() {
    return this.api.getItem(this.key);
  }

  set(value) {
    this.api.setItem(this.key, value);
  }
}
```

Наш новый класс `Store` — обобщенный, его можно использовать как с API `localStorage`, так и API `sessionStorage`. Он представляет собой «тонкую» обертку для интерфейсов программирования приложений Web Storage. При создании экземпляра мы указываем, какой именно API хранения мы хотели бы использовать.

Обратите внимание, что в вышеприведенном коде имеются ссылки на свойство `this.key`, значение которому в конструкторе не присваивается. Данная реализация класса `Store` не предназначена для использования сама по себе, она должна применяться путем создания подкласса, в котором определяется свойство `key`.

Создайте с помощью ключевого слова `extends` подкласс, который можно было бы затем использовать для хранения имени пользователя в `sessionStorage`:

```
class Store {
  constructor(storageApi) {
    this.api = storageApi;
  }
  get() {
    return this.api.getItem(this.key);
  }

  set(value) {
    this.api.setItem(this.key, value);
  }
}

export class UserStore extends Store {
  constructor(key) {
    super(sessionStorage);
    this.key = key;
  }
}
```

Класс `UserStore` лишь задает конструктор, выполняющий два действия. Во-первых, вызывает метод `super`, который, в свою очередь, вызывает конструктор класса `Store`, передавая ему ссылку на `sessionStorage`. Во-вторых, задает значение свойства `this.key`.

Теперь для `Store` задано значение свойства `api`, а для экземпляра `UserStore` — значение свойства `key`. Это значит, что все готово для вызова экземпляром `UserStore` методов `get` и `set`.

Класс `UserStore` будет использоваться в файле `app.js`, так что необходимо его экспортировать.

Теперь пора задействовать наш новый класс `UserStore`. Импортируйте `UserStore` в файле `app.js`, создайте экземпляр и воспользуйтесь им для сохранения имени пользователя:

```
import socket from './ws-client';
import {UserStore} from './storage';
import {ChatForm, ChatList, promptForUsername} from './dom';

const FORM_SELECTOR = '[data-chat="chat-form"]';
const INPUT_SELECTOR = '[data-chat="message-input"]';
const LIST_SELECTOR = '[data-chat="message-list"]';

let username = '';
let userStore = new UserStore('x-chattrbox/u');
let username = userStore.get();
if (!username) {
  username = promptForUsername();
  userStore.set(username);
}

class ChatApp {
  ...
```

Запустите приложение Chattrbox еще раз в браузере. На этот раз имя пользователя будет запрошено у вас только один раз — при первой загрузке страницы. При последующих обновлениях страницы должно использоваться введенное вами имя пользователя.

Чтобы убедиться в том, что ваше имя пользователя было сохранено в `sessionStorage`, можете воспользоваться панелью ресурсов в DevTools. Щелчком активизируйте панель ресурсов — и слева увидите список. Щелкнув на ▶ рядом с пунктом Session Storage в списке, разверните адрес http://localhost:3000. Щелкните на этом URL для разворачивания данных, хранящихся в `UserStore` (рис. 18.7).

Внизу этого списка пар «ключ/значение» находятся кнопки для обновления списка и удаления из него элементов. Их можно использовать, когда необходимо вручную поменять сохраненные данные.

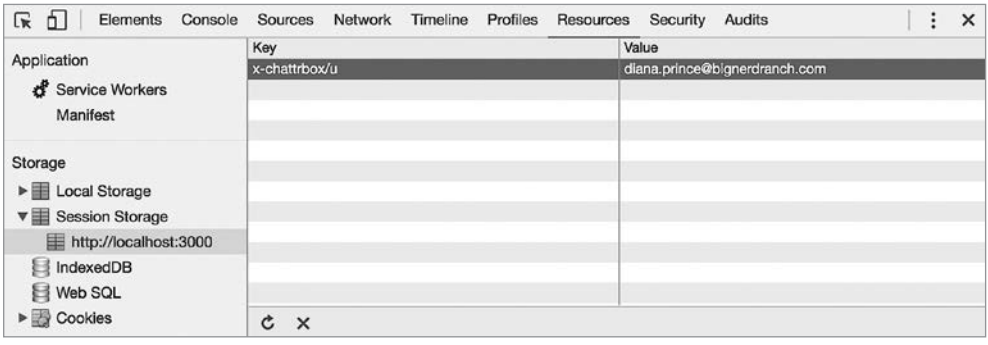


Рис. 18.7. Панель ресурсов в DevTools

Форматирование и изменение меток даты/времени в сообщениях

Метки даты/времени в наших сообщениях пока что не очень приятны для глаз пользователей (серьезно, кому придет в голову называть время в миллисекундах, прошедших с 1 января 1970 года?). Чтобы сделать их более удобными (например, «10 минут назад»), добавим модуль `moment`. Установите его с помощью npm и сохраните в виде зависимости, используемой для разработки:

```
npm install --save-dev moment
```

Все наши сообщения хранят метки даты/времени в виде атрибутов данных. Напишите метод `init` для класса `ChatList`, который бы вызывал встроенную функцию `setInterval`, принимающую на входе два аргумента: запускаемую функцию и частоту ее запуска. Эта функция будет обновлять сообщения, заменяя метки даты/времени на удобные для чтения.

Чтобы задать строку метки даты/времени, воспользуемся в модуле `dom` библиотекой jQuery для поиска всех элементом с атрибутом `data-time`, чье значение представляет собой числовую метку даты/времени. Создайте новый объект `Date` на основе этой числовой метки даты/времени и передайте его методу `moment`. Затем вызовите метод `fromNow` для получения итоговой строки метки даты/времени и сделайте ее элементом текста HTML.

```
...
  drawMessage({user: u, timestamp: t, message: m}) {
    ...
  }

  init() {
    this.timer = setInterval(() => {
      $('[data-time]').each((idx, element) => {
        let $element = $(element);
```

```
        let timestamp = new Date().setTime($element.attr('data-time'));
        let ago = moment(timestamp).fromNow();
        $element.html(ago);
      });
    }, 1000);
  }
}
```

Мы запускаем эту функцию каждые 1000 миллисекунд. Чтобы обеспечить немедленное отображение удобной для чтения метки даты/времени, исправьте метод `drawMessage`. Воспользуйтесь методом `moment` для создания отформатированной строки метки даты/времени при первой отрисовке сообщения в списке чата.

```
...
  drawMessage({user: u, timestamp: t, message: m}) {
    ...
    $message.append($('<span>', {
      'class': 'timestamp',
      'data-time': t,
      text: (new Date(t)).getTime()
      text: moment(t).fromNow()
    }));
    ...
```

Наконец, измените файл `app.js`, добавив обращение к методу `this.chatList.init` в обратном вызове функции `socket.registerOpenHandler`:

```
...
class ChatApp {
  constructor () {
    this.chatForm = new ChatForm(FORM_SELECTOR, INPUT_SELECTOR);
    this.chatList = new ChatList(LIST_SELECTOR, username);

    socket.init('ws://localhost:3001');
    socket.registerOpenHandler(() => {
      this.chatForm.init((text) => {         let message = new
ChatMessage({message: text});
        socket.sendMessage(message.serialize());
      });
      this.chatList.init();
    });
    ...
```

Сохраните изменения и дайте возможность сценариям npm скомпилировать их. Обновите страницу в браузере и начните переписку. Вы увидите, как обновленные метки даты/времени появляются вместе с набираемым текстом. Спустя пару минут вы увидите, как метки даты/времени ваших сообщений обновляются (рис. 18.8).

Мы завершаем работу над приложением Chattrbox. Хотя она заняла всего несколько глав, сделано немало. Мы научились создавать два вида серверов в Node.js — простые веб-серверы и серверы WebSockets. Мы создали с помощью ES6 клиентское при-

ложение, применяя утилиты Babel и Browserify для компиляции кода в ES5, чтобы можно было использовать Chattrbox в устаревших браузерах, а также автоматизировали процесс разработки с помощью сценариев системы управления пакетами npm.

Рис. 18.8. Не слишком тайные личности

Приложение Chattrbox — вершина изученных нами пока что технологий. В следующем приложении, Tracker, мы познакомимся с Ember.js — фреймворком для создания масштабных приложений. Он будет основываться на наших добытых нелегким трудом знаниях модульной разработки, асинхронного программирования и инструментов разработки.

Бронзовое упражнение: добавление в сообщения визуальных эффектов

Добавьте в новые сообщения визуальные эффекты. Можете их постепенно усиливать или ослаблять (обратитесь к документации по эффектам библиотеки jQuery, чтобы узнать, какие существуют возможности).

В качестве дополнительного упражнения применяйте эти эффекты только к новым сообщениям, а не к тем, которые уже имеются в чате и загружаются приложением при первом входе пользователей в приложение или обновлении страницы браузера.

Как можно отличить старые сообщения от новых? У каждого сообщения есть атрибут даты, по которому вы можете определить, старше ли оно, чем секунда-другая.

Серебряное упражнение: кэширование сообщений

Если вы в процессе переписки захотите обновить страницу браузера, то все ваши сообщения исчезнут. То, что экземпляр `UserStore` помнит ваше имя пользователя, конечно, чудесно, но было бы еще лучше, если бы у вас был аналогичный механизм для кэширования сообщений чата.

Создайте наследующий `Store` класс `MessageStore`. Он должен сохранять сообщения по мере их поступления, причем один раз.

При загрузке страницы приложение Chattrbox должно извлекать все закешированные сообщения из экземпляра `MessageStore`. Подумайте, хотите ли вы, чтобы сообщения сохранялись даже при закрытии и повторном открытии вкладки браузера (если да, то какую альтернативу API `sessionStorage` вы должны использовать?).

Золотое упражнение: отдельные комнаты чата

Это упражнение потребует изменения как серверного, так и клиентского приложения.

Предоставьте вашим пользователям возможность открывать отдельные комнаты чата. При вводе ими имени пользователя предлагайте им указать имя комнаты чата, в которой они хотели бы общаться.

При входе в комнату чата пользователи должны получать через соединение по протоколу WebSockets только предназначенные для этой комнаты сообщения. Вам может понадобиться поменять способ хранения сообщений на сервере или способ отправки сообщений клиенту, а возможно, и то и другое.

В качестве дополнительного упражнения добавьте отображение выпадающего списка доступных комнат чата в пользовательском интерфейсе клиентского приложения, чтобы пользователи могли переключаться с одной комнаты на другую. При переходе в другую комнату обеспечьте получение с сервера и отображение в списке чата всех новых сообщений.

Часть IV. Архитектура приложения

19 Введение в MVC и Ember

«Модель — представление — контроллер» (Model — View — Controller, MVC) — исключительно удобный паттерн проектирования программного обеспечения. Он отлично работает для веб-приложений, позволяя создавать структуру по слоям. Данная глава познакомит вас с паттерном MVC. Вы освоите установку и настройку Ember — фреймворка для MVC. Следующие несколько глав посвящены отдельным частям этого паттерна, демонстрируемым по мере того, как мы будем создавать приложение слой за слоем.

Существует множество реализаций MVC, особенно в сфере клиентской части. Рисунок 19.1 демонстрирует, какую его трактовку будем использовать мы.

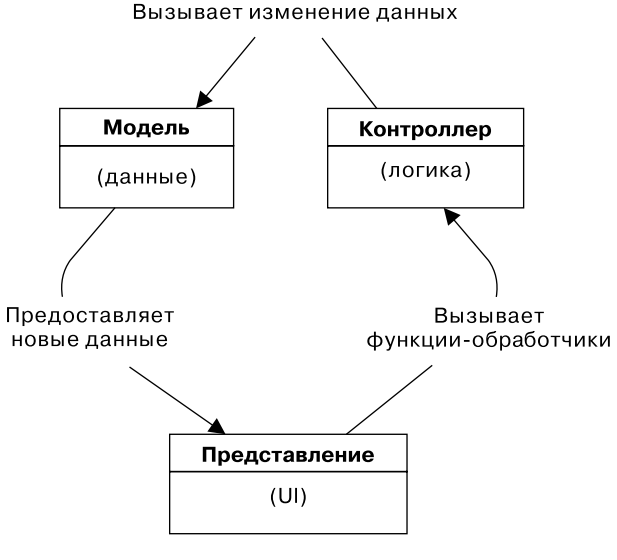


Рис. 19.1. Паттерн «Модель — представление — контроллер»

Вот действия, выполняемые каждым из слоев:

- *модели* отвечают за данные. При изменении данных модель сообщает об этом всем прослушивателям;
- *представления* отвечают за пользовательский интерфейс. Они занимаются представлением моделей и прослушивают на предмет изменений. При генерации событий UI в ответ на вводимые пользователем данные они вызывают в контроллере функции-обработчики;
- в *контроллерах* содержится логика приложения. Они извлекают экземпляры моделей и передают их представлениям. В них также содержатся функции-обработчики, выполняющие изменения в экземплярах моделей.

Этот процесс создает впечатление кругового взаимодействия. К слову, так оно и есть. Эти три части работают совместно. Данные приложения перемещаются от моделей к представлениям. Данные событий перемещаются от представлений к контроллерам. Контроллеры вызывают изменение данных в модели в соответствии с событиями пользовательского интерфейса.

Наверное, вам интересно, с чего мы начнем работу над этим круговым паттерном MVC. В главе 8 мы создали приложение CoffeeRun и заключили всю необходимую функциональность в объект `Window.App`. У каждого добавленного модуля — своя роль в приложении и название, соответствующее его функциональности. Для паттерна MVC необходима функция начальной настройки, аналогичная созданию нового экземпляра `Truck`, для загрузки контроллеров. Контроллеры, в свою очередь, загрузят модели и представления.

Наше следующее приложение — Tracker — будет загружать начальное состояние DOM в файл HTML в виде просто пустого тега `<body>`. Сценарии для инициализации нашего приложения будут также загружаться из этого HTML-файла. В паттерне MVC представления (контент HTML) динамически визуализируются в зависимости от маршрута и состояния данных (моделей).

Приложению, которое мы будем создавать, потребуется больше семи модулей, имевшихся у приложения CoffeeRun. Паттерн MVC помогает разбивать модули по соответствующим функциональности файлам и поддерживать согласованность независимо от того, десять у вас модулей или сто.

Tracker

Наше приложение Tracker будет включать *URL — маршрутизацию* — одну из лучших возможностей веб-приложений. У него будут модели для описания данных, контроллеры для обработки действий пользователя, шаблоны для описания пользовательского интерфейса и маршруты для установки соответствия моделей шаблонам. По мере создания приложения мы научимся новым паттернам и методам, которые сделают наш код проще и элегантнее.

Заказчик приложения Tracker — криптозоолог, путешествующий по миру в поисках существ вроде снежного человека, лохнесского чудовища и единорогов. Ему требуется приложение для отслеживания этих загадочных существ и сохранения информации о любых случаях их визуального наблюдения. Требования могут меняться (обычно на практике так и происходит), но начнем с того, что у пользователя должна быть возможность:

- вывести список имеющихся случаев наблюдений;
- добавить новые случаи наблюдения;
- связать существ с наблюдениями;
- получать информацию о свежих случаях наблюдения в мгновенных сообщениях.

У каждого наблюдения должны быть следующие данные модели и взаимосвязи.

Атрибут модели наблюдения	**Тип атрибута**
Дата наблюдения существа	Объект даты
Место	Строка
Существо	Ключ модели существа
Очевидец (-цы)	Массив ключей очевидцев

У каждого наблюдения должны быть следующие данные модели.

Атрибут модели существа	**Тип атрибута**
Название	Строка
Тип	Строка
Путь к изображению	Строка

И у каждого свидетеля должны быть следующие данные модели и взаимосвязи.

Атрибуты модели очевидца	**Тип атрибута**
Имя	Строка
Фамилия	Строка
Полное имя	Строка: конкатенация имени и фамилии
Адрес электронной почты	Строка
Наблюдение (-я)	Массив ключей наблюдений

Создание Tracker будет немного отличаться от создания предыдущих приложений. Оно будет больше напоминать создание реального приложения: станет больше

кода в разделах и меньше немедленной обратной связи. Зато вы по-настоящему поучаствуете в разработке и создадите достаточно сложное приложение (рис. 19.2).

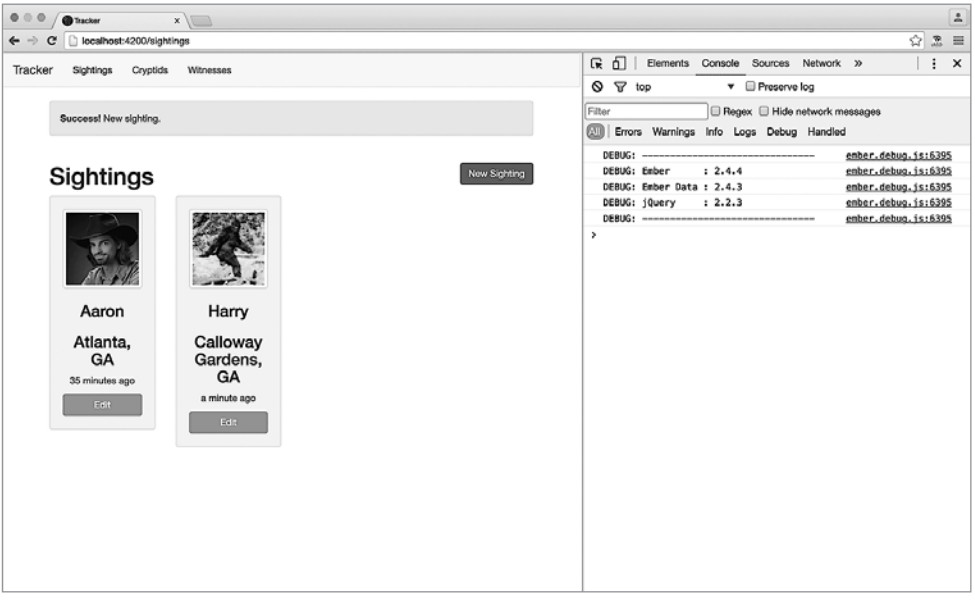


Рис. 19.2. Завершенное приложение Tracker

Ember: MVC-фреймворк

Создавая Tracker, мы изучим простой паттерн разработки веб-приложений с помощью Ember — одного из ведущих MVC-фреймворков. Ember включает понятия и соглашения по именам, обеспечивающие быструю разработку. Мы изучим основы Ember в процессе создания приложения.

Как написано на домашней странице Ember (emberjs.com), Ember — это «фреймворк для создания многообещающих веб-приложений». В отличие от таких библиотек, как jQuery, подобные Ember фреймворки оживляют структуру приложения и часто включают в себя *инструменты для скаффолдинга*, представляющие собой сценарии для создания шаблонных файлов в нужных каталогах. Начиная с 2011 года сообщество Ember создало разветвленную систему библиотек и инструментария для ускорения разработки.

Мы начнем знакомство с фреймворком Ember с Ember CLI — утилиты, предназначенной для скаффолдинга, разработки, тестирования и сборки. Если вам незнаком термин *CLI*, то он расшифровывается как «интерфейс командной строки». Мы создадим новый проект, загрузим зависимости, сгенерируем объекты Ember, выполним сборку и запустим приложение Tracker из этой утилиты.

Установка Ember

Для начала нам понадобится установить некоторые инструменты.

Во-первых, убедитесь, что используете последнюю версию платформы Node.js (выше 0.12.0). Проверить это можно с помощью команды терминала: `node --version`. На момент написания данной главы текущая версия Node — 5.5.0 (да, это довольно сильно отличается от минимальной требуемой версии 0.12.0. Прочитать подробнее о том, как и почему Node перепрыгнул от версии 0.12.0 к 4.0.0, можно в статье «Википедии» по адресу ru.wikipedia.org/wiki/Node.js).

При необходимости скачайте обновленную версию платформы Node.js с nodejs.org.

После обновления Node можно установить утилиту Ember CLI с помощью следующей команды терминала:

```
npm install -g ember-cli@2.4
```

Установка может занять несколько минут. Если вы получили ошибку Please try running this command again as root/Administrator (Пожалуйста, запустите эту команду повторно с правами администратора), значит, существует проблема с правами владельца. **Не запускайте повторно** установку с помощью команды `sudo`, так как команды `npm` и `sudo` плохо сочетаются. Вместо этого используйте следующую команду: `sudo chown -R $USER /usr/local`. Она выполнит повторно исходные команды установки (без `sudo`).

Вам могут встретиться и другие ошибки при установке утилиты Ember CLI из-за несовместимости с вашей системой. Большинство ошибок содержат инструкции по исправлению процесса установки. Некоторые ошибки установки могут потребовать несложного поиска в Интернете и обновления запущенных на вашем компьютере программ. За дополнительной информацией обратитесь к имеющейся на сайте Ember CLI странице, посвященной общим вопросам (находится по адресу ember-cli.com/user-guide/#commonissues).

Далее установите Bower — еще одну полезную утилиту для администрирования:

```
npm install -g bower
```

Bower и система управления пакетами npm необходимы для создания приложения Ember.

Следом установите Ember Inspector — плагин для Chrome. Для этого откройте Chrome и введите в адресной строке `chrome://extensions/`. Внизу страницы расширений Chrome щелкните на команде Get more extensions (Дополнительные расширения). Выполните поиск по запросу `Ember Inspector` (рис. 19.3), нажмите кнопку Add to Chrome (Добавить в Chrome) и следуйте дальнейшим указаниям для его установки.

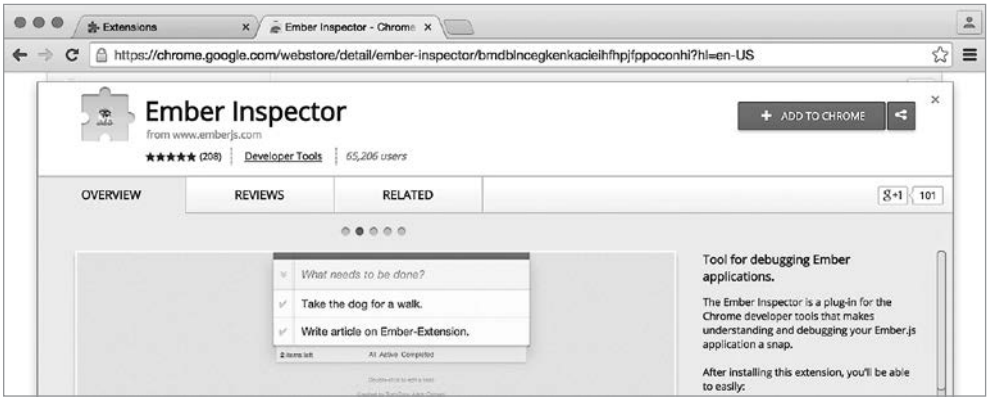


Рис. 19.3. Установка расширения для Chrome Ember Inspector

Утилита Ember CLI в ходе работы использует программу Watchman. Watchman — утилита командной строки, интегрирующаяся с браузерами и предоставляющая возможность горячей перезагрузки приложений.

Установить Watchman на Mac можно с помощью Homebrew. Homebrew (система управления пакетами для OS X) вы можете скачать с помощью команды терминала (скопируйте ее с сайта brew.sh). После установки Homebrew установите Watchman (версия 3.0.0 или выше) с помощью следующей команды терминала:

```
brew install watchman
```

Инструкции по установке Watchman для Windows можно найти на сайте facebook.github.io/watchman/docs/install.html.

Теперь у нас в наличии все инструменты, необходимые для начала работы над нашим проектом Ember — приложением Tracker.

Создание приложения Ember

Акцент, сделанный Ember на условных соглашениях и паттернах, позволяет создавать приложения при написании минимума кода. Этот фреймворк выполняет огромное количество работы за кулисами, генерируя множество объектов и событий при запуске приложения. В дальнейшем при работе над приложением Tracker мы будем использовать собственные объекты вместо созданных для нас фреймворком Ember.

Перейдите в терминале к нашему каталогу проектов. Команда `ember new [`*`название проекта`*`]` создаст каталог и основные файлы, необходимые для начала разработки.

Создайте новое приложение Ember под названием tracker:

```
ember new tracker
```

Создание нового приложения Ember может занять несколько минут. Как видно из выводимой в терминале информации (часть ее приведена ниже), команда `ember new` создает основные файлы проекта и структуру каталогов и использует утилиты npm и Bower для загрузки внешних библиотечных ресурсов. Эти библиотеки необходимы для запуска приложения Ember, а также для компиляции, сборки и тестирования приложения.

```
installing app
  create .bowerrc
  create .editorconfig
  create .ember-cli
  create .jshintrc
  create .travis.yml
  create .watchmanconfig
  create README.md
  create app/app.js
  create app/components/.gitkeep
  create app/controllers/.gitkeep
  create app/helpers/.gitkeep
  create app/index.html
  create app/models/.gitkeep
  create app/router.js
  create app/routes/.gitkeep
  create app/styles/app.css
  create app/templates/application.hbs
  . . .
Successfully initialized git.
Installed packages for tooling via npm.
Installed browser packages via Bower.
```

После того как Ember завершит настройку приложения Tracker, проверим, что все работает, запустив локальный сервер.

Запуск сервера

Чуть ниже мы воспользуемся командой `ember server` (или `ember s`, если хотите сократить количество нажатий клавиш) для сборки приложения и запуска локально доступного сервера. Команда `ember server` отслеживает изменения в файлах и перезапускает процесс сборки/выдачи/отслеживания, чтобы гарантировать, что в браузере вам видна самая свежая версия кода (аналогично утилите browser-sync, которую мы использовали в Ottergram и CoffeeRun).

Ember CLI использует для компиляции программу Broccoli. Если вам доводилось программировать на таких языках, как Java или Objective-C, то под словом «компиляция» вы можете понимать нечто иное, чем то, что принято в JavaScript. В данном случае Broccoli объединяет все необходимые для работы приложения файлы, обеспечивая в то же время все нужные зависимости.

Пришло время *побороться за пользователя*. Перейдите в каталог Tracker и запустите сервер:

```
cd tracker
ember server
```

В Chrome откройте новое окно браузера и перейдите по адресу http://localhost:4200, чтобы посмотреть наше новое приложение Ember в действии (рис. 19.4). Необходимо также открыть вкладку Ember в DevTools, как показано на рисунке.

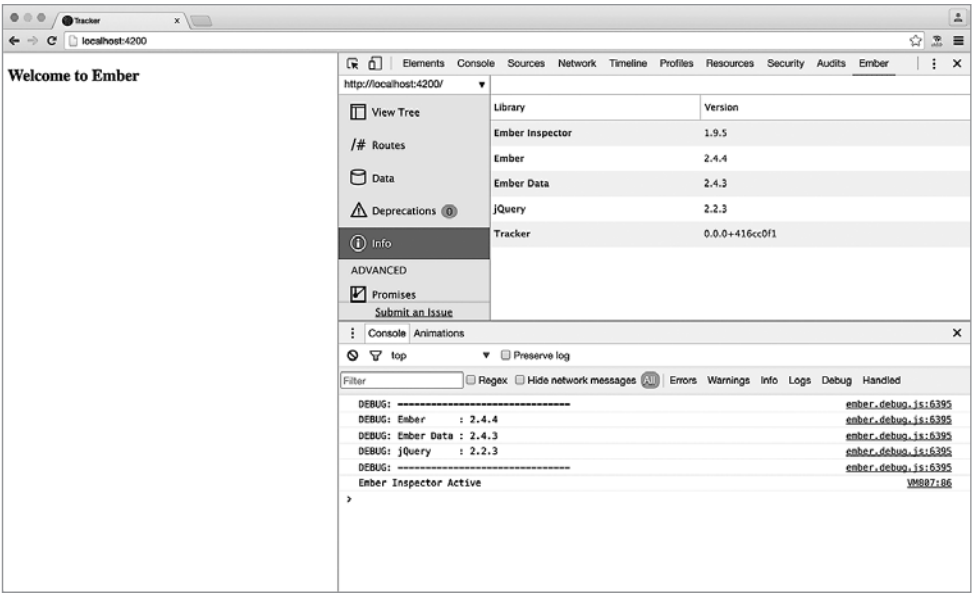


Рис. 19.4. Сервер Ember

Как уже упоминалось выше, утилита Ember CLI обновляет страницу браузера при внесении изменений в файлы приложения. Эта возможность носит название Livereload (горячая перезагрузка), и вы можете увидеть, что она упоминается среди выводимой в терминал информации как:

```
Livereload server on http://localhost:49152
```

Обратите внимание на рис. 19.4, что консоль и Ember Inspector выводят различные сгенерированные компоненты, а также их версии. Для данной книги мы использовали версии 2.4 (и Ember, и Ember Data). На момент публикации Ember CLI генерирует приложение Ember 2.x (это прослеживается по номерам версий на рисунке). Если вы видите номера версий, начинающиеся с 1.x.x, то, вероятно, вы пропустили этап установки или обновления Ember-CLI.

Обратите внимание, что после запуска сервера в терминале вы видите версию (Version) Ember CLI, а не самого запускаемого вами приложения Ember.

Внешние библиотеки и дополнения

Утилита Ember CLI нацелена на ускорение работы разработчиков различными способами, включая добавление кода из сообщества свободно распространяемого ПО. В предыдущих главах мы добавляли в локальную среду модули node с помощью системы управления пакетами npm. Ранее в данной главе мы обсуждали загрузку внешних библиотек через утилиту Bower (еще одна система управления пакетами).

Ember CLI отлично работает с обеими этими системами управления пакетами. Библиотеки и инструменты можно установить с помощью простых команд, например:

```
npm install [название пакета] --save-dev
npm install [название пакета] --save
bower install [название пакета] --save
```

При работе с внешними библиотеками эти инструменты командной строки загружают файлы в каталоги `bower_components` и `node_modules`.

Мы использовали для приложения CoffeeRun фреймворк Bootstrap, а теперь запустим и Tracker, задействовав этот фреймворк. Для добавления Bootstrap с помощью утилиты Bower введите следующую команду в терминале:

```
bower install bootstrap-sass --save
```

Мы загрузили библиотеку Bootstrap локально — со всеми ее файлами JavaScript и стилями. Теперь мы включим эту библиотеку в процесс сборки Ember CLI, чтобы обеспечить наше приложение необходимыми ресурсами Bootstrap.

Современный процесс разработки сценариев и стилей для веб-приложений включает компиляцию. Мы собираемся добавить утилиту, которая поможет нам упростить компиляцию: ember-cli-sass будет заниматься преобразованием таблиц стилей SCSS в CSS во время выполняемого утилитой Ember CLI процесса сборки. SCSS, часто называемый Sass, добавляет в таблицы стилей многие хорошо вам знакомые программные конструкции, такие как переменные, функции, циклы и пары «ключ/значение», без потери синтаксиса CSS.

Установите ember-cli-sass через терминал:

```
ember install ember-cli-sass
```

Утилита ember-cli-sass — пример дополнения Ember. Дополнения (www.emberaddons.com) — это проекты, добавляющие внешние библиотеки или конфигурационный код, создающие вспомогательные функции или компоненты либо решающие за вас какие-то еще сложные задачи. Ember CLI предоставляет возможность удобного добавления имеющихся проектов в наш с помощью команды ember install.

Обратите внимание, что Ember CLI — относительно новая утилита, поэтому дополнения могут оказаться несинхронизированными. Если вы обнаружили проблемы

с конкретным дополнением, посетите страницу, посвященную сложным вопросам, в GitHub-репозитории этого дополнения.

Мы только что добавили возможность компиляции файлов SCSS. Теперь нам нужно сделать файл `app/styles/app.css` файлом `.scss`. Переименуйте файл `app/styles/app.css` в `app/styles/app.scss`. После этого перезапустите сервер Ember для инициализации новых утилит CLI.

Для проверки новой утилиты CLI добавим в таблицу стилей переменную SCSS. Создайте в файле `app/styles/app.scss` пару «ключ/значение», указав перед ней `$`, для проверки компиляции SCSS:

```
$bg-color: coral;
html {
  background: $bg-color;
}
```

Загляните в браузер. У нашей страницы теперь появился цвет фона (рис. 19.5).

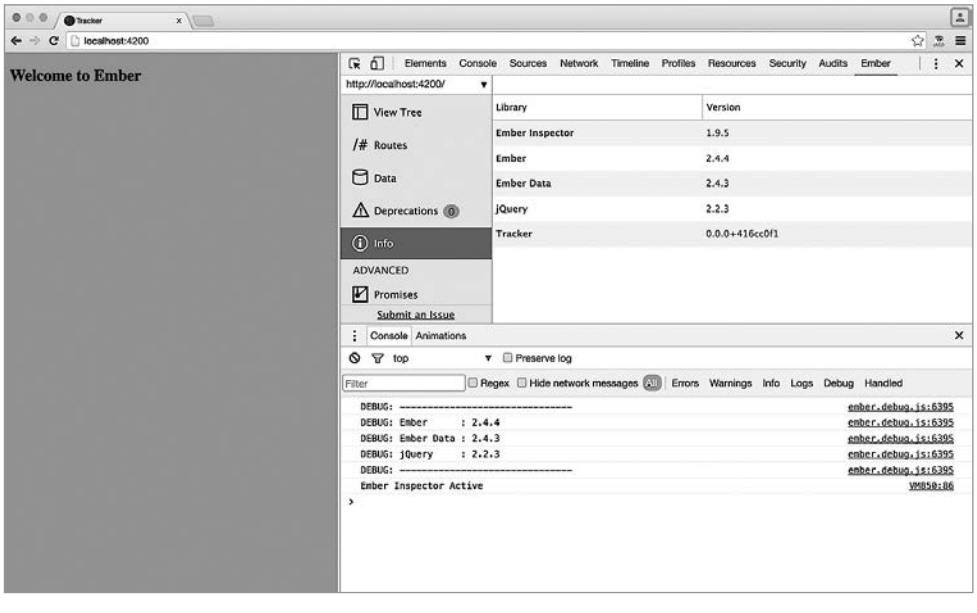


Рис. 19.5. Компиляция SCSS: проверка

Далее нам предстоит внести в наш проект стили и сценарии Bootstrap. Ранее мы добавили SCSS-версию Bootstrap с помощью команды `bower install bootstrap-sass`. Чтобы добавить эту библиотеку в нашу таблицу стилей, понадобится импортировать библиотеку стилей в файл и сконфигурировать утилиту Ember CLI для сборки приложения с этими ресурсами.

Конфигурация

Упомянутому нами ранее механизму компиляции Broccoli требуются дополнительные настройки при добавлении новых ресурсов — JavaScript и таблиц стилей.

Ember CLI генерирует файл конфигурации `ember-cli-build.js`. Именно в этот файл можно внедрить зависимости и именно в нем можно настроить структуру вывода приложения. Для Tracker мы добавим только внешние библиотеки и настройки для компиляции SCSS.

Откройте файл `ember-cli-build.js`, присвойте переменной значение пути к каталогу `bootstrap`, и добавьте каталог `stylesheets` библиотеки Bootstrap в ключ `includePaths` в `sassOptions`:

```
...
var EmberApp = require('ember-cli/lib/broccoli/ember-app');

module.exports = function(defaults) {

  var bootstrapPath = 'bower_components/bootstrap-sass/assets/';

  var app = new EmberApp(defaults, {
    // Добавить здесь параметры
    sassOptions: {
      includePaths: [
        bootstrapPath + 'stylesheets'
      ]
    }
  });

  // ... Комментарии шаблона ...

  // Задайте пути к ресурсам Bootstrap
  // Добавьте ресурсы в приложение с помощью import
  app.import(bootstrapPath + 'javascripts/bootstrap.js');
  return app.toTree();
};
```

Мы добавили в конфигурацию указание для Ember CLI искать файлы `*.scss` в каталоге `bower_components/bootstrap-sass/assets/stylesheets`. Сохраните файл и перезапустите сервер Ember, чтобы новая конфигурация загрузилась вместе с приложением.

Теперь мы можем воспользоваться директивой `@import` в файле `app.scss`, чтобы импортировать стили Bootstrap:

```
$bg-color: coral;
html {
  background: $bg-color;
}

// ---------------------------
```

```
// переопределение переменных bootstrap
// ----------------------------

// конец переопределения переменных bootstrap
@import 'bootstrap';
```

Директива `@import` добавляет содержимое файла `bootstrap.scss` в файл `app.scss`, создаваемый при выполняемом утилитой Ember CLI процессе сборки. Этот файл Bootstrap находится в каталоге `bower_components/bootstrap-sass/assets/stylesheets/`.

Мы добавили в процесс сборки приложения с помощью оператора `app.import(bootstrapPath + 'javascripts/bootstrap.js');` в файле `ember-cli-build.js` JavaScript-компоненты библиотеки Bootstrap. Импорт в конфигурации сборки CLI вносит файл в список ресурсов, которые будут объединяться в единый файл `dist/assets/vendor.js`. В файле `bootstrap.js` библиотеки Bootstrap имеются отдельные JavaScript-модули для «схлопывания» элементов DOM, модальных окон, вкладок, выпадающих меню и многого другого в один файл. Добавлять все JavaScript-компоненты, наверное, излишне, но в будущем мы сможем подправить конфигурацию в файле `ember-cli-build.js`, чтобы выбирать только конкретные необходимые нам компоненты.

После добавления ресурсов желательно, прежде чем двигаться дальше, всегда проверять, работают ли они. В каталоге `app` имеется файл `index.html`, но проверять работу нового кода Bootstrap следует не с его помощью. Этот файл предназначен в основном для процесса сборки.

Вместо этого необходимо добавить все элементы HTML в *шаблоны* приложения каталога `app/templates`. Более подробно мы расскажем о шаблонах в главе 23.

А пока что добавьте в файл `app/templates/application.hbs` компонент `NavBar` библиотеки Bootstrap:

```
<h2 id="title">Welcome to Ember</h2>

{{outlet}}
<header>
  <nav class="navbar navbar-default">
    <div class="container-fluid">
      <!-- Название и переключатели сгруппированы для лучшего отображения
        на мобильных устройствах -->
      <div class="navbar-header">
        <button type="button" class="navbar-toggle collapsed"
          data-toggle="collapse" data-target="#top-navbar-collapse">
          <span class="sr-only">Toggle navigation</span>
          <span class="icon-bar"></span>
          <span class="icon-bar"></span>
          <span class="icon-bar"></span>
        </button>
```

```
        <a class="navbar-brand">Tracker</a>
      </div>

      <!-- Собираем навигационные ссылки, формы и остальной контент
       для включения/отключения -->
      <div class="collapse navbar-collapse" id="top-navbar-collapse">
        <ul class="nav navbar-nav">
          <li>
            <a href="#">Test Link</a>
          </li>
          <li>
            <a href="#">Test Link</a>
          </li>
        </ul>
      </div><!-- /.navbar-collapse -->
    </div><!-- /.container-fluid -->
  </nav>
</header>
<div class="container">
  {{outlet}}
</div>
```

Мы добавили компонент `NavBar` библиотеки Bootstrap с конкретными HTML-атрибутами: идентификаторами, именами классов и атрибутами данных. Мы также переместили существующий `{{outlet}}` внутрь элемента `<div>`. Этот фрагмент кода отвечает за структуру дочерних шаблонов. Мы познакомимся с `{{outlet}}` в следующей главе.

Результат этого кода показан на рис. 19.6.

Компонент `NavBar` адаптивен и отображает кнопку сворачивания при размере окна браузера меньше чем 768 пикселов в ширину. Эта кнопка реагирует на событие щелчка кнопкой мыши путем открывания и закрывания списка ссылок (рис. 19.7).

Отличная новость — мы довели приложение Ember до работоспособного состояния. Мы установили утилиты для генерации кода, компиляции ресурсов, загрузки зависимостей и выдачи приложения. Теперь у нас есть отправная точка для создания в следующих главах оставшейся части нашего приложения.

Для самых любознательных: установка систем управления пакетами npm и Bower

Параметры `--save-dev` и `--save` в конце команд `npm install` и `bower install` добавляют в файл JSON каждой из этих утилит пары «ключ/значение», состоящие из имени библиотеки и ее версии. Для Bower имя файла JSON — `bower.json`, для npm (как вы уже видели в Chattrbox) — `package.json`.

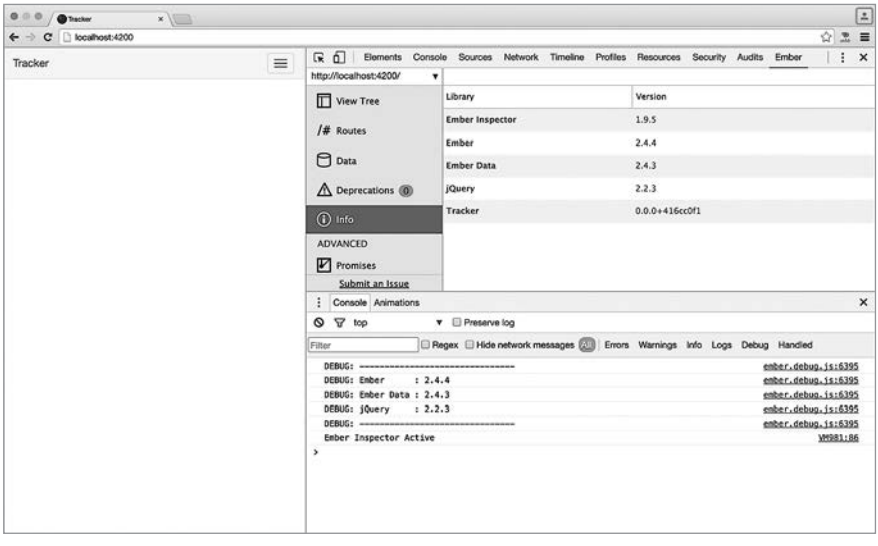


Рис. 19.6. Компонент NavBar библиотеки Bootstrap

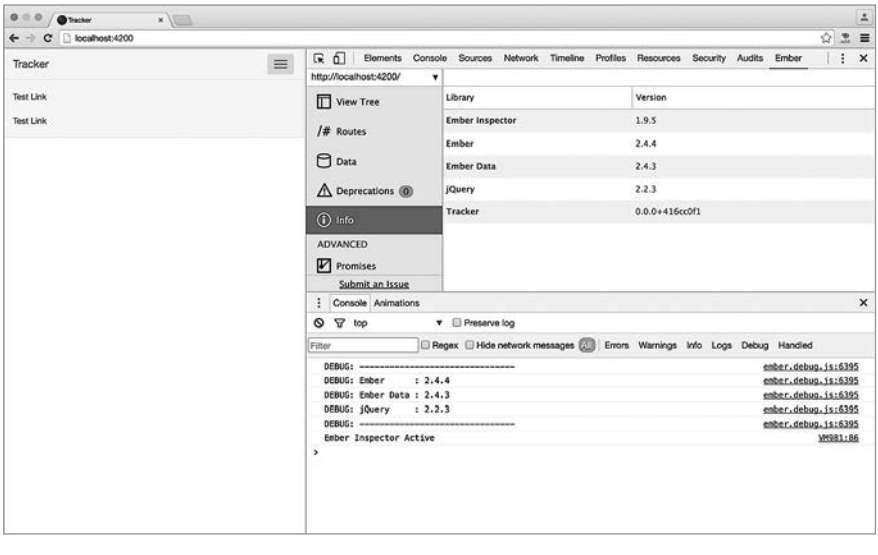


Рис. 19.7. Тестирование сворачивающего подкомпонента компонента NavBar библиотеки Bootstrap

Вот пример добавления в этой главе в файл `bower.json` пар «ключ/значение»:

```
{
  "name": "tracker",
  "dependencies": {
    "ember": "~2.4.3",
```

```
    "ember-cli-shims": "0.1.1",
    "ember-cli-test-loader": "0.2.2",
    "ember-qunit-notifications": "0.1.0",
    "bootstrap-sass": "^3.3.6"
  }
}
```

В файле `bower.json` указаны зависимость `ember.js` и минимальный номер ее версии. В предназначенном для разработки репозитории проекта или системе контроля версий не будут сохраняться перечисленные тут библиотеки и ресурсы, только файл `bower.json`. Разработчик, получив код, сможет выполнить команды `bower install` и `npm install` для создания локальной среды разработки.

Бронзовое упражнение: ограничьте количество импортов

Измените файл `ember-cli-build.js`, чтобы импортировать только файлы `collapse.js` и `transition.js`. После этого файл `vendor.js` уменьшится в размере, но компонент `NavBar` не потеряет работоспособности.

Прежде чем вносить изменения, найдите файл `dist/assets/vendor.js` и запомните количество строк кода (или размер файла). Сравните это число с новым размером.

Серебряное упражнение: добавьте шрифт Awesome

Шрифт Awesome хранится в библиотеке UI, предназначенной для добавления в проект часто используемых значков. Эти значки можно масштабировать, как обычный шрифт. Добавьте Awesome через дополнения Ember CLI, а также значок в файл `app/templates/application.hbs`. Загляните в репозиторий этого дополнения на GitHub за более детальной информацией.

Золотое упражнение: пользовательская настройка NavBar

Фреймворк Bootstrap написан на SCSS и достаточно демократичен в использовании переменных и функций. Применяя SCSS в своем проекте, вы можете управлять процессом компиляции правил оформления библиотекой Bootstrap. Можно даже создавать собственные темы Bootstrap для изменения переменных по умолчанию.

Поменяйте у NavBar значения свойств `background-color`, `border-radius` и `padding`, просто добавив или изменив переменные в файле `app/stylesheets/app.scss`.

20 Маршрутизация, маршруты и модели

На текущий момент у нас есть каркас приложения Tracker. Теперь необходимо решить, какие страницы — или *маршруты* — будет включать наше приложение.

Маршрутизация подобна дорожному регулировщику: при выборе пользователем определенного URL маршрутизация указывает дорогу к составляющим эту страницу данным. В предыдущих проектах мы создавали прослушиватели событий для подтверждения отправки формы и нажатий на кнопки. Маршрутизация аналогична прослушивателю события, но отслеживает изменения текущего URL.

Все сайты используют какую-либо разновидность маршрутизации. Например, если вы перейдете по адресу www.bignerdranch.com/we-teach/, то сервер установит соответствие маршрута `/we-teach/` HTML-файлам, находящимся в каталоге `we-teach` на сервере. Другие серверы работают иначе: вместо извлечения статистических HTML-файлов могут запускать выводящую какую-то разметку HTML функцию.

Приложение Ember может делать то же самое, но без запроса HTML у сервера. Когда приложению понадобится перейти на другую экранную форму, оно просто поменяет имя маршрута в URL. У `Router` — дочернего объекта основного объекта приложения — есть прослушиватели событий и обработчики изменений URL. Используя новый маршрут, он выполняет поиск в таблице маршрутизации и находит объект `Ember.Route`. Потом `Router` вызывает несколько методов этого объекта маршрута, запуская процесс получения необходимых для следующей экранной формы данных. Такие последовательные обратные вызовы *в жизненном цикле маршрута* называются *точками подключения*.

Создание маршрутов — фундаментальная часть разработки приложений Ember. Соглашения о наименованиях фреймворка Ember подразумевают, что разработчик будет создавать контроллеры и шаблоны, чьи имена будут соответствовать связанным

с ними маршрутам. Так, например, при создании маршрута `sightings` маршрутизатор устанавливает соответствие запроса для `/sightings` маршруту `SightingsRoute`, который, в свою очередь, обращается к контроллеру `SightingsController` и, наконец, визуализирует шаблон `app/templates/sightings.hbs`.

В этой главе вы узнаете о структурных компонентах приложений Ember и научитесь использовать утилиту Ember CLI, чтобы создать файлы модулей для маршрутов и шаблонов приложения Tracker. Маршруты — ключ к приложению Ember, и работа в данной главе подготовит вас к разработке приложения в следующих пяти главах.

Рисунок 20.1 демонстрирует Tracker по состоянию на конец данной главы.

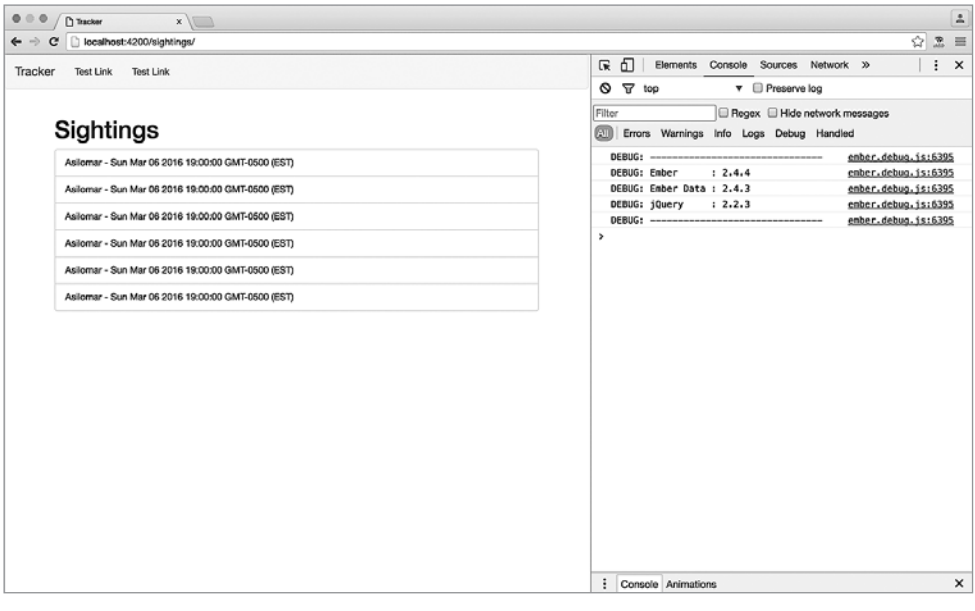


Рис. 20.1. Приложение Tracker

Утилита generate фреймворка Ember

Ember CLI предоставляет утилиту generate для скаффолдинга, которая будет полезна при изучении соглашений и шаблонов наименований Ember. Мы будем использовать команду `ember generate` (в сокращенной форме — `ember g`) для создания файлов и шаблонного кода нашего проекта.

Напомним, что задача приложения Tracker — фиксация случаев наблюдения криптидов (таких существ, как снежный человек). Оно будет отслеживать информацию о случаях наблюдения, криптидах и очевидцах. Приложению понадобится немало маршрутов (табл. 20.1).

Таблица 20.1. Маршруты для приложения

Маршрут	Путь маршрута	Данные маршрута
index	/index	Без данных — перенаправление на наблюдения (sightings)
sightings	/sightings	Список наблюдений
cryptids	/cryptids	Список криптидов
witnesses	/witnesses	Список очевидцев
sighting	/sighting	Подробная информация об отдельном наблюдении
cryptid	/cryptid	Подробная информация об отдельном криптиде
witness	/witness	Подробная информация об отдельном очевидце
sightings index	/sightings/index	Целевая страница списка наблюдений
sightings new	/sightings/new	Форма для создания новых наблюдений
sighting index	/sighting/:sighting_id/index	Целевая страница отдельного наблюдения
sighting edit	/sighting/:sighting_id/edit	Форма редактирования отдельного наблюдения

Вы сможете создать их с помощью команды `ember generate`. Откройте терминал и перейдите в каталог `tracker`. Выполните следующие команды (по строке за раз) для генерации маршрутов:

```
ember g route index
ember g route sightings
ember g route sightings/index
ember g route sightings/new
ember g route sighting
ember g route sighting/index
ember g route sighting/edit
ember g route cryptids
ember g route cryptid
ember g route witnesses
ember g route witness
```

Это будет выглядеть примерно так, как показано на рис. 20.2.

Теперь посмотрим, что создала для нас команда `ember g`. У нас должны были появиться новые файлы в каталогах `routes/` и `templates/`.

Откройте файл `app/routes/index.js` и обратите внимание, что этот модуль импортирует класс `Ember` и экспортирует класс `Ember.Route`:

```
import Ember from 'ember';

export default Ember.Route.extend({
});
```

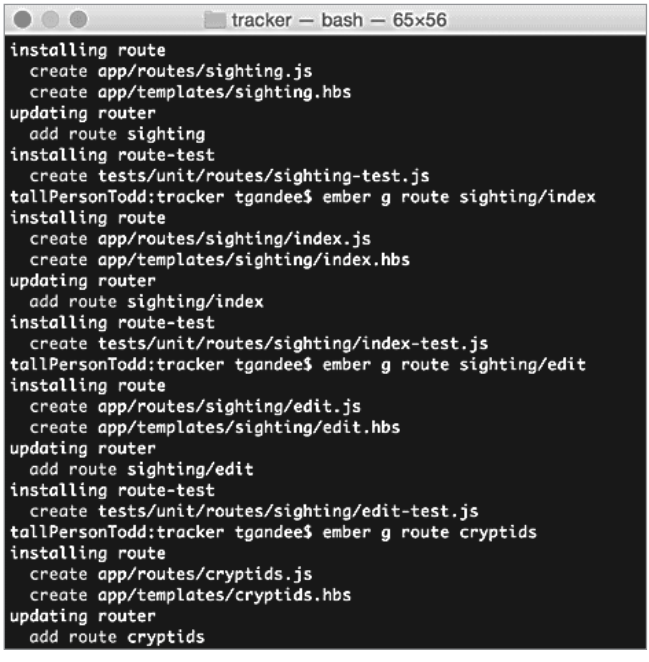


Рис. 20.2. Генерация маршрутов

Метод `.extend` создает новый подкласс класса `Ember.Route`, принимая на входе в качестве аргумента объект JavaScript. Благодаря синтаксису ES6 можно создать отдельные модули для каждого маршрута.

Утилита Ember CLI будет автоматически находить только что созданный нами модуль `Ember.Route` и импортировать его в наше приложение, независимо от того, используем ли мы команду `generate`, как здесь, или создаем модули вручную. Работать с командой `generate` удобнее, поскольку она добавляет в файлы определенный шаблонный код.

Откройте шаблон `app/templates/index.hbs`, также сгенерированный для нас Ember CLI. Этот шаблон будет использоваться для `IndexRoute`. В нем содержится одна-единственная строка: `{{outlet}}`.

Вероятно, вы помните этот фрагмент кода по прошлой главе, где вы редактировали файл `templates/application.hbs`. Этот вспомогательный элемент помогает шаблонам выполнять вложение контента между маршрутами. Через несколько абзацев вы узнаете о нем больше.

Пока что оставьте эту строку в файле `app/templates/index.hbs` в покое и добавьте над ней HTML-элемент `<h1>`:

```
<h1>Index Route</h1>
{{outlet}}
```

После этого запустите приложение с помощью команды `ember server`. Пусть сервер будет запущен во время вашей работы над проектом. Если вам понадобится утилита Ember CLI (например, для генерации дополнительных модулей), просто откройте еще одно окно терминала. Сервер подгрузит новые модули в наше приложение Ember и обновит страницу браузера.

Перейдите в браузере Chrome по адресу http://localhost:4200. Приложение должно выглядеть так, как показано на рис. 20.3.

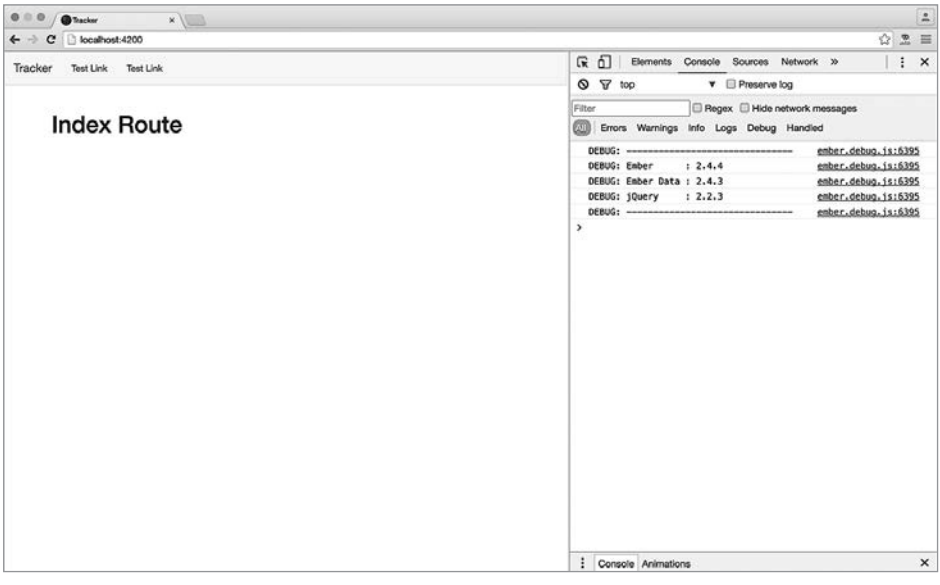


Рис. 20.3. Маршрут Index

Вы должны увидеть элементы `NavBar` из `app/templates/application.hbs` и элемент `<h1>` из `app/templates/index.hbs`. Как же этот элемент сюда попал?

Когда мы создавали приложение, для нас было сгенерировано немало файлов, включая `app.js` и `router.js`. Файл `app.js` — отправная точка всего приложения, он отвечает за инициализацию. В нем имеются функции для создания нового приложения Ember (аналогично созданию нового экземпляра `Truck` в приложении CoffeeRun).

В частности, приложение Ember будет создавать объекты `Router` и `ApplicationRoute` при загрузке или перезапуске приложения. Эти два объекта управляют нашим приложением.

В файле `router.js` мы будем регистрировать маршруты для связи URL с конкретными страницами. В конфигурации каждого маршрута (`route`) можно указать несколько параметров. Можно даже создавать вложенные маршруты. Эта ценная

возможность Ember позволяет повторно использовать контент и логику в различных экранных формах.

Откройте файл `router.js` и взгляните на метод, регистрирующий маршруты:

```
import Ember from 'ember';
import config from './config/environment';

const Router = Ember.Router.extend({
  location: config.locationType
});

Router.map(function() {
  this.route('sightings', function() {
    this.route('new');
  });
  this.route('sighting', function() {
    this.route('edit');
  });
  this.route('cryptids');
  this.route('cryptid');
  this.route('witnesses');
  this.route('witness');
});
export default Router;
```

В метод `Router.map` передается обратный вызов. Внутри этого обратного вызова метод `route` регистрирует маршруты. Он также принимает на входе обратные вызовы, которые Ember преобразует в иерархию маршрутов. Наверху этой иерархии находится маршрут `ApplicationRoute`.

При посещении URL вложенного маршрута Ember использует HTML из родительского шаблона. Ember ищет в этом родительском шаблоне вспомогательный элемент `{{outlet}}`, указывающий, в каком месте должен быть вложен HTML из дочернего шаблона.

Посмотрим, как это будет работать в нашем приложении.

Ember визуализирует контент маршрута `ApplicationRoute` с вложенным в него контентом маршрута `IndexRoute`. За кулисами Ember проверяет каталог `routes/` проекта на наличие файла `index.js`. Добавив `index.js` в соответствующий каталог, можно создать целевую страницу для любого маршрута. По сути, это общепринятая практика, рекомендуемая к использованию в приложениях Ember.

Возможно, вы обратили внимание, что в файле `router.js` нет ссылок на маршруты `index`. Ember автоматически генерирует маршрут `index` для всех родительских маршрутов с вложенными дочерними маршрутами, как будто они находятся в `router.js`:

```
...
Router.map(function() {
  this.route('index');
```

```
  this.route('sightings', function() {
    this.route('index');
    this.route('new');
  });
  this.route('sighting', function() {
    this.route('index');
    this.route('edit');
  });
  this.route('cryptids');
  this.route('cryptid');
  this.route('witnesses');
  this.route('witness');
});
...
```

Вложенные маршруты

Маршруты дают возможность структурировать данные в представлениях. Аналогично каталогам вложенные маршруты группируют вместе взаимосвязанные маршруты под единым URL. Удобно рассматривать родительские маршруты как существительные, а дочерние — как глаголы или прилагательные:

```
// Родительский маршрут – существительное
this.route('sightings', function() {
  // Дочерний маршрут – глагол или прилагательное
  this.route('new');
});
```

`sightings` — родительский маршрут, представляющий *sightings (наблюдения)*, которые представляют собой сущности (существительные). `new` — вложенный маршрут, представляющий действие по *созданию* наблюдения (глагол). Свойство `this.route` используется для формирования URL, включающего как родительский, так и дочерний маршрут.

С помощью вложения шаблонов можно визуализировать одни части нашего сайта (например, навигацию) на всех маршрутах, а другие — отображать только по определенным маршрутам (например, `IndexRoute` по корневому URL). Мы будем задавать для каждого маршрута способ извлечения данных с помощью функций обратного вызова.

Теперь нам нужно отредактировать некоторые сгенерированные вместе с маршрутами файлы шаблонов и перейти по адресам различных страниц приложения. Код, который мы добавим в этом разделе, — временный, но он позволит нам увидеть взаимосвязи между нашими маршрутами.

Для начала откройте файл шаблона `app/templates/sightings.hbs` и добавьте в него элемент `<h1>` над имеющимся там вспомогательным элементом `{{outlet}}`:

```
<h1>Sightings</h1>
{{outlet}}
```

Далее отредактируйте шаблон `app/templates/sightings/index.hbs`. Добавьте в него еще один элемент `<h1>`, теперь уже удалив вспомогательный элемент `{{outlet}}`. Родительские шаблоны используют `{{outlet}}` для вложения дочерних представлений. `app/templates/sightings/index.hbs` — дочерний шаблон без каких-либо вложенных маршрутов, так что ему вспомогательный элемент `{{outlet}}` не нужен:

```
~~{{outlet}}~~
<h1>Index Route</h1>
```

Сохраните файлы и перейдите в браузере по адресу http://localhost:4200/sightings/, чтобы увидеть результат (рис. 20.4).

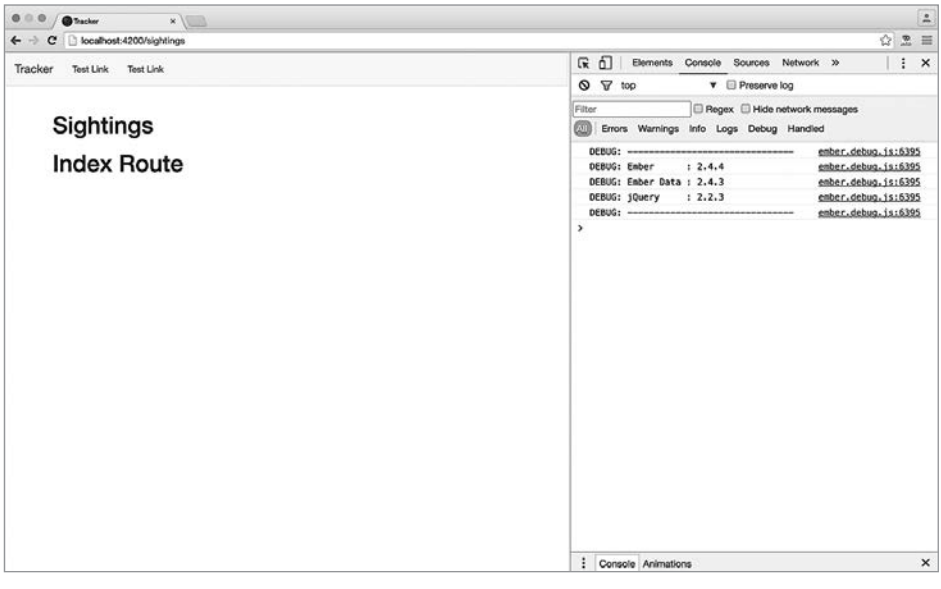


Рис. 20.4. Наблюдения: вложенные маршруты

Отредактируйте шаблон `app/templates/sightings/new.hbs`. Это дерево маршрутов также оканчивается дочерним маршрутом, так что удалите `{{outlet}}` и добавьте элемент `<h1>`:

```
~~{{outlet}}~~
<h1>Index Route</h1>
```

Поменяйте URL в вашем браузере на http://localhost:4200/sightings/new (рис. 20.5).

Теперь наше приложение Tracker содержит вложенные маршруты, визуализирующие шаблон для родительского файла `app/templates/sightings.hbs` и обоих дочерних — `app/templates/sightings/index.hbs` и `app/templates/sightings/new.hbs`. Родительский шаблон использует вспомогательный элемент `{{outlet}}` для вложения представлений.

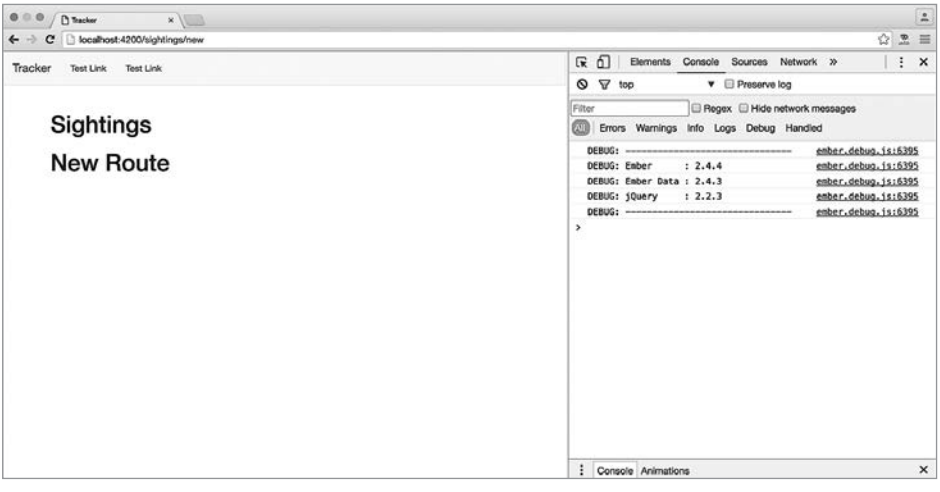


Рис. 20.5. Наблюдения: новый маршрут

Утилита Ember Inspector

Ember Inspector обеспечивает удобный способ просмотра всех маршрутов приложения. Щелкните в Ember Inspector на пункте меню Routes, чтобы просмотреть их (рис. 20.6).

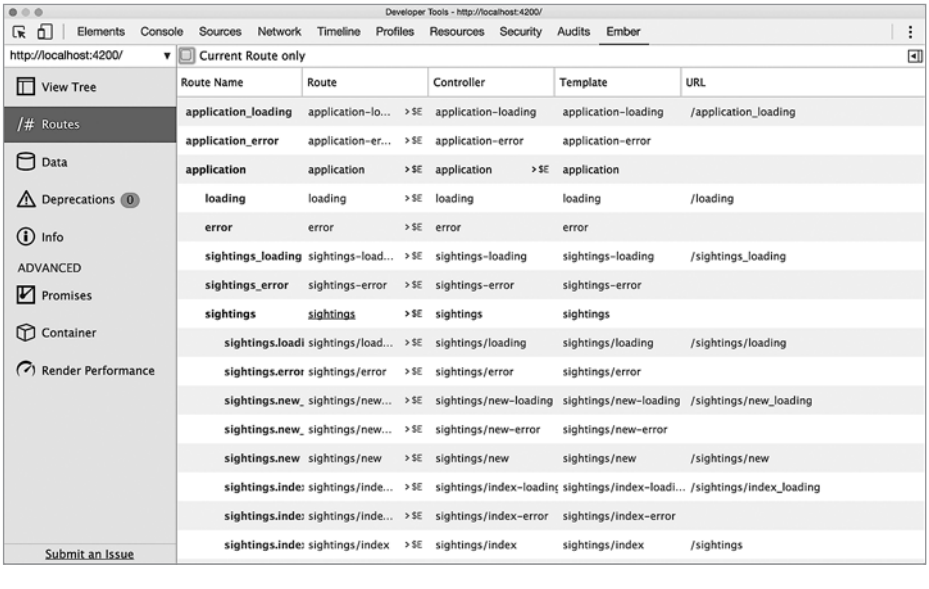


Рис. 20.6. Структура маршрутов

Ого, сколько маршрутов! Даже больше, чем мы сгенерировали. Обратите внимание на многочисленные маршруты, оканчивающиеся на `loading` и `error`. Это автоматически сгенерированные маршруты для состояний жизненного цикла по загрузке данных в маршрутах. Аналогично маршрутам `index` эти объекты были сгенерированы фреймворком Ember для заполнения промежутков при переходе от одного состояния маршрута к другому.

Назначение моделей

Следующий шаг — обеспечение всех маршрутов данными с помощью обратного вызова модели маршрута. У каждого объекта `Ember.Route` имеется метод для связывания модели (она, напомним, представляет собой данные, которыми обеспечивается шаблон) с контроллером.

За кулисами приложение Ember инициализирует объект `Route` при изменении URL. У объекта `Route` имеются четыре *точки подключения* для настройки: `beforeModel`, `model`, `afterModel` и `setupController`.

Пока что мы сосредоточим внимание на обратном вызове точки подключения `model`.

Добавьте в обратном вызове `model` какие-нибудь фиктивные данные в маршрут `SightingsRoute` в файле `app/routes/sightings.js`:

```
import Ember from 'ember';

export default Ember.Route.extend({
  model(){
    return [
      {
        id: 1,
        location: 'Asilomar',
        sightedAt: new Date('2016-03-07')
      },
      {
        id: 2,
        location: 'Asilomar',
        sightedAt: new Date('2016-03-07')
      },
      {
        id: 3,
        location: 'Asilomar',
        sightedAt: new Date('2016-03-07')
      },
      {
        id: 4,
        location: 'Asilomar',
        sightedAt: new Date('2016-03-07')
```

```
      },
      {
        id: 5,
        location: 'Asilomar',
        sightedAt: new Date('2016-03-07')
      },
      {
        id: 6,
        location: 'Asilomar',
        sightedAt: new Date('2016-03-07')
      }
    ];
  }
});
```

Обратите внимание на синтаксис точки подключения `model`:

```
model() {
  [тут располагается ваш код]
}
```

Это сокращенная форма записи ES6 для:

```
model: function() {
  [тут располагается ваш код]
}
```

В следующих главах мы будем использовать этот синтаксис для описания методов наших объектов в Ember.

Обратный вызов `model` — подходящее место для извлечения данных, необходимых для визуализации шаблона. Методы жизненного цикла шаблона в `Ember.Route` возвращают для каждой из точек подключения объекты. Точка подключения `model` в итоге вернет данные точке подключения `setupController`, устанавливающей значение свойства `model` контроллера `SightingsController`. Вы можете обращаться к этим данным в шаблонах `app/templates/sightings.hbs` и `app/templates/sightings/index.hbs`.

Отредактируйте `app/templates/sightings/index.hbs` следующим образом. Мы дадим пояснения к этому коду после того, как вы его введете.

```
<h1>Index Route</h1>
<div class="panel panel-default">
  <ul class="list-group">
    {{#each model as |sighting|}}
      <li class="list-group-item">
        {{sighting.location}} - {{sighting.sightedAt}}
      </li>
    {{/each}}
  </ul>
</div>
```

Если вы никогда не использовали языки шаблонов, этот код может показаться вам странным. Слова в двойных фигурных скобках (`{{ }}`), по существу, JavaScript-функции, замаскированные под операторы. Эти строки как бы говорят: «Для каждого `sighting` в свойстве `model` (которое должно быть массивом) визуализировать элемент `<li>` с местоположением `location` и датой `sightedAt` элемента `sighting`».

Мы рассмотрим синтаксис `{{ }}` в целом и синтаксис `{{#each}}` в частности в главе 23.

Перейдите в браузере по адресу http://localhost:4200/sightings. Приложение должно выглядеть так, как показано на рис. 20.7.

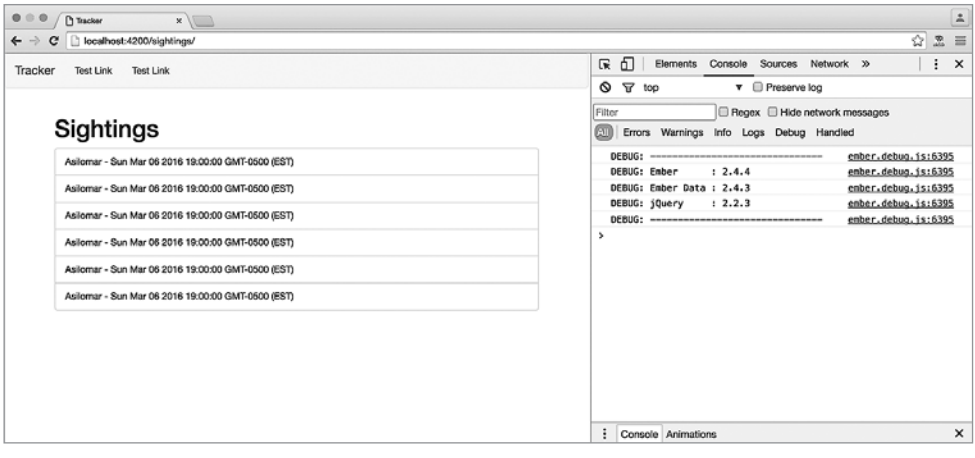


Рис. 20.7. Перечень модели index

Мы завершили первую половину цикла маршрута, передав данные шаблону для его визуализации. В следующей главе мы изучим язык шаблонизации `Handlebars`. Он показывает состояние приложения с помощью свойств контроллера, визуализируя только нужные элементы DOM при изменении состояния приложения.

Точка подключения beforeModel

Как описывалось выше, объект маршрута вызывает последовательность функций, начиная с `beforeModel`. Эта функция хорошо подходит для того, чтобы проверить состояние приложения до извлечения данных, а также для перенаправления пользователя, который не должен находиться на данной странице, например для выполнения его аутентификации.

Мы будем использовать точку подключения `beforeModel` для безусловного перенаправления пользователя на новую страницу. Это удобно сделать в маршруте `IndexRoute`. Возможно, в будущем нам захочется добавить панель инструментов, но пока что целевой страницей будет `sightings`.

Добавьте в файле app/routes/index.js обратный вызов beforeModel:

```
import Ember from 'ember';
export default Ember.Route.extend({
  beforeModel(){
    this.transitionTo('sightings');
  }
});
```

Теперь при переходе по адресу http://localhost:4200/ URL меняется на http://localhost:4200/sightings и вы должны увидеть список наблюдений из шаблона app/templates/sightings/index.hbs.

Последние две точки подключения, afterModel и setupController, не будут использоваться в приложении Tracker. При создании файла маршрута делается копия объекта Ember.Route с перезаписью метода (аналогично интерфейсам в таких языках программирования, как Java). Точка подключения setupController запускается по умолчанию для задания значения свойства model объекта controller маршрута.

В настоящий момент у нашего приложения есть несколько простых маршрутов, обрисовывающих контуры его функциональности: целевая страница, перечень наблюдений и маршрут для добавления нового наблюдения. Мы создали шаблоны для маршрутов и добавили данные моделей в маршрут sightings. Мы перенаправляем маршрут index на маршрут sightings index. Неплохое начало! В следующей главе мы познакомимся с Ember.Models, адаптерами, вычисляемыми свойствами и механизмами хранения.

Для самых любознательных: setupController и afterModel

Точка подключения setupController предназначена для заданий значений свойств контроллера, который будет их визуализировать. Можно использовать поведение по умолчанию: значение свойства model контроллера задается при задании значений других активных свойств контроллера с помощью метода this._super:

```
setupController(controller, model) {
  this._super(controller, model);
  // this.controllerFor('[иной контроллер]').set("[имя свойства]", [значение]);
}
```

Точка подключения afterModel выполняется после разрешения точки подключения model (представляющей собой объект Promise). Обратите внимание, что существуют особые случаи, когда точка подключения model не будет вызвана, поскольку объект Promise уже был разрешен. В таких случаях точка подключения afterModel вызывается перед setupController и может использоваться в качестве метода для проверки целостности данных модели перед передачей их контроллеру.

21 Модели и привязка данных

При работе над следующей частью приложения Tracker сосредоточим внимание на слое данных.

Мы уже работали с данными в виде объектных литералов, создавали и изменяли объекты и их свойства, а также задавали функции для быстрого создания объектов на основе значений по умолчанию. Вам уже известно, как сохранять данные в API `localStorage` и `sessionStorage`.

В приложении Tracker мы будем работать с данными в виде моделей. Модели, по сути, представляют собой функции, создающие объекты со специальными свойствами и методами. Они представляют архитектуру данных, движущихся по нашему приложению.

В фреймворке Ember имеется класс, который позволяет описать архитектуру данных приложения, — `Ember.Object`. Все классы Ember наследуют его. Ember предоставляет нам при простом синтаксисе и схеме именования возможность создавать, извлекать, изменять и уничтожать экземпляры моделей во время работы приложения.

Однако нашему современному приложению понадобится больше возможностей, чем предоставляет класс `Ember.Object`. Наши модели должны уметь сохраняться при получении от бизнес-логики запроса на извлечение или сохранение данных из источника данных.

Мы познакомимся с *Ember Data* — JavaScript-библиотекой, представляющей собой надстройку над классом `Ember.Object`, которая позволяет добавлять относящуюся к моделям функциональность. Библиотека Ember Data добавляет основанные на `Ember.Object` классы, скрывающие от нас всю сложность работы с различными источниками данных: воплощающими REST API, localStorage и даже статическими данными фикстур.

Библиотека Ember Data также предоставляет возможность хранения данных в памяти. Хранилище данных — это место, где создаются, извлекаются, изменяются и уничтожаются экземпляры моделей.

Описания моделей

Утилита Ember CLI уже загрузила библиотеку Ember Data, так что мы готовы приступить к созданию моделей.

В предыдущей главе мы использовали команду `ember g route [`*`имя маршрута`*`]`, чтобы указать утилите Ember CLI создать файлы маршрутов. Команду `ember generate` можно использовать и для создания файлов моделей, используя синтаксис `ember g model [`*`имя модели`*`]`.

Создайте файлы моделей, необходимые для маршрутов криптидов, наблюдений и очевидцев:

```
ember g model cryptid
ember g model sighting
ember g model witness
```

Описание модели криптидов будет находиться в файле `app/models/cryptid.js`. Откройте этот файл и добавьте атрибуты для имени, типа криптида (биологического вида), изображения профиля и наблюдений в модель `cryptid`:

```
import DS from 'ember-data';

export default DS.Model.extend({
  name: DS.attr('string'),
  cryptidType: DS.attr('string'),
  profileImg: DS.attr('string'),
  sightings: DS.hasMany('sighting')

});
```

В библиотеке Ember Data, на которую мы тут сослались как на DS (от data store — «хранилище данных»), есть метод `attr`, применяемый для задания атрибутов модели. Метод `attr` возвращает значение после выполнении синтаксического разбора данных из источника. Если передать методу `attr` тип атрибута, значение будет приведено к этому типу. Если не указывать тип атрибута, данные будут переданы соответствующему ключу в неизменном виде.

Существует несколько встроенных типов атрибутов: `string`, `number`, `boolean` и `date`. Можно создавать пользовательские атрибуты модели с помощью *преобразований*, о которых мы расскажем в главе 22.

Метод `attr` может также принимать на входе второй аргумент для задания значения по умолчанию. Этот необязательный аргумент представляет собой хеш с одним ключом — `defaultValue`. Вот несколько примеров:

```
name: DS.attr('string', {defaultValue: 'Bob'}),
isNew: DS.attr('boolean', {defaultValue: true}),
createdAt: DS.attr('date', {defaultValue: new Date()}),
numOfChildren: DS.attr('number', {defaultValue: 1})
```

В описании криптида мы использовали тип атрибута `string` для атрибутов `name`, `cryptidType` и `profileImg`. (Почему тип `string` для `profileImg`? Дело в том, что он будет ссылаться на путь к изображению, а не на само изображение.)

Атрибут `sightings` использует другой метод для описания своих данных — `hasMany`. Этот метод относится к методам описания отношений библиотеки Ember Data. При запросе криптида у API, воплощающего REST, у него окажутся ассоциированные с ним наблюдения. Эта ассоциация будет возвращена в виде массива идентификаторов наблюдений, ссылающихся на экземпляр модели наблюдений.

В библиотеке Ember Data имеются методы для работы с такими типами отношений, как «один» к «одному», «один» ко «многим» и «многие» ко «многим» (табл. 21.1).

Таблица 21.1. Типы отношений

Отношение	Модель-владелец	Модель, к которой относится
Один к одному	DS.hasOne	DS.belongsTo
Один ко многим	DS.hasMany	DS.belongsTo
Многие ко многим	DS.hasMany	DS.hasMany

Первый аргумент — модель, с которой необходимо ассоциировать. В нашем приложении у криптида может быть несколько экземпляров атрибута `sightings` (вы удивитесь, когда узнаете, сколько людей видят этих существ). Второй аргумент — необязательный хеш, который аналогично второму аргументу метода `attr` представляет собой конфигурационный объект для задания значений при вычислении функции. Он содержит ключ `async` и значение (которое по умолчанию равно `true`).

Отношения моделей могут потребовать выполнения запросов к серверу для извлечения остальных данных модели. В случае криптид необходим запрос к атрибуту `sightings` для отображения данных наблюдений для каждого криптида. То же самое справедливо для обратного отношения — наблюдений, *относящихся к* криптиду. Значение по умолчанию `async: true` требует отдельного запроса и конечной точки API для извлечения связанных данных.

Если API умеет отправлять все данные вместе, можно задать значение `async` равным `false`. Для приложения Tracker оставьте значение по умолчанию `true`.

Далее откройте модель очевидцев в файле `app/models/witness.js` и добавьте атрибуты для имени и фамилии очевидца, его адреса электронной почты и зафиксированных наблюдений:

```
import DS from 'ember-data';

export default DS.Model.extend({
  fName: DS.attr('string'),
  lName: DS.attr('string'),
  email: DS.attr('string'),
  sightings: DS.hasMany('sighting')
});
```

Мы определили очевидца как объект, содержащий имя (атрибут `fName`), фамилию (атрибут `lName`), адрес электронной почты (атрибут `email`) и отношение к наблюдениям как многие ко многим (атрибут `sightings`).

Наконец, откройте файл нашей третьей модели `app/models/sighting.js`. Добавьте в модель `sightings` атрибуты для данных, кто, что, где и когда наблюдал, а также для даты фиксации наблюдения:

```
import DS from 'ember-data';

export default DS.Model.extend({
  location: DS.attr('string'),
  createdAt: DS.attr('date'),
  sightedAt: DS.attr('date'),
  cryptid: DS.belongsTo('cryptid'),
  witnesses: DS.hasMany('witness')
});
```

Наблюдения весьма схожи с очевидцами и криптидами, основные свойства описаны как строки. Свойство `location` — значение, которое пользователь будет вводить в приложении, а свойства `createdAt` и `sightedAt` будут добавляться на сервере при внесении наблюдения в базу данных.

Для свойства `cryptid` предусмотрен метод `DS.belongsTo('cryptid')`. Этот метод поддерживает отношение «один» ко «многим», связывающее экземпляр `cryptid` с экземпляром `sighting`. Выбрано именно это отношение потому, что у каждого криптида может быть много наблюдений.

Метод createRecord

При инициализации приложения библиотека Ember Data создает `store` — объект локального хранилища. `this.store` — объект, создающий, извлекающий, обновляющий и удаляющий все требуемые записи моделей приложения Tracker. Ember внедряет объект `store` во все `Routes`, `Controllers` и `Components`. В области видимости методов маршрутов доступ к `store` будет осуществляться через переменную `this`.

Для создания записи мы будем вызывать метод `this.store.createRecord`. Этот метод ожидает на входе два аргумента: имя модели в виде строки и данные записи в виде объекта.

Откройте файл `app/routes/sightings.js`. Удалите фиктивные наблюдения и создайте три новые записи `sighting`, каждую со значением `location` в виде строки и значением `sightedAt` с помощью `new Date`:

```
import Ember from 'ember';

export default Ember.Route.extend({
  model() {
    return [
      {
        id: 1,
        location: 'Asilomar',
        sighted_at: new Date('2016-03-07')
      },
      ...
      {
        id: 6,
        location: 'Asilomar',
        sightedAt: new Date('2016-03-07')
      }
    ];
    let record1 = this.store.createRecord('sighting', {
      location: 'Atlanta',
      sightedAt: new Date('2016-02-09')
    });

    let record2 = this.store.createRecord('sighting', {
      location: 'Calloway',
      sightedAt: new Date('2016-03-14')
    });

    let record3 = this.store.createRecord('sighting', {
      location: 'Asilomar',
      sightedAt: new Date('2016-03-21')
    });

    return [record1, record2, record3];
  }
});
```

В главе 20 модель маршрута `sightings` возвращала массив. Вместо возвращения JavaScript-объектов мы создали три записи наблюдений и вернули их в массиве. Выполните команду `ember server` для запуска сервера, если он еще не запущен, чтобы увидеть наши новые записи по маршруту `sightings`, http://localhost:4200/sightings (рис. 21.1).

Этот пример демонстрирует, что создание моделей Ember Data напоминает создание объектов JavaScript. Преимущество объектов моделей Ember Data состоит в предоставляемых ими методах. Начнем с методов `get` и `set`.

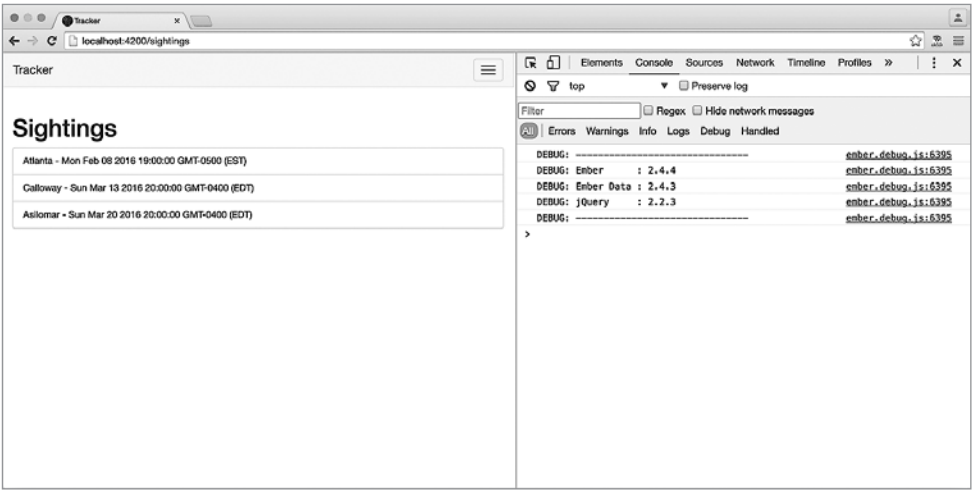


Рис. 21.1. Создаем (create) sightings

Методы get и set

В основе записей модели Ember Data лежит класс `Ember.Object`. Описание этого объекта содержит методы `get` и `set`. В отличие от большинства других языков программирования использовать для экземпляров объектов геттеры и сеттеры в JavaScript не обязательно. Ember использует идею геттеров и сеттеров в методах для принудительного выполнения функций при изменении свойств объекта. Это дает Ember возможность добавлять в метод `set` триггеры событий, поэтому программистам нужно внимательнее указывать значения свойств.

Метод `get` принимает на входе один аргумент — имя свойства, значение которого необходимо извлечь. Оцените, как он работает, на обратном вызове модели `app/routes/sightings.js`:

```
import Ember from 'ember';

export default Ember.Route.extend({
  model() {
    let record1 = this.store.createRecord('sighting', {
      location: 'Atlanta',
      sightedAt: new Date('2016-02-09')
    });
    console.log("Record 1 location: "  +  record1.get('location') );
    ...
    return [record1, record2, record3];
  }
});
```

Обновите страницу браузера. Убедитесь, что DevTools открыты, и выберите вкладку консоли JavaScript. Вы должны увидеть записи журнала Ember, оканчивающиеся строкой Record 1 location: Atlanta (Запись 1. Местоположение: Атланта).

Далее снова в файле `app/routes/sightings.js` установите с помощью метода `set` значение свойства `location` для записи `record1` после ее создания, но перед тем, как вывести в журнал значение свойства:

```
import Ember from 'ember';

export default Ember.Route.extend({
  model() {
    let record1 = this.store.createRecord('sighting', {
      location: 'Atlanta',
      sightedAt: new Date('2016-02-09')
    });
    record1.set('location', 'Paris, France');
    console.log("Record 1 location: " + record1.get('location'));
    ...
    return [record1, record2, record3];
  }
});
```

Обновите страницу браузера — и вы увидите, что консоль отразила установленное значение Record 1 location: Paris, France (Запись 1. Местоположение: Париж, Франция) (рис. 21.2).

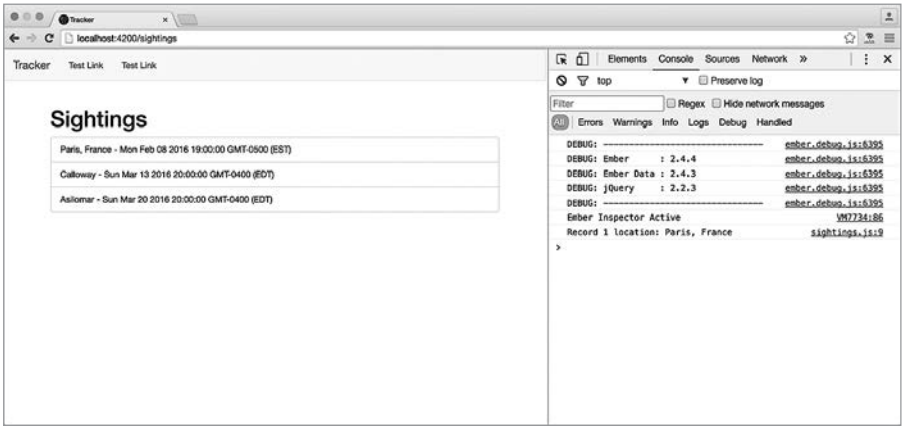


Рис. 21.2. Задаем location

Это простейшие примеры применения методов `get` и `set`. При задании значения свойства записи модели можно также связать другие записи модели со свойством этой записи с целью создания отношения между двумя записями модели для свойств, описанных с помощью методов `hasMany` или `belongsTo`.

Вычисляемые свойства

Вычисляемые свойства играют важную роль в управлении свойствами модели для шаблонов и компонентов. Метод `Ember.computed` принимает на входе значения свойств из области видимости и возвращает после завершения работы значение. Например, при вызове следующего фрагмента кода мы получаем вычисляемое свойство с изменением регистра свойства `fName` объекта на нижний:

```
Ember.computed(' fName ', function(){
  return this.get('fName').toLowerCase();
});
```

В этом примере метод `Ember.computed` выступает в качестве прослушивателя события изменений свойства `fName`. Вам не нужно менять метод `set`, чтобы вызвать срабатывание события, добавлять прослушиватель события и менять свойство `fName`. Все, что надо сделать, — создать новое свойство, возвращающее нужное вам значение.

Вычисляемые свойства используются в Ember практически повсеместно. Мы будем создавать вычисляемые свойства и для компонентов (в главе 25). Вычисляемое свойство применяется или в качестве *декоратора* для представления/компонента, как в вышеприведенном примере, или для извлечения конкретных данных, хранящихся в глубине объекта модели.

Декорирование данных означает их форматирование определенным образом, например приведение строки к нижнему регистру. Данные, поступающие от API, не всегда отформатированы так, как нам бы хотелось. Декораторы — это функции, чьи входящие аргументы и выходные объекты/массивы специально предназначены для использования в слое представления приложения. Отформатированные или свежесформированные декорированные данные, как правило, не возвращаются в базу данных. Поэтому декораторы обычно добавляются в контроллер, за исключением случая, когда все страницы визуализируют данные из модели, отформатированной в базе данных.

Добавьте вычисляемое свойство для `fullName` в модель очевидца в файле `app/models/witness.js`:

```
import DS from 'ember-data';

export default DS.Model.extend({
  fName: DS.attr('string'),
  lName: DS.attr('string'),
  email: DS.attr('string'),
  sightings: DS.hasMany('sighting'),
  fullName:  Ember.computed('fName', 'lName', function(){
    return this.get('fName') + ' ' + this.get('lName');
  })
});
```

Если автоперезапуск сервера выдает ошибку, убедитесь, что не забыли добавить запятую после объявления свойства `sightings`. Ее легко случайно пропустить.

Добавленное вами в модель `witness` свойство — функция, вызываемая при каждом изменении свойств `fName` и `lName`. Вычисляемые свойства могут принимать на входе любое количество аргументов — отслеживаемых свойств (последний аргумент — функция, возвращающая значение). Каждый являющийся свойством аргумент сможет инициировать вызов функционального аргумента.

Откройте файл `app/routes/witnesses.js` и создайте новую запись очевидца, чтобы проверить вычисляемое свойство модели `witness`:

```
import Ember from 'ember';

export default Ember.Route.extend({
  model() {
    let witnessRecord = this.store.createRecord('witness', {
      fName: "Todd",
      lName: "Gandee",
      email: "fake@bignerdranch.com"
    });
    return [witnessRecord];
  }
});
```

Чтобы вывести данные об очевидце на экран, отредактируйте шаблон `app/templates/witnesses.hbs`, чтобы использовать тот же итератор `{{#each}}`, что и в шаблоне `app/templates/sightings/index.hbs`:

```
{{outlet}}
<h1>Witnesses</h1>
<div class="row">
  {{#each model as |witness|}}
    <div class="col-xs-12 col-sm-6 col-md-4">
      <div class="well">
        <div class="thumbnail">
          <div class="caption">
            <h3>{{witness.fullName}}</h3>
            <div class="panel panel-danger">
              <div class="panel-heading">Sightings</div>
            </div>
          </div>
        </div>
      </div>
    </div>
  {{/each}}
</div>
```

Перейдите по адресу http://localhost:4200/witnesses и оцените результат (рис. 21.3).

В этом представлении мы добавили список очевидцев (который пока что содержит только одного очевидца) и воспользовались вычисляемым свойством для отображения свойства очевидца `fullName`. Это свойство было сгенерировано из добавленных нами при создании записи очевидца значений. Чтобы увидеть изменение

свойства, мы можем с помощью метода `witnessRecord.set` подставить другое имя или фамилию, прежде чем обратный вызов модели вернет запись.

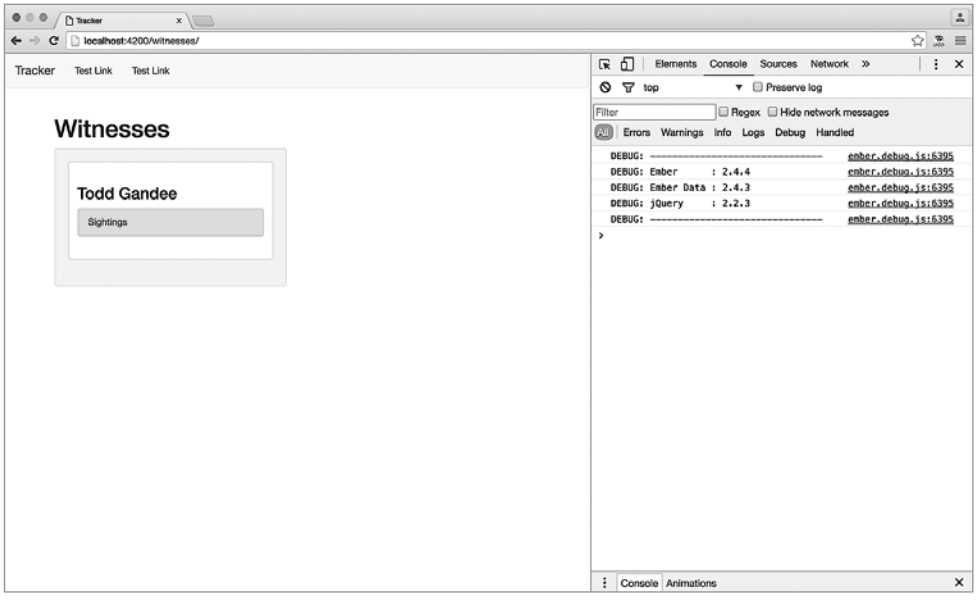


Рис. 21.3. Список очевидцев

Мы уже продвинулись довольно далеко — и это всего за несколько посвященных Ember глав! Вы уже научились описывать модели данных, создавать записи и вычисляемые свойства, а также получать и задавать значения свойств. А буквально через минуту прочитаете об извлечении, обновлении и удалении записей посредством API.

В следующей главе вы научитесь использовать адаптеры, сериализаторы и преобразования, чтобы связывать свои модели с данными в Интернете.

Для самых любознательных: извлечение данных

Как уже упоминалось выше, хранилище данных управляет данными модели и умеет их извлекать. В предыдущей главе мы возвращали данные в обратном вызове модели `SightingRoute` с помощью массива. В главе 22 мы будем в этом маршруте извлекать данные модели и возвращать их в виде объекта `Promise` с помощью метода `this.store.findAll` и других методов извлечения данных.

В табл. 21.2 приведены методы объекта `store` библиотеки Ember Data, предназначенные для извлечения данных из API, сохранения их в памяти и возвращения инициатору запроса.

Таблица 21.2. Методы объекта store библиотеки Ember Data

Тип запроса	Извлечь все записи	Извлечь отдельную запись
Найти сохраненные и локальные записи	findAll	findRecord
Найти только локальные записи	peekAll	peekRecord
Найти отфильтрованные записи	query	queryRecord

Методы извлечения бывают нескольких типов: сохраняющие и локальные, только локальные, а также фильтрующие сохраняющие и локальные. В большинстве сценариев использования требуются методы `findAll` и `findRecord`. Аргументы для каждого из них в точности соответствуют конечным точкам API, создаваемым библиотекой Ember Data для запроса данных из API.

Единственный обязательный аргумент метода `findAll` — имя модели. Например, запрос всех очевидцев выглядел бы так: `findAll('witness')`. Обратили внимание на имя в единственном числе? Помните, что этот аргумент представляет собой имя модели. Ember Data позаботится о том, чтобы запрос был с именем во множественном числе, когда будет формировать URL Ajax `/witnesses/`.

Методу `findRecord` требуется дополнительный аргумент для указания на конкретную запись. Этот аргумент представляет собой идентификатор, обычно `id` записи, например `this.store.findRecord('witness', 5)`. При вызове `findRecord('witness', 5)` создаст запрос для данных по пути `/witnesses/5`.

Для извлечения данных функциями `peekAll` и `peekRecord` необходимы такие же аргументы. При вызове этих методов данные будут возвращены немедленно, а не в виде объекта `Promise`.

Запрос данных у API — еще один способ формирования запросов. Если ваш API поддерживает параметры запроса *params* для отдельных конечных точек, хороший вариант — использовать методы `query` и `queryRecord` библиотеки Ember Data. Аналогично другим методам объекта `store` эти методы принимают имя модели в качестве первого аргумента. Последний аргумент — объект запроса, чьи пары «ключ/значение» преобразуются в значения строки запроса. В случае `query` запрос найдет все записи, отфильтрованные по ключам и значениям, а `queryRecord` используется, если известно, что запрос вернет только одну запись.

Например, вызов:

```
this.store.query('user',{fName: "todd"})
```

создаст следующий запрос:

```
/users/?f_name=todd
```

С другой стороны:

```
this.store.queryRecord('user', {email: 'me@test.com'})
```

создаст вот такой запрос:

```
/users?email=me@test.com
```

Все эти методы объекта `store` используют адаптеры, о которых вы прочтете в следующей главе.

Для самых любознательных: сохранение и удаление данных

Обновление (то есть сохранение) и удаление записей — следующие логические шаги после создания и извлечения (существует мнемоническая схема для этого: CRUD, что расшифровывается как «создание, чтение, обновление, удаление» (create, read, update, destroy)). Методы `save` и `destroyRecord` доступны непосредственно у экземпляров моделей. Эти методы инициируют через адаптер запросы для обновления хранилища данных. Они возвращают объекты `Promise`, а значит, предоставляют возможность связывать обратные вызовы цепочкой с помощью функции `.then` для выполнения каких-либо действий с возвращенными данными.

Как мы видели в этой главе, метод `set` позволяет изменять значения свойств для записей модели. Если поменять значение локально в приложении, данные будут отличаться от хранимого источника. Следовательно, после изменения значения с помощью `set` необходимо сохранять данные. Сделать это можно с помощью метода `modelRecord.save`. Сохранение модели указывает `store` на необходимость выполнения запроса к API (`POST` или `PUT` — в зависимости от состояния записи).

Хотя выше мы этого не упоминали, но при извлечении данных `store` будет выполнять `get`-запросы данных. При сохранении данных запросы отправляются с типом `POST`, если данные не существуют в хранимом источнике, и `PUT`, если существуют и просто обновляются.

Как уже упоминалось ранее, при вызове метода `createRecord` мы не сохраняем данные в базе данных. Мы просто создаем объект в оперативной памяти. Для создания и обновления их необходимо вызывать метод `save`.

Последний шаг — удаление записей. Аналогично методам для извлечения и обновления записей метод `modelRecord.destroyRecord` использует метод запроса — в данном случае `DELETE` — для удаления записи из хранимого источника. После этого объект `store` удаляет запись из памяти. Аналогично методу `save` метод `destroyRecord` представляет собой два вызова функций в одном (`deleteRecord` и `save`), поскольку `deleteRecord` удаляет только локальную запись. Метод `destroyRecord` используется чаще, поскольку объединяет эти шаги в одно действие.

Бронзовое упражнение: изменение вычисляемого свойства

В настоящий момент вычисляемое свойство `fullName` использует свойства `fName` и `lName`. Поменяйте связанные с ним свойства на `email` и `fName` для создания `fullName`, которое бы отображалось в виде `Todd - tgandee@bignerdranch.com`.

Серебряное упражнение: пометьте флагом новые наблюдения

Добавьте к модели наблюдений новый булев атрибут `isNew`[1]. Задайте для него значение по умолчанию `defaultValue`, равное `false`. Добавьте это свойство в одну из созданных нами записей и установите его значение равным `true`. Перейдите по маршруту `sightings` и воспользуйтесь Ember Inspector в браузере Chrome, чтобы посмотреть данные для этого маршрута. Только у одного экземпляра наблюдения свойство `isNew` должно быть равно `true`.

Золотое упражнение: добавление форм обращения

Очевидцы должны придерживаться вежливого обращения, например *«мистер Ганди»*. Добавьте в модель свойство `title`, после чего задайте его значение для всех записей очевидцев, кроме одной. Добавьте объектный аргумент со значением по умолчанию `defaultValue`. Придумайте интересную форму обращения по умолчанию для экземпляров, у которых форма обращения не указана. После создания свойства `title` и внесения новых данных в запись добавьте вычисляемое свойство для отображения `titleName`.

Поэкспериментируйте с формой обращения по умолчанию. В «Википедии» есть отличный список форм обращения (https://ru.wikipedia.org/wiki/Титул). «Махатма Ганди» звучит очень неплохо...

[1] Ember уже включает в модели булев атрибут isNew. Добавление его вручную приведет к ошибке. — *Примеч. пер.*

22 Данные: адаптеры, сериализаторы и преобразования

Приложения должны иметь возможность получать и передавать данные через интерфейс. Подключение к источнику данных — важная часть разработки приложения. Без него вы получите сложную систему форм, событий и списков — и никаких данных, которые можно было бы отобразить.

В данной главе мы рассмотрим основы подключения источников данных в фреймворке Ember. Мы будем использовать созданный для этой книги API и создадим адаптер для нашего приложения.

Эта глава немного отличается от других глав книги. В ней больше информации и меньше кода. Однако она даст вам реальное представление о разработке приложений, имеющих доступ к серверу и базе данных, находящимся вне вашего контроля. В следующей главе мы вернемся к написанию кода.

Как уже упоминалось в главе 21, адаптеры — «переводчики» для нашего приложения. При связи с источником данных приложению нужно запрашивать и отправлять данные различными способами. Библиотека Ember Data поставляется с встроенными объектами адаптеров для обработки некоторых наиболее распространенных сценариев работы с данными: JSON API и универсальными воплощающими REST API.

Мы воспользуемся объектом `JSONAPIAdapter` для подключения к источнику данных и будем возвращать данные в формате JSON API. Объект `RESTAdapter` представляет собой набор методов для работы с отформатированными данными из различных API, сгенерированными фреймворком Rails и плагином ActiveRecord для Rails.

Спецификации JSON API предоставляют возможность обмениваться данными в формате JSON. Они предлагают работающий предсказуемым образом и хорошо

масштабируемый паттерн для отправки данных серверам и получения данных от них. Хотя существует множество серверных языков программирования и имеются свои соглашения по паттернам объектов API для каждого, паттерн JSON API может использовать любой язык, так что смена технологий на сервере не должна влиять на приложения клиентской части. Узнать больше о JSON API можно на сайте jsonapi.org.

В данной главе мы также обсудим вопросы безопасности, а еще *сериализаторы* — слой трансляции в потоке выполнения адаптеров. Наконец, мы познакомимся с *преобразованиями* — средством приведения данных к типам, ожидаемым моделями. Совместная работа адаптеров, сериализаторов и преобразований показана на рис. 22.1.

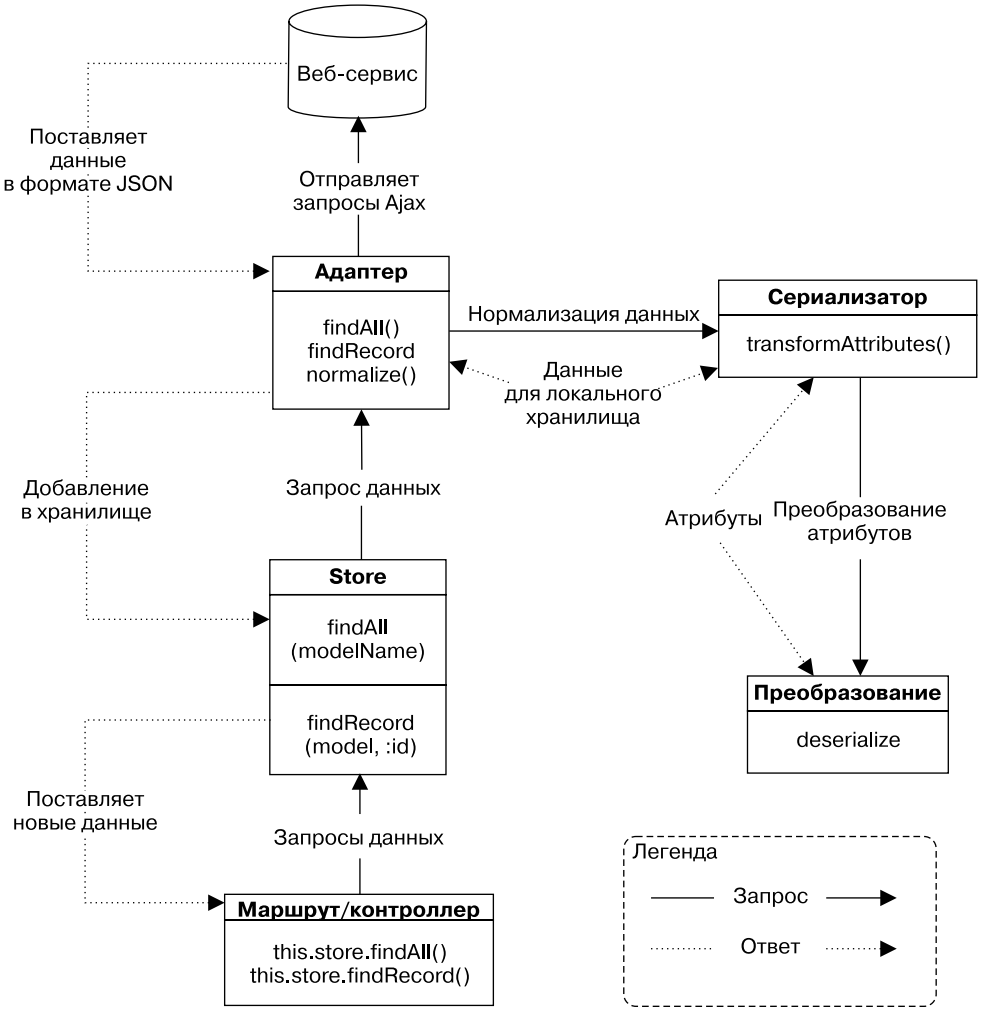


Рис. 22.1. Адаптер, сериализаторы и преобразования

К концу данной главы приложение Tracker будет выглядеть так, как показано на рис. 22.2.

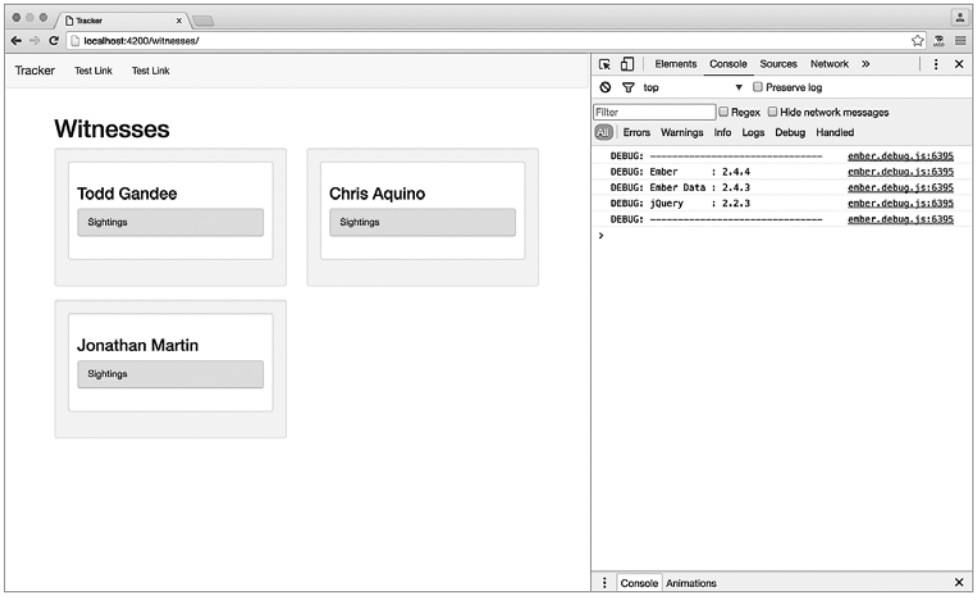


Рис. 22.2. Приложение Tracker по состоянию на конец данной главы

Адаптеры

Команда фреймворка Ember разрабатывала его в расчете на конкретные соглашения. Значительная часть этих соглашений — адаптеры. Объект `JSONAPIAdapter` будет связываться с API REST для всех запросов, имеющих источником объект `store`. Каждый запрос будет добавлять имя модели и соответствующие данные атрибутов к относительному пути домена.

Для генерации конкретных URL для Ajax-запросов адаптеру требуются свойства `host` и `namespace`. Адаптер выполняет Ajax-запросы и ожидает ответа в формате JSON с определенной структурой. Например, объект `JSONAPIAdapter` будет ожидать, что ответ на запрос очевидцев при запросе `GET` будет выглядеть следующим образом:

```
{
"links": {
  "self": "http://bnr-tracker-api.herokuapp.com/api/witnesses"
},
"data": [
  {
    "id": "5556013e89ad2a030066f6e0",
```

```
    "type": "witnesses",
    "attributes": {
      "lname": "Gandee",
      "fname": "Todd"
    },
    "links": {
      "self": "/api/witnesses/5556013e89ad2a030066f6e0"
    },
    "relationships": {
      "sightings": {
      "data": [],
      "links": {
        "self":
        "/api/witnesses/5556013e89ad2a030066f6e0/relationships/sightings"
        }
      }
    }
  }
]
}
```

Для каждого ответа свойство `type` каждого объекта должно быть именем запрашиваемой модели, чтобы можно было разрешить все записи для этого типа модели. Также для каждого отдельного объекта модели свойство `id` должно быть первичным ключом.

Начнем с генерации адаптера приложения:

```
ember g adapter application
```

Наше приложение будет выполнять запросы к API Tracker Big Nerd Ranch. Как мы говорили, среди значений свойств `JSONAPIAdapter` должны быть URL хоста и пространство имен, добавляемые после хоста при выполнении Ajax-запроса данных модели. Откройте файл `app/adapters/application.js` и объявите хост и пространство имен:

```
import JSONAPIAdapter from 'ember-data/adapters/json-api';

export default DS.JSONAPIAdapter.extend({
  host: 'https://bnr-tracker-api.herokuapp.com',
  namespace: 'api'
});
```

Подобно другим классам фреймворка Ember, паттерны наименования также применяются и к адаптерам, поэтому можно создавать адаптеры для подгонки под свои нужды любой модели, которая требуется API. Нестандартные случаи в структуре данных на сервере можно поместить в одну модель, вместо того чтобы подгонять все модели под исключительные случаи.

Код, добавленный нами в файл `app/adapters/application.js`, — глобальная настройка для всех запросов данных. Это все, что нам нужно для приложения Tracker, поскольку API отправляет JSON-ответы из одного хоста и пространства имен.

Но если для модели очевидцев, например, потребуется другое пространство имен или другой хост, можно создать файл `app/adapters/witness.js` и сконфигурировать именно этот адаптер для запросов очевидцев.

Далее необходимо извлечь данные из API через интерфейс `store`. Криптиды и очевидцы в API готовы к отправке.

Откройте файл `app/routes/witnesses.js`. Удалите фиктивные данные и замените их вызовом метода извлечения, с которым вы познакомились в главе 21.

```
...
  model(){
    let witnessRecord = this.store.createRecord('witness', {
      fname: "Todd",
      lname: "Gandee",
      email: "fake@bignerdranch.com"
    });
    return [witnessRecord];
    return this.store.findAll('witness');
  }
});
```

Теперь перезапустите приложение из терминала с помощью команды `ember server` и перейдите в браузере по адресу http://localhost:4200/witnesses (рис. 22.3).

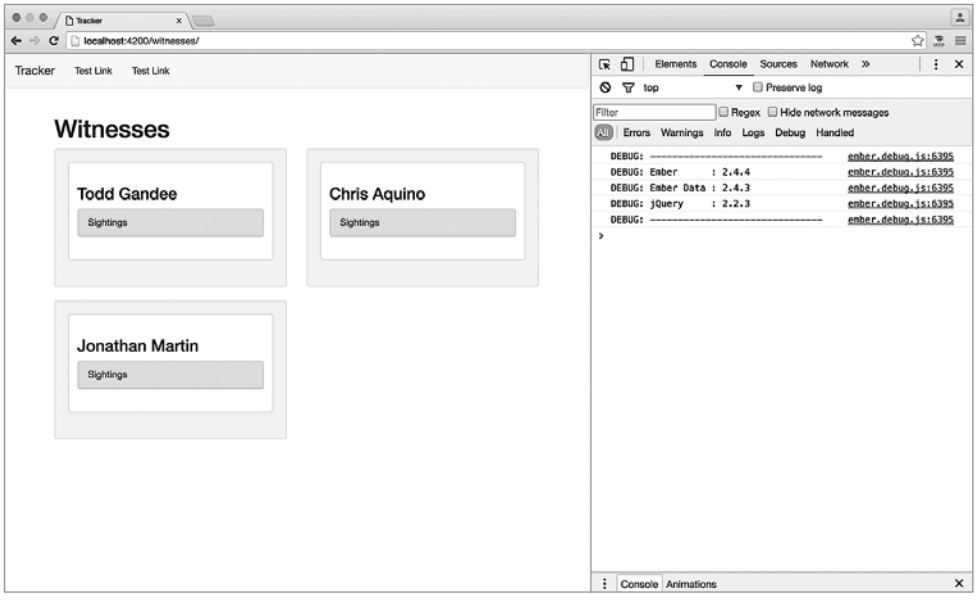


Рис. 22.3. Список очевидцев

Аналогично большинству потоков жизненного цикла в фреймворке Ember у адаптеров имеется несколько методов, передающих данные из API в объект

`store` и далее в маршруты, контроллеры и шаблоны. Цель конкретных адаптеров, таких как объект `JSONAPIAdapter`, заключается в работе с обширным паттерном данных и обработке предполагаемого ввода/вывода для каждой модели. В нашем примере используется сервер Node.js в сочетании с базой данных MongoDB, а для работающих с JSON API-данными конечных точек API применяется модуль json-api node.

Работа с API, который следует конкретному паттерну (с незначительными отклонениями в экстремальных случаях), — сплошное удовольствие для программиста. Вам может также понадобиться обрабатывать какие-либо дополнительные данные в запросе, например аутентификацию или заголовки запроса. Адаптеры можно настроить в расчете на любой сценарий использования.

Ранее в Ember были встроенные адаптеры для сценариев использования с другими источниками данных, такими как объект localStorage и данные фикстур. Эти и другие адаптеры теперь являются дополнениями. Их можно добавлять через утилиту Ember CLI, если потребуется синхронизировать данные модели с другими источниками.

Если появилась необходимость создать адаптер, загляните в документацию, чтобы найти ответы на свои вопросы. Некоторые из заслуживающих упоминания методов и свойств: `ajaxOptions`, `ajaxError`, `handleResponse` и `headers`.

Пока наше приложение выполняет запросы к серверу для получения набора очевидцев. Прежде чем перейти к изучению политик безопасности контента, сериализаторов и преобразований, мы воспользуемся методом `this.store.findAll` для извлечения всех записей `cryptid` из API. Поскольку шаблона представления для `cryptids` нет, будем исследовать возвращаемые записи с помощью Ember Inspector.

Добавьте вызов этого метода в `app/routes/cryptids.js`:

```
import Ember from 'ember';

export default Ember.Route.extend({
  model(){
    return this.store.findAll('cryptid');
  }
});
```

Теперь, когда данные возвращаются, перезагрузите страницу приложения в браузере и перейдите по адресу http://localhost:4200/cryptids. Воспользуйтесь Ember Inspector, чтобы изучить возвращенные данные: в DevTools выберите вкладку Ember и щелкните на Data, а затем на cryptid(4) (рис. 22.4).

До сих пор не было необходимости в продвинутом использовании Ember Inspector. Мы просто продемонстрировали, как можно исследовать находящиеся в оперативной памяти данные. Ember извлекает данные из API и заполняет объект `store` правильными данными модели. При решении проблемных вопросов с данными

модели отслеживание пути запроса может оказаться утомительным делом. Именно к Ember Inspector вам следует обратиться в первую очередь при решении подобной проблемы.

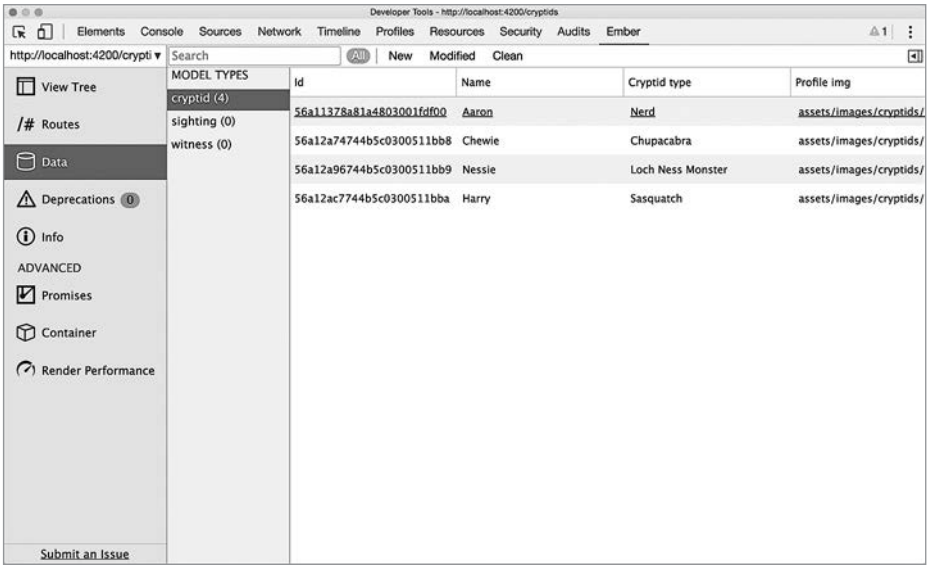


Рис. 22.4. Данные cryptid

Политика обеспечения безопасности контента

Фреймворк Ember использует новый слой безопасности JavaScript для обнаружения кросс-доменных запросов до того, как они достигнут вашего сервера. Соответствующий рабочий стандарт носит название «Политика безопасности контента» (Content Security Policy). В Ember CLI имеется объект `contentSecurityPolicy` для добавления соответствующей информации. Настройки по умолчанию весьма жесткие, когда речь идет о запросе данных, сценариев, изображений, стилей и других типов файлов извне домена вашего приложения.

Существует дополнение для задания некоторых настроек по умолчанию и интеграции политики безопасности в приложение: `ember-cli-content-security-policy`. Для приложения Tracker оно нам не понадобится, но полезно знать о его существовании. Это дополнение облегчает добавление переменных среды для настройки политики безопасности. Объект политики безопасности работает со спецификацией браузера под названием `content-security-policy`. Эта спецификация имеется в некоторых наиболее современных браузерах и представляет собой стандарт, разработанный для предотвращения создания кросс-доменных сценариев и иных атак с внедрением кода в результате выполнения в приложении вредоносного контента.

Вот пример объекта `contentSecurityPolicy`:

```
module.exports = function(environment) {
  ...
  // config/environment.js
  ENV.contentSecurityPolicy = {
    'default-src': "",
    'script-src': "",
    'font-src': "",
    'connect-src': "",
    'img-src': "",
    'style-src': "",
    'media-src': null
  }
  ...
}
```

Каждая строка политики безопасности создает *белый список* — набор безопасных путей для каждого типа запросов. `default-src` — настройка для всех прочих типов, изначально равная `null`, чтобы заставить программистов заносить в белые списки те настройки, которые им нужны. Остальные настройки, например `script-src` и `connect-src`, предназначены для внешних по отношению к домену приложения запросов, таких как https://bnr-tracker-api.herokuapp.com.

Для получения более детальной информации обратитесь к странице Content Security Policy в MDN и репозиторию GitHub дополнения Ember CLI.

Сериализаторы

При поступлении и отправке данных структура JSON сериализуется и десериализуется. Адаптер использует сериализатор для передачи данных из хранилища и в него, а также для создания и разрешения запросов и ответов.

Создать сериализатор можно с помощью команды `ember g serializer [`*`имя приложения или модели`*`]`. В результате появится сериализатор с шаблонным кодом, например:

```
import DS from 'ember-data';

export default DS.JSONAPISerializer.extend({
});
```

Сериализатор — тот объект, который присваивается свойству `serializer` адаптера. При отсутствии конкретного файла сериализатора в приложении фреймворк Ember использует адаптер и сериализатор по умолчанию — `JSONAPIAdapter` и `JSONAPISerializer` соответственно.

При включении нового сериализатора в приложение он станет сериализатором по умолчанию (`defaultSerializer`) для соответствующего файла `app/adapters/`

application.js. Используя имя модели в качестве параметра команды ember g serializer, можно настраивать под себя сериализацию данных для конкретной модели.

Как и для объекта JSONAPIAdapter, менять конфигурацию придется только в случае, когда API не соответствует спецификации JSON API или в нестандартных ситуациях. Если вам понадобится в проекте менять данные запроса или ответа, посмотрите в документации библиотеки Ember Data следующие методы: keyForAttribute, keyForRelationship, modelNameFromPayloadKey и serialize.

Метод keyForAttribute предназначен для преобразования имен атрибутов модели в имена ключей, отправляемых в запросе. Он принимает на входе три аргумента: key, typeClass и method. В случае JSONAPISerializer этот метод возвращает key отформатированным с помощью дефисов. Это значит, что все подчеркивания и так называемый верблюжий регистр будут преобразованы в дефисы. Например, если в модели было свойство с именем first_name, то в объекте запроса будет ключ first-name. Если ваш API ожидает на входе first_name, вам понадобится внести изменения в метод keyForAttribute, чтобы решить проблему с наименованием.

Метод keyForRelationship делает то же самое, только для ключей отношений. Если имя модели содержит подчеркивания, необходимо отредактировать этот метод для объекта JSONAPISerializer в случае, когда модели связаны отношениями belongsTo или hasMany. Некоторые API предполагают, что отношения будут содержать суффикс _id или _ids. Именно с помощью указанного метода можно это поменять.

Применяемый для приложения Tracker bnr-tracker-api использует соответствующие ключи JSON API с дефисом, например cryptid-type. Добавлять сериализатор в приложение не потребуется. Однако в качестве примера рассмотрим использование вспомогательного метода фреймворка Ember под названием Ember.String.underscore для изменения ключей атрибутов входящих и исходящих JSON-данных:

```
import Ember from 'ember';
import DS from 'ember-data';
var underscore = Ember.String.underscore;
export default DS.JSONAPISerializer.extend({
  keyForAttribute(attr) {
    return underscore(attr);
  },
  keyForRelationship(rawKey) {
    return underscore(rawKey);
  }
});
```

Фреймворк Ember предоставляет в объекте Ember.String множество методов работы со строками. Метод Ember.String.underscore преобразует строку с тире или верблюжьим регистром, используя подчеркивания для разделения слов.

На протяжении жизненного цикла запроса данных вызываются методы сериализатора, например this.store.findAll('witness'). Один из способов исследования

потока обратных вызовов запроса данных — добавление в этот код оператора `debugger`, позволяющего видеть, что поступает в метод и что из него возвращается.

Приведем пример:

```
import Ember from 'ember';
import DS from 'ember-data';

var underscore = Ember.String.underscore;

export default DS.JSONAPISerializer.extend({
  keyForAttribute(attr) {
    let returnValue = underscore(rawKey);
    debugger;
    return returnValue;
    return underscore(rawKey);
  },
  keyForRelationship(rawKey) {
    return underscore(rawKey);
  }
});
```

При работе с новым API такой стиль отладки может оказаться весьма удобным. Когда вы обратитесь к аргументу `attr` для сериализации имен ключей, у вас будет доступ к объекту `Ember.String`. К счастью, Ember и сообщество разработчиков Ember создали адаптеры и сериализаторы для многих паттернов API.

Преобразования

Библиотека Ember Data предоставляет нам возможность преобразовывать поступающие из API данные, требуемые для работы нашего приложения. Вы уже видели в главе 21, что в библиотеке Ember Data имеются встроенные преобразования — методы `DS.attr('string')`, `DS.attr('boolean')`, `DS.attr('number')` и `DS.attr('date')`.

При добавлении преобразования в приложение можно вызвать метод `DS.attr` с нужным типом атрибута. Преобразования подобны приведению типов JavaScript — они принимают на входе значение и возвращают значение в виде заданного типа.

Вот пример простейшего преобразования `DS.attr('object')`:

```
export default DS.Transform.extend({
  deserialize(value) {
    if (!Ember.$.isPlainObject(value)) {
      return {};
    } else {
      return value;
    }
  },
```

```
  serialize(value) {
    if (!Ember.$.isPlainObject(value)) {
      return {};
    } else {
      return value;
    }
  }
});
```

У этого преобразования имеются два метода: `deserialize` и `serialize`. Первый проверяет, являются ли входные данные объектом, и возвращает его. В противном случае он возвращает пустой объект. Второй метод возвращает исходящие данные, если они представляют собой объект, в противном случае возвращает пустой объект. Преобразование гарантирует, что возвращаемые из API и отправляемые в API данные относятся к описанному в модели типу.

Для самых любознательных: дополнение Ember CLI Mirage

Слабое звено приложений клиентский части — API. Вы можете столкнуться с разными проблемами: API может еще не быть создано, API может находиться в разработке, API может находиться за брандмауэром во время разработки...

Простейшее решение в случае недоступности API — использовать статический текст или данные фикстур. Но это может привести к появлению других проблем, например необходимости изменения логики приложения для учета недоступности данных, изменения запросов к объекту `this.store`, изменения адаптера или сериализатора.

Дополнение Ember CLI Mirage выступает в качестве прокси для запросов к заданным маршрутам API. Оно позволяет настраивать модели для формирования отношений, настраивать фабрики для задания первоначальных значений данных, создавать данные фикстур для конкретных ответов и описывать маршруты CRUD для перехвата конкретных запросов к API. После этих настроек можно вести разработку, словно с API все в порядке.

После запуска дополнения Mirage все запросы будут перенаправляться к локальному Mirage.

На момент выхода данной книги текущая версия ember-cli-mirage — 0.2.0-beta.8. Очень скоро вы будете исследовать Mirage в качестве упражнения. Чтобы узнать об Ember CLI Mirage больше, посетите сайт www.ember-cli-mirage.com.

В следующей главе мы соберем воедино все понятия из последних трех глав и создадим работающее приложение, маршрутизирующее запросы и отображающее данные. В главе 24 мы будем создавать, редактировать и удалять контент.

Вы оцените мощь библиотеки Ember Data, адаптеров и сериализаторов, когда будете вызывать для моделей методы `save` и `destroyRecord`, ведь после этого данные без всяких проблем будут отправляться в API.

Серебряное упражнение: безопасность контента

Добавление слоев безопасности в приложение — весьма важная задача. Как уже упоминалось, существует новый API браузера для политики безопасности контента и у фреймворка Ember имеется объект среды для конфигурирования политики белых списков приложения. Установите это дополнение и просмотрите ошибки в консоли, чтобы убедиться, что внешние конечные точки для запросов приложения добавлены в политику.

Золотое упражнение: Mirage

Ember CLI Mirage — великолепное дополнение к нашему арсеналу разработки. Оно предоставит API для приложения до того, как команда, занимающаяся прикладной частью, завершит разработку своего стека.

Установите ember-cli-mirage из терминала, чтобы приступить к его использованию:

```
ember install ember-cli-mirage
```

Для включения и отключения Mirage добавьте переменную среды в файл `config/environment.js`:

```
if (environment === 'development') {
  // ENV.APP.LOG_RESOLVER = true;
  // ENV.APP.LOG_ACTIVE_GENERATION = true;
  // ENV.APP.LOG_TRANSITIONS = true;
  // ENV.APP.LOG_TRANSITIONS_INTERNAL = true;
  // ENV.APP.LOG_VIEW_LOOKUPS = true;
  ENV['ember-cli-mirage'] = {
    enabled: true
  }
}
```

И наконец, добавьте фиктивные данные для конечных точек очевидцев и криптид в виде *фабрик*.

Для работы с приложением Tracker можете извлечь уже сконфигурированный каталог `app/mirage` из предоставленной нами библиотеки примеров в каталоге `Tracker/Data_Chapter/mirage-example`. Мы будем обновлять этот пример, чтобы он соответствовал текущей версии дополнения.

23 Представления и шаблоны

Буква V в аббревиатуре MVC означает *представления*. В приложении Tracker представления будут шаблонами. Шаблоны обрабатываются с помощью JavaScript для создания элементов HTML. Это дает нам возможность менять DOM без отправки новых запросов к серверу.

В данной главе мы будем создавать файлы шаблонов и добавлять данные, извлеченные в точке подключения `model` маршрута. Язык шаблонов и встроенные в Ember вспомогательные функции дают возможность создавать шаблоны с минимальными отклонениями от обычного синтаксиса HTML.

К концу данной главы мы создадим списки, подобные показанным на рис. 23.1.

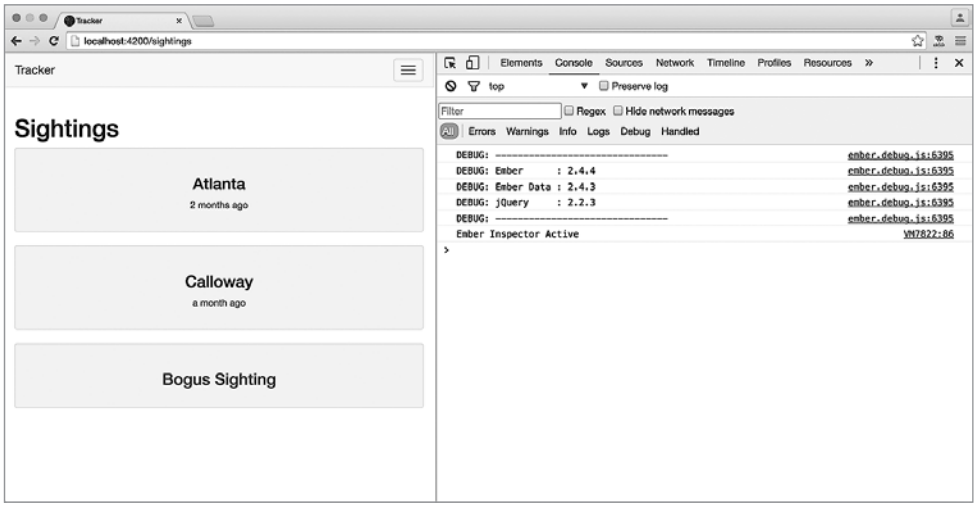


Рис. 23.1. Список наблюдений

Handlebars

Мы будем использовать Handlebars — обладающий широкими возможностями язык для создания динамических шаблонов. Он похож на серверные языки шаблонизации, такие как PHP, JSP, ASP и ERB. Он включает теги для элементов HTML и разделители для работы с объектами данных.

В языке Handlebars разделителями являются двойные фигурные скобки `{{}}`. Внутри них можно визуализировать строки данных и выполнять логику с помощью вспомогательных методов. Мы уже видели два подобных вспомогательных метода: `{{outlet}}` и `{{#each}}`.

Реализация языка Handlebars фреймворка Ember недавно перешла на новый механизм под условным названием HTMLBars. Некоторые детали языка в этой главе взяты из HTMLBars, но совместимы и с более старыми версиями Ember (по крайней мере вплоть до версий 1.13.x).

Модели

В фреймворке Ember за шаблонами всегда стоят модели. Это значит, что объект (или массив объектов) передается в качестве аргумента при визуализации шаблона в виде строки HTML и присоединяется к DOM.

У объекта модели могут быть свойства со строками, массивами и другими объектами. При написании шаблонов с помощью фреймворка Ember и языка Handlebars обращение к этому объекту производится с указанием двойных фигурных скобок.

Если нужно отобразить значение свойства модели, можно написать `{{model.name}}`, где `name` — свойство объекта `model`. Синтаксис с использованием точки должен быть вам уже знаком, но не думайте, что внутри фигурных скобок может находиться какой-либо JavaScript-код.

Вспомогательные методы

Шаблоны Handlebars представляют собой строки, дополняемые JavaScript-функциями. Когда функция встречает двойные фигурные скобки, она пытается получить экземпляр разделителя с помощью свойства объекта или вызвать вложенную функцию, возвращающую строку. Эти вложенные функции носят название *вспомогательных методов* и создаются в приложении. Существует несколько встроенных в библиотеку Handlebars вспомогательных методов, к которым Ember добавляет парочку своих.

Вспомогательные методы могут принимать две формы. Первая — встраиваемые вспомогательные методы, использующие синтаксис `{{[`*`имя вспомогательного метода`*`] [`*`аргументы`*`]}}`. Аргументы могут включать необязательные параметры, например:

```
{{input type="text" value=firstName disabled=entryNotAllowed size="50"}}
```

Для более сложных вспомогательных методов характерен блочный синтаксис:

```
{{#[имя вспомогательного метода] [аргументы]}}
  [контент блока]
{{/[имя вспомогательного метода]}2}
```

Например, если необходимо отображать ссылку для входа в приложение только тем пользователям, которые еще не выполнили вход, можно использовать данный блок:

```
{{#if notSignedIn}}
  <a href="/">Sign In</a>
{{/if}}
```

В случае блочных вспомогательных методов можно передавать контент в блок для дополнения вывода динамическими сегментами.

Встроенные условные операторы языка Handlebars, описанные в следующем разделе, представляют собой блочные вспомогательные методы.

Мы будем использовать вспомогательные методы для визуализации разделов наших шаблонов, касающихся наблюдений и криптид, а также для визуализации NavBar.

Условные операторы

Условные операторы предоставляют возможность включать в шаблоны языка Handlebars простейшие потоки команд. Их синтаксис выглядит следующим образом:

```
{{#if аргумент}}
  [визуализация контента блока]
{{else}}
  [визуализация прочего контента]
{{/if}}
```

Или в качестве альтернативы:

```
{{#unless аргумент}}
  [визуализация контента блока]
{{/unless}}
```

Условные операторы принимают на входе один аргумент, получающий истинное или ложное значение (под *истинным* мы понимаем значение, вычисление которого дает `true` в булевом контексте. Истинны все значения, кроме определенных как *ложные*: значение `false`, `0`, `""`, `null`, `undefined` и `NaN`).

Время приступить к работе. Откройте шаблон `app/templates/sightings/index.hbs` и добавьте условный оператор, чтобы записи наблюдений отображали или их место, или (если данных о месте нет) предупреждение об отсутствии данных:

```
<div class="panel panel-default">
  <ul class="list-group">
    {{#each model as |sighting|}}
```

```
      <li class="list-group-item">
        {{sighting.location}} - {{sighting.sightedAt}}
      </li>
    {{/each}}
  </ul>
</div>
<div class="row">
  {{#each model as |sighting|}}
    <div class="col-xs-12 col-sm-3 text-center">
      <div class="media well">
        <div class="caption">
          {{#if sighting.location}}
            <h3>{{sighting.location}} - {{sighting.sightedAt}}</h3>
          {{else}}
            <h3 class="text-danger">Bogus Sighting</h3>
          {{/if}}
        </div>
      </div>
    </div>
  {{/each}}
</div>
```

Мы поменяли структуру DOM нашего шаблона наблюдений, чтобы информация о наблюдениях выводилась в виде элементов, стилизованных с помощью wells-стиля фреймворка Bootstrap. Использование `{{#if}}` и `{{else}}` позволяет визуализировать различные HTML в случае, когда место наблюдения не было указано.

Теперь нам необходимо изменить данные, отправляемые из маршрута в шаблон, чтобы увидеть результаты работы нашего нового условного оператора. Задайте в модели маршрута наблюдений в файле `app/routes/sightings.js` свойство `location` одного из наблюдений равным пустой строке:

```
...
  model(){
    ...
    let record3 = this.store.createRecord('sighting', {
      location: 'Asilomar',
      sightedAt: new Date('2016-03-21')
    });

    return [record1, record2, record3];
  }
});
```

Запустите сервер и перейдите в браузере по веб-адресу `http://localhost:4200/sightings`, чтобы увидеть новый список наблюдений (рис. 23.2).

Этот условный оператор вычисляет истинное значение пустой строки для последнего экземпляра наблюдения. Таким образом, шаблон визуализирует контент блока в виде текста Bogus sighting (Фиктивное наблюдение).

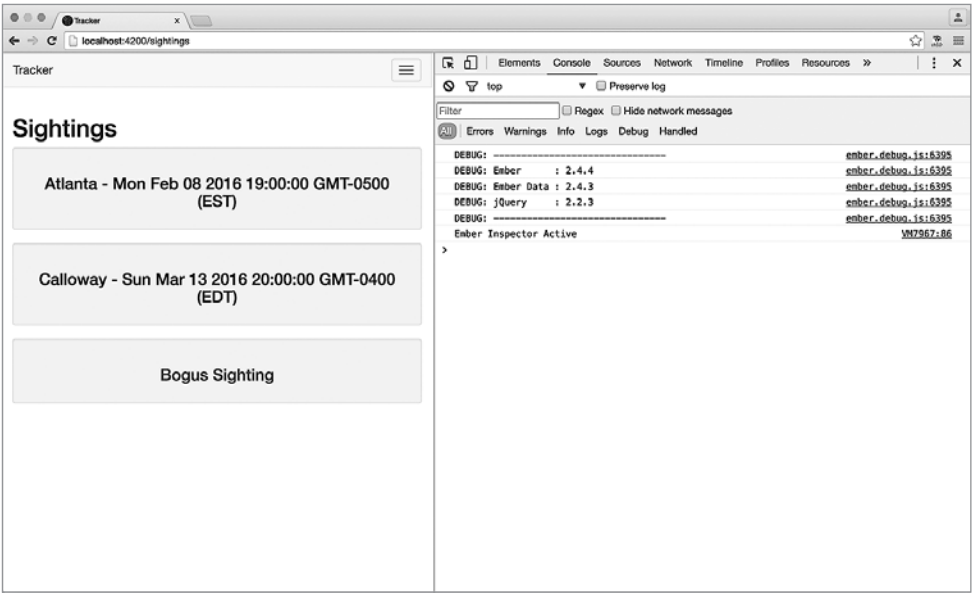


Рис. 23.2. Фиктивное наблюдение

Организация циклов с помощью {{#each}}

Мы уже использовали блочный вспомогательный метод `{{#each}}` в индексном шаблоне. Аргумент для этого вспомогательного метод — `array` в виде `|`*`существо`*`|`, содержащегося в аргументе блока. Блок будет визуализироваться только тогда, когда передаваемый в `{{#each}}` аргумент представляет собой массив, содержащий хотя бы один элемент.

Аналогично блочному вспомогательному методу `{{#if}}` данный вспомогательный метод поддерживает блок `{{else}}` для визуализации в случае пустого аргумента-массива.

Воспользуемся `{{#each}} {{else}} {{/each}}` для создания списка всех зафиксированных криптид или (если их нет) текста `No Creatures` (Существ не найдено) в шаблоне `app/templates/cryptids.hbs`:

```
~~{{outlet}}~~
<div class="row">
  {{#each model as |cryptid|}}
    <div class="col-xs-12 col-sm-3 text-center">
      <div class="media well">
        <div class="caption">
          <h3>{{cryptid.name}}</h3>
        </div>
      </div>
```

```
    </div>
  {{else}}
    <div class="jumbotron">
      <h1>No Creatures</h1>
    </div>
  {{/each}}
</div>
```

Аналогично наблюдениям мы выводим список криптидов со стилем `wells`. Блок `{{else}}` предоставляет нам возможность включать в шаблон условие на случай отсутствия элементов в выводимом списке. Наше приложение может визуализировать какой-нибудь другой элемент при выполнении этого условия для пустого массива модели.

Перейдите по адресу http://localhost:4200/cryptids, чтобы посмотреть на новый список криптидов (рис. 23.3).

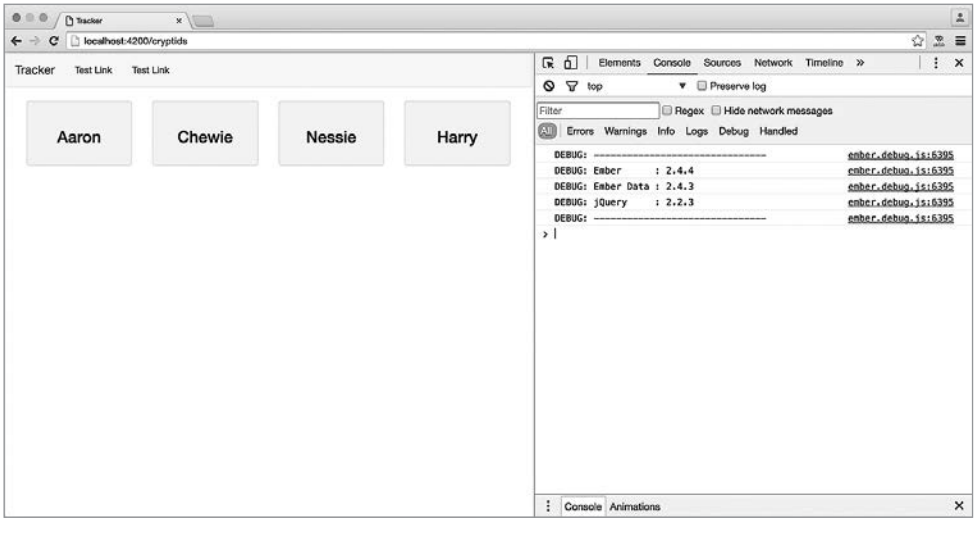


Рис. 23.3. Список криптидов

Теперь можете удалить из файла `app/routes/cryptids.js` обратный вызов метода `model`, закомментировав оператор `return`, чтобы выполнился блок `else` условного оператора:

```
...
  model(){
    // return this.store.findAll('cryptid');
  }
});
```

Обновите страницу браузера http://localhost:4200/cryptids. Список криптидов теперь пуст (рис. 23.4).

Рис. 23.4. Пустой список криптидов

С помощью `{{each}} {{else}} {{/each}}` можно создавать условные представления на основе наличия/отсутствия данных и простой условной логики. Теперь, когда вы уже увидели (и протестировали) чудесные возможности условных итераторов, верните файл `app/routes/cryptids.js` в исходное состояние:

```
...
  model(){
    // return this.store.findAll('cryptid');
  }
});
```

Привязка атрибутов элементов

Значения атрибутов элементов можно визуализировать из свойств контроллеров аналогично тому, как контент элементов визуализируется между тегами элементов DOM. В предыдущих версиях фреймворка Ember существовал вспомогательный метод `{{bind-attr}}` для привязки атрибутов. Теперь же благодаря HTMLBars можно просто использовать `{{}}` для привязки свойства к атрибуту.

Привязка атрибутов часто производится с такими свойствами элементов, как `class` и `src`. Среди данных криптидов имеется путь к изображению, так что мы можем динамически привязывать их свойство `src` к `model`.

Добавьте изображение в список криптидов в шаблоне `app/templates/cryptids.hbs`:

```
<div class="row">
  {{#each model as |cryptid|}}
    <div class="col-xs-12 col-sm-3 text-center">
      <div class="media well">
        <img class="media-object thumbnail" src="{{cryptid.profileImg}}"
          alt="{{cryptid.name}}" width="100%" height="100%">
```

```
        <div class="caption">
          <h3>{{cryptid.name}}</h3>
        </div>
      </div>
    </div>
  {{else}}
    <div class="jumbotron">
      <h1>No Creatures</h1>
    </div>
  {{/each}}
</div>
```

Нам понадобится добавить изображения криптидов из наших ресурсов в каталоге `tracker/public/assets/image/cryptids`. При работе сервера Ember каталог `public` является корневым каталогом ресурсов. Для находящегося в широкой эксплуатации приложения может потребоваться настроить эти пути под себя, но для целей разработки путь `public/assets` вполне подойдет. Эти файлы копируются в каталог `dist`, в котором приложение компилируется и из которого выдается браузеру.

При развертывании приложения на сервере и добавлении изображений к хранимым данным необходимо четко знать фактический путь к файлу изображения. В нашем примере изображения выдаются из того же каталога, что и приложение, и в базе данных хранится относительный путь изображения по отношению к нашему приложению.

До появления HTMLBars можно было использовать вспомогательный метод `{{bind-attr}}` в качестве встраиваемой тернарной операции для присваивания свойств в зависимости от значения булева свойства. Сейчас же можно применять встраиваемый вспомогательный метод `{{if}}`. Распространенным приемом является наличие в UI стилей для представления истинного и ложного состояний конкретного свойства.

Воспользуемся тернарной формой в шаблоне `app/templates/cryptids.hbs` для обработки случая отсутствующих изображений:

```
<div class="row">
  {{#each model as |cryptid|}}
    <div class="col-xs-12 col-sm-3 text-center">
      <div class="media well">
        <img class="media-object thumbnail" src="{{cryptid.profileImg}}"
          src="{{if cryptid.profileImg cryptid.profileImg
          'assets/images/cryptids/blank_th.png'}}"
          alt="{{cryptid.name}}" width="100%" height="100%">
        <div class="caption">
        ...
```

В отличие от блочного вспомогательного метода `{{#if}}`, встраиваемый вспомогательный метод `{{if}}` не выдает блочного контента. Встраиваемый вспомогательный метод рассматривает и вычисляет первый аргумент как булев, после чего выводит второй или третий аргумент.

В данном случае мы вычисляем, истинен или ложен первый аргумент `{{cryptid.profileImg}}`. Если он истинен, выводим путь к изображению криптида. В противном случае используется указанное изображение-заполнитель.

В качестве аргумента для всех встраиваемых вспомогательных методов можно указывать динамическое значение. Можно также передавать в качестве аргумента значения любых простых типов данных, например строк, чисел или булевых значений.

Прежде чем оценить результаты работы нашего условного оператора, создайте криптид без пути изображения в точке подключения `beforeModel` в файле `app/routes/cryptids.js`:

```
import Ember from 'ember';

export default Ember.Route.extend({
  beforeModel(){
    this.store.createRecord('cryptid', {
      "name": "Charlie",
      "cryptidType": "unicorn"
    });
  },
  model(){
    return this.store.findAll('cryptid');
  }
});
```

Теперь обновите страницу http://localhost:4200/cryptids и проверьте, как выводятся новые изображения (рис. 23.5).

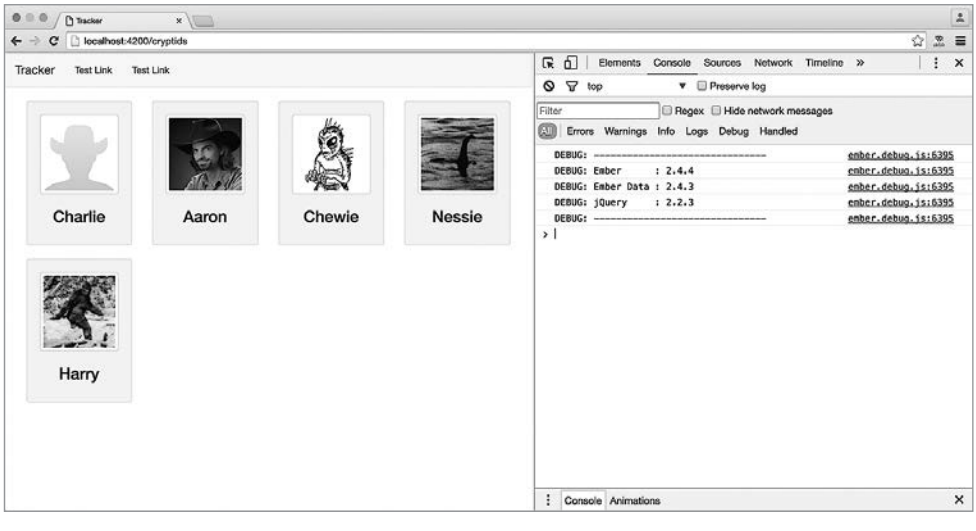


Рис. 23.5. Список криптидов с изображениями

Ссылки

Как обсуждалось в главе 20, маршрутизация уникальна для браузерных приложений. Ember прослушивает пару точек подключения для событий в целях управления маршрутизацией в приложении. Поэтому следует создавать ссылки с помощью блочных вспомогательных методов `{{#link-to}}`. Этот вспомогательный метод принимает на входе маршрут (в виде строки) в качестве первого аргумента для создания элемента-якоря. Например, `{{#link-to 'index'}}Home{{/link-to}}` создаст ссылку на корневую индексную страницу.

Чтобы увидеть, как это работает, изменим нашу основную навигационную панель, используя вспомогательные методы `{{#link-to}}`.

Начнем вносить изменения с шаблона `app/templates/application.hbs`. Замените тестовые ссылки компонента NavBar ссылками на наши наблюдения, криптиды и очевидцев:

```
...
      <div class="collapse navbar-collapse" id="top-navbar-collapse">
        <ul class="nav navbar-nav">
          <li>
            <a href="#">Test Link</a>
          </li>
          <li>
            <a href="#">Test Link</a>
          </li>
          <li>
            {{#link-to 'sightings'}}Sightings{{/link-to}}
          </li>
          <li>
            {{#link-to 'cryptids'}}Cryptids{{/link-to}}
          </li>
          <li>
            {{#link-to 'witnesses'}}Witnesses{{/link-to}}
          </li>
        </ul>
      </div><!-- /.navbar-collapse -->
```

Теперь, когда есть ссылки на страницы списка, можно перезагрузить приложение и протестировать их. Пощелкайте на них. Нажмите кнопку возврата. Наше веб-приложение работает! Отпразднуем это.

Следующая задача — сделать так, чтобы ссылки с изображений на странице криптидов вели на соответствующие конкретные страницы существ. Вспомогательный метод `{{#link-to}}` может принимать несколько аргументов для настройки ссылки, так что воспользуемся этим.

В файле `app/templates/cryptids.hbs` оберните тег `<img>` в ссылку на конкретного криптида, использовав в качестве второго аргумента `cryptid.id`:

```
...
    <div class="media well">
      {{#link-to 'cryptid' cryptid.id}}
        <img class="media-object thumbnail"
        src="{{if cryptid.profileImg cryptid.profileImg
        'assets/images/cryptids/blank_th.png'}}"
        alt="{{cryptid.name}}" width="100%" height="100%">
      {{/link-to}}
      <div class="caption">
        <h3>{{cryptid.name}}</h3>
      </div>
    </div>
...
```

Теперь, когда у нас есть ссылки на маршрут `cryptid`, а в якоре имеется путь к `cryptids/[cryptid_id]`, необходимо отредактировать файл `router.js`, чтобы `CryptidRoute` ожидал на входе динамическое значение:

```
...
Router.map(function() {
  this.route('sightings', function() {
    this.route('new');
  });
  this.route('sighting', function() {
    this.route('edit');
  });
  this.route('cryptids');
  this.route('cryptid', {path: 'cryptids/:cryptid_id'});
  this.route('witnesses');
  this.route('witness');
});
...
```

Проверьте на практике. После щелчка на любом изображении криптидов вы должны увидеть пустую страницу. Это хорошо. Приложение маршрутизирует вас на `CryptidRoute`, который визуализирует шаблон `app/templates/cryptid.hbs` (единственное число). Этот файл пока что пуст.

Если вы щелкнете на изображении единорога Чарли, то, вероятно, будет выдана ошибка. Дело в том, что его запись была создана в точке подключения `beforeModel` и у нее отсутствует идентификатор записи `id`. Соответственно, происходит попытка передать вспомогательному методу `{{#link-to}}` значение, равное `null`.

Самое время удалить эту точку подключения `beforeModel`. Творчество лучше отложить до работы над страницами создания новых криптидов, о которых мы расскажем в главе 24. Удалите эту точку подключения из файла `app/routes/cryptids.js`:

```
...
  beforeModel(){
    this.store.createRecord('cryptid', {
      "name": "Charlie",
```

```
      "cryptidType": "unicorn"
    });
  },
  model(){
    return this.store.findAll('cryptid');
  }
...
```

Далее добавьте запрос данных криптида в файл `app/routes/cryptid.js` (снова единственное число):

```
import Ember from 'ember';
export default Ember.Route.extend({
  model(params){
    return this.store.findRecord('cryptid', params.cryptid_id);
  }
});
```

В качестве аргумента точке подключения `model` маршрута передается динамический параметр `cryptid_id`. Он используется для вызова метода `findRecord` хранилища `store`.

Теперь отредактируйте шаблон для отдельного криптида `app/templates/cryptid.hbs`, чтобы отображать его изображение и название:

```
{{outlet}}
<div class="container text-center">
  <img class="img-rounded" src="{{model.profileImg}}" alt="{{model.type}}">
  <h3>{{model.name}}</h3>
</div>
```

Передаваемый в этот шаблон `model` — отдельный объект, а не массив объектов, по которому можно пройти в цикле. Метод `this.store.findRecord` возвращает отдельный экземпляр `cryptid`. В шаблоне этим экземпляром является `model`, а свойства извлекаются с помощью `{{model.[property-name]}}`.

Воспользуйтесь компонентом NavBar в браузере для перехода к маршруту `Cryptids`, после чего щелкните на одном из изображений криптидов для просмотра страницы с подробной информацией о нем (рис. 23.6).

Мы изучим `{{#link-to}}` подробнее в следующих главах. Помните, что вспомогательные методы — это функции, вызываемые при визуализации шаблона. Вместе с фреймворком Ember поставляются некоторые из них, но ничто не ограничивает вашу работу лишь встроенными вспомогательными методами.

Пользовательские вспомогательные методы

Свойство `sightedAt` наблюдения отображается в виде неприглядной строки с неформатированной датой. Для приведения дат к более понятному виду мы воспользуемся той же библиотекой, что и в приложении Chattrbox.

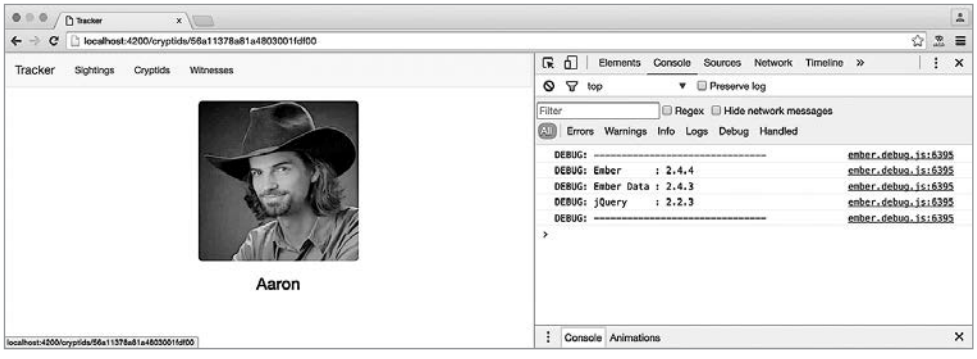


Рис. 23.6. Страница с подробной информацией о криптиде

Добавьте из терминала модуль `moment`:

```
bower install moment --save
```

Далее воспользуйтесь оператором `app.import`, чтобы добавить модуль `moment` в ресурсы сторонних поставщиков в файле `ember-cli-build.js`:

```
...
  // Добавление в приложение ресурсов с помощью import
  app.import(bootstrapPath + 'javascripts/bootstrap.js');
  app.import('bower_components/moment/moment.js');

  return app.toTree();
};
```

Изменения в конфигурации сервера требуют перезапуска, так что, завершив работу с файлом `ember-cli-build.js`, остановите сервер Ember (Control+C) и запустите его снова (`ember server`):

Теперь вам необходимо из терминала сгенерировать модуль для вспомогательного метода:

```
ember g helper moment-from
```

Там вы создадите функцию, которая будет возвращать HTML в виде строки. У функции будет аргумент-дата, и она станет работать с этой датой с помощью библиотеки `moment.js`. Мы окружим дату HTML-тегом `<span>` и применим к данному элементу одну из утилит Bootstrap для текста:

```
import Ember from 'ember';

export function momentFrom(params/*, hash*/) {
  return params;
}
export function momentFrom(params) {
  var time = window.moment(...params);
```

```
  var formatted = time.fromNow();
  return new Ember.Handlebars.SafeString(
    '<span class="text-primary">'
    + formatted + '</span>'
  );
}

export default Ember.Helper.helper(momentFrom);
```

Теперь, создав вспомогательный метод, воспользуйтесь им в шаблоне `app/templates/sightings/index.hbs`:

```
...
          {{#if sighting.location}}
            <h3>{{sighting.location}} - {{sighting.sightedAt}}</h3>
            <p>{{moment-from sighting.sightedAt}}</p>
          {{else}}
...
```

Вспомогательный метод `moment-from` принимает на входе один аргумент и возвращает отформатированную дату в виде строки HTML. При использовании пользовательского вспомогательного метода для визуализации HTML-элементов фреймворк Ember предоставляет возможность применить метод `Ember.Handlebars.SafeString` для вывода чистой разметки.

Посмотрите в браузере на новые отформатированные даты (рис. 23.7).

Библиотека `moment.js` принимает на входе объект даты и форматирует ее во что-то вроде `2 months ago` (2 месяца назад). Это производит гораздо более приятное впечатление, чем `Tue Feb 9 2016 19:00:00 GMT-0500 (EST)` (в `moment.js` есть множество параметров форматирования дат — загляните на momentjs.com, чтобы узнать о них больше).

Создание вспомогательного метода для вывода конкретного текста позволяет убрать из шаблона часть логики. Пользовательские вспомогательные методы позволяют сократить дублирование и централизовать форматирование UI. Это первый шаг к абстрагированию нашего кода в Ember Components, которое мы обсудим в главе 25.

В данной главе вы научились отображать простые свойства моделей и настраивать шаблоны с помощью условных операторов и циклов. Мы связали атрибуты элементов HTML со свойствами и создали новые маршруты с динамическими атрибутами для загрузки отдельных записей из обеспечивающей шаблон данными модели. Наконец, вы воспользовались вспомогательными методами для создания ссылок на страницы и на практике изучили работу вспомогательных методов, создав собственный пользовательский вспомогательный метод для форматирования дат.

В следующей главе мы завершим жизненный цикл приложения созданием и редактированием данных с помощью контроллеров. Вы узнаете о действиях, извлечении наборов данных из хранилища данных и создании декораторов.

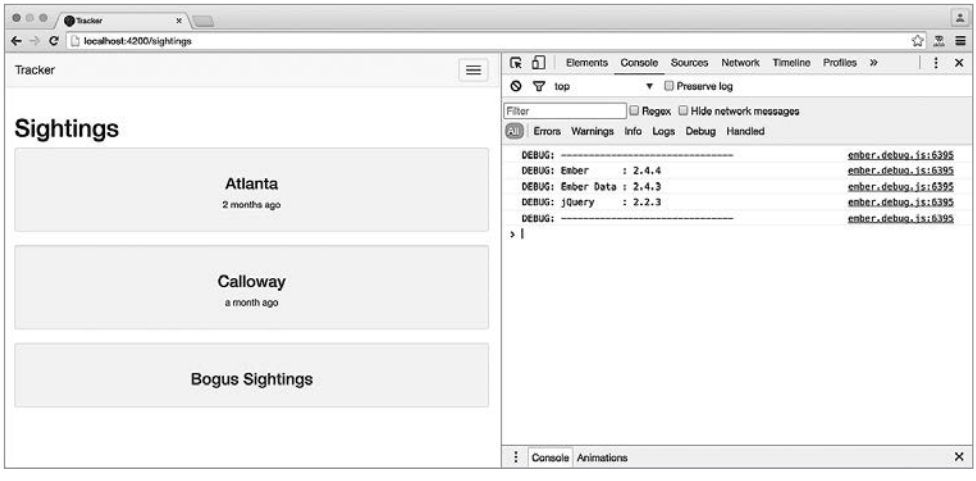


Рис. 23.7. Даты наблюдений, отформатированные с помощью moment.js

Бронзовое упражнение: добавление эффектов перекатывания для ссылок

Для созданных ссылок необходим какой-нибудь контент с эффектом перекатывания. Чтобы добиться этого, добавьте во вспомогательные методы `{{#link-to}}` атрибут `title`. Он должен быть названием наблюдаемого криптида.

Серебряное упражнение: изменение формата даты

Вспомогательный метод `{{moment-from}}` сделал даты более приятными глазу, но теперь они стали недостаточно информативными. Просмотрите документацию по методу `moment` и измените выводимую вспомогательным методом информацию, чтобы дата форматировалась следующим образом: `Sunday May 31, 2016`.

Золотое упражнение: создание пользовательского вспомогательного метода для миниатюр

Разметка для миниатюр криптидов длинновата. Создайте пользовательский вспомогательный метод для отображения миниатюр криптидов, в который бы передавался путь к изображению. Почистите свой код, использовав этот вспомогательный метод везде, где только встречаются изображения криптидов.

24 Контроллеры

Контроллеры — последняя часть паттерна MVC. Как вы уже знаете из главы 19, контроллеры содержат логику приложения, извлекают экземпляры модели и передают их представлениям, а также включают в себя функции-обработчики, выполняющие изменения в экземплярах моделей.

Контроллеры, которые мы создадим в этой главе, не будут особенно большими фрагментами кода. Именно в этом и заключается идея MVC: распределение сложности приложения соответствующим образом. Управление данными — задача моделей, а работа с пользовательским интерфейсом — сфера деятельности представлений. Контроллерам нужно всего лишь, так сказать, *управлять* моделями и представлениями.

Фреймворк Ember добавлял объекты контроллеров в приложение во время его работы, даже не ставя вас в известность. Контроллеры выступают посредниками между объектами маршрутов и шаблонами, выполняя передачу модели. Если вы не создаете объект контроллера, Ember знает, что данных модели достаточно для передачи шаблону, и делает это для вас.

Создание контроллера в Ember дает возможность задавать прослушиваемые (во время использования маршрута) события или действия, позволяет описывать свойства декораторов для пополнения модели данными, которые вы хотели бы отображать без их сохранения.

Одна из целей приложения Tracker — дать пользователю возможность создавать новые наблюдения. Для этого нам понадобится создать маршрут, контроллер, свойства контроллера и действия контроллера.

Мы уже создавали новый маршрут наблюдения `app/routes/sightings/new.js`. Далее мы создадим для этой страницы новую запись наблюдения и загрузим наборы криптидов и очевидцев. Для каждого нового наблюдения понадобится отношение принадлежности к криптиду и наличие одного или нескольких очевидцев. Форма, которую мы создадим, будет выглядеть так, как показано на рис. 24.1.

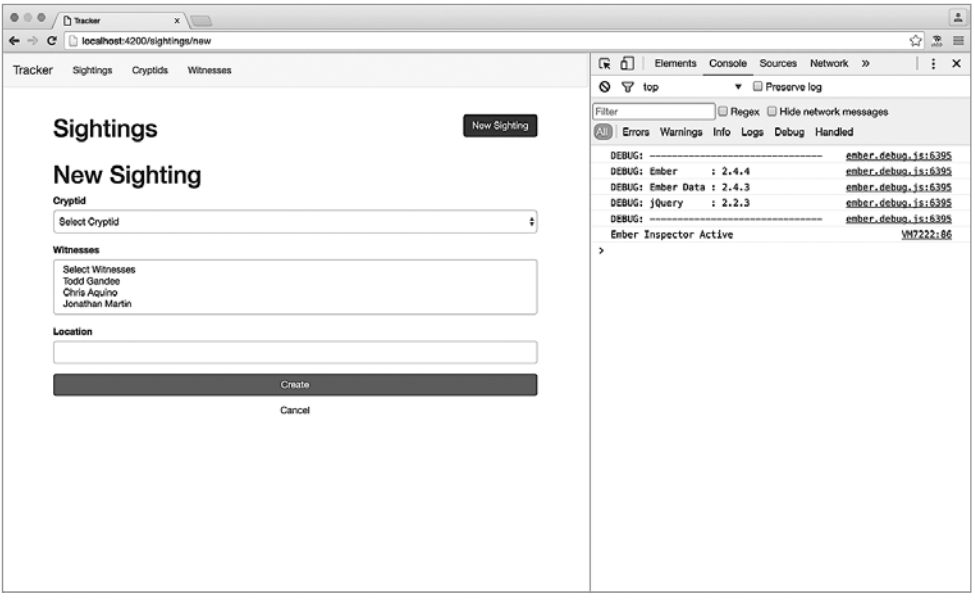


Рис. 24.1. Форма New Sighting (Новое наблюдение) приложения Tracker

Когда все это будет сделано, мы создадим контроллер для управления событиями формы нового наблюдения. Мы сделаем даже больше — обеспечим возможность редактирования и удаления существующих наблюдений.

Новые наблюдения

Настроенная нами модель маршрута `SightingsRoute` возвращает все наблюдения. А в модели нового наблюдения мы будем получать результат создания отдельного нового (пустого) наблюдения. Кроме того, мы будем возвращать набор объектов `Promise` для криптидов и очевидцев. Для этой цели мы будем использовать метод `Ember.RSVP.hash({})`.

Что ж, приступим. Откройте файл `app/routes/sightings/new.js` и добавьте точку подключения модели для возвращения набора объектов `Promise` в виде `Ember.RSVP.hash`:

```
...
export default Ember.Route.extend({
  model() {
    return Ember.RSVP.hash({
      sighting: this.store.createRecord('sighting')
    });
  }
});
```

При использовании этого маршрута возвращается новая запись наблюдения. Если бы вам пришлось вернуться на маршрут `sightings`, вы бы увидели пустой элемент, поскольку создали новую запись. Мы займемся *«грязными» записями* (данными модели, которые были изменены, но не сохранены в источнике данных) ближе к концу данной главы. Пока что достаточно знать, что метод `createRecord` добавил в локальный набор новое наблюдение.

При создании нового наблюдения нам понадобится список криптидов и очевидцев. Метод `Ember.RSVP.hash({})` используется тут, чтобы указать, что возвращается хеш объектов `Promise`. Единственный ключ — `sighting`, а значит, вашей ссылке для `model` в шаблоне необходимо выполнять поиск по свойству `model.sighting`, чтобы сослаться на созданную запись о наблюдении.

Добавьте в этот хеш методы извлечения для криптидов и очевидцев (не забудьте про запятую после вызова метода `this.store.createRecord('sighting')`):

```
...
export default Ember.Route.extend({
  model() {
    return Ember.RSVP.hash({
      sighting: this.store.createRecord('sighting'),
      cryptids: this.store.findAll('cryptid'),
      witnesses: this.store.findAll('witness')
    });
  }
}
```

Далее мы собираемся использовать теги `<select>` в нашем новом шаблоне для наблюдений, чтобы отобразить пользователю список криптидов и очевидцев. Но прежде, чем формировать шаблон с новыми данными модели, нам понадобится установить плагин Ember CLI, облегчающий использование тегов `<select>` вместе с относящимися к ним свойствами.

Установите плагин emberx-select из командной строки:

```
ember install emberx-select
```

Мы воспользуемся этим компонентом, обычно называемым `x-select`, в нашем шаблоне. Это позволит нам не создавать действия `onchange` для каждого тега `<select>`.

Перезапустите сервер (`ember server`), прежде чем использовать компонент `emberx-select`.

Теперь, подготовив все данные модели, вы можете отредактировать шаблон, `app/templates/sightings/new.hbs`, чтобы создать новую форму наблюдения:

```
<h1>New Route</h1>
<h1>New Sighting</h1>
<form>
  <div class="form-group">
```

```
    <label for="name">Cryptid</label>
    {{#x-select value=model.sighting.cryptid class="form-control"}}
      {{#x-option}}Select Cryptid{{/x-option}}
      {{#each model.cryptids as |cryptid|}}
        {{#x-option value=cryptid}}{{cryptid.name}}{{/x-option}}
      {{/each}}
    {{/x-select}}
  </div>
  <div class="form-group">
    <label>Witnesses</label>
    {{#x-select value=model.sighting.witnesses multiple=true
                class="form-control"}}
      {{#x-option}}Select Witnesses{{/x-option}}
      {{#each model.witnesses as |witness|}}
        {{#x-option value=witness}}{{witness.fullName}}{{/x-option}}
      {{/each}}
    {{/x-select}}
  </div>
  <div class="form-group">
    <label for="location">Location</label>
    {{input value=model.sighting.location
      type="text" class="form-control" name="location" required=true}}
  </div>
</form>
```

Вау! Мы добились всего, что нам было нужно, и даже больше. Маршрут использует новый метод `Ember.RSVP`, шаблон использует вспомогательные методы, кроме того, задействованы новые компоненты `{{x-select}}` и `{{x-option}}`.

Компонент `{{x-select}}` предназначен для использования элемента `<select>` в целях присваивания значения свойству. Он спроектирован таким образом, чтобы работать точно так же, как элемент `<select>` в среде связывания данных Ember. Мы задаем `value` равным свойству `cryptid` модели наблюдения — и при выборе этой новой опции компонент будет обрабатывать событие `onchange`. Это работает, поскольку свойству `cryptid` требуется запись из модели криптида в качестве значения.

Что касается очевидцев, то тут имеется дополнительный атрибут `multiple=true`, разрешающий пользователям выбирать для наблюдения нескольких очевидцев. Они будут транслироваться в группу очевидцев с отношением `hasMany`.

Прежде чем двигаться дальше, нам понадобится добавить ссылку в шаблон для маршрута наблюдений, чтобы получить путь для перехода на маршрут `sightings.new`. Откройте шаблон `app/templates/sightings.hbs` и позаботьтесь об этом:

```
~~<h1>Sightings</h1>~~
<div class="row">
  <div class="col-xs-6">
    <h1>Sightings</h1>
  </div>
  <div class="col-xs-6 h1">
    {{#link-to "sightings.new" class="pull-right btn btn-primary"}}
```

```
      New Sighting
    {{/link-to}}
  </div>
</div>
{{outlet}}
```

Для создания кнопки используются простые ссылки из Bootstrap. Загрузите http:/localhost:4200/sightings, чтобы ее увидеть (рис. 24.2).

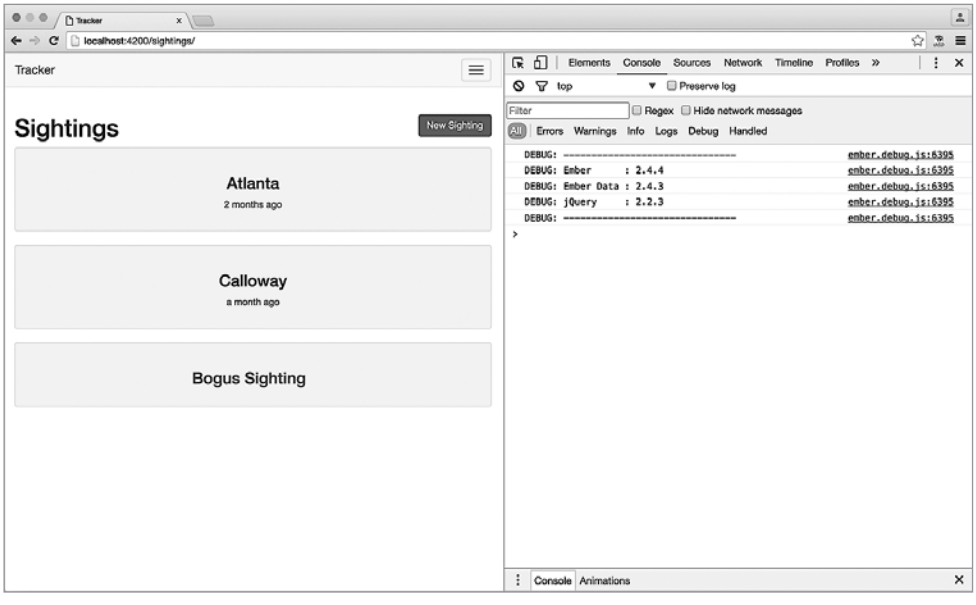


Рис. 24.2. Кнопка New Sighting (Новое наблюдение)

Теперь у вас есть возможность перейти по маршруту `sightings.new`. Новая кнопка добавляет ссылку для создания нового наблюдения из любой точки структуры дерева маршрутов наблюдений.

Обратите внимание, что, когда вы находитесь на маршруте `sightings.new`, кнопка активна. В фреймворке Ember все продумано, и ссылке присваивается класс, соответствующий активному событию, даже в том случае, когда текущий маршрут совпадает с маршрутом этой ссылки. Активное состояние кнопки или ссылки отражает текущий маршрут в виде визуальных подсказок UI.

Действия — ключ к обработке событий формы или любых других событий в приложении. Свойство `actions` — хеш, в котором выполняется присваивание функций ключам. Ключи затем используются в шаблонах для запуска обратных вызовов.

Мы готовы создать контроллер для маршрута `sightings.new` с помощью команды терминала:

```
ember g controller sightings/new
```

Ember создает файл app/controllers/sightings/new.js. Откройте его и добавьте действия create и cancel, необходимые для создания наблюдений:

```
import Ember from 'ember';

export default Ember.Controller.extend({
  actions: {
    create() {
    },
    cancel() {
    }
  }
});
```

При создании элемента формы у атрибута action обычно имеется URL, по которому нужно отправлять форму. В случае фреймворка Ember элементу формы требуется только название функции, которую следует выполнить.

Сделайте в шаблоне app/templates/sightings/new.hbs, чтобы элемент формы задавал действие для события submit. Добавьте также кнопки Create (Создать) и Cancel (Отменить):

```
<h1>New Sighting</h1>
<form {{action "create" on="submit"}}>
...
  <div class="form-group">
    <label for="location">Location</label> {{input value=model.location
      type="text" class="form-control" name="location" required=true}}
  </div>
  <button type="submit" class="btn btn-primary btn-block">Create</button>
  <button {{action 'cancel'}} class="btn btn-link btn-block">Cancel</button>
</form>
```

Редактор Atom может выдать предупреждение относительно синтаксиса строк со вспомогательными методами {{action}}. Это предупреждение можно проигнорировать. Можете также установить пакет Language-Mustache и активизировать его в настройках редактора Atom, чтобы Atom понимал такой синтаксис. Этот пакет можно найти на странице atom.io/packages/language-mustache.

Строковые аргументы двух вспомогательных методов {{action}} соответствуют созданным вами в файле app/controllers/sightings/new.js действиям. Действия привязаны к обработчикам событий. У вспомогательного метода {{action}} имеется аргумент on, соответствующий событию, для которого назначено данное действие. Если аргумент on во вспомогательный метод не добавлен, действию назначается прослушиватель события щелчка кнопкой мыши.

В форме новых наблюдений мы добавили on="submit", чтобы указать, что действие action должно вызываться при подтверждении отправки формы. С другой стороны, мы не указали аргумент on для кнопки отмены, так что связанным с этим действием событием будет щелчок кнопкой мыши.

Наша форма теперь умеет подтверждать отправку и отмену с помощью действий контроллера, но для этих действий нужно написать код. Начнем с действия `create`. Исправьте файл `app/controllers/sightings/new.js` для окончательного создания наблюдения, сохраните его и вернитесь к списку наблюдений.

```
...
  actions: {
    create() {
      var self = this;
      this.get('model.sighting').save().then(function() {
        self.transitionToRoute('sightings');
      });
    },
    cancel() {
    }
  }
});
```

Теперь после подтверждения отправки формы вызывается действие `create`. Во-первых, мы задали ссылку на контроллер с помощью переменной `self`, во-вторых, получили модель наблюдения и вызвали метод `save`.

Последний шаг — сохранение в хранимом источнике. В модели имеется флаг для наличия «грязных» атрибутов (`hasDirtyAttributes`), устанавливаемый в `false` при ее сохранении.

Сохранение модели приводит к возврату объекта `Promise`. Мы связали этот `Promise` в цепочку с методом `then`, принимающим на входе функцию, вызываемую в случае успешного сохранения модели. Наконец, мы возвращаемся к списку наблюдений с помощью метода `transitionToRoute`.

Проверьте работу формы по адресу http://localhost:4200/sightings/new (рис. 24.3).

Заполните форму и нажмите кнопку Create (Создать). Хотя мы успешно добавили новое наблюдение, модель маршрута нашего списка наблюдений по-прежнему возвращает созданные встраиваемым кодом записи. Откройте файл `app/routes/sightings.js`, удалите фиктивные данные и замените их обращением к объекту `store` для извлечения наблюдений:

```
import Ember from 'ember';

export default Ember.Route.extend({
  model() {
    let record1 = this.store.createRecord('sighting', {
      location: 'Atlanta',
      sightedAt: new Date('2016-02-09')
    });
    record1.set('location', 'Paris, France');
    console.log("Record 1 location: " + record1.get('location'));
    let record2 = this.store.createRecord('sighting', {
      location: 'Calloway',
```

```
      sightedAt: new Date('2016-03-14')
    });
    let record3 = this.store.createRecord('sighting', {
      location: '',
      sightedAt: new Date('2016-03-21')
    });

    return [record1, record2, record3];
    return this.store.findAll('sighting', {reload: true});
  }
});
```

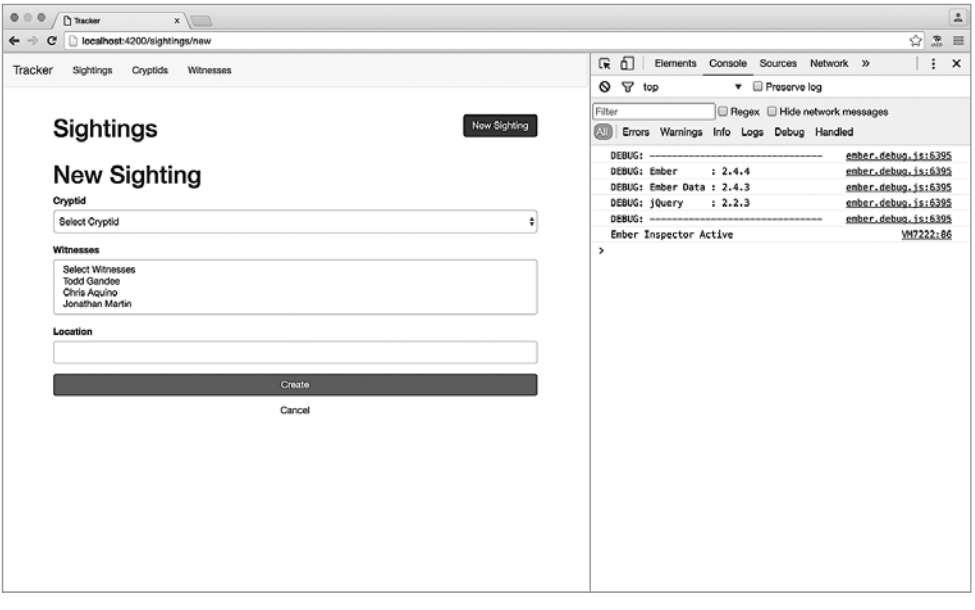


Рис. 24.3. Форма New Sighting (Новое наблюдение)

Обратите внимание на второй аргумент метода `findAll` — объектный литерал с единственным ключом `reload`. Этот аргумент указывает объекту `store` запрашивать свежие данные из API при каждом вызове модели маршрута. Добавление этого аргумента отражает наше желание видеть самые свежие данные всякий раз, когда мы просматриваем список.

Действие `cancel` должно удалять находящиеся в памяти «грязные» экземпляры наблюдений. Мы воспользуемся методом `model.deleteRecord`, как и в главе 21. Добавьте его в файл `app/controllers/sightings/new.js`:

```
...
  actions: {
    create() {
      var self = this;
```

```
      this.get('model.sighting').save().then(function() {
        self.transitionToRoute('sightings');
      });
    },
    cancel() {
      this.get('model.sighting').deleteRecord();
      this.transitionToRoute('sightings');
    }
  }
...
```

После удаления записи пользователь будет возвращен к списку. Этот сценарий работает при нажатии пользователем кнопки `cancel`, но что будет, если для возврата к списку или перехода на другой маршрут используется навигация вверху страницы?

Если мы не уничтожим «грязную» запись, она останется в памяти на время активности сеанса пользователя. Для уничтожения записи воспользуемся действием в маршруте.

В жизненном цикле маршрута существуют действия, вызываемые при различных состояниях и переходах. Эти действия можно подменить, чтобы настроить по своему усмотрению обратные вызовы для различных этапов перехода по маршруту.

Откройте файл `app/controllers/sightings/new.js` и подмените действие `willTransition`, чтобы гарантировать удаление «грязных» записей:

```
...
  model(){
    return Ember.RSVP.hash({
      sighting: this.store.createRecord('sighting'),
      cryptids: this.store.findAll('cryptid'),
      witnesses: this.store.findAll('witness')
    });
  },
  actions: {
    willTransition() {
      var sighting = this.get('controller.model.sighting');
      if(sighting.get('hasDirtyAttributes')){
        sighting.deleteRecord();
      }
    }
  }
});
```

Действие `willTransition` будет срабатывать при изменении маршрута. Применение метода `deleteRecord` удалит объект модели, но только в том случае, если свойство модели `hasDirtyAttributes` равно `true`.

Мы рассмотрели все вопросы, связанные с «грязными» записями при создании наблюдения. Мы также обеспечили сохранение данных в хранимый источник при минимальных изменениях контроллера.

Редактирование наблюдения

После создания и чтения данных следующим шагом, в соответствии с CRUD, является обновление. У нас есть маршрут редактирования для наблюдения, но его необходимо активизировать, добавив в список наблюдений кнопку для перехода к маршруту редактирования. Нам также нужно добавить форму с полями наблюдения в шаблон редактирования (сделать так, чтобы точка подключения `model` маршрута извлекала очевидцев, криптидов и наблюдения) и параметры маршрута — в объект редактирования маршрута в файле `app/router.js`. Наконец, нам нужно создать контроллер для добавления действий в форму редактирования.

Начнем с добавления кнопки Edit (Редактировать) в шаблон `app/templates/sightings/index.hbs`. Чтобы в списке было больше иформации, добавьте также название и изображение увиденного криптида:

```
...
      <div class="media well">
        <img class="media-object thumbnail"
         src="{{if sighting.cryptid.profileImg
          sighting.cryptid.profileImg 'assets/images/cryptids/blank_th.png'}}"
          alt="{{sighting.cryptid.name}}" width="100%" height="100%">
        <div class="caption">
          <h3>{{sighting.cryptid.name}}</h3>
          {{#if sighting.location}}
            <h3>{{sighting.location}}</h3>
            <p>{{moment-from sighting.sightedAt}}</p>
          {{else}}
            <h3 class="text-danger">Bogus Sighting</h3>
          {{/if}}
        </div>
        {{#link-to 'sighting.edit' sighting.id tagName="button"
          class="btn btn-success btn-block"}}
          Edit
        {{/link-to}}
      </div>
...
```

Загрузите страницу http://localhost:4200/sightings и проверьте работу новой кнопки Edit (Редактировать) (рис. 24.4).

Теперь в файле `router.js` добавьте в маршрут редактирования динамические параметры:

```
...
  this.route('sighting', function() {
    this.route('edit', {path: "sightings/:sighting_id/edit"});
  });
...
```

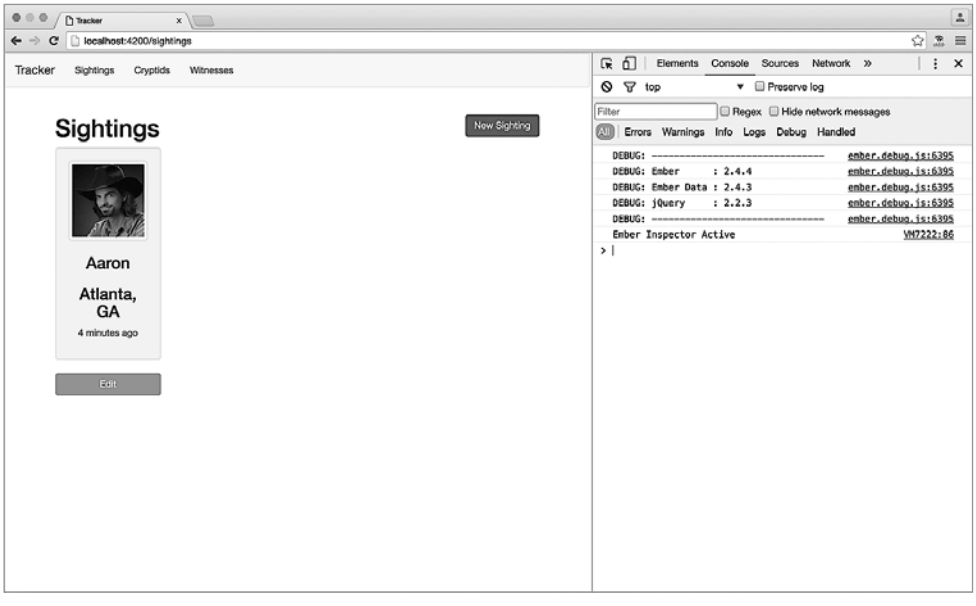


Рис. 24.4. Список наблюдений с кнопкой Edit (Редактировать)

Добавьте методы извлечения для наблюдений, криптидов и очевидцев в маршрут в файле `app/routes/sighting/edit.js`:

```
...
export default Ember.Route.extend({
  model(params) {
    return Ember.RSVP.hash({
      sighting: this.store.findRecord('sighting', params.sighting_id),
      cryptids: this.store.findAll('cryptid'),
      witnesses: this.store.findAll('witness')
    });
  }
});
```

Теперь добавьте элементы формы в шаблон `app/templates/sighting/edit.hbs` аналогично тому, как делали при создании нового наблюдения:

```
{{outlet}}
<h1>Edit Sighting:
  <small>
    {{model.sighting.location}} -
    {{moment-from model.sighting.sightedAt}}
  </small>
</h1>
<form {{action "update" model on="submit"}}>
  <div class="form-group">
    <label for="name">Cryptid</label>
```

```
    {{input value=model.sighting.cryptid.name type="text" class="form-control"
      name="location" disabled=true}}
  </div>
  <div class="form-group">
    <label>Witnesses</label>
    {{#each model.sighting.witnesses as |witness|}}
      {{input value=witness.fullName type="text" class="form-control"
        name="location" disabled=true}}
    {{/each}}
  </div>
  <div class="form-group">
    <label for="location">Location</label>
    {{input value=model.sighting.location type="text" class="form-control"
      name="location" required=true}}
  </div>
  <button type="submit" class="btn btn-info btn-block">Update</button>
  <button {{action 'cancel'}} class="btn btn-block">Cancel</button>
</form>
```

Почти готово, остался последний шаг — контроллер. Необходимо задать `{{action}}` формы равным `update`. Поскольку Tracker пока что разрешает лишь изменять местоположение, криптидов и очевидцев, мы будем визуализировать в отключенных полях ввода.

Создайте контроллер:

```
ember g controller sighting/edit
```

Откройте файл `app/controllers/sighting/edit.js` и добавьте действия `update` и `cancel`:

```
import Ember from 'ember';

export default Ember.Controller.extend({
  sighting: Ember.computed.alias('model.sighting'),
  actions: {
    update() {
      if(this.get('sighting').get('hasDirtyAttributes')){
        this.get('sighting').save().then(() => {
          this.transitionToRoute('sightings');
        });
      }
    },
    cancel() {
      if(this.get('sighting').get('hasDirtyAttributes')){
        this.get('sighting').rollbackAttributes();
      }
      this.transitionToRoute('sightings');
    }
  }
});
```

Для обновления достаточно вызвать метод `save`. Вызывать API необходимо только при изменении записи, поэтому мы использовали метод `sighting.get('hasDirtyAttributes')`. Фреймворк Ember обо всем позаботился!

Обратите внимание на использование свойства `Ember.computed.alias`. Присваивание этого вычисляемого свойства позволит вам набирать не так много кода при обращении к используемому в данный момент объекту `sighting`. Вы можете создать псевдоним любого свойства для быстрого доступа к нему, что особенно удобно для вложенных свойств.

Удаление наблюдения

Время от времени сообщение о наблюдении криптида оказывается уткой. Хотя это и редкость, вам все равно нужен способ удаления старых или недостоверных данных. Как вы помните из главы 21, уничтожение записей выполняется с помощью метода `record.destroyRecord`. Все достаточно просто.

Добавьте кнопку удаления в шаблон `app/templates/sighting/edit.hbs`:

```
  <button type="submit" class="btn btn-info btn-block">Update</button>
  <button {{action 'cancel'}} class="btn btn-block">Cancel</button>
</form>

<hr>
<button {{action 'delete'}} class="btn btn-block btn-danger">
  Delete
</button>
```

Помимо этой новой кнопки, мы добавили горизонтальную линейку (обычную линию) между элементами формы, чтобы отделить кнопки Update (Обновить) и Cancel (Отмена) от новой кнопки Delete (Удалить). Элемент `<hr>` — инструмент UI, используемый нами для указания, что удаление наблюдения — это отдельный от редактирования наблюдения процесс.

Далее добавьте в контроллер `app/controllers/sighting/edit.js` действие `delete`:

```
...
    cancel() {
      if(this.get('sighting').get('hasDirtyAttributes')){
        this.get('sighting').rollbackAttributes();
      }
      this.transitionToRoute('sightings');
    },
    delete() {
      var self = this;
      if (window.confirm("Are you sure you want to delete this sighting?")) {
        this.get('sighting').destroyRecord().then(() => {
```

```
          self.transitionToRoute('sightings');
        });
      }
    }
  }
});
```

Действие `delete` добавляет вызов метода `window.confirm` для получения подтверждения действия от пользователя. Не считая условного оператора, это действие аналогично остальным: получить модель, вызвать метод и добавить асинхронный вызов `then`, выполняемый после окончания выполнения запроса API.

Теперь перейдите по адресу http://localhost:4200/sightings и нажмите кнопку Edit (Редактировать) одного из наблюдений, чтобы просмотреть файл `app/templates/sightings/edit.hbs` и его новую кнопку Delete (Удалить) (рис. 24.5).

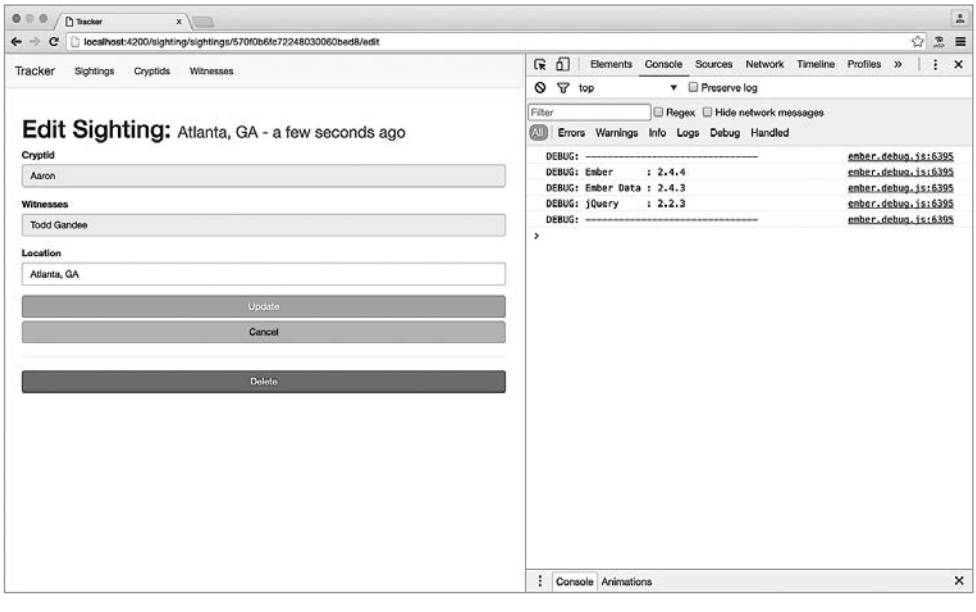


Рис. 24.5. Форма редактирования наблюдения (Edit Sighting)

Наш технологический цикл завершен, сделано все: от создания до удаления.

Действия маршрутов

Действия могут быть не только у контроллеров. Маршруты могут объявлять действия для шаблонов и подменять действия жизненного цикла. При вызове действие «всплывает» из шаблона в контроллер, далее — в маршрут и родительские маршруты.

Соответственно, маршрут может работать в качестве контроллера в случае, когда описание контроллера не требуется. Это может прозвучать неоднозначно в свете наших разговоров о разделении обязанностей в приложении, но фреймворк Ember действительно делит задачу контроллера на две части: маршрутизацию информации и логику контроллера. Иногда больше логики содержится в файле маршрута, а иногда — в контроллере. Разделение файлов позволяет отделять небольшие «удобоваримые» фрагменты кода, если в остальных файлах находятся многочисленные действия или декораторы.

Чтобы увидеть, как это работает, нужно переместить действия `create` и `cancel` из файла `app/controllers/sightings/new.js` в файл `app/routes/sightings/new.js`. Это изменение также расширит ваше представление о перемещаемых между этими файлами методах и объектах, что, согласитесь, будет хорошим опытом на случай, если вы решите ограничиться только файлом маршрута и управляющими представлением приложения компонентами (о компонентах вы узнаете в следующей главе).

Для начала добавьте указанные действия в файл `app/routes/sightings/new.js`:

```
import Ember from 'ember';

export default Ember.Route.extend({
  model() {
  ...
  },
  sighting: Ember.computed.alias('controller.model.sighting'),
  actions: {
    willTransition() {
      var sighting = this.get('controller.model.sighting');
      if(sighting.get('hasDirtyAttributes')) {
        sighting.deleteRecord();
      }
    },
    create() {
      var self = this;
      this.get('sighting').save().then(function(data) {
        self.transitionTo('sightings');
      });
    },
    cancel() {
      this.get('sighting').deleteRecord();
      this.transitionToRoute('sightings');
    }
  }
});
```

Мы создали вычисляемое свойство `Ember.computed.alias` для объекта наблюдения в маршруте. Основное отличие от обращения к объекту `model.sighting` контроллера из маршрута заключается в местонахождении этого объекта. Объект `model` на

«новом» маршруте не идентичен таковому на «новом» контроллере. Для обращения к объекту `sighting` мы использовали вызов `get('controller.model.sighting')`. Создание псевдонима для этого объекта будет каждый раз экономить вам набор этого кода.

Теперь удалите эти действия из файла `app/controllers/sightings/new.js`:

```
import Ember from 'ember';

export default Ember.Controller.extend({
  actions: {
    create() {
      var self = this;
      this.get('model.sighting').save().then(function() {
        self.transitionToRoute('sightings');
      });
    },
    cancel() {
      this.get('model.sighting').deleteRecord();
      this.transitionToRoute('sightings');
    }
  }
});
```

Убедитесь, что приложение было перезагружено (или перезапустите сервер: `ember server`). Перейдите на http://localhost:4200/sightings/new и создайте новое наблюдение, чтобы убедиться, что вы по-прежнему можете выполнять действия маршрута.

На текущий момент файл `app/controllers/sightings/new.js` не нужен. Можете его удалить. Фреймворк Ember все равно будет создавать объект контроллера при работе приложения, а пустой файл в каталоге `app` станет вам только мешать.

В данной главе мы сосредоточили внимание на контроллерах Ember, чтобы продемонстрировать, как создавать действия и свойства для шаблонов, переходить на новые маршруты после сохранения данных и удалять записи при отмене или смене маршрута. Действия предоставляют возможность менять с помощью простого обратного вызова данные модели нашего приложения. И наконец, мы закончили создание основных действий CRUD обновлением и удалением записей из маршрута редактирования.

Фреймворк Ember обеспечивает возможность быстрой разработки, причем создавать контроллеры не обязательно (кроме случаев, когда вам нужны какие-то конкретные их возможности для ваших маршрутов). Кроме того, контроллеры позволяют более детально настраивать представления и управлять данными модели. Действия — ключ к взаимодействию с пользователем.

Действия маршрутов, контроллеров и компонентов — тема следующей (последней) главы.

Бронзовое упражнение: страница детальной информации о наблюдении

Создайте страницу детальной информации о наблюдении с помощью шаблона `app/templates/sighting/index.hbs` и модуля `app/routes/sighting.js` для отображения изображения криптида, его местоположения и списка очевидцев. Добавьте кнопку Edit (Редактировать) (совет: добавить действия можно для маршрута).

Серебряное упражнение: дата наблюдения

При создании и редактировании наблюдения добавьте в контроллер свойство `sightingDate` и поле ввода, связанное с этим свойством. Воспользуйтесь или простым текстовым полем ввода, или `input[type="date"]` для ввода даты наблюдения. Преобразуйте дату в строку ISO8601 с помощью метода `moment` и присвойте свойству наблюдения `sightingAt` эту строку с датой.

Золотое упражнение: добавление и удаление очевидцев

При создании нового наблюдения сформируйте список очевидцев из действия `select onchange`. Создайте новое свойство для списка очевидцев и добавьте объекты в свойство `witnesses` наблюдения.

Сделайте так, чтобы при работе пользователя с этой страницей набор выбранных очевидцев визуализировался как список с кнопкой Remove (Удалить). Воспользуйтесь элементом `select` для добавления опций в список наряду с удалением из элемента `select`. Вам понадобятся два действия: `addWitness` и `removeWitness`.

25 Компоненты

Компоненты — объекты фреймворка Ember, в которых хранятся свойства контроллеров и представлений. Основная цель их применения — возможность повторного использования элементов DOM с собственным контекстом или областью видимости. Для настройки визуализируемого внутри их шаблонов контента компоненты принимают на входе атрибуты. Компоненты также предоставляют возможность связывать действия со свойствами родительского контроллера или родительского маршрута (рис. 25.1).

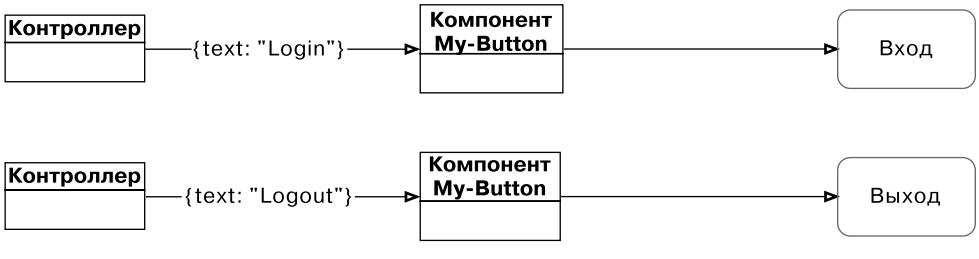


Рис. 25.1. Свойство компонента

В будущем при обзоре архитектуры приложения вы начнете во многих маршрутах и шаблонах видеть сгруппированные вместе лишь немного различающиеся элементы. Если есть возможность убрать группу элементов из шаблона и описать ее с помощью отражающих состояние элементов переменных, то это явный кандидат на роль компонента.

В данной главе вы увидите относительно простые примеры оборачивания элементов DOM в объекты JavaScript для использования в нескольких маршрутах. Подобно упоминавшимся в главе 23 вспомогательным методам, компоненты для формирования HTML принимают аргументы в виде атрибутов. В качестве дополнительной возможности у них могут быть собственные действия и свойства

с областью видимости, служащие для самообновления по мере взаимодействия с пользователем.

В этой главе вы увидите только верхушку айсберга разработки масштабируемых приложений. Приложения Ember при промышленной эксплуатации в значительной степени полагаются на компоненты для создания единообразных пользовательских интерфейсов и удобных в сопровождении баз кода, насчитывающих иной раз сотни маршрутов. Однако примеры, которые рассматриваются в этой главе, дадут фундаментальное понимание, необходимое для будущих проектов, где может понадобиться сделать из ваших шаблонов маршрутов пригодные для повторного использования страницы или отдельные элементы.

Элементы итераторов как компоненты

Примером сгруппированных элементов, из которых часто делают компонент, могут быть элементы, визуализируемые в итераторах `{{#each}}`. Это может быть контейнер `<div>` с конкретным `className`, основанным на объекте итератора, заголовком, путем к изображению, кнопкой с действием и/или стилями, зависящими от состояний свойства объекта. Чтобы облегчить их повторное использование, можно обернуть весь этот код DOM в шаблон компонента и создать JavaScript-файл компонента для обработки декораторов и действий, которые обычно обрабатывает контроллер.

Наш список наблюдений создан с использованием итератора `{{#each}}`, и наш первый созданный компонент будет представлять собой элемент списка наблюдений. Начнем с генерации компонента с помощью утилиты Ember CLI в терминале:

```
ember g component listing-item
```

Эта команда создает три файла: `app/components/listing-item.js`, `app/templates/components/listing-item.hbs` и тестовый файл `tests/integration/components/listing-item-test.js`.

Теперь, когда компонент создан, необходимо найти код, который должен заменять компонент. Откройте шаблон `app/templates/sightings/index.hbs` и найдите следующий блок кода. Через минуту вы переместите часть этого кода в компонент. А пока что просто посмотрите на него:

```
<div class="row">
  {{#each model as |sighting|}}
    <div class="col-xs-12 col-sm-3 text-center">
      <div class="media well">
        <img class="media-object thumbnail"
         src="{{if sighting.cryptid.profileImg
          sighting.cryptid.profileImg 'assets/images/cryptids/blank_th.png'}}"
          alt="{{sighting.cryptid.name}}" width="100%" height="100%">
```

```
        <div class="caption">
          <h3>{{sighting.cryptid.name}}</h3>
          {{#if sighting.location}}
            <h3>{{sighting.location}}</h3>
            <p>{{moment-from sighting.sightedAt}}</p>
          {{else}}
            <h3 class="text-danger">Bogus Sighting</h3>
          {{/if}}
        </div>
        {{#link-to 'sighting.edit' sighting.id tagName="button"
          class="btn btn-success btn-block"}}
          Edit
        {{/link-to}}
      </div>
    </div>
  {{/each}}
</div>
```

На данной конкретной странице должен быть контейнер `<div>` с классом `col-xs-12` (то есть элемент `<div>`, выделенный серым в листинге). Он представляет собой визуальный контейнер для макета и задает размер для контента. Если бы мы добавили этот контейнер в компонент для списка наблюдений, размер контейнера оказался бы фиксированным для каждого экземпляра компонента на сайте.

А вот контейнер `<div class="media well">` и его содержимое можно перенести в компонент. Затем можно будет добавить этот компонент в контейнер любого размера с сохранением характеристик отдельного элемента списка. Компонент будет содержать основные перечисленные части этого элемента — контейнер `media` и изображение, заголовок и кнопку Edit (Редактировать).

Откройте шаблон `app/templates/components/listing-item.hbs` и начните с добавления изображения и названия криптида:

```
<img class="media-object thumbnail" src="{{imagePath}}" alt="{{name}}"
  width="100%" height="100%">
<div class="caption">
  <h3>{{name}}</h3>
  {{yield}}
</div>
```

Компонент представляет отдельный элемент DOM. Следовательно, все в шаблоне компонента — дочерний элемент этого элемента DOM. По умолчанию HTMLBars фреймворка Ember использует JavaScript для создания элементов `<div>` и визуализации шаблонов компонентов внутри них.

Только что написанный нами код добавляет изображение и заголовок с названием в элемент `<div>`, созданный компонентом `{{#listing-item}}`. Аналогично другим шаблонам компонент содержит динамические части — передаваемые в него переменные (мы обсудим `{{yield}}` в следующем абзаце).

Если добавить этот компонент в шаблон маршрута, код будет выглядеть так:

```
<div>
  <img class="media-object thumbnail" src="[cryptid's imagePath string]"
    alt="[cryptid's name string]" width="100%" height="100%">
  <div class="caption">
    <h3>[cryptid's name string]</h3>
    {{yield}}
  </div>
</div>
```

Динамические части шаблона `{{name}}` и `{{imagePath}}` — это передаваемые компоненту атрибуты. С помощью `{{yield}}` можно передавать в компонент дочерние элементы для визуализации в конкретном месте структуры узлов DOM. Файл шаблона компонента играет роль макета или базового набора элементов, а `{{yield}}` предназначен для элементов, которые требуется добавлять вместе с каждым экземпляром компонента. Мы воспользуемся им далее в данной главе.

Хотя это еще не вполне та разметка, которую мы хотим заменить в шаблоне `app/templates/sightings/index.hbs`, но уже неплохое начало.

Замените этим компонентом существующий код в шаблоне `app/templates/sightings/index.hbs`.

```
<div class="row">
  {{#each model as |sighting|}}
    <div class="col-xs-12 col-sm-3 text-center">
      <div class="media well">
        <img class="media-object thumbnail"
         src="{{if sighting.cryptid.profileImg
          sighting.cryptid.profileImg 'assets/images/cryptids/blank_th.png'}}"
          alt="{{sighting.cryptid.name}}" width="100%" height="100%">
        <div class="caption">
          <h3>{{sighting.cryptid.name}}</h3>
          {{#if sighting.location}}
            <h3>{{sighting.location}}</h3>
            <p>{{moment-from sighting.sightedAt}}</p>
          {{else}}
            <h3 class="text-danger">Bogus Sighting</h3>
          {{/if}}
        </div>
        {{#link-to 'sighting.edit' sighting.id tagName="button"
          class="btn btn-success btn-block"}}
          Edit
        {{/link-to}}
      </div>
      {{#listing-item imagePath=sighting.cryptid.profileImg
        name=sighting.cryptid.name}}
      {{/listing-item}}
    </div>
  {{/each}}
</div>
```

Мы заменили с помощью одной строки значительный фрагмент кода. По ходу дела мы потеряли часть функциональности, но скоро мы это исправим. Следующий шаг — возвращение обратно стилей контейнера для элементов компонента. Речь идет об отсутствующих стилях `media` и `well` фреймворка Bootstrap.

Откройте файл `app/components/listing-item.js` и добавьте в компонент `listing-item` свойство `classNames`:

```
import Ember from 'ember';

export default Ember.Component.extend({
  classNames: ["media", "well"]
});
```

Значение свойства `classNames` передается атрибуту `classNames` создаваемого компонентом элемента `<div>`.

Теперь визуализированный компонент выглядит следующим образом:

```
<div class="media well">
  <img class="media-object thumbnail" src="[imagePath string]"
   alt="[name string]"
    width="100%" height="100%">
  <div class="caption">
    <h3>[name string]</h3>
    {{yield}}
  </div>
</div>
```

Далее мы добавим элементы в раздел `{{yield}}` шаблона компонента, чтобы сделать реализацию экземпляра этого компонента уникальной для маршрута наблюдений. В первую очередь в файле `app/templates/sightings/index.hbs` добавьте контекстно зависимый контент компонента, который будет визуализироваться в `{{yield}}`:

```
...
      {{#listing-item imagePath=sighting.cryptid.profileImg
        name=sighting.cryptid.name}}
        {{#if sighting.location}}
          <h3>{{sighting.location}}</h3>
            <p>{{moment-from sighting.sightedAt}}</p>
        {{else}}
          <h3 class="text-danger">Bogus Sighting</h3>
        {{/if}}
        {{#link-to 'sighting.edit' sighting.id tagName="button"
          class="btn btn-success btn-block"}}
          Edit
        {{/link-to}}
      {{/listing-item}}
```

Этот код (он должен быть вам знаком) возвращает на свои места местоположение и время наблюдения, а также кнопку Edit (Редактировать).

Компоненты для кода DRY

Пришло время небольшой «усушки» кода. Что-что? DRY[1] — это принцип программирования; его аббревиатура расшифровывается как «не повторяйся» (Don't Repeat Yourself). Другими словами, если вы что-то пишете, то делайте это только в одном месте.

И листинг для наблюдений, и листинг для криптидов используют элемент с классом `class="media well"` для изображений и названий. Так что этот код явно требует небольшой «усушки». Данный компонент не соответствует в точности элементу списка криптидов, но как раз тут-то нам и пригодится `{{yield}}`.

Добавьте новый компонент `{{#listing-item}}` в список криптидов. Замените в шаблоне `app/templates/cryptids.hbs` элемент `<div class="media well">` и его дочерние элементы:

```
<div class="row">
  {{#each model as |cryptid|}}
    <div class="col-xs-12 col-sm-3 text-center">
      <div class="media well">
        {{#link-to 'cryptid' cryptid.id}}
          <img class="media-object thumbnail"
            src="{{if cryptid.profileImg cryptid.profileImg
            'assets/images/cryptids/blank_th.png'}}"
            alt="{{cryptid.name}}" width="100%" height="100%">
        {{/link-to}}
        <div class="caption">
          <h3>{{cryptid.name}}</h3>
        </div>
      </div>
      {{#link-to 'cryptid' cryptid.id}}
        {{listing-item imagePath=cryptid.profileImg name=cryptid.name}}
      {{/link-to}}
    </div>
  {{else}}
    <div class="jumbotron">
      <h1>No Creatures</h1>
    </div>
  {{/each}}
</div>
```

Обратили ли вы внимание, что мы по-разному ссылались на компоненты в этих двух шаблонах? Итератор наблюдений использует `{{yield}}` для добавления элементов внутри `<div class="caption">`, а шаблон криптидов — нет. Если нам нужно, чтобы компонент обращался только к коду из файла шаблона компонента, можно воспользоваться *встраиваемым компонентом*, описываемым без использования знака решетки (`#`), то есть так, как мы и сделали: `{{listing-item}}`. При таком синтаксисе закрывающий элемент HTMLBars (`{{/listing-item}}`) не требуется.

[1] Dry (англ.) — «высушивать». — *Примеч. пер.*

Чтобы добавить в этот компонент ссылку, мы обернули его в `{{#link-to}}`. В старом варианте с итерируемыми элементами на страницу детальной информации о криптиде вела ссылка только от изображения. Теперь же весь элемент ссылается на страницу детальной информации о криптиде. Этот пример демонстрирует гибкость компонентов, их приспосабливаемость к контексту различных шаблонов маршрутов, помимо возможностей по визуализации сходного контента. Можно также добавить в компонент дополнительные атрибуты на случай, если требуется ссылка внутри шаблона компонента.

Данные вниз, действия вверх

Далее мы добавим компонент, который будет зависеть от изменений состояния приложения. А именно — создадим мгновенное уведомление, которое будет отображаться при создании нового списка.

Один из основополагающих принципов компонентов — «данные (или состояние) вниз, действия вверх». В отличие от контроллеров компоненты не должны менять состояние приложения — им следует передавать изменения посредством действий (рис. 25.2).

Передав модель маршрута непосредственно компоненту, можно легко заменить им контроллер, отказавшись от использования декораторов или действий контроллера.

Мы создадим новый компонент и действия, после чего добавим этот компонент в файл `app/templates/application.hbs` для визуализации глобального сообщения при создании нового наблюдения.

Для начала сгенерируйте новый компонент в терминале:

```
ember g component flash-alert
```

Компонент `{{flash-alert}}` будет элементом-контейнером для `<strong>`[1] заголовка уведомления и текста сообщения.

Для этого откройте и отредактируйте файл `app/templates/components/flash-alert.hbs`:

```
~~{{yield}}~~
<strong>{{typeTitle}}!</strong> {{message}}
```

Отредактируйте файл компонента `app/components/flash-alert.js`, добавив свойство `classNames`:

```
import Ember from 'ember';

export default Ember.Component.extend({
  classNames: ["alert"]
});
```

[1] Выделен жирным шрифтом. — *Примеч. пер.*

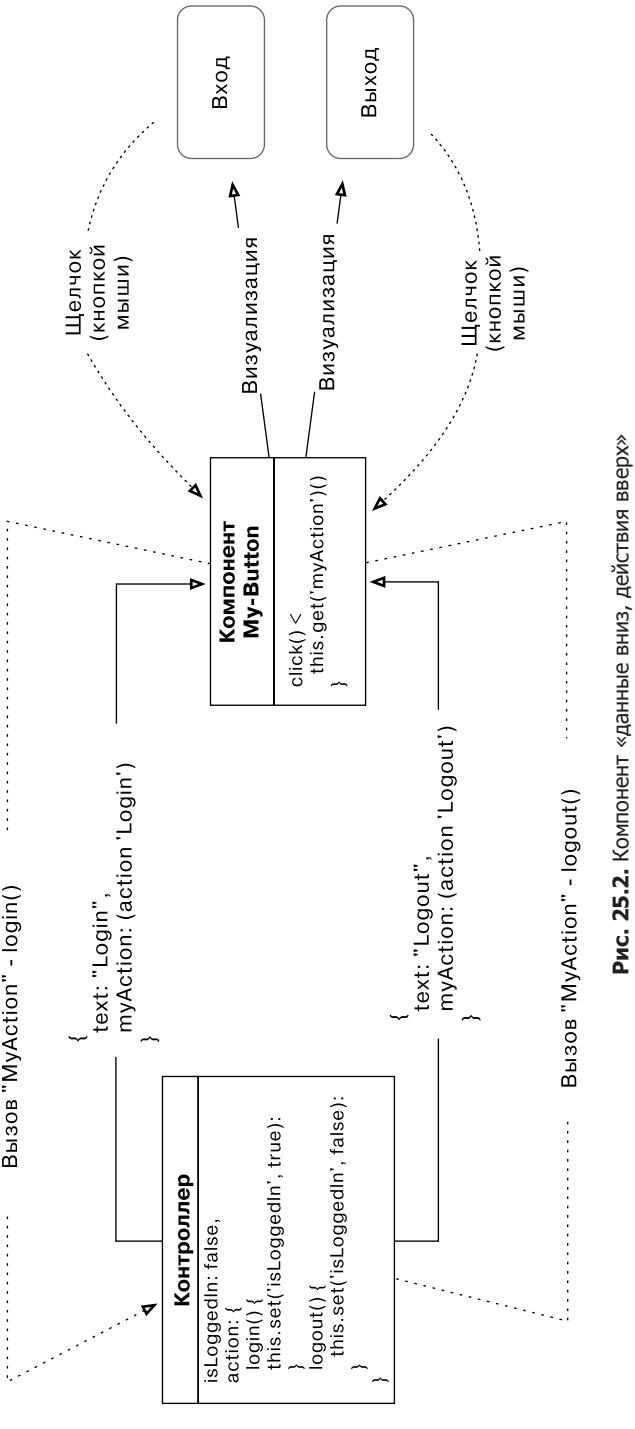


Рис. 25.2. Компонент «данные вниз, действия вверх»

Теперь компонент визуализирует следующие элементы:

```
<div class="alert">
  <strong>{{typeTitle}}!</strong> {{message}}
</div>
```

Привязки имени класса

Похоже, что этот контейнер будет выводить правильное сообщение, но такая стилизация не предоставляет контекст, чтобы понять, каков тип уведомления. В Bootstrap имеются различные варианты стилей для компонентов уведомлений: `"alert-success"`, `"alert-info"`, `"alert-warning"` и `"alert-danger"`.

Для указания правильного типа уведомления мы добавим один из этих классов с помощью вычисляемого свойства и привязки имени класса.

Начнем с добавления вычисляемого свойства в файл `app/components/flash-alert.js`:

```
...
export default Ember.Component.extend({
  classNames: ["alert"],
  typeClass: Ember.computed('alertType', function() {
    return "alert-" + this.get('alertType');
  })
});
```

Мы добавили вычисляемое свойство для `typeClass`, которое будет применяться для указания имени класса в элементе `<div>` компонента. Вычисляемое свойство ожидает получения свойства `alertType`, которое мы добавим позднее, и возвращает строку с предшествующими `"alertType"` символами `"alert-"`. Это дает нам возможность передавать свойство со значением `"success"`, `"info"`, `"warning"` или `"danger"` (для предоставления пользователю контекста уведомления). Мы воспользуемся этим же свойством `alertType` и для другого вычисляемого свойства.

Наконец, добавьте в файл `app/components/flash-alert.js` привязки имен классов для `classNames`, привязанных к свойствам компонента:

```
...
export default Ember.Component.extend({
  classNames: ["alert"],
  classNameBindings: ['typeClass'],
  typeClass: Ember.computed('alertType', function() {
    return "alert-" + this.get('alertType');
  })
});
```

Этот код добавляет в массив `classNames` имя класса из значения вычисляемого свойства `typeClass`. При задании свойства `alertType` стиль компонента будет меняться.

Свойство `classNameBindings` предназначено для использования только с атрибутом `classNames` элемента. Существует дополнительное свойство компонентов Ember для других атрибутов — `attributeBindings`. Можно связывать свойства компонента с атрибутами элементов компонента. Простейший пример — задание значения вычисляемого свойства `computedProperty` для атрибута `href` ссылки. Например:

```
export default Ember.Component.extend({
  attributeBindings: ['href', 'customHREF:href'],
  href: "http://www.mydomain.com",
  customHREF: "http://www.mydomain.com"
});
```

Благодаря привязке атрибутов можно заполнить все атрибуты компонента передаваемыми ему данными состояния. Чтобы получить больше информации по этому вопросу, загляните в онлайн-документацию Ember.

Далее добавьте вычисляемое свойство для отображения типа уведомления в виде строки. Шаблон компонента ожидает получения свойства `typeTitle`, так что нам нужно добавить это вычисляемое свойство в файл `app/components/flash-alert.js`:

```
import Ember from 'ember';
export default Ember.Component.extend({
  ...
  typeClass: Ember.computed('alertType', function() {
    return "alert-" + this.get('alertType');
  }),
  typeTitle: Ember.computed('alertType', function() {
    return Ember.String.capitalize(this.get('alertType'));
  })
});
```

Теперь в нашем сообщении есть связанная с типом уведомления строка в верхнем регистре и элемент `<strong>` в шаблоне. Мы связали имя класса и декорированное свойство с компонентом путем вычисления значения каждого из них из данных, передаваемых в компонент через свойство `alertType`.

Далее нам нужно добавить наш компонент на страницу в шаблоне `app/templates/application.hbs`:

```
<header>
  ...
</header>
<div class="container">
  {{flash-alert}}
  {{outlet}}
</div>
```

Это встраиваемый компонент, то есть в нем не будет элементов для визуализации в `{{yield}}`, а значит `{{yield}}` в шаблоне компонента не нужен.

Данные вниз

На текущий момент у уведомления есть только контейнеры и отсутствуют какие-либо данные состояния для визуализации контента и `classNames`. Передадим в него (в шаблоне `app/templates/application.hbs`) состояние компонента:

```
...
  {{flash-alert message="This is the Alert Message" alertType="success"}}
  {{outlet}}
</div>
```

Запустите сервер и откройте в браузере адрес http://localhost:4200/sightings, чтобы увидеть, как визуализировался компонент `{{flash-alert}}` с предоставленными нами встроенными данными (рис. 25.3).

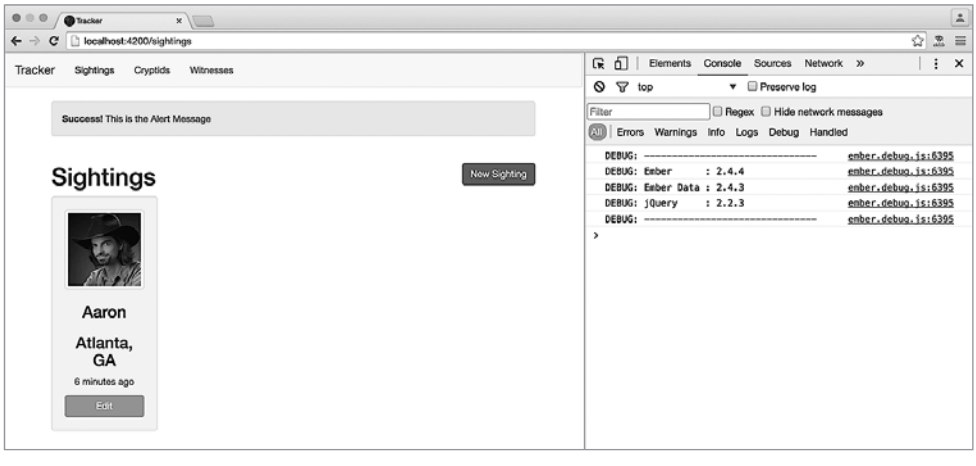


Рис. 25.3. Мгновенное уведомление

Теперь, когда уведомление отображается на экране, хотелось бы задавать значения атрибутов `message` и `alertType` динамически. Нам понадобится для этого контроллер приложения. Добавьте контроллер, воспользовавшись генератором Ember CLI:

```
ember g controller application
```

Контроллер приложения будет сохранять динамическое состояние компонента `{{flash-alert}}`. Для сохранения состояния уведомления контроллеру понадобятся свойства. Добавьте следующее свойство в файл `app/controllers/application.js`:

```
import Ember from 'ember';

export default Ember.Controller.extend({
  alertMessage: null,
  alertType: "success",
  isAlertShowing: false
});
```

Мы добавили свойства, значение которых будут задаваться действием. Теперь у нас есть свойства контроллера, которые можно передать в компонент `flash-alert` в шаблоне `app/templates/application.hbs`:

```
...
</header>
<div class="container">
  {{#if isAlertShowing}}
    {{flash-alert message="This is the Alert Message" alertType="success"}}
                  alertMessage alertType=alertType}}
  {{/if}}
  {{outlet}}
</div>
```

Сейчас можно задавать значение свойств контроллера с помощью действия. Приложению нужно визуализировать компонент только при равенстве `true` значения свойства `isAlertShowing` и наличии значений у остальных свойств. Действие, задающее значения этих свойств, будет поступать от контроллеров из всего приложения. Как же мы обеспечим «всплытие» этих действий? Это для нас сделает фреймворк Ember (рис. 25.4).

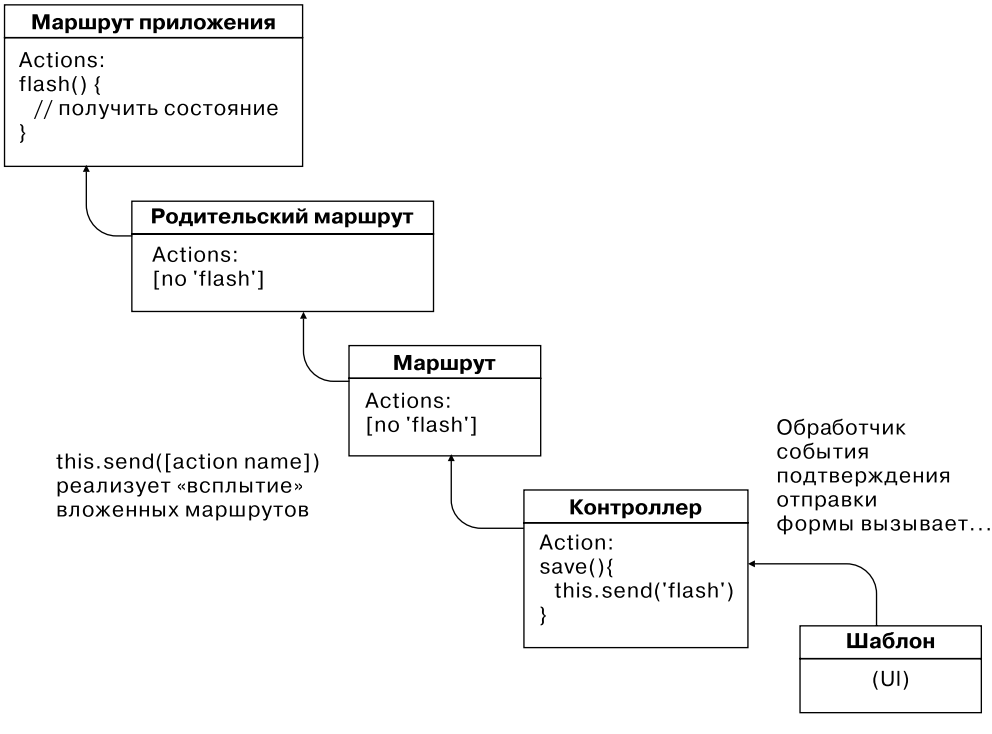


Рис. 25.4. Процесс создания мгновенного уведомления

Необходимо добавить в маршрут действие. Можно вызвать действие из контроллера с последующим его обращением к текущему контроллеру, затем к текущему маршруту, родительским маршрутам и, наконец, к маршруту приложения.

Поскольку компонент `flash-alert` находится в шаблоне `app/templates/application.hbs`, нам понадобится маршрут приложения. Создайте его:

```
ember g route application
```

Вы увидите выведенное в командной строке сообщение:

```
[?] Overwrite app/templates/application.hbs?
```

Введите `n` или `no` для продолжения. Мы не хотим перезаписывать только что созданный файл шаблона. Нам нужно только добавить JavaScript-файл `app/routes/application.js`.

Далее нам необходимо отредактировать маршрут в файле `app/routes/application.js`:

```
import Ember from 'ember';

export default Ember.Route.extend({
  actions: {
    flash(data){
      this.controller.set('alertMessage', data.message);
      this.controller.set('alertType', data.alertType);
      this.controller.set('isAlertShowing', true);
    }
  }
});
```

Действия вверх

Теперь у нас есть действие, задающее уведомление `flash-alert`, которое отображает соответствующее сообщение. Нужно только передать этому действию данные. Эти данные представляют собой объект с ключами для свойств `alertType` и `message`.

Свойство `alertType` будет частью как `message`, так и стиля с вариантами уведомления Bootstrap: `"success"`, `"info"`, `"warning"` или `"danger"`. Вызов действия из контроллера будет выглядеть так:

```
this.send('flash', {alertType: "success", message: "You Did It! Hooray!"});
```

Добавлять вызов действия приложения мы будем после создания нового наблюдения. Добавьте в файле `app/routes/sightings/new.js` следующий код:

```
...
    create() {
      var self = this;
      this.get('sighting').save().then(function(data){
        self.send('flash', {alertType: "success", message: "New sighting."});
```

```
        self.transitionTo('sightings');
      });
...
```

Теперь перейдите к созданию нового наблюдения, нажав кнопку New Sighting (Новое наблюдение). Выберите криптида и очевидца, наберите новое местоположение и нажмите Save (Сохранить). После сохранения наблюдения в базе данных приложение переправит вас к списку наблюдений с новым мгновенным уведомлением вверху списка (рис. 25.5).

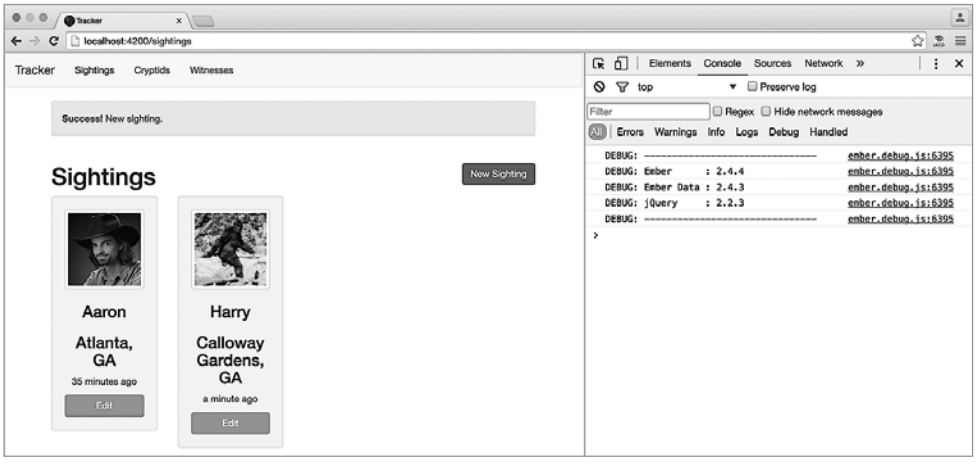


Рис. 25.5. Мгновенное уведомление: новое наблюдение

Последний шаг — добавить действие и событие, которые удаляют уведомление с экрана. Показывать уведомление `flash-alert` необходимо только после создания нового сообщения, так что нам нужно поменять значение переключателя, чтобы скрыть уведомление после его удаления.

Добавьте в файле `app/controllers/application.js` действие `removeAlert`:

```
...
export default Ember.Controller.extend({

  alertMessage: null,
  alertType: null,
  isAlertShowing: false,
  actions: {
    removeAlert(){
      this.set('alertMessage', "");
      this.set('alertType', "success");
      this.set('isAlertShowing', false);
    }
  }
});
```

Значение свойства `isAlertShowing` будет установлено равным `false`, `alertMessage` — равным пустой строке, а `alertType` — `"success"`.

Далее необходимо отправить действие `removeAlert` компоненту. Добавьте следующий код в шаблон `app/templates/application.hbs`:

```
...
</header>
<div class="container">
  {{#if isAlertShowing}}
    {{flash-alert message=alertMessage alertType=alertType}}
      close=(action "removeAlert")}}
  {{/if}}
  {{outlet}}
</div>
```

Синтаксис `close=(action "removeAlert")`, вероятно, выглядит странно. Это нововведение в Ember 2.0 названо *действием замыкания*. В нем происходит передача функции-литерала, которая будет вызвана из компонента в виде атрибута с именем `close`. Это напоминает использование псевдонима.

В более ранних версиях Ember этот код выглядел сложнее. Действия замыканий не просто функции, передаваемые из объекта как аргумент. Чтобы узнать о них больше, посмотрите в блоге EmberJS сообщение о возможностях версий 1.13 и 2.0 (по адресу emberjs.com/blog/2014/06/12/ember-1-13-0-released.html).

Далее нам нужно вызвать это действие из компонента. Компоненты являются экземплярами элементов DOM, поэтому у них есть пары «ключ/значение», представляющие события элементов DOM. Можно добавить объявление прослушивателя события `click` — и фреймворк Ember включит прослушиватель в элемент `<div>`, который обертывает шаблон. Добавьте этот код в файл `app/components/flash-alert.js`:

```
import Ember from 'ember';
export default Ember.Component.extend({
  ...
  typeTitle: Ember.computed('alertType', function() {
    return Ember.String.capitalize(this.get('alertType'));
  }),
  click() {
    this.get('close')();
  }
});
```

При вызове свойства `close` контроллер приложения будет обращаться к действию `"removeAlert"`. Благодаря использованию действия замыкания мы связали свойство компонента с описанной в родительском контроллере функцией и привязали функциональность компонента к области видимости его родителя. Теперь можно добавлять уведомление `flash-alert` на любом уровне и придавать свойству `close` различную функциональность в зависимости от контекста.

Мы добавили компонент, реагирующий на перемещающиеся вниз данные (для настройки компонента) и перемещающиеся вверх из родительского контроллера действия (для задания внешнего состояния компонента). Это жизненный цикл компонента: данные вниз, действия вверх. Помните про него при создании компонентов.

В посвященных архитектуре приложения главах вы узнали о структуре современных приложений, создаваемых с помощью фреймворка Ember, о паттерне MVC и о разделении обязанностей с помощью заранее созданных JavaScript-объектов этого фреймворка. Поспособствует он сохранению душевного здоровья и при работе с паттернами наименования, скаффолдингом, инструментами сборки и условными соглашениями. Теперь вы можете уверенно создавать новые приложения с помощью команды `ember new`.

Сообщество разработчиков Ember продолжает сопровождать этот замечательный фреймворк и наращивать его эффективность по мере развития JavaScript. Этот фреймворк создан людьми, столкнувшимися с теми же проблемами, которые выпадут и на вашу долю в процессе совершенствования навыков работы с JavaScript. Смело задавайте вопросы, когда что-то не получается, помогайте исправлять ошибки, если обнаружите, что что-то не работает, но не только берите, а и давайте сами. Вы теперь часть большого сообщества JavaScript-разработчиков.

Бронзовое упражнение: настройка предупреждающего сообщения

Срабатывающее при добавлении наблюдения `{{flash-alert}}` представляет собой общее уведомление. Добавьте в него место наблюдения и дату.

Серебряное упражнение: сделайте из NavBar компонент

Сделайте NavBar компонентом в шаблоне приложения. Добавьте свойство `state` для отображения двух вариантов навигации. Добавьте в компонент NavBar условные операторы для отображения конкретных ссылок.

Золотое упражнение: массив предупреждающих сообщений

Переделайте компонент `flash-alert`, чтобы он принимал массив предупреждающих сообщений с различными типами и текстами сообщений. Возможно, потребуется, чтобы на экране отображалось несколько предупреждений одновременно. При создании предупреждающего сообщения воспользуйтесь `Ember.ArrayProxy` вместо отдельных свойств. Добавьте в массив вместе с текстом сообщения и типом новое свойство `index`, чтобы иметь возможность удалять элемент из массива по щелчку кнопкой мыши.

Послесловие

Поздравляем! Вы добрались до конца этого руководства. Не у всех хватает силы воли сделать то, что вы сделали, и выучить то, что выучили вы. Возьмите за это с полки пирожок.

Ваш нелегкий труд вознагражден: теперь вы разработчик клиентской части.

Последнее упражнение

У нас осталось последнее (самое сложное) задание для вас: станьте *хорошим* разработчиком клиентской части. Все хорошие разработчики хороши по-своему, так что начиная с этого момента вам придется искать свой собственный путь.

С чего же вам начать? Вот несколько идей по этому поводу.

Пишите код. Начните прямо сейчас. Вы быстро забудете все, чему научились, если не будете применять свои знания на практике. Примите участие в каком-нибудь проекте или напишите простое приложение. Что бы вы ни делали, не теряйте времени: пишите код.

Учитесь. Вы узнали из этой книги о многом по чуть-чуть. Заставило ли что-то ваши глаза загореться? Напишите небольшой код, чтобы потренироваться с наиболее заинтересовавшей вас функцией. Найдите и прочитайте об этом всю имеющуюся документацию или целую книгу, если она существует. Загляните также в подкаст Jabber по языку программирования JavaScript, чтобы окунуться в увлекательное обсуждение последних событий в сфере разработки клиентской части (`devchat.tv/js-jabber`).

Встречайтесь с людьми. Местные встречи разработчиков — отличное место, чтобы найти единомышленников. Многие высококлассные разработчики клиентской части активно ведут блоги в Twitter. Вы также можете посещать конференции на тему разработки клиентской части, чтобы встретить там программистов (возможно, даже нас!).

Подберите для себя сообщество разработчиков программного обеспечения с открытым исходным кодом. Разработка клиентской части просто кипит на www.github.com. Отыскав хорошую библиотеку, поинтересуйтесь другими проектами

ее авторов. Распространяйте и свой собственный код, ведь никогда не знаешь, кто посчитает его полезным. Мы считаем, что список рассылки WRDL (Web Development Reading List, список для чтения по веб-разработке) — отличная возможность быть в курсе происходящего в сообществе разработчиков клиентской части (`wdrl.info`).

Нескромная реклама

Вы можете найти нас в Twitter: Криса — под именем @radishmouse, а Тодда — @tgandee.

Если эта книга вам понравилась, обратите внимание на другие руководства Big Nerd Ranch на www.bignerdranch.com/books. У нас также имеется широкий выбор недельных курсов для разработчиков, на которых мы помогаем выучить всего за неделю материал, достойный целой книги. И конечно, если вам просто нужен кто-то, кто бы написал крутой код, мы также выполняем работы по контракту. Для получения дальнейшей информации загляните на www.bignerdranch.com.

Спасибо

Без таких читателей, как вы, наша работа была бы невозможна. Спасибо вам за то, что купили и прочитали нашу книгу.