3.1

DJANGO

for

PROFESSIONALS

Production websites with Python & Django

WILLIAM S. VINCENT

Django для профессионалов

ITcoder

От переводчика...

После изучения основ **Django** возникает огромный разрыв между созданием простых **«**игрушечных приложений**»** и тем**,** что требуется для создания **«**готового к работе**»** веб**-**приложения**,** подходящего для развертывания тысячами или даже миллионами пользователей**.**

В этой книге подробно рассказывается**,** как профессиональные программисты **Django** выполняют свою работу и создают настоящие веб**-**приложения**.** Мы создадим с нуля онлайн**-**магазин книг**.** И мы подробно рассмотрим следующие темы**:**

- **Django** и **Python**
- расширенная регистрация пользователей**:** пользовательские модели пользователей**,** социальная аутентификация
- **Docker & PostgreSQL** локально
- переменные окружения **(**не более нескольких файлов настроек**!)**
- разрешения и загрузка файлов **/** изображений
- поиск
- безопасность и производительность
- развертывание

Действительно хорошо освоите **docker** плюс к тому же научитесь писать тесты и запускать их внутри **docker.** Книга действительно научит вас поднимать проект с нуля до продакшена**.**

Примечание **:** у автора вышла новая обновленная версия книги **4.0 ,** так что у официального репозитория версия уже другая**.**

Код с этой версии книги вы сможете найти у меня на **Github .**

Всем желаю успехов в разработке.

ITcoder

Django для профессионалов

Создание веб-сайтов с помощью **Python** и **Django**

William S. Vincent

Эта книга продается на сайте **http://leanpub.com/djangoforprofessionals**

Эта версия была опубликована **2020-09-18**

Это книга **Leanpub. Leanpub** предоставляет авторам и издателям возможность использовать процесс публикации **Lean Publishing.**

Издательство **Lean Publishing -** это публикация электронной книги в процессе работы с использованием легких инструментов и множества итераций для получения отзывов читателей, внесения изменений до тех пор, пока у вас не получится нужная книга, а после этого вы сможете добиться успеха.

Другие работы **William S. Vincent**

Django for beginners

Django for API

Перевод которых вы можете найти найти у нас в телеграм канале

Backend | Python - @python_itkg

Содержание

Глава 3: Проект книжного магазина

- **Docker**
- **PostgreSQL**
- Собственная модель пользователя
- Собственные формы пользователя
- Пользовательская администрация
- Суперпользователь
- Тесты
- Юнит-тесты
- **Git**
- Заключение

Глава 4: Приложение Pages

- Шаблоны
- **URL** и представления
- Тесты
- Тестирование шаблонов
- Тестирование **HTML**
- метод **setUp**
- Решение
- **Git**
- Заключение

Глава 5: Регистрация пользователей

- Приложение **Auth**
- Авторизованные **URL-**адреса и представления
- Домашняя страница
- Исходный код **Django**
- Вход в систему **(Log In)**
- Переадресация
- Выход из системы **(Log Out)**
- Регистрация
- Тесты
- **setUpTestData()**
- **Git**
- Заключение

Глава 6: Статические файлы

- приложение **staticfiles**
- **STATIC_URL**
- **STATICFILES_DIRS**
- **STATIC_ROOT**
- **STATICFILES_FINDERS**
- Статические директории
- Изображения
- **JavaScript**
- **collectstatic**
- **Bootstrap**
- Страница о сайте
- **Django Crispy Forms**
- Тесты
- **Git**
- Заключение

Глава 7: Расширенная регистрация пользователей

- **django-allauth**
- **AUTHENTICATION_BACKENDS**
- **EMAIL_BACKEND**
- **ACCOUNT_LOGOUT_REDIRECT**
- **URLs**
- Шаблоны
- Вход в систему **(Log In)**
- Выйти из системы **(Log Out**
- Регистрация
- Администратор
- Вход только по электронной почте
- Тесты
- **Social**
- **Git**
- Заключение

Глава 8: Переменные окружения

- **environs[django**
- **SECRET_KEY**
- **DEBUG** и **ALLOWED_HOSTS**
- **DATABASES**
- **Git**
- Заключение

Глава 9: Электронная почта

- Собственные письма-подтверждения
- Страница подтверждения электронной почты
- Сброс пароля и смена пароля
- Служба электронной почты
- **Git**
- Заключение

Глава 10: Книжное приложение

- Модели
- Администратор
- **URLs**
- Представления
- Шаблоны
- **object_list**
- Страница индивидуальной книги
- **contex_object_name**
- **get_absolute_url**
- Первичные ключи против **ID**
- **Slugs** против **UUID**
- Навигационная панель
- Тесты
- **Git**
- Заключение

Глава 11: Приложение для отзывов

- **Foreign Keys**
- Модель отзывов
- Администратор
- Шаблоны
- Тесты
- **Git**
- Заключение

Глава 12: Загрузка файлов/изображений

- Медиафайлы
- Модели
- Администратор
- Шаблон
- Следующие шаги
- **Git**
- Заключение

Глава 13: Права доступа (Permissions)

- Только зарегистрированные пользователи
- Разрешения
- Собственные разрешения
- Права доступа пользователей
- **PermissionRequiredMixin**
- **Groups & UserPassesTestMixin**
- Тесты
- **Git**
- Заключение

Глава 14: Поиск

- Страница результатов поиска
- Базовая фильтрация
- Объекты **Q**
- Формы
- Форма поиска
- **Git**
- Заключение

Глава 15: Производительность

- **django-debug-toolbar**
- Анализ страниц
- **select_related** и **prefetch_related**
- Кэширование
- Индексы
- **django-extensions**
- Фронтенд-активы
- **Git**
- Заключение

Глава 16: Безопасность

- Социальная инженерия
- Обновления **Django**
- Контрольный список развертывания
- **docker-compose-prod.yml**
- **DEBUG**
- Параметры по умолчанию
- **SECRET_KEY**
- **Web** Безопасность
- **SQL-**инъекция
- **XSS (**межсайтовый скриптинг**)**
- Подделка межсайтовых запросов **(CSRF)**
- Защита от **Clickjacking**
- **HTTPS/SSL**
- **HTTP Strict Transport Security (HSTS)**
- Безопасные файлы **cookie**
- Повышение надежности администратора
- **Git**
- Заключение

Глава 17: Развертывание

- **PaaS** против **IaaS**
- **WhiteNoise**
- Медиафайлы
- **Gunicorn**
- **Heroku**
- Развертывание с помощью **Docker**
- **heroku.yml**
- Развертывание **Heroku**
- **SECURE_PROXY_SSL_HEADER**
- Журналы **Heroku**
- Дополнения **Heroku**
- Заключение

Заключение

Обучающие ресурсы

Обратная связь

Введение

Добро пожаловать в **Django** для профессионалов **-** руководство по созданию профессиональных веб**-**сайтов с помощью веб**-**фреймворка **Django.** Существует огромная пропасть между созданием простых "игрушечных приложений"**,** которые можно быстро создать и развернуть**,** и тем**,** что требуется для создания "готового к производству" веб**-**приложения**,** пригодного для развертывания для тысяч или даже миллионов пользователей**.** Эта книга покажет вам**,** как преодолеть этот пробел**.**

Когда вы впервые устанавливаете **Django** и создаете новый проект**,** настройки по умолчанию ориентированы на быструю локальную разработку**.** И в этом есть смысл**:** нет необходимости добавлять все дополнительные функции**,** необходимые для большого веб**-**сайта**,** пока вы не поймете**,** что они вам нужны**.** Эти настройки по умолчанию включают **SQLite** в качестве базы данных по умолчанию**,** локальный веб**-**сервер**,** локальный хостинг статичных свойств**,** встроенную модель пользователя и включенный режим **DEBUG.**

Но для производственного проекта многие**,** если не большинство**,** из этих настроек должны быть перенастроены**.** И даже тогда может возникнуть разочаровывающее отсутствие соглашения между экспертами**.** Например**,** какую базу данных лучше всего использовать в производственном проекте**?** Многие разработчики **Django**, в том числе и я**,** выбирают **PostgreSQL.** Именно его мы будем использовать в этой книге**.** Однако в зависимости от проекта можно привести аргументы в пользу **MySQL.** На самом деле все зависит от конкретных требований проекта**.**

Вместо того**,** чтобы перегружать читателя полным набором доступных вариантов**,** эта книга показывает один подход**,** основанный на современных лучших практиках сообщества **Django**, для создания профессионального веб**-**сайта**.** Рассматриваются такие темы**,** как использование **Docker** для локальной разработки и развертывания**, PostgreSQL**, пользовательская модель пользователей**,** надежная аутентификация пользователей с помощью электронной почты**,** всестороннее тестирование**,** переменные окружения**,** повышение безопасности и производительности и многое другое**.**

К концу этой книги вы создадите профессиональный веб**-**сайт и освоите все необходимые для этого шаги**.** Начинаете ли вы новый проект**,** который надеется стать таким же большим**,** как **Instagram (**в настоящее время это самый большой **Django-**сайт в мире**),** или же вносите столь необходимые обновления в существующий **Django-**проект**,** у вас будут все необходимые инструменты и знания для этого**.**

Необходимые требования

Если вы совсем новичок в **Django** или веб-разработке, эта книга не для вас. Темп будет слишком быстрым. Хотя вы можете читать, копировать весь код и в конце получить работающий сайт, я рекомендую начать с моей книги **"Django** для начинающих**".** Она начинается с самых основ и постепенно вводит понятия через создание пяти все более сложных приложений **Django.** После прочтения этой книги вы будете готовы к успеху с этой книгой.

Я также написал книгу о преобразовании веб-сайтов **Django** в веб-интерфейсы **API** под названием **Django for APIs.** На практике большинство разработчиков **Django** работают в командах с другими разработчиками и фокусируются на **back-end API,** а не на полнофункциональных веб-приложениях, требующих специальных **JavaScript front-end.** Поэтому чтение **Django for APIs** будет полезно для вашего образования как **Django-**разработчика, но не обязательно перед чтением этой книги.

Мы будем использовать **Docker** на протяжении большей части этой книги, но все еще полагаемся на то, что **Python 3**, **Django** и **Pipenv** будут установлены локально. **Git** также является необходимой частью инструментария разработчика. Наконец, в этой книге мы также будем широко использовать командную строку, поэтому, если вам нужно освежить ее в памяти, пожалуйста, посмотрите здесь.

Структура книги

Глава **1** начинается с введения в **Docker** и рассказывает о том, как " применить **Docker"** к традиционному проекту **Django.** В главе **2** представлена **PostgreSQL -** база данных, готовая к использованию на производстве, которую мы можем запустить локально в нашей среде **Docker.** Затем в главе **3** начинается основной проект книги: онлайн-магазин книг с собственной моделью пользователя, поиском, загрузкой изображений, правами доступа и множеством других возможностей.

Глава **4** посвящена созданию приложения **Pages** для базовой домашней страницы, а также надежному тестированию, которое включается в каждую новую функцию сайта. В главе **5** с нуля реализован полный поток регистрации пользователей с использованием встроенного приложения **auth** для регистрации, входа и выхода. Глава **6** знакомит с правильной конфигурацией статических ресурсов для **CSS, JavaScript** и изображений, а также с добавлением **Bootstrap** для создания стиля.

В главе **7** акцент переносится на расширенную регистрацию пользователей, а именно: вход только по электронной почте и социальную аутентификацию с помощью стороннего пакета **django-allauth.**

В главе **8** представлены переменные окружения **-** ключевой компонент разработки **12-**факторного приложения и передовой опыт**,** широко используемый в сообществе веб**-**разработчиков**.** Завершая настройку нашего проекта**,** глава **9** посвящена электронной почте и добавлению специализированного стороннего провайдера**.**

Структура первой половины книги является намеренной**.** Когда придет время создавать свои собственные проекты **Django,** есть вероятность**,** что вы будете повторять многие из тех же шагов из глав **3-9.** В конце концов**,** каждый новый проект нуждается в правильной конфигурации**,** аутентификации пользователя и переменных окружения**.** Поэтому рассматривайте эти главы как подробное объяснение и руководство**.** Вторая половина книги посвящена конкретным функциям**,** связанным с нашим сайтом книжного магазина**.**

Глава **10** начинается с создания моделей**,** тестов и страниц для нашего книжного магазина с помощью приложения **Books.** Также обсуждается **URL-**адреса и переход от **id** к **slug** и **UUID (Universally Unique IDentifier)** в **URL-**адресах**.** В главе **11** добавлено добавление обзоров в наш книжный магазин и рассматриваются внешние ключи**.**

В главе **12** добавляется загрузка изображений**,** а в главе **13** устанавливаются разрешения на весь сайт для его блокировки**.** Для любого сайта**,** но особенно для электронной коммерции**,** поиск является жизненно важным компонентом**,** и в главе **14** рассматривается создание формы и все более сложных фильтров поиска для сайта**.**

В главе **15** внимание переключается на оптимизацию производительности**,** включая добавление панели инструментов **django- debug-toolbar** для проверки запросов и шаблонов**,** индексов базы данных**,** внешних ресурсов**,** а также множества встроенных опций кэширования**.** Глава **16** посвящена безопасности в **Django,** как встроенным опциям**,** так и дополнительным конфигурациям**,** которые могут и должны быть добавлены для производственной среды**.** Последний раздел**,** глава **17,** посвящен развертыванию**,** стандартным обновлениям**,** необходимым для перехода с веб**-**сервера **Django,** локальной работе со статическими файлами и настройке **ALLOWED_HOSTS.**

В Заключении рассматриваются различные дальнейшие шаги по проекту и дополнительные лучшие практики **Django.**

В главе **8** представлены переменные окружения **-** ключевой компонент разработки **12-**факторного приложения и передовой опыт**,** широко используемый в сообществе веб**-**разработчиков**.** Завершая настройку нашего проекта**,** глава **9** посвящена электронной почте и добавлению специализированного стороннего провайдера**.**

Структура первой половины книги является намеренной**.** Когда придет время создавать свои собственные проекты **Django**, есть вероятность**,** что вы будете повторять многие из тех же шагов из глав **3-9.** В конце концов**,** каждый новый проект нуждается в правильной конфигурации**,** аутентификации пользователя и переменных окружения**.** Поэтому рассматривайте эти главы как подробное объяснение и руководство**.** Вторая половина книги посвящена конкретным функциям**,** связанным с нашим сайтом книжного магазина**.**

Глава **10** начинается с создания моделей**,** тестов и страниц для нашего книжного магазина с помощью приложения **Books.** Также обсуждается **URL-**адреса и переход от **id** к **slug** и **UUID (Universally Unique IDentifier)** в **URL-**адресах**.** В главе **11** добавлено добавление обзоров в наш книжный магазин и рассматриваются внешние ключи**.**

В главе **12** добавляется загрузка изображений**,** а в главе **13** устанавливаются разрешения на весь сайт для его блокировки**.** Для любого сайта**,** но особенно для электронной коммерции**,** поиск является жизненно важным компонентом**,** и в главе **14** рассматривается создание формы и все более сложных фильтров поиска для сайта**.**

В главе **15** внимание переключается на оптимизацию производительности**,** включая добавление панели инструментов **django- debug-toolbar** для проверки запросов и шаблонов**,** индексов базы данных**,** внешних ресурсов**,** а также множества встроенных опций кэширования**.** Глава **16** посвящена безопасности в **Django,** как встроенным опциям**,** так и дополнительным конфигурациям**,** которые могут и должны быть добавлены для производственной среды**.** Последний раздел**,** глава **17,** посвящен развертыванию**,** стандартным обновлениям**,** необходимым для перехода с веб**-**сервера **Django,** локальной работе со статическими файлами и настройке **ALLOWED_HOSTS.**

В Заключении рассматриваются различные дальнейшие шаги по проекту и дополнительные лучшие практики **Django.**

Макет книги

В этой книге много примеров кода**,** которые оформлены следующим образом**:**

code

```
 # Это код Python
print(Hello, World)
```

Для краткости мы будем использовать точки **...** для обозначения существующего кода, который остается неизменным, например, в функции, которую мы обновляем.

code

```
def make_my_website:
    ...
    print("All done!")
```

Мы также будем часто использовать консоль командной строки для выполнения команд, которые в традиционном стиле **Unix** имеют форму префикса **$.**

Командная строка

```
$ echo "hello, world"
```

В результате выполнения этой команды в следующей строке будет указано:

Командная строка

```
"hello, world"
```

Обычно команда и ее вывод объединяются для краткости. Перед командой всегда ставится символ **$,** а перед результатом **-** нет. Например, приведенная выше команда и результат будут представлены следующим образом:

Командная строка

```
$ echo "hello, world"
hello, world
```

Текстовый редактор

Современный текстовый редактор - обязательная часть инструментария любого разработчика программного обеспечения. Среди прочих возможностей они поставляются с плагинами, которые помогают форматировать и исправлять ошибки в коде **Python.** Популярные варианты включают **Black**, **autopep8** и **YAPF.**

Опытные разработчики, возможно, по-прежнему предпочитают использовать **Vim** или **Emacs**, но новички и все более опытные программисты также предпочитают современные текстовые редакторы, такие как **VSCode**, **Atom**, **Sublime Text** или **PyCharm.**

Заключение

Django - отличный выбор для любого разработчика, который хочет создавать современные, надежные веб-приложения с минимальным количеством кода. Он популярен, активно развивается и тщательно протестирован крупнейшими веб-сайтами в мире.

Полный исходный код книги можно найти в официальном репозитории **Github.**

В следующей главе мы узнаем, как настроить любой компьютер для разработки **Django** с помощью **Docker.**

Глава **1: Docker**

Правильная настройка локальной среды разработки остается сложной задачей, несмотря на все другие достижения современного программирования. Просто существует слишком много переменных: разные компьютеры, операционные системы, версии **Django**, варианты виртуального окружения и так далее. Если добавить к этому сложность работы в команде, где все должны иметь одинаковые настройки, то проблема только усугубляется.

В последние годы появилось решение: **Docker**. Хотя **Docker** существует всего несколько лет, он быстро стал выбором по умолчанию для многих разработчиков, работающих над проектами производственного уровня.

С помощью **Docker** наконец-то стало возможным точно и надежно воспроизвести производственную среду локально, начиная от правильной версии **Python** до установки **Django** и запуска дополнительных сервисов, таких как база данных производственного уровня. Это означает, что больше не имеет значения, на каком компьютере вы работаете **- Mac**, **Windows** или **Linux.** Все работает в самом **Docker**.

Docker также значительно упрощает совместную работу в командах. Прошли времена обмена длинными, устаревшими файлами **README** для добавления нового разработчика в групповой проект. Вместо этого с помощью **Docker** вы просто делитесь двумя файлами **- Dockerfile** и **docker-compose.yml -** и разработчик может быть уверен, что его локальная среда разработки точно такая же, как и у остальных членов команды.

Docker не является совершенной технологией. Она все еще относительно новая, сложная внутри и находится в стадии активной разработки. Но перспектива, к которой она стремится **-** согласованная и совместно используемая среда разработки, которую можно запустить как локально на любом компьютере, так и развернуть на любом сервере **-** делает ее надежным выбором.

В этой главе мы узнаем немного больше о самом **Docker** и " докеризируем" наш первый проект **Django.**

Что такое Docker?

Docker - это способ изолировать всю операционную систему с помощью контейнеров **Linux**, которые являются одним из видов виртуализации. Виртуализация уходит своими корнями в начало компьютерной науки, когда большие и дорогие компьютеры типа мэйнфрейм были нормой. Как несколько программистов могли использовать одну и ту же машину? Ответом стала технология виртуализации и, в частности, виртуальные машины, которые представляют собой полные копии компьютерной системы, начиная с операционной системы.

Если вы арендуете место у облачного провайдера, например, **Amazon Web Services (AWS)**, они, как правило, не предоставляют вам выделенную часть оборудования. Вместо этого вы используете один физический сервер совместно с другими клиентами. Но поскольку каждый клиент имеет свою виртуальную машину, работающую на сервере, клиенту кажется, что у него есть свой собственный сервер.

Именно эта технология позволяет быстро добавлять или удалять серверы у облачного провайдера. В основном за этой технологией стоит скрытое программное обеспечение, а не фактическое оборудование.

В чем недостаток виртуальной машины? Размер и скорость. Типичная гостевая операционная система может легко занять **700** МБ. Поэтому если один физический сервер поддерживает три виртуальные машины, это как минимум **2,1** ГБ дискового пространства, а также отдельные потребности в ресурсах процессора и памяти.

На помощь приходит **Docker**. Основная идея заключается в том, что большинство компьютеров работают на базе одной и той же операционной системы **Linux**, так что если бы мы виртуализировали, начиная с уровня **Linux**? Разве это не обеспечит легкий и быстрый способ дублирования большей части тех же функций? Ответ - да. И в последние годы контейнеры **Linux** стали широко популярны. Для большинства приложений - особенно веб-приложений - виртуальная машина предоставляет гораздо больше ресурсов, чем требуется, и контейнера более чем достаточно.

По сути, это и есть **Docker**: способ реализации **Linux-**контейнеров!

Здесь можно использовать аналогию с домами и квартирами. Виртуальные машины - это как дома: отдельные здания с собственной инфраструктурой, включая водопровод и отопление, а также кухню, ванные комнаты, спальни и так далее. Контейнеры **Docker** похожи на квартиры: они имеют общую инфраструктуру, такую как водопровод и отопление, но бывают разных размеров, которые соответствуют точным потребностям владельца.

Контейнеры против виртуальных сред

Как программист **Python** вы уже должны быть знакомы с концепцией виртуальных сред, которые представляют собой способ изолировать пакеты **Python**. Благодаря виртуальным средам на одном компьютере можно локально запускать несколько проектов. Например, проект А может использовать **Python 3.4** и **Django 1.11** среди прочих зависимостей; в то время как проект Б использует **Python 3.8** и **Django 3.1**. Настроив выделенную виртуальную среду для каждого проекта, мы можем управлять этими различными программными пакетами, не загрязняя при этом нашу глобальную среду.

Смущает то, что сейчас существует множество популярных инструментов для реализации виртуальных сред: от **virtualenv** до **venv** и **Pipenv**, но по сути все они делают одно и то же.

Важное различие между виртуальными окружениями и **Docker** заключается в том, что виртуальные окружения могут изолировать только пакеты **Python.** Они не могут изолировать не-**Python** программное обеспечение, например, базу данных **PostgreSQL** или **MySQL.** И они все еще полагаются на глобальную установку **Python** на уровне системы (другими словами, на вашем компьютере). Виртуальная среда указывает на существующую установку **Python;** она не содержит самого **Python.**

Контейнеры **Linux** идут на шаг дальше и изолируют всю операционную систему, а не только части **Python.** Другими словами, мы установим сам **Python** в **Docker**, а также установим и запустим базу данных производственного уровня.

Docker сам по себе является сложной темой, и мы не будем так глубоко погружаться в нее в этой книге, однако понимание его происхождения и ключевых компонентов очень важно. Если вы хотите узнать об этом больше, я рекомендую видеокурс "Погружение в **Docker**".

Установите Docker

Итак, достаточно теории. Давайте начнем использовать **Docker** и **Django** вместе. Первым шагом будет регистрация бесплатного аккаунта на **Docker Hub,** а затем установка настольного приложения **Docker** на вашу локальную систему:

- **Docker** для **Mac**
- **Docker** для **Windows**
- **Docker** для **Linux**

Загрузка может занять некоторое время, так как это большой файл! Не стесняйтесь размять ноги на этом этапе.

Обратите внимание, что в версии для **Linux** пользователь имеет права **root**, другими словами, вы можете делать все, что угодно. Это часто не идеально, и при желании вы можете настроить **Docker** на запуск от имени пользователя, не являющегося **root**.

После завершения установки **Docker** мы можем убедиться, что запущена правильная версия, набрав в командной строке команду **docker --version.** Она должна быть не ниже версии **18.**

Командная строка

```
$ docker --version
Docker version 19.03.12, build 48a66213fe
```

Docker часто используется с дополнительным инструментом, **Docker Compose**, который помогает автоматизировать команды. **Docker Compose** входит в состав загрузок для **Mac** и **Windows,** но если вы работаете на **Linux,** вам придется добавить его вручную. Это можно сделать, выполнив команду **sudo pip install docker-compose** после завершения установки **Docker.**

Docker Hello, World

Docker поставляется с собственным образом **"Hello, World",** который будет полезно запустить на первом этапе**.** В командной строке введите **docker run hello-world .** Это позволит загрузить официальный образ **Docker,** а затем запустить его в контейнере**.** Мы обсудим как образы**,** так и контейнеры в ближайшее время**.**

Командная строка

```
$ docker run hello-world
Unable to find image 'hello-world:latest' locally
latest: Pulling from library/hello-world
1b930d010525: Pull complete
Digest:
sha256:b8ba256769a0ac28dd126d584e0a2011cd2877f3f76e093a7ae560f2a5301c00
Status: Downloaded newer image for hello-world:latest
Hello from Docker!
This message shows that your installation appears to be working correctly.
To generate this message, Docker took the following steps:
 1. The Docker client contacted the Docker daemon.
 2. The Docker daemon pulled the "hello-world" image from the Docker Hub. (amd64)
 3. The Docker daemon created a new container from that image which runs the
    executable that produces the output you are currently reading.
 4. The Docker daemon streamed that output to the Docker client, which sent it to your
    terminal.
To try something more ambitious, you can run an Ubuntu container with:
$ docker run -it ubuntu bash
Share images, automate workflows, and more with a free Docker ID:
https://hub.docker.com/
For more examples and ideas, visit:
https://docs.docker.com/get-started/
```

Команда **docker info** позволяет нам проверить **Docker.** Она будет содержать много выходных данных**,** но сосредоточьтесь на верхних строках**,** которые показывают**,** что сейчас у нас есть **1** контейнер**,** который остановлен**,** и **1** образ**.**

Командная строка

```
$ docker info
Client:
    Debug Mode: false
Server:
    Containers: 1
        Running: 0
        Paused: 0
        Stopped: 1
     Images: 1
...
```

Это означает**,** что **Docker** успешно установлен и запущен**.**

Django Hello, World

Сейчас мы создадим проект **Django "Hello, World",** который будет работать локально на нашем компьютере, а затем полностью перенесем его в **Docker,** чтобы вы могли увидеть, как все части подходят друг другу.

Первым шагом будет выбор места для нашего кода. Это может быть любое место на вашем компьютере, но если вы работаете на **Mac,** то легко найти это место на рабочем столе. Из командной строки перейдите на Рабочий стол и создайте директорию кода для всех примеров кода в этой книге.

Командная строка

```
$ cd ~/Desktop
$ mkdir code && cd code
```

Затем создайте директорию **hello** для этого примера и установите **Django** с помощью **Pipenv,** который создает **Pipfile** и файл **Pipfile.lock.** Активируйте виртуальную среду с помощью команды **shell.**

Командная строка

```
$ mkdir hello && cd hello
$ pipenv install django~=3.1.0
$ pipenv shell
(hello) $
```

> Если вам нужна помощь в установке **Pipenv** или **Python 3,** вы можете найти более подробную информацию здесь.

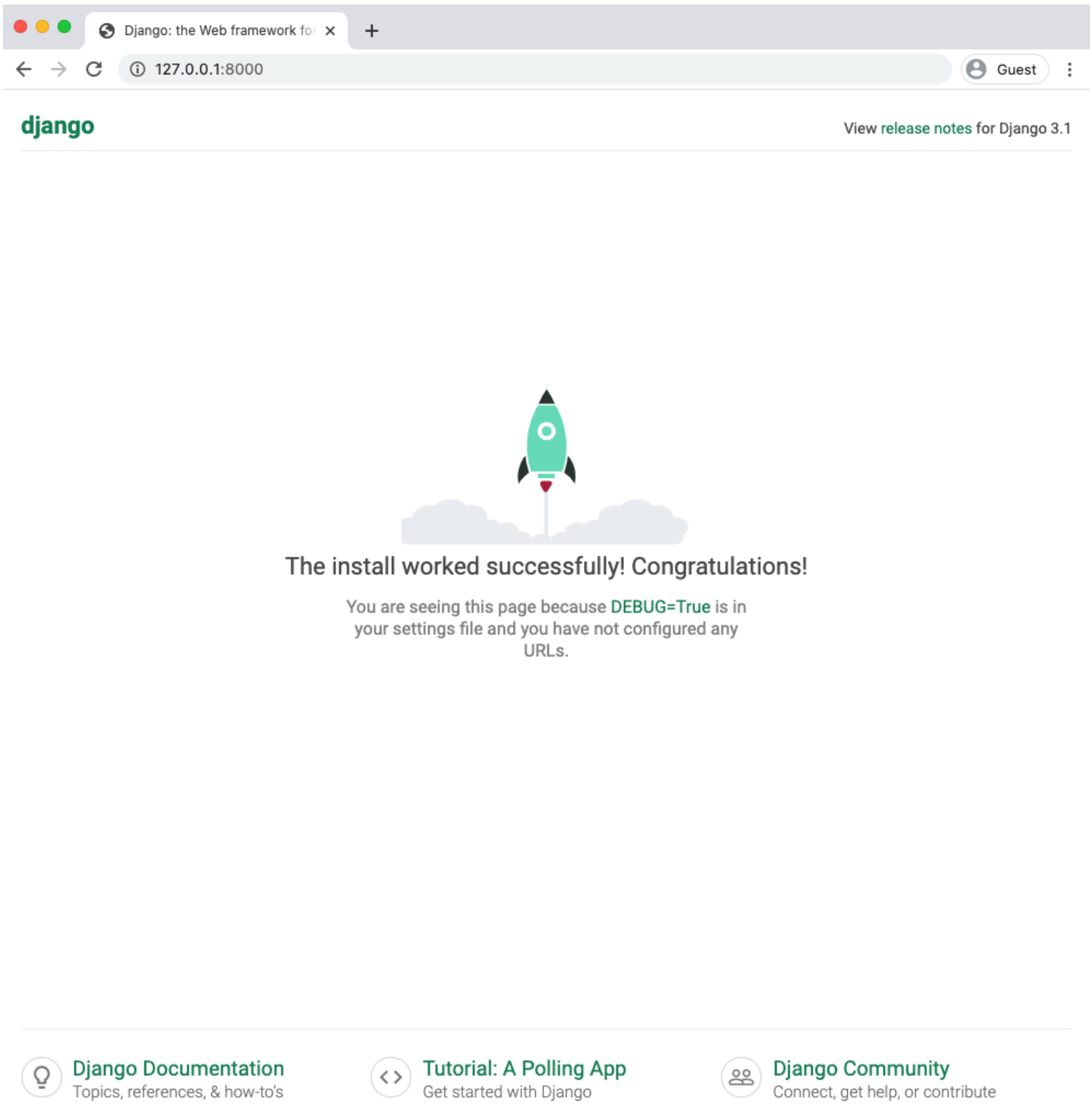
Теперь мы можем использовать команду **startproject** для создания нового проекта **Django** под названием **config .** Добавление точки, **.** в конце команды **-** это необязательный шаг, но многие разработчики **Django** делают его. Без точки **Django** добавляет дополнительный каталог к проекту, а с точкой **-** нет.

Наконец, используйте команду **migrate** для инициализации базы данных и запустите локальный веб-сервер командой **runserver.**

Командная строка

```
(hello) $ django-admin startproject config .
(hello) $ python manage.py migrate
(hello) $ python manage.py runserver
```

Предполагая, что все сработало правильно, вы должны теперь иметь возможность перейти к просмотру страницы **Django Welcome** по адресу **http://127.0.0.1:8000/** в вашем веб-браузере.



Django welcome page

Приложение Pages(страница)

Теперь мы сделаем простую домашнюю страницу**,** создав для нее специальное приложение **pages.** Остановите локальный сервер**,** набрав **Control+c,** а затем используйте команду **startapp,** добавив имя нужной страницы**.**

Командная строка

```
(hello) $ python manage.py startapp pages
```

Django автоматически устанавливает для нас новый директорий **pages** и несколько файлов**.** Но даже если приложение было создано**,** наш конфигуратор не узнает его**,** пока мы не добавим его в конфигурацию **INSTALLED_APPS** в файле **config/settings.py. Django** загружает приложения сверху вниз**,** поэтому**,** вообще говоря**,** хорошей практикой является добавление новых приложений ниже встроенных приложений**,** на которые они могут полагаться**,** таких как **admin**, **auth** и все остальные**.**

Code

```
# config/settings.py
INSTALLED_APPS = [
        'django.contrib.admin',
        'django.contrib.auth',
        'django.contrib.contenttypes',
        'django.contrib.sessions',
        'django.contrib.messages',
        'django.contrib.staticfiles',
        'pages',  # new
]
```

Теперь мы можем задать маршрут **URL** для приложения **pages.** Поскольку мы хотим**,** чтобы наше сообщение появилось на домашней странице**,** мы будем использовать пустую строку **''.** Не забудьте так же добавить импорт **include** во вторую строку**.**

Code

```
# config/urls.py
from django.contrib import admin
from django.urls import path, include #new

urlpatterns = [
      path( 'admin/', admin.site.urls),
      path('', include('pages.urls')), # new
]
```

Вместо того чтобы устанавливать шаблон на данном этапе**,** мы можем просто жестко закодировать сообщение в нашем уровне представления в **pages/views.py,** которое будет выводить строку **"Hello, World!".**

Code

```
# pages/views.py
from django.http import HttpResponse

def home_page_view(request):
    return HttpResponse('Hello, World!')
```

Что дальше**?** Наш последний шаг **-** создать файл **urls.py** в приложении **pages** и связать его с **home_- page_view.** Если вы работаете на компьютере **Mac** или **Linux,** команда **touch** может быть использована из командной строки для создания новых файлов**.** В **Windows** создайте новый файл с помощью текстового редактора**.**

Command Line

```
(hello) $ touch pages/urls.py
```

В пути импорта вашего текстового редактора в верхней строке добавьте **home_page_view ,** а затем задайте его маршрут**,** чтобы он снова был пустой строкой **''.** Обратите внимание**,** что мы также предоставляем дополнительное имя**, home,** для этого маршрута**,** что является лучшей практикой**.**

Code

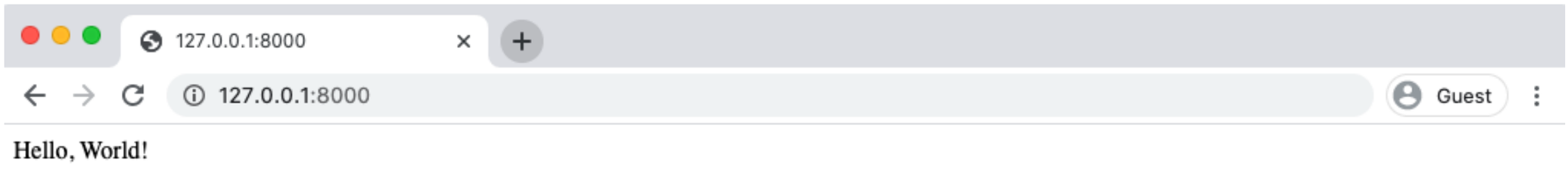
```
# pages/urls.py
from django.urls import path
from .views import home_page_view

urlpatterns = [
    path('', home_page_view, name='home')
]
```

Полный поток нашей домашней страницы **Django** выглядит следующим образом**: *** когда пользователь заходит на главную страницу**,** он сначала попадает на **config/urls.py *** затем на **pages/urls.py *** и**,** наконец**,** направляется на **home_page_view,** который возвращает строку **"Hello, World!".** Наша работа закончена для базовой домашней страницы**.** Запустите локальный сервер снова**.**

Command Line

```
(hello) $ python3 manage.py runserver
```

Если вы обновите веб**-**браузер на сайте **http://127.0.0.1:8000/,** он теперь выведет наше желаемое сообщение**.**



Hello World

Теперь пришло время переключиться на **Docker.** Снова остановите локальный сервер с помощью **Control+c** и выйдите из нашей виртуальной среды**,** поскольку она нам больше не нужна**,** набрав **exit.**

Command Line

```
(hello) $ exit
$
```

Как узнать**,** что виртуальная среда больше не активна**?** Больше не будет круглых скобок вокруг имени каталога в строке командной строки**.** Любые обычные команды **Django,** которые вы попытаетесь запустить в этот момент**,** будут неудачными**.** Например**,** попробуйте выполнить **python manage.py runserver** и посмотрите**,** что произойдет**.**

Command Line

```
$ python manage.py runserver
File "./manage.py", line 14
    ) from exc
^
SyntaxError: invalid syntax
```

Это означает**,** что мы полностью вышли из виртуальной среды и готовы к работе с **Docker.**

Образы, контейнеры и хост Docker

Образ **Docker -** это снимок во времени того**,** что содержит проект**.** Он представлен **Dockerfile**и буквально является списком инструкций**,** которые н браза**.** Если продолжить нашу аналогию с квартирой**,** то образ **-** это чертеж или набор планы квартиры**;** контейнер **-** это реальное**,** полностью построенное здание**.** Третья основная концепция **-** это "хост **Docker",** который является базовой ОС**.** Можно иметь несколько контейнеров**,** запущенных на одном хосте **Docker.** Когда мы говорим о коде или процессах выполняющихся в **Docker,** это означает**,** что они выполняются на хосте **Docker.** Давайте создадим наш первый **Dockerfile,** чтобы увидеть всю эту теорию в действии

Command Line

$ touch Dockerfile

В **Dockerfile** добавьте следующий код**,** который мы рассмотрим построчно ниже**.**

Dockerfile

```
# Pull base image
FROM python:3.8

# Set environment variables
ENV PYTHONDONTWRITEBYTECODE 1
ENV PYTHONUNBUFFERED 1

# Set work directory
WORKDIR /code

# Install dependencies
COPY Pipfile Pipfile.lock /code/
RUN pip install pipenv && pipenv install --system

# Copy project
COPY . /code/
```

При создании образа **Dockerfile** читаются сверху вниз**.** Первой командой должна быть команда **FROM,** которая позволяет нам импортировать базовый образ для использования в нашем образе**,** в данном случае **Python 3.8.**

Затем мы используем команду **ENV** для установки двух переменных окружения**:**

- PYTHONUNBUFFERED обеспечивает привычный вид вывода нашей консоли и не буферизируется **Docker,** чего мы не хотим**.**
- PYTHONDONTWRITEBYTECODE означает**,** что **Python** не будет пытаться писать **.pyc** файлы**,** чего мы также не хотим

Далее мы используем **WORKDIR,** чтобы установить путь к рабочей директории по умолчанию в нашем образе под названием **code,** где мы будем хранить наш код**.** Если бы мы не сделали этого**,** то каждый раз**,** когда мы хотели бы выполнить команды в нашем контейнере**,** нам пришлось бы вводить длинный путь**.**

Вместо этого **Docker** будет считать**,** что мы хотим выполнять все команды из этой директории**.** Для наших зависимостей мы используем **Pipenv,** поэтому мы скопируем файлы **Pipfile** и **Pipfile.lock** в каталог **/code/** в **Docker.**

Стоит немного объяснить**,** почему **Pipenv** создает **Pipfile.lock.** Концепция блокировки файлов не является уникальной для **Python** или **Pipenv;** на самом деле она уже присутствует в менеджерах пакетов для большинства современных языков программирования**: Gemfile.lock** в **Ruby**, **yarn.lock** в **JavaScript, composer.lock** в **PHP** и так далее**.**

Pipenv был первым популярным проектом, который включил их в упаковку **Python**.

Преимущество файла блокировки в том, что это приводит к детерминированной сборке: сколько бы раз вы ни устанавливали пакеты программ, результат будет один и тот же. Без файла блокировки, который "фиксирует" зависимости и их порядок, это не обязательно так. Это означает, что у двух членов команды, устанавливающих один и тот же список программных пакетов, могут получиться немного разные сборки.

Когда мы работаем с **Docker**, где есть код как локально на нашем компьютере, так и внутри **Docker**, потенциальная возможность конфликтов **Pipfile.lock** возникает при обновлении программных пакетов.

Мы рассмотрим это должным образом в следующей главе. Двигаясь дальше, мы используем команду **RUN**, чтобы сначала установить **Pipenv**, а затем **pipenv install** для установки программных пакетов, перечисленных в нашем **Pipfile.lock**, в настоящее время только **Django**. Важно также добавить флаг **--system**, поскольку по умолчанию **Pipenv** будет искать виртуальную среду для установки любого пакета, но поскольку мы сейчас находимся в **Docker**, технически никакой виртуальной среды нет. В некотором смысле, контейнер **Docker -** это наша виртуальная среда и даже больше. Поэтому мы должны использовать флаг **--system**, чтобы наши пакеты были доступны во всем **Docker** для нас.

В качестве последнего шага мы копируем остальную часть нашего локального кода в каталог **/code/** в **Docker**. Почему мы копируем локальный код дважды, сначала **Pipfile** и **Pipfile.lock**, а затем все остальное? Причина в том, что образы создаются на основе инструкций сверху вниз, поэтому мы хотим, чтобы вещи, которые часто меняются **-** например, наш локальный код **-** были последними. Таким образом, нам придется регенерировать только эту часть образа, когда произойдет изменение, а не переустанавливать все при каждом изменении. А поскольку пакеты программ, содержащиеся в наших **Pipfile** и **Pipfile.lock**, меняются нечасто, имеет смысл скопировать их и установить раньше.

Наши инструкции по созданию образа уже готовы, поэтому давайте соберем образ с помощью команды **docker build .** Точка, **.** указывает на текущий каталог, в котором нужно выполнить команду. Здесь будет много вывода; я включил только первые две строки и последние три.

Command Line

```
$ docker build .
Sending build context to Docker daemon
Step 1/7 : FROM python:3.8
3.8: Pulling from library/python
...
Successfully built 8d85b5d5f5f6
```

Теперь нам нужно создать файл **docker-compose.yml** для управления запуском контейнера**,** который будет создан на основе нашего образа **Dockerfile.**

Command Line

```
$ touch docker-compose.yml
```

Он будет содержать следующий код**.**

docker-compose.yml

```
version: '3.8'
services:
    web:
    build: .
     command: python /code/manage.py runserver 0.0.0.0:8000
     volumes:
            - .:/code
    ports:
            - 8000:8000
```

В верхней строке мы указываем последнюю версию **Docker Compose,** которая на данный момент составляет **3.8.** Пусть вас не смущает тот факт**,** что **Python** также находится на версии **3.8** в данный момент**;** между ними нет никакого совпадения**!** Это просто совпадение**.**

Затем мы указываем**,** какие сервисы **(**или контейнеры**)** мы хотим запустить на нашем хосте **Docker.** Можно запустить несколько сервисов**,** но сейчас у нас только один для **web.** Мы указываем**,** как создать контейнер**,** говоря**: "**Ищите в текущей директории **.** файл **Dockerfile .** Затем внутри контейнера выполним команду для запуска локального сервера**.** Монтирование томов автоматически синхронизирует файловую систему **Docker** с файловой системой нашего локального компьютера**.**

Это означает**,** что нам не нужно перестраивать образ каждый раз**,** когда мы меняем один файл**!** Наконец**,** мы указываем порты**,** которые будут открыты в **Docker,** это будет **8000,** который используется в **Django** по умолчанию**.**

Если вы впервые используете **Docker**, то, скорее всего, вы сейчас в замешательстве. Но не волнуйтесь. Мы создадим множество образов **Docker** и контейнеров в течение этой книги, и с практикой поток начнет обретать смысл. Вы увидите, что мы используем очень похожие файлы **Dockerfile** и **docker-compose.yml** в каждом из наших проектов.

Последний шаг - запуск нашего **Docker-**контейнера с помощью команды **docker-compose up .** Эта команда приведет к еще одному длинному потоку выходного кода в командной строке.

Command Line

```
$ docker-compose up
Creating network "hello_default" with the default driver
Building web
Step 1/7 : FROM python:3.8
...
Creating hello_web_1 ... done
Attaching to hello_web_1
web_1 | Watching for file changes with StatReloader
web_1 | Performing system checks...
web_1 |
web_1 | System check identified no issues (0 silenced).
web_1 | August 03, 2020 - 19:28:08
web_1 | Django version 3.1, using settings 'config.settings'
web_1 | Starting development server at http://0.0.0.0:8000/
web_1 | Quit the server with CONTROL-C.
```

Чтобы убедиться, что это действительно сработало, вернитесь на сайт **http://127.0.0.1:8000/** в своем веб-браузере. Обновите страницу, и страница **"Hello, World"** должна появиться.

Теперь **Django** работает исключительно в контейнере **Docker.** Мы не работаем в виртуальной среде локально. Мы не выполняли команду **runserver.** Весь наш код теперь существует, и наш сервер **Django** работает внутри автономного контейнера **Docker.** Успешно**!**

Остановите контейнер с помощью **Control+c (**нажмите одновременно кнопки **"Control"** и **"c")** и дополнительно введите **docker-compose down .** Контейнеры **Docker** занимают много памяти, поэтому хорошей идеей будет остановить их таким образом, когда вы закончите их использовать. Контейнеры должны быть без статических данных, поэтому мы используем тома, чтобы копировать наш код локально, где он может быть сохранен.

Command Line

```
$ docker-compose down
Removing hello_web_1 ... done
Removing network hello_default
```

Git

В наши дни **Git** является наиболее популярной системой контроля версий**,** и мы будем использовать ее в этой книге**.**
Сначала добавьте новый файл **Git** с помощью **git init ,** затем проверьте статус изменений**,** добавьте обновления и включите сообщение о фиксации**.**

Command Line

```
$ git init
$ git status
$ git add .
$ git commit -m 'test1'
```

Вы можете сравнить свой код для этой главы с официальным репозиторием**,** доступным на **Github.**

Заключение

Docker - это самодостаточная среда**,** которая включает в себя всё**,** что нам нужно для локальной разработки**:** веб**-**сервисы**,** базы данных и многое другое**,** если мы захотим**.** Общая схема всегда будет одинаковой при использовании его с **Django:**

- создать виртуальную среду локально и установить **Django**
- создать новый проект
- выйти из виртуальной среды
- написать **Dockerfile** и затем собрать начальный образ
- написать файл **docker-compose.yml** и запустить контейнер с помощью **docker-compose up**

Мы создадим еще несколько проектов **Django** с помощью **Docker,** так что этот поток будет иметь больше смысла**,** но на этом все**.** В следующей главе мы создадим новый проект **Django** с помощью **Docker** и добавим **PostgreSQL** в отдельный контейнер в качестве нашей базы данных**.**

Глава 2: PostgreSQL

Одно из самых непосредственных различий между работой над "игрушечным приложением" на **Django** и готовым к производству **-** это база данных**. Django** поставляется с **SQLite** в качестве выбора по умолчанию для локальной разработки**,** потому что она маленькая**,** быстрая и файловая**,** что делает ее простой в использовании**.** Никакой дополнительной установки или настройки не требуется**.**

Однако за это удобство приходится платить**.** Вообще говоря**, SQLite -** не лучший выбор базы данных для профессиональных веб**-**сайтов**.** Поэтому**,** хотя **SQLite** можно использовать локально при создании прототипа идеи**,** редко кто использует **SQLite** в качестве базы данных в рабочем проекте**.**

Django поставляется со встроенной поддержкой четырех баз данных**: SQLite, PostgreSQL, MySQL** и **Oracle.** В этой книге мы будем использовать **PostgreSQL,** поскольку это наиболее популярный выбор среди разработчиков **Django,** однако**,** красота **ORM Django** заключается в том**,** что даже если бы мы захотели использовать **MySQL** или **Oracle,** фактический код **Django,** который мы напишем**,** будет практически идентичен**. Django ORM** обрабатывает перевод кода **Python** в базы данных за нас**,** что довольно удивительно**,** если подумать**.**

Сложность использования этих трех баз данных заключается в том**,** что каждая из них должна быть установлена и запущена локально**,** если вы хотите точно имитировать производственную среду на вашем локальном компьютере**.** А мы этого хотим**!** Хотя **Django** обрабатывает детали переключения между базами данных за нас**,** неизбежно возникают небольшие**,** трудноуловимые ошибки**,** которые могут проявиться**,** если вы используете **SQLite** для локальной разработки**,** но другую базу данных в производстве**.** Поэтому лучшей практикой является использование одной и той же базы данных локально и в производстве**.**

В этой главе мы начнем новый проект **Django** с базы данных **SQLite,** а затем переключимся на **Docker** и **PostgreSQL.**

Начало работы

В командной строке убедитесь**,** что вы вернулись к папке **code** на нашем рабочем столе**.** Вы можете сделать это двумя способами**.** Либо введите **cd ...,** чтобы переместиться на уровень "вверх"**,** так что если вы сейчас находитесь в **Desktop/code/hello,** то переместитесь в **Desktop/code .** Или вы можете просто набрать **cd ~ /Desktop/code/,** что приведет вас прямо в нужный директорий**.** Затем создайте новый директорий с именем **postgresql** для кода этой главы**.**

Command Line

```
$ cd ..
$ mkdir postgresql && cd postgresql
```

Теперь установите **Django,** запустите командную оболочку и создайте базовый проект **Django** под названием **config.** Не забудьте про точку **.** в конце команды**!**

Command Line

```
$ pipenv install django~=3.1.0
$ pipenv shell
(postgresql) $ django-admin startproject config .
```

Пока все хорошо**.** Теперь мы можем перенести нашу базу данных для ее инициализации и использовать **runserver** для запуска локального сервера**.**

> Обычно я не рекомендую запускать migrate в новых проектах до тех пор, пока не будет настроена пользовательская модель пользователя. В противном случае Django привяжет базу данных к встроенной модели User, которую трудно модифицировать в дальнейшем в проекте. Мы рассмотрим этот вопрос в главе 3, но поскольку эта глава предназначена в основном для демонстрационных целей, использование модели User по умолчанию здесь является одноразовым исключением.

Command Line

```
(postgresql) $ python manage.py migrate
(postgresql) $ python manage.py runserver
```

Подтвердите**,** что все сработало**,** перейдя на сайт **http://127.0.0.1:8000/** в своем веб**-**браузере**.** Вам возможно**,** вам придется обновить страницу**,** но вы должны увидеть знакомую страницу приветствия **Django.**

Остановите локальный сервер с помощью **Control+c,** а затем используйте команду **ls,** чтобы перечислить все файлы и директории**.**

Command Line

```
(postresql) $ ls
Pipfile  Pipfile.lock  config  db.sqlite3  manage.py
```

Docker

Для перехода на **Docker** сначала выйдите из нашей виртуальной среды**,** а затем создайте файлы **Dockerfile** и **docker-compose.yml,** которые будут управлять нашим образом и контейнером **Docker** соответственно**.**

Command Line

```
(postgresql) $ exit
$ touch Dockerfile
$ touch docker-compose.yml
```

Dockerfile такой же**,** как и в главе **1.**

Dockerfile

```
# Pull base image
FROM python:3.8
# Set environment variables
ENV PYTHONDONTWRITEBYTECODE 1
ENV PYTHONUNBUFFERED 1
# Set work directory
WORKDIR /code
# Install dependencies
COPY Pipfile Pipfile.lock /code/
RUN pip install pipenv && pipenv install --system
# Copy project
COPY . /code/
```

Теперь выполните сборку начального образа с помощью команды **docker build .**

Command Line

```
$ docker build .
```

Вы заметили**,** что в этот раз **Dockerfile** создал образ гораздо быстрее**?** Это потому**,** что **Docker** сначала ищет локально на вашем компьютере определенный образ**.** Если он не находит образ локально**,** то загружает его**.** А поскольку многие из этих образов уже были на компьютере из предыдущей главы**, Docker** не нужно было загружать их снова**!**

Теперь пришло время для файла **docker-compose.yml,** который также соответствует тому**,** что мы видели ранее в главе **1.**

docker-compose.yml

```
version: '3.8'
services:
web:
    build: .
    command: python /code/manage.py runserver 0.0.0.0:8000
    volumes:
        - .:/code
    ports:
        - 8000:8000
```

Режим отсоединения

Теперь мы запустим наш контейнер**,** но на этот раз в отсоединенном режиме**,** для чего потребуется флаг **-d** или **-detach (**они делают одно и то же**).**

Command Line

```
$ docker-compose up -d
```

Отдельный режим запускает контейнеры в фоновом режиме, что означает, что мы можем использовать одну вкладку командной строки без необходимости открывать еще одну. Это избавляет нас от необходимости постоянно переключаться между двумя вкладками командной строки. Недостатком является то, что если/когда произойдет ошибка, вывод не всегда будет виден. Поэтому, если в какой-то момент ваш экран не соответствует этой книге, попробуйте набрать **docker-compose logs**, чтобы увидеть текущий вывод и отладить любые проблемы.

Скорее всего, вы увидите сообщение **"Warning: Image for service web was built because it did not already exist"** в нижней части команды. **Docker** автоматически создал для нас новый образ внутри контейнера. Как мы увидим далее в книге, добавление флага **--build** для принудительной сборки образа необходимо при обновлении пакетов программного обеспечения, поскольку по умолчанию **Docker** будет искать локальную кэшированную копию программного обеспечения и использовать ее, что повышает производительность.

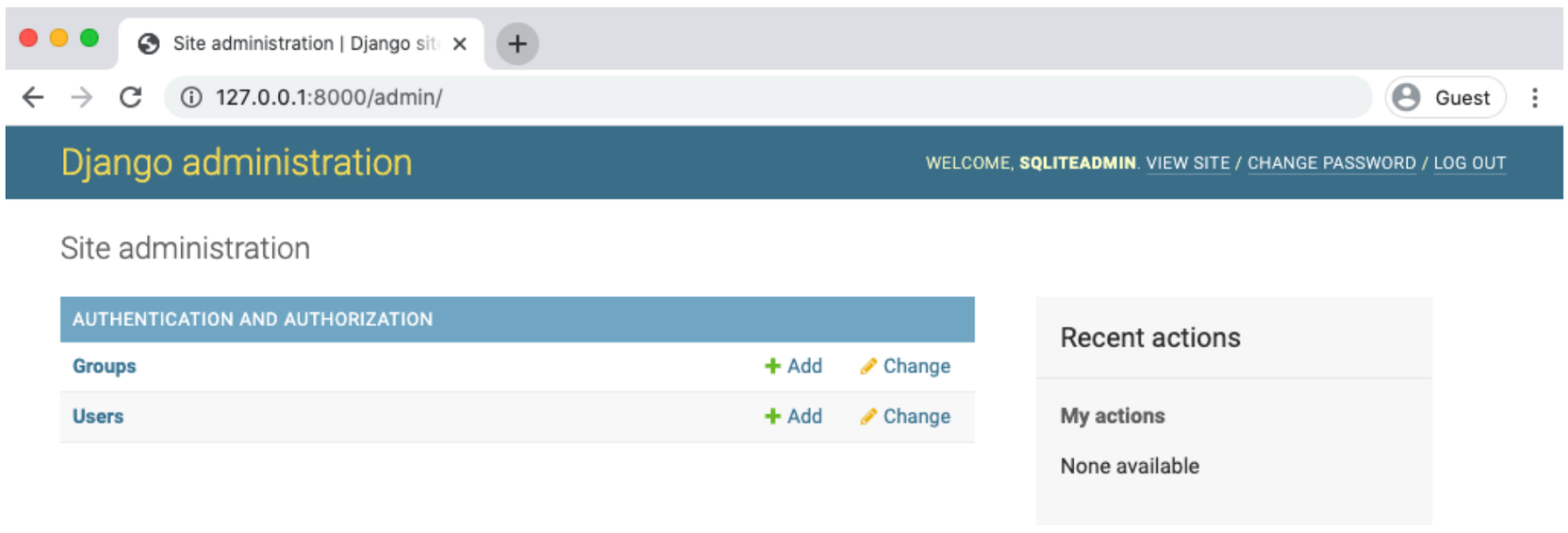
Чтобы убедиться, что все работает правильно, вернитесь на сайт **http://127.0.0.1:8000/** в своем веб-браузере. Обновите страницу, чтобы снова увидеть страницу приветствия **Django**.

Поскольку теперь мы работаем в **Docker**, а не локально, мы должны предварять традиционные команды командой **docker-compose exec [service],** где мы указываем имя сервиса. Например, для создания учетной записи суперпользователя вместо ввода **python manage.py createsuperuser** обновленная команда будет выглядеть следующим образом, используя веб-сервис.

Command Line

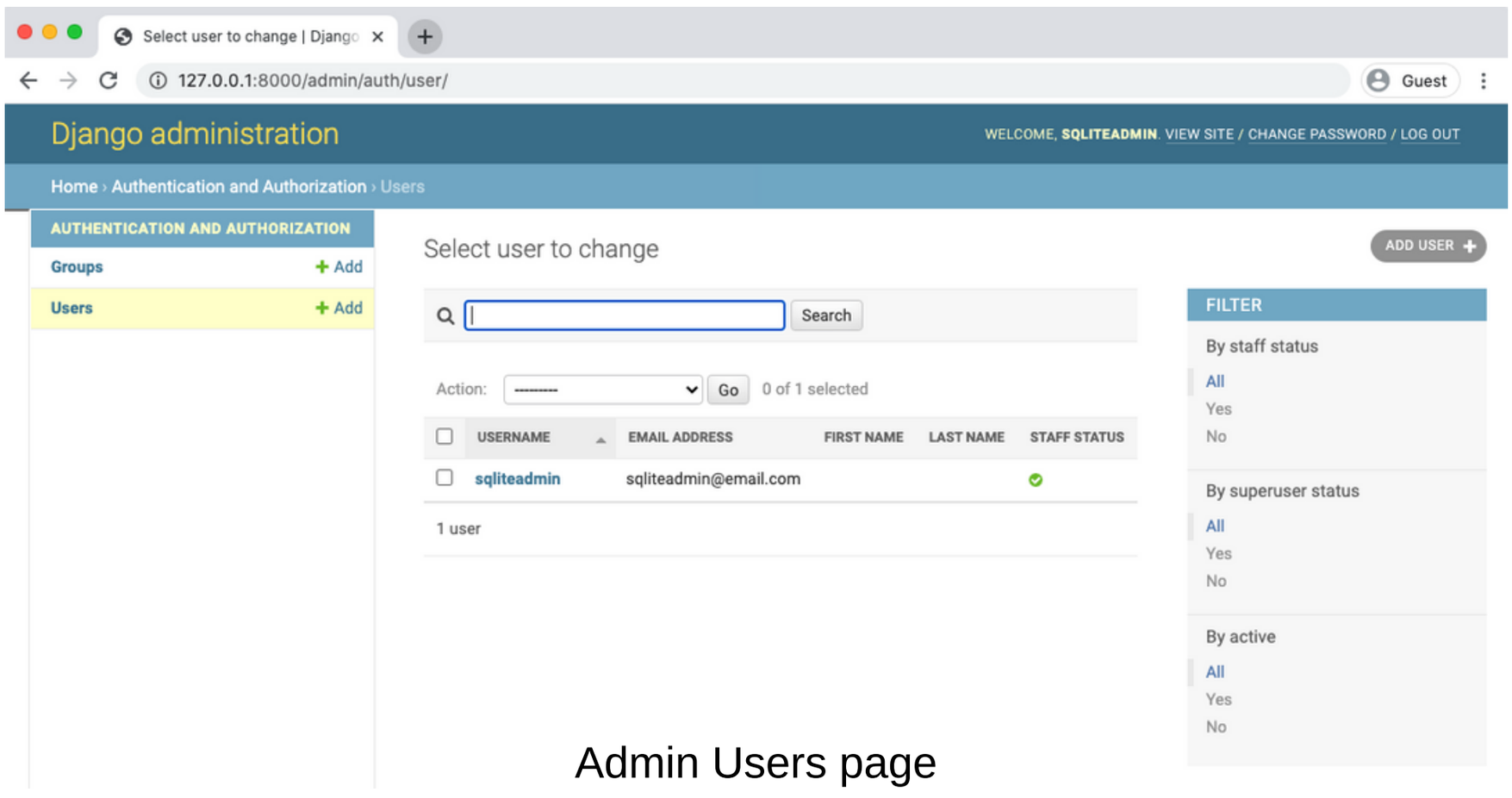
```
$ docker-compose exec web python manage.py createsuperuser
```

Для имени пользователя выберите **sqliteadmin , sqliteadmin@email.com** в качестве адреса электронной почты и выберите пароль по своему усмотрению.
Я часто использую **testpass123 .**
Затем перейдите непосредственно в админку по адресу **http://127.0.0.1:8000/admin** и войдите в систему. Вы будете перенаправлены на главную страницу администратора. Обратите внимание на имя пользователя **sqliteadmin** в правом верхнем углу.



Django sqliteadmin

Если вы нажмете на кнопку Пользователи, то попадете на страницу Пользователи, где мы сможем подтвердить, что был создан только один пользователь.



Admin Users page

Важно отметить еще один аспект **Docker:** до сих пор мы обновляли нашу базу данных**,** которая в настоящее время представлена файлом **db.sqlite3**, в **Docker.** Это означает**,** что фактический файл **db.sqlite3** каждый раз меняется**.** И благодаря монтированию томов в конфигурации **docker-compose.yml** каждое изменение файла копируется в файл **db.sqlite3** на нашем локальном компьютере**.** Вы можете выйти из **Docker**, запустить оболочку**,** запустить сервер с помощью **python manage.py runserver** и увидеть точно такой же логин администратора в этот момент**,** потому что база данных **SQLite** одна и та же**.**

PostgreSQL

Теперь пришло время перейти на **PostgreSQL** для нашего проекта**,** что требует трех дополнительных шагов**:**

- установить адаптер базы данных**, psycopg2 ,** чтобы **Python** мог общаться с **PostgreSQL**
- обновить конфигурацию **DATABASE** в нашем файле **settings.py**
- установить и запустить **PostgreSQL** локально

Готовы**?** Начинаем**.** Остановите запущенный контейнер **Docker** с помощью **docker-compose down .**

Command Line

```
$ docker-compose down
Stopping postgresql_d_1 ... done
Removing postgresql_web_1 ... done
Removing network postgresql_default
```

Затем в нашем файле **docker-compose.yml** добавьте новый сервис под названием **db .** Это означает, что на нашем хосте **Docker** будут работать два отдельных сервиса, каждый из которых является контейнером: **web** для локального сервера **Django** и **db** для нашей базы данных **PostgreSQL.**

Версия **PostgreSQL** будет привязана к последней версии, **11.** Если бы мы не указали номер версии, а использовали просто **postgres** , то была бы загружена последняя версия **PostgreSQL**, даже если в будущем эта версия будет **12, 13** или другой номер. Всегда полезно привязываться к конкретному номеру версии, как для баз данных, так и для пакетов.

Вторая часть **-** добавление переменной окружения **POSTGRES_HOST_AUTH_METHOD=trust ,** которая позволяет нам подключаться без пароля. Это удобно для локальной разработки.

Наконец, мы добавляем строку **depends_on** в наш веб-сервис, поскольку он буквально зависит от базы данных. Это означает, что **db** будет запущен до **web.**

---

docker-compose.yml

```
version: '3.8'
services:
web:
      build: .
      command: python /code/manage.py runserver 0.0.0.0:8000
      volumes:
            - .:/code
      ports:
          - 8000:8000
      depends_on:
          - db
      db:
        image: postgres:11
        environment:
            - "POSTGRES_HOST_AUTH_METHOD=trust"
```

Теперь запустите **docker-compose up -d,** который перестроит наш образ и запустит два контейнера**,** в одном из которых будет работать **PostgreSQL** внутри **db,** а в другом **-** наш веб**-**сервер **Django.**

Command Line

```
$ docker-compose up -d
```

Важно отметить**,** что такая производственная база данных**,** как **PostgreSQL**, не основана на файлах**.** Она полностью работает в службе **db** и является вечной**;** когда мы выполним **docker-compose down,** все данные в ней будут потеряны**.** В отличие от нашего кода в веб**-**контейнере**,** который имеет монтирование томов для синхронизации локального и **Docker-**кода**.**

В следующей главе мы узнаем**,** как добавить монтирование томов для нашей службы **db,** чтобы сохранить информацию о базе данных**.**

Настройки

С помощью текстового редактора откройте файл **config/settings.py** и прокрутите вниз до конфигурации **DATABASES.**
Текущая настройка следующая**:**

Code

```
# config/settings.py
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.sqlite3',
        'NAME': BASE_DIR / 'db.sqlite3',
      }
}
```

По умолчанию **Django** определяет **sqlite3** в качестве движка базы данных**,** дает ему имя **db.sqlite3 ,** и помещает его в **BASE_DIR,** что означает в директории уровня нашего проекта**.**

Поскольку структура директорий часто вызывает путаницу**, " project-level"** означает верхний директорий нашего проекта**,** который содержит **config , manage.py , Pipfile , Pipfile.lock ,** и файл **db.slite3.**

Command Line

```
(postgresql) $ ls
Dockerfile   Pipfile.lock   docker-compose.yml config
Pipfile      db.sqlite3     manage.py
```

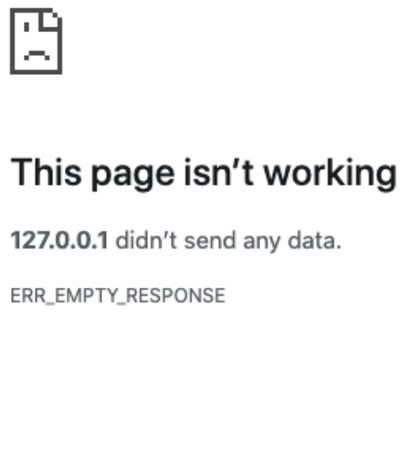
Для перехода на **PostgreSQL** мы обновим конфигурацию **ENGINE. PostgreSQL** требует **NAME , USER , PASSWORD , HOST ,** и **PORT .**

Для удобства мы зададим первые три параметра как **postgres, HOST - db,** что является именем нашего сервиса**,** заданным в **docker-compose.yml**, а **PORT - 5432,** который является портом **PostgreSQL** по умолчанию**.**

Code

```
# config/settings.py
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.postgresql',
        'NAME': 'postgres',
        'USER': 'postgres',
        'PASSWORD': 'postgres',
        'HOST': 'db',
        'PORT': 5432
    }
}
```

Вы увидите ошибку**,** если обновите веб**-**страницу по адресу **http://127.0.0.1:8000/**



Django error

Что происходит**?** Поскольку мы запускаем **Docker** в отсоединенном режиме с флагом **-d,** это не сразу понятно**.** Пора проверить наши журналы**.**

Command Line

```
$ docker-compose logs
...
web_1 | django.core.exceptions.ImproperlyConfigured: Error loading psycopg2 module: No module named 'psycopg2'
```

Там будет много вывода**,** но в нижней части раздела **web_1** вы увидите приведенные выше строки**,** которые говорят о том**,** что мы еще не установили драйвер **psycopg2.**

Psycopg

PostgreSQL это база данных, которая может использоваться практически любым языком программирования. Но если вы подумаете об этом, то языки программирования так или иначе различаются и каким образом они подключается к самой базе данных?

Ответ - через адаптер базы данных! И именно таким адаптером является **Psycopg** - самый популярный адаптер баз данных для **Python.** Если вы хотите узнать больше о том, как работает **Psycopg**, вот ссылка на более полное описание на официальном сайте.

Мы можем установить **Pyscopg** с помощью **Pipenv.** В командной строке введите следующую команду, чтобы он был установлен на нашем хосте **Docker.**

Command Line

```
$ docker-compose exec web pipenv install psycopg2-binary==2.8.3
```

Надеюсь, вы спрашиваете, зачем устанавливать в **Docker**, а не локально? Короткий ответ заключается в том, что последовательная установка новых программных пакетов в **Docker** и последующая пересборка образа с нуля избавит нас от потенциальных конфликтов **Pipfile.lock.**

Генерация **Pipfile.lock** в значительной степени зависит от используемой ОС. Мы указали всю нашу ОС в **Docker**, включая использование **Python 3.8.** Но если вы установите **psycopg2** локально на своем компьютере, где используется другое окружение, полученный файл **Pipfile.lock** также будет отличаться. Но тогда монтирование томов в нашем файле **docker-compose.yml**, которое автоматически синхронизирует локальную и **Docker** файловые системы, приведет к тому, что локальный **Pipfile.lock** перезапишет версию в **Docker.** Таким образом, теперь наш контейнер **Docker** пытается запустить некорректный файл **Pipfile.lock.** Черт!

Один из способов избежать этих проблем - последовательно устанавливать новые пакеты программного обеспечения в **Docker**, а не локально.

Если вы сейчас обновите веб-страницу, вы все равно увидите ошибку. Хорошо, давайте проверим журналы.

Command Line

```
$ docker-compose logs
```

Это то же самое, что и раньше! Почему это происходит? **Docker** автоматически кэширует образы, если ничего не меняется по соображениям производительности. Мы хотим, чтобы он автоматически перестроил образ с нашими новыми **Pipfile** и **Pipfile.lock**, но поскольку последняя строка нашего **Dockerfile - COPY . /code/** будут скопированы только файлы; основной образ не будет перестраиваться, пока мы не заставим его это сделать. Это можно сделать, добавив флаг **--build**.

Итак, обзор: при добавлении нового пакета сначала установите его в **Docker**, остановите контейнеры, принудительно перестройте образ, а затем снова запустите контейнеры. Мы будем использовать этот поток неоднократно на протяжении всей книги.

Command Line

```
$ docker-compose down
$ docker-compose up -d --build
```

Если вы снова обновите домашнюю страницу, то страница приветствия **Django** по адресу **http://127.0.0.1:8000/** теперь работает! Это потому, что **Django** успешно подключился к **PostgreSQL** через **Docker**.

Отлично, все работает.

Новая база данных

Однако, поскольку мы теперь используем **PostgreSQL**, а не **SQLite**, наша база данных пуста. Если вы снова посмотрите текущие журналы, набрав **docker-compose logs**, вы увидите сообщения типа **"You have 18 unapplied migrations(s)".**

Чтобы подкрепить эту мысль, зайдите в админку по адресу **http://127.0.0.1:8000/admin/** и войдите в систему. Будет ли работать наша предыдущая учетная запись суперпользователя **sqliteadmin** и **testpass123?**

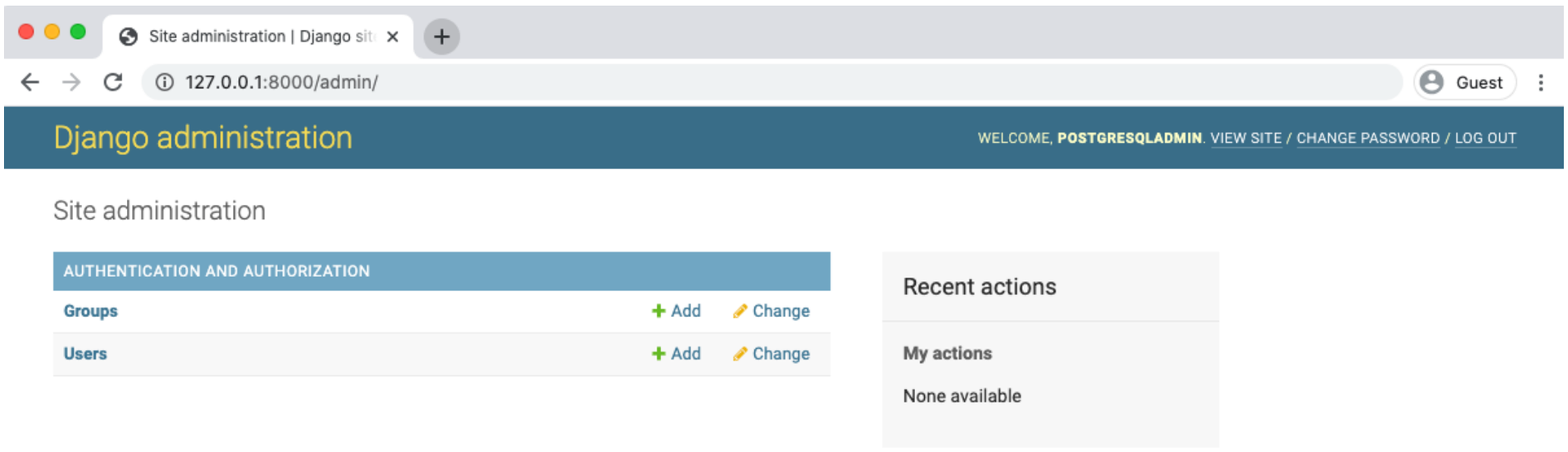
Нет! Мы видим **ProgrammingError** по адресу **/admin .** Чтобы исправить эту ситуацию, мы можем как мигрировать, так и создать суперпользователя в **Docker**, который будет иметь доступ к базе данных **PostgreSQL.**

Command Line

```
$ docker-compose exec web python manage.py migrate
$ docker-compose exec web python manage.py createsuperuser
```

Как мы назовем нашего суперпользователя? Давайте воспользуемся **postgresqladmin** и для тестирования зададим **email postgresqladmin@email.com** и пароль **testpass123 .**

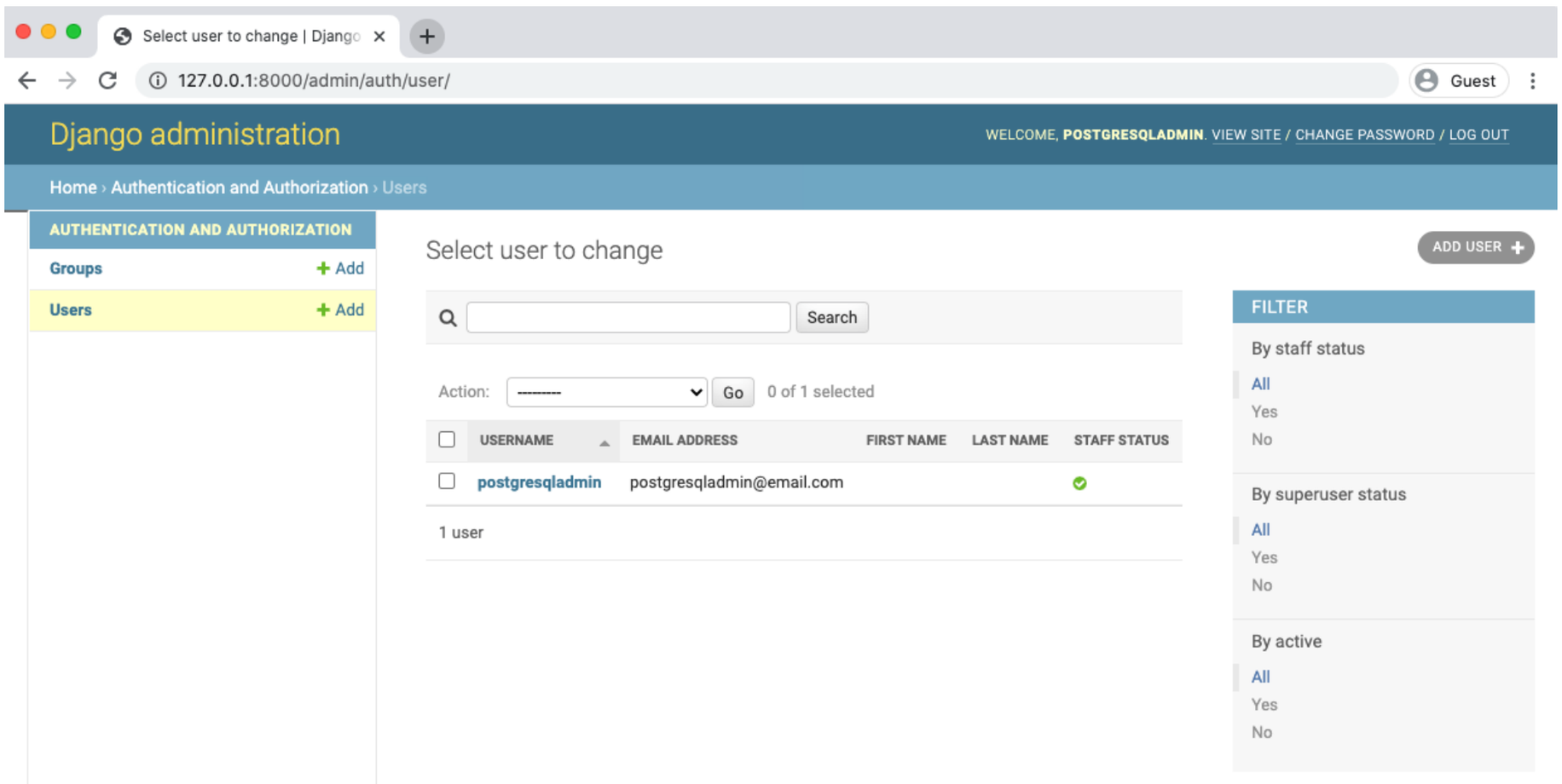
В веб-браузере перейдите на страницу администратора по адресу **http://127.0.0.1:8000/admin/** и введите данные для входа нового суперпользователя.



Административная панель с postgresadmin

В правом верхнем углу показано, что мы вошли в систему с правами **postgresadmin**, а не **sqliteadmin.**

Кроме того, вы можете нажать на вкладку **Users** на главной странице и посетить раздел **Users**, чтобы увидеть, что наш единственный пользователь **-** это новая учетная запись суперпользователя.



Административная панель с пользователями

Не забудьте остановить наш запущенный контейнер с помощью **docker-compose down.**

Command Line

```
$ docker-compose down
```

Git

Давайте снова сохраним наши изменения, инициализировав **Git** для этого нового проекта, добавив наши изменения и включив сообщение о фиксации

Command Line

```
$ git init
$ git status
$ git add .
$ git commit -m 'ch2'
```

Официальный исходный код главы **2** доступен на **Github**.

Заключение

Целью этой главы было продемонстрировать, как **Docker** и **PostgreSQL** работают вместе в проекте **Django**. Переход от базы данных **SQLite** к **PostgreSQL** для многих разработчиков поначалу является ментальным скачком.

Ключевым моментом является то, что с **Docker** нам больше не нужно находиться в локальной виртуальной среде. **Docker -** это наша виртуальная среда... и наша база данных, и многое другое по желанию. Хост **Docker** по сути заменяет нашу локальную операционную систему, и в нем мы можем запускать несколько контейнеров, например, для нашего веб-приложения и для нашей базы данных, которые могут быть изолированы и запускаться отдельно.

В следующей главе мы запустим наш проект книжного интернет-магазина. Давайте начнем!

Глава 3: Проект книжного магазина

Пришло время создать главный проект этой книги - интернет-магазин книг. В этой главе мы начнем новый проект, переключимся на **Docker**, добавим пользовательскую модель и проведем первые тесты.

Давайте начнем с создания нового проекта **Django** с **Pipenv** локально, а затем переключимся на **Docker**. Скорее всего, вы сейчас находитесь в директории **postgresql** из главы **2**, поэтому в командной строке наберите **cd ..**, что вернет вас в нужную директорию кода на рабочем столе (если вы на **Mac**). Мы создадим каталог **books** для нашего кода, а затем установим **django .** Мы также знаем, что будем использовать **PostgreSQL**, поэтому мы можем установить адаптер **psycopg2**. Только после того, как мы создали наш начальный образ, мы можем приступить к установке будущих пакетов программного обеспечения в самом **Docker**. Наконец, используйте команду **shell** для входа в новую виртуальную среду.

Command Line

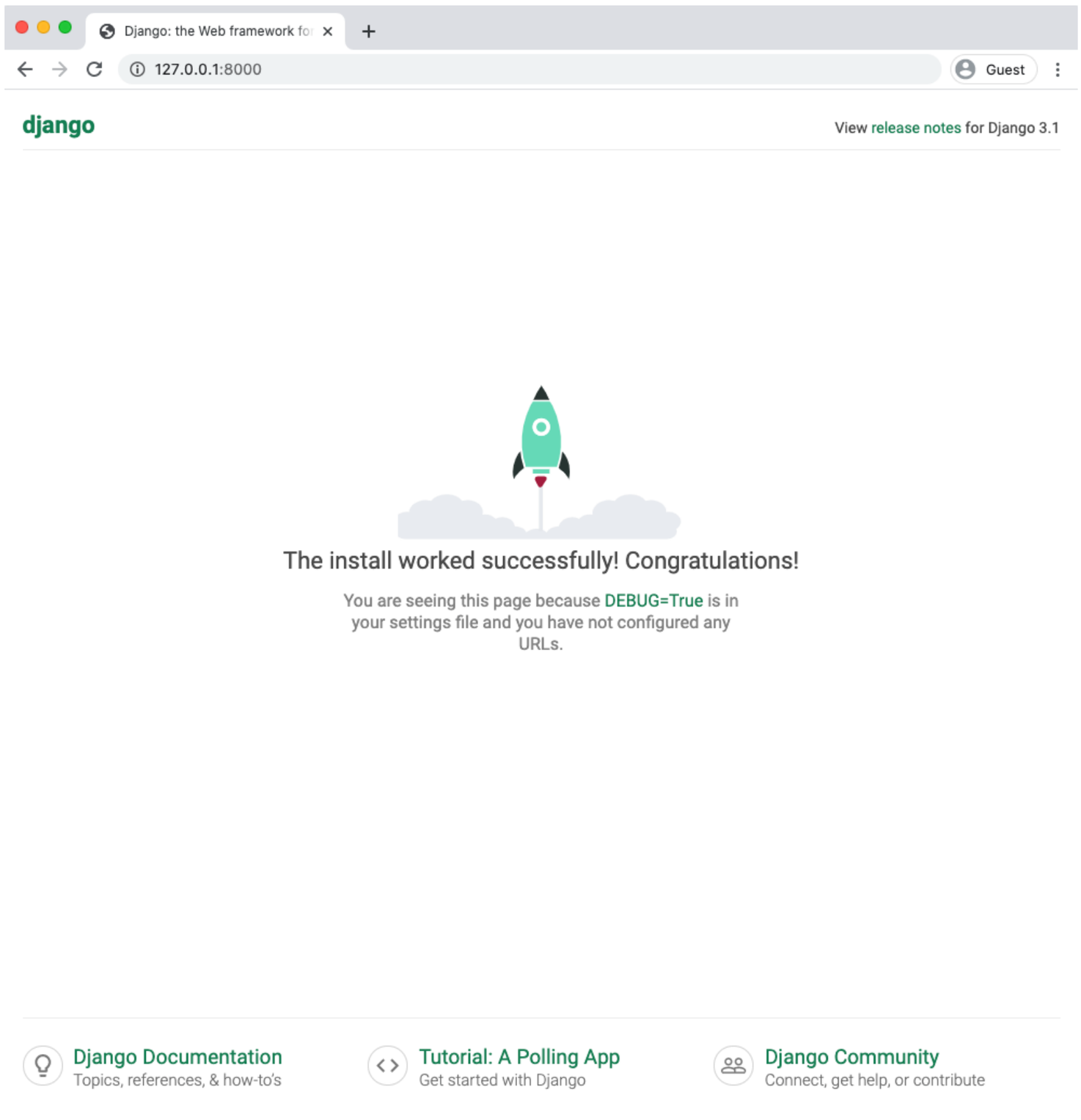
```
$ cd ..
$ mkdir books && cd books
$ pipenv install django~=3.1.0 psycopg2-binary==2.8.5
$ pipenv shell
```

Мы назовем наш новый проект **Django config .** Убедитесь, что вы не забыли точку, **.** в конце команды, иначе **Django** создаст дополнительный каталог, который нам не нужен. Затем с помощью **runserver** запустим локальный веб-сервер **Django** и убедимся, что все работает правильно.

Command Line

```
(books) $ django-admin startproject config .
(books) $ python manage.py runserver
```

В вашем веб-браузере перейдите по адресу **http://127.0.0.1:8000/** и вы должны увидеть приветственную страницу **Django**.



Приветственная страница Django

В командной строке вы, скорее всего, увидите предупреждение о **"18 непримененных миграциях".** На данный момент его можно проигнорировать, поскольку мы собираемся перейти на **Docker** и **PostgreSQL.**

Docker

Теперь мы можем переключиться на **Docker** в нашем проекте. Остановите локальный сервер **Control+c**, а также выйдите из оболочки виртуальной среды.

Command Line

```
(books) $ exit
$
```

Docker уже должен быть установлен, а десктопное приложение из предыдущей главы запущено. Как обычно, нам нужно создать **Dockerfile** и файл **docker-compose.yml.**

Command Line

```
$ touch Dockerfile
$ touch docker-compose.yml
```

Dockerfile будет таким же, как и раньше.

Dockerfile

Получить базовый образ

FROM python:3.8

Установите переменные среды

ENV PYTHONDONTWRITEBYTECODE 1

ENV PYTHONUNBUFFERED 1

Установите рабочую директорию

WORKDIR /code

Установите зависимости

COPY Pipfile Pipfile.lock /code/

RUN pip install pipenv && pipenv install --system

Копирование проекта

COPY . /code/

Контейнеры **Docker** по своей природе непостоянны**.** Они существуют только во время работы**,** и все данные в них удаляются**,** когда контейнер останавливается**.** Мы обходим эту проблему**,** используя тома для постоянных данных**.** В веб**-**сервисе у нас уже есть том**,** который связывает наш локальный код с запущенным контейнером и наоборот**.** Но у нас нет выделенного тома для нашей базы данных **PostgreSQL,** поэтому любая информация в ней будет потеряна**,** когда контейнер перестанет работать**.** Решение состоит в том**,** чтобы добавить том и для базы данных**.** Для этого мы указываем местоположение томов в службе **db,** а также тома**,** которые живут вне контейнеров**.**

Это**,** вероятно**,** довольно запутанно**,** и полное объяснение выходит за рамки этой книги**,** поскольку она посвящена **Django,** а не **Docker.** Однако суть в том**,** что контейнеры **Docker** не хранят постоянных данных**,** поэтому все**,** что мы хотим сохранить**,** например**,** исходный код или информацию базы данных**,** должен иметь выделенный том**,** иначе он будет потерян каждый раз**,** когда контейнер будет остановлен**.** Если вам интересно**,** вы можете прочитать документацию **Docker** по томам для более технического объяснения того**,** как все это работает**.**

В любом случае**,** вот обновленный код для нашего файла **docker-compose.yml,** который теперь поддерживает том базы данных**.**

docker-compose.yml

```
version: '3.8'
services:
    web:
        build: .
        command: python /code/manage.py runserver 0.0.0.0:8000
        volumes:
            - .:/code
        ports:
            - 8000:8000
        depends_on:
            - db
db:
    image: postgres:11
    volumes:
      - postgres_data:/var/lib/postgresql/data/
    environment:
      - "POSTGRES_HOST_AUTH_METHOD=trust"
volumes:
   postgres_data:
```

Мы можем собрать наш образ и запустить контейнеры с помощью одной команды.

Command Line

```
$ docker-compose up -d --build
```

Если вы видите здесь ошибку типа **Bindfor 0.0.0.0:8000 failed: port is already allocated**, значит, вы не полностью остановили контейнер **Docker** из главы **2.** Попробуйте запустить **docker-compose** в директории, где вы его ранее запускали, вероятно, **postgresql .** Затем снова попытайтесь собрать и запустить наш новый образ и контейнер. Если этот способ все еще не сработал, вы можете полностью выйти из десктопного приложения **Docker**, а затем открыть его снова.

Перейдите в веб-браузер по адресу **http://127.0.0.1:8000/** и нажмите обновить. Это должна быть та же самая приветственная страница **Django**, но теперь запущенная внутри **Docker.**

PostgreSQL

Несмотря на то, что мы уже установили **psycopg** и **PostgreSQL** доступен в нашем файле **docker-compose.yml**, мы все еще должны указать **Django** переключиться на него вместо базы данных **SQLite** по умолчанию. Сделайте это сейчас. Код такой же, как и в предыдущей главе.

Code

```
# config/settings.py
DATABASES = {
    'default': {
            'ENGINE': 'django.db.backends.postgresql',
             'NAME': 'postgres',
             'USER': 'postgres',
             'PASSWORD': 'postgres',
             'HOST': 'db',
             'PORT': 5432
    }
}
```

Обновите веб-браузер для домашней страницы, чтобы убедиться, что все по-прежнему работает правильно.

Собственная модель пользователя

Пора внедрить собственную модель, которую официальная документация **Django** "настоятельно рекомендует". Почему? Потому что вам придется вносить изменения во встроенную модель пользователя в какой-то момент жизни вашего проекта.

Если вы не начали работу с собственной моделью пользователя с самой первой команды **migrate**, то вас ждут неприятности, потому что **User** тесно переплетается с остальными компонентами **Django**. Очень сложно переключиться на собственную модель пользователя в середине проекта.

Многих смущает тот факт, что собственные модели пользователей были добавлены только в **Django 1.5**. До этого момента рекомендованным подходом было добавление поля **OneToOneField**, часто называемого моделью профиля, к **User** . Вы можете часто видеть такую настройку в старых проектах.

Но в наши дни использование собственные модели пользователя является более распространенным подходом. Однако, как и во многих других вещах, связанных с **Django**-технологиями, есть выбор реализации: либо расширить **AbstractUser**, который сохраняет поля и разрешения пользователя по умолчанию, либо расширить **AbstractBaseUser**, который еще более детализирован и гибок, но требует больше работы.

В этой книге мы будем придерживаться более простого **AbstractUser**, поскольку **AbstractBaseUser** можно добавить позже, если потребуется. Существует четыре шага для добавления собственной модели пользователя в наш проект:

1. Создайте модель **CustomUser**
2. Обновите **config/settings.py**
3. Настройте **UserCreationForm** и **UserChangeForm**
4. Добавьте собственную модель пользователя в файл **admin.py**

Первым шагом будет создание модели **CustomUser,** которая будет жить в своем собственном приложении**.** Мне нравится называть это приложение **accounts .** Мы можем сделать это либо локально в оболочке нашего виртуального окружения**,** то есть запустить **pipenv shell** и затем выполнить **python manage.py startapp accounts .** Однако для единообразия мы будем выполнять большинство команд в самом **Docker.**

Command Line

```
$ docker-compose exec web python manage.py startapp accounts
```

Создайте новую модель **CustomUser,** которая расширяет **AbstractUser.** Это означает**,** что мы создаем копию**,** где **CustomUser** теперь наследует все функциональные возможности **AbstractUser,** но мы можем переопределять или добавлять новые функции по мере необходимости**.** Мы пока не делаем никаких изменений**,** поэтому включим в **Python** оператор **pass,** который служит в качестве заполнителя для нашего будущего кода**.**

Code

```
# accounts/models.py
from django.contrib.auth.models import AbstractUser
from django.db import models

class CustomUser(AbstractUser):
        pass
```

Теперь зайдите и обновите наш файл **settings.py** в разделе **INSTALLED_APPS,** чтобы сообщить **Django** о нашем новом приложении для работы с учетными записями**.** Мы также хотим добавить конфигурацию **AUTH_USER_MODEL** в нижней части файла**,** которая заставит наш проект использовать **CustomUser** вместо модели **User** по умолчанию**.**

Code

```
# config/settings.py
INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
# Local
    'accounts', # new
]
...
AUTH_USER_MODEL = 'accounts.CustomUser' # new
```

Пора создать файл миграций для изменений. Мы добавим в команду необязательные учетные записи имени приложения, чтобы включить только изменения в этом приложении.

Command Line

```
$ docker-compose exec web python manage.py makemigrations accounts
Migrations for 'accounts':
    accounts/migrations/0001_initial.py
        - Create model CustomUser
```

Затем запустите **migrate** для инициализации базы данных в первый раз.

Command Line

```
$ docker-compose exec web python manage.py migrate
```

Собственные пользовательские формы

Модель пользователя может быть как создана, так и отредактирована в админке **Django.** Поэтому нам нужно будет обновить встроенные формы, чтобы они указывали на **CustomUser** вместо **User .**

Создайте файл **accounts/forms.py.**

Command Line

```
$ touch accounts/forms.py
```

В текстовом редакторе введите следующий код, чтобы переключиться на **CustomUser .**

Code

```
# accounts/forms.py
from django.contrib.auth import get_user_model
from django.contrib.auth.forms import UserCreationForm, UserChangeForm

class CustomUserCreationForm(UserCreationForm):
    class Meta:
        model = get_user_model()
        fields = ('email', 'username',)

class CustomUserChangeForm(UserChangeForm):
    class Meta:
        model = get_user_model()
        fields = ('email', 'username',)
```

В самом верху мы импортировали модель **CustomUser** через **get_user_model,** которая обращается к нашему параметру **AUTH_- USER_MODEL** в **settings.py .** Это может показаться более круговым**,** чем прямой импорт **CustomUser** здесь**,** но это поддерживает идею создания одной единственной ссылки на собственную модель пользователя**,** а не прямого обращения к ней по всему нашему проекту**.**

Далее мы импортируем **UserCreationForm** и **UserChangeForm,** которые будут расширены**.**

Затем создадим две новые формы **- CustomUserCreationForm** и **CustomUserChangeForm -** которые расширяют базовые формы пользователей**,** импортированные выше**,** и указывают на замену нашей модели **CustomUser** и отображение полей **email** и **username.** Поле пароля неявно включено по умолчанию и поэтому не нуждается в явном именовании**.**

Пользовательская админка пользователя

Наконец**,** мы должны обновить наш файл **accounts/admin.py.** Администратор **-** это обычное место для манипулирования данными пользователя**,** и существует тесная связь между встроенным **User** и администратором**.** Мы расширим существующий **UserAdmin** до **CustomUserAdmin** и скажем **Django** использовать наши новые формы**,** собственную модель пользователя и перечислять только **email** и **username** пользователя**.** Если мы захотим**,** мы можем добавить в **list_display** больше существующих полей **User,** например**, is_staff .**

Code

```
# accounts/admin.py
from django.contrib import admin
from django.contrib.auth import get_user_model
from django.contrib.auth.admin import UserAdmin
from .forms import CustomUserCreationForm, CustomUserChangeForm

CustomUser = get_user_model()
class CustomUserAdmin(UserAdmin):
        add_form = CustomUserCreationForm
        form = CustomUserChangeForm
        model = CustomUser
        list_display = ['email', 'username',]
admin.site.register(CustomUser, CustomUserAdmin)
```

Фух**.** Немного кода на начальном этапе**,** но это спасает от душевной боли в дальнейшем**.**

Суперпользователь

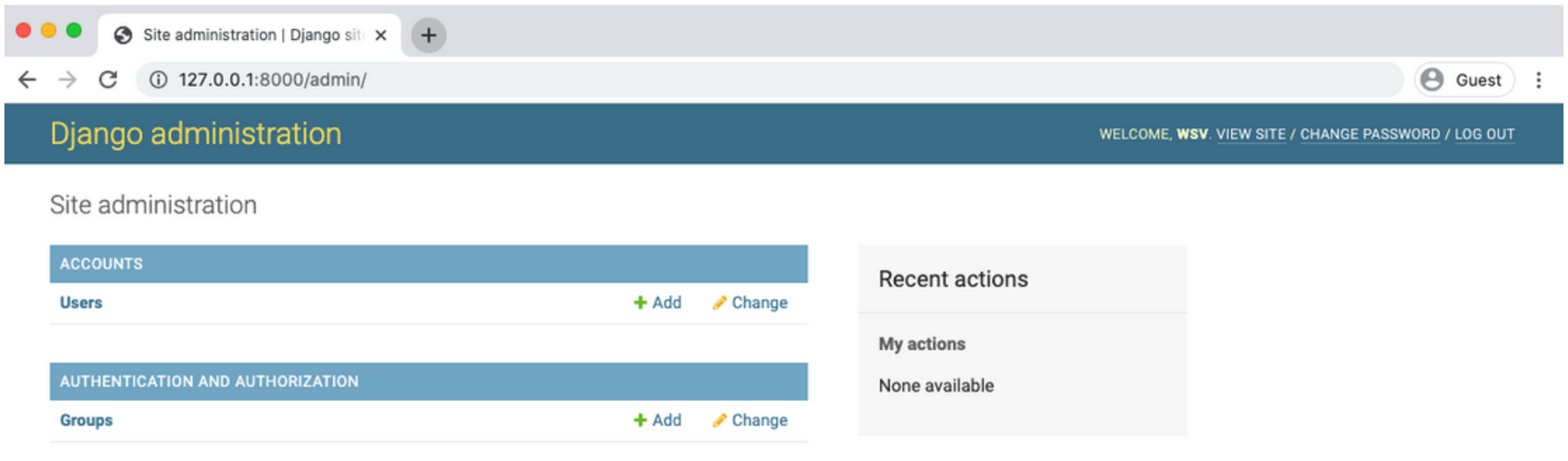
Хороший способ убедиться, что наша собственная модель пользователя работает правильно, - это создать учетную запись суперпользователя, чтобы мы могли войти в админку. Эта команда даст доступ к **CustomUserCreationForm** под капотом.

Command Line

```
$ docker-compose exec web python manage.py createsuperuser
```

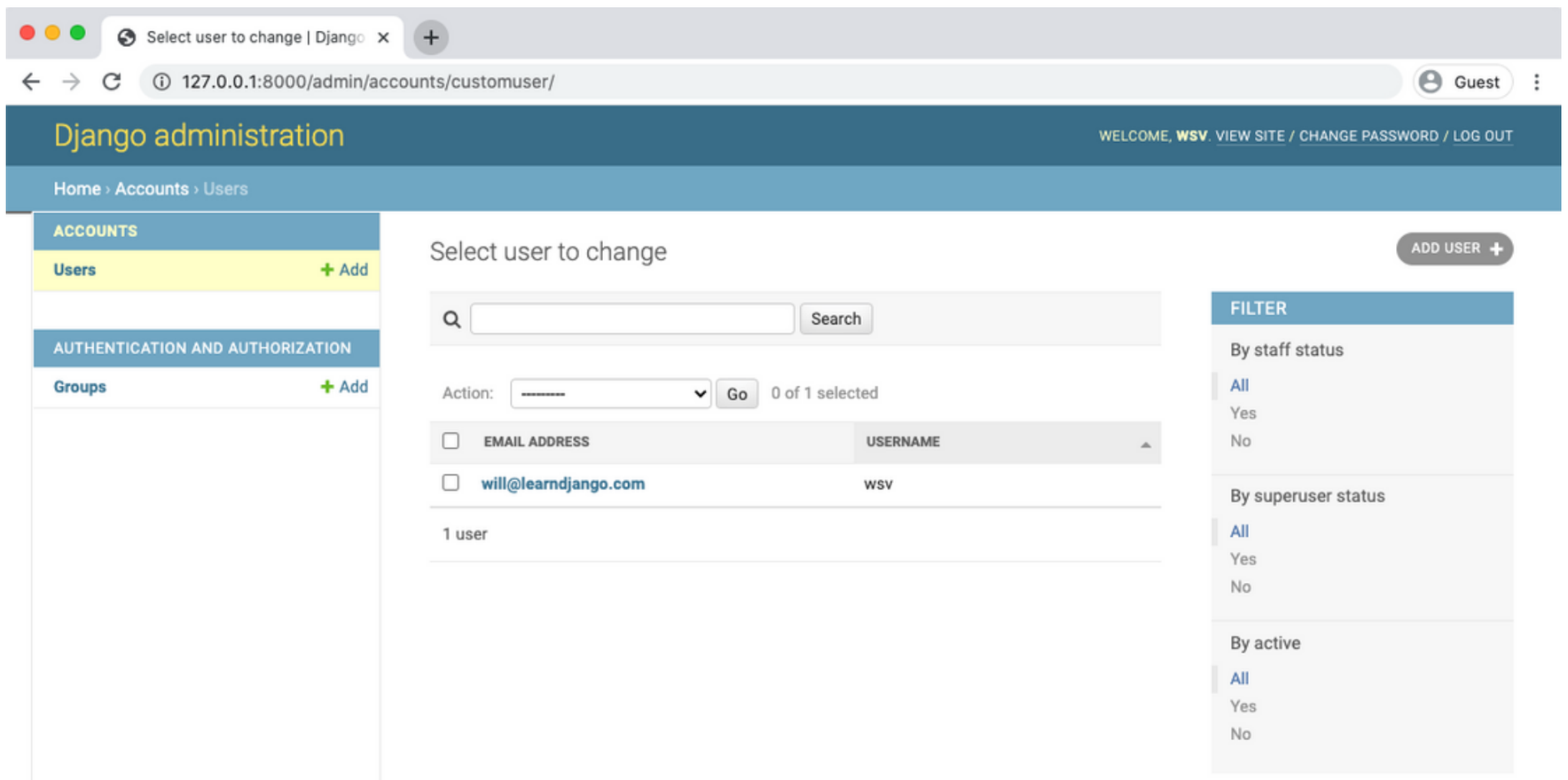
Я использовал имя пользователя **wsv**, электронную почту **will@learndjango.com** и пароль **testpass123**. Вы можете использовать здесь свои собственные предпочтительные варианты.

Теперь зайдите на **http://127.0.0.1:8000/admin** и подтвердите, что вы можете войти в систему. Вы должны увидеть свое имя суперпользователя в правом верхнем углу на странице входа в систему.



Домашняя страница администратора Django

Вы также можете нажать на раздел **"Users"**, чтобы увидеть электронную почту и имя пользователя вашей учетной записи суперпользователя.



Страница пользователя администратора Django

Тесты

Поскольку мы добавили новую функциональность в наш проект, мы должны протестировать ее. Независимо от того, являетесь ли вы разработчиком-одиночкой или работаете в команде, тесты очень важны. По словам сооснователя **Django** Джейкоба Каплан-Мосса, "Код без тестов ломается, как задумано".

Существует два основных типа тестов:

- Юнит-тесты - это небольшие, быстрые и изолированные от конкретной части функциональности тесты.
- Интеграционные тесты - большие, медленные и используются для тестирования всего приложения или пользовательского потока, например, платежей, которые охватывают несколько экранов.

Вы должны написать много модульных тестов и небольшое количество интеграционных тестов.

Язык программирования **Python** содержит свою собственную структуру модульного тестирования, а система автоматизированного тестирования **Django** расширяет ее, добавляя множество тестов в веб-контекст. Нет оправдания тому, что вы не пишете много тестов; они сэкономят ваше время.

Важно отметить, что не все нужно тестировать. Например, все встроенные функции **Django** уже содержат тесты в исходном коде. Если бы мы использовали в нашем проекте модель **User** по умолчанию, нам не нужно было бы ее тестировать. Но так как мы создали модель **CustomUser**, нам необходимо это сделать.

Unit (модульный тест)

Тесты Для написания модульных тестов в **Django** мы используем **TestCase**, который сам по себе является расширением **Python's TestCase**. Наше приложение для аккаунтов уже содержит файл **tests.py**, который автоматически добавляется при использовании команды **startapp**. В настоящее время он пуст. Давайте исправим это!

Каждый метод должен быть предварен словом **test**, чтобы быть запущенным тестовым пакетом **Django**. Также не помешает быть слишком описательным в названиях ваших юнит-тестов, поскольку зрелые проекты имеют сотни, если не тысячи тестов!

Code

```
# accounts/tests.py
from django.contrib.auth import get_user_model
from django.test import TestCase

class CustomUserTests(TestCase):
    def test_create_user(self):
        User = get_user_model()
        user = User.objects.create_user(
          username='will',
          email='will@email.com',
          password='testpass123'
```

```
        )
            self.assertEqual(user.username, 'will')
            self.assertEqual(user.email, 'will@email.com')
            self.assertTrue(user.is_active)
            self.assertFalse(user.is_staff)
            self.assertFalse(user.is_superuser)
    def test_create_superuser(self):
            User = get_user_model()
            admin_user = User.objects.create_superuser(
                username='superadmin',
                email='superadmin@email.com',
                password='testpass123'
    )
            self.assertEqual(admin_user.username, 'superadmin')
            self.assertEqual(admin_user.email, 'superadmin@email.com')
            self.assertTrue(admin_user.is_active)
            self.assertTrue(admin_user.is_staff)
            self.assertTrue(admin_user.is_superuser)
```

Мы импортировали **get_user_model** и **TestCase** перед созданием класса **CustomUserTests.** В нем есть два отдельных теста**. test_create_user** подтверждает**,** что новый пользователь может быть создан**.** Сначала мы устанавливаем нашу модель пользователя в переменную **User,** а затем создаем его с помощью метода менеджера **create_user,** который выполняет фактическую работу по созданию нового пользователя с соответствующими разрешениями**.**

Для **test_create_superuser** мы следуем аналогичной схеме**,** но вместо **create_user** ссылаемся на **create_superuser** **.** Разница между этими двумя пользователями заключается в том**,** что у суперпользователя и **is_staff,** и **is_superuser** должны иметь значение **True.**

Для запуска наших тестов в **Docker** мы добавим префикс **docker-compose exec web** к традиционной команде **python manage.py test .**

Command Line

```
$ docker-compose exec web python manage.py test
Creating test database for alias 'default'...
System check identified no issues (0 silenced).
..
----------------------------------------------------------------------
Ran 2 tests in 0.268s
OK
Destroying test database for alias 'default'...
```

Все тесты пройдены, поэтому мы можем продолжить.

Git

Мы многого достигли в этой главе, поэтому сейчас самое время сделать паузу и зафиксировать нашу работу, инициализировав новый репозиторий **Git**, добавив изменения и включив сообщение о фиксации.

Command Line

```
$ git init
$ git status
$ git add .
$ git commit -m 'ch3'
```

Вы можете сравнить с официальным исходным кодом этой главы на **Github**.

Заключение

Наш проект **Bookstore** теперь работает с **Docker** и **PostgreSQL**, и мы настроили пользовательскую модель. Следующим проектом будет приложение **pages** для наших статических страниц.

Глава 4: Приложение Pages

Давайте создадим домашнюю страницу для нашего нового проекта. Пока это будет статическая страница, то есть она никак не будет взаимодействовать с базой данных. Позже это будет динамическая страница, отображающая книги на продажу, но... по очереди. Обычно даже в продвинутом проекте есть несколько статических страниц, например, страница **About**, поэтому давайте создадим для них специальное приложение **pages.** В командной строке снова используйте команду **startapp**, чтобы создать приложение **pages.**

Command Line

```
$ docker-compose exec web python manage.py startapp pages
```

Затем добавьте его в наш параметр **INSTALLED_APPS.** Мы также обновим **TEMPLATES,** чтобы **Django** искал папку шаблонов на уровне проекта. По умолчанию **Django** ищет папку шаблонов внутри каждого приложения, но организовать все шаблоны в одном месте проще.

Code

```
# config/settings.py
INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
# Local
     'accounts',
     'pages', # new
  ]
TEMPLATES = [
{
...
   'DIRS': [str(BASE_DIR.joinpath('templates'))], # new
...
  }
]
```

Обратите внимание, что обновление параметра **DIRS** означает, что **Django** также будет искать в этой новой папке; он по-прежнему будет искать любые папки шаблонов внутри приложения.

Шаблоны

Настало время создать новый каталог шаблонов и поместить в него два файла**:** **_base.html** и **home.html .** Первый файл базового уровня будет наследоваться всеми остальными файлами**; home.html** будет нашей домашней страницей**.**

Command Line

```
$ mkdir templates
$ touch templates/_base.html
$ touch templates/home.html
```

Зачем называть базовый шаблон **_base.html** с подчеркиванием вместо **base.html?** Это необязательно**,** но некоторые разработчики предпочитают добавлять знак подчеркивания _ для обозначения файла**,** который предназначен для наследования другими файлами**,** а не отображается сам по себе**.**

В базовый файл мы включим необходимый минимум и добавим блочные теги для заголовка и содержимого**.** Блочные теги дают возможность шаблонам более высокого уровня переопределять только содержимое внутри тегов**.** Например**,** главная страница будет иметь заголовок **"Home"**, но мы хотим**,** чтобы он отображался между тегами **html <title></title>.** Использование блочных тегов упрощает обновление этого содержимого в унаследованных шаблонах**.**

Зачем использовать имя **content** для основного содержимого нашего проекта**?** Это имя может быть любым **- main** или каким**-**то другим общим показателем **-** но использование **content** является общепринятым соглашением об именовании в мире **Django.** Можете ли вы использовать что**-**то другое**?** Безусловно**.** Является ли **content** самым распространенным из тех**,** что вы увидите**?** Да**.**

Code

```
<!-- templates/_base.html -->
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8">
    <title>{% block title %}Bookstore{% endblock title %}</title>
</head>
<body>
   <div class="container">
        {% block content %}
        {% endblock content %}
   </div>
</body>
</html>
```

Теперь о домашней странице**,** которая пока будет называться просто **" Homepage".**

Code

```
<!-- templates/home.html -->
{% extends '_base.html' %}
{% block title %}Home{% endblock title %}
{% block content %}
    <h1>Homepage</h1>
{% endblock content %}
```

URL и Views

Каждая веб-страница в нашем проекте **Django** нуждается в файлах **urls.py** и **views.py**, которые должны идти вместе с шаблоном. Для новичков тот факт, что порядок здесь не имеет значения - нам нужны все три файла и очень часто четвертый, **models.py**, для базы данных - сбивает с толку. Обычно я предпочитаю начинать с ссылок и работать оттуда, но не существует "правильного пути" для построения этой связанной паутины файлов **Django**.

Давайте начнем с **urls.py** на уровне проекта, чтобы установить правильный путь для веб-страниц в приложении **pages**. Поскольку мы хотим создать домашнюю страницу, мы не добавляем никакого дополнительного префикса к маршруту **URL**, который обозначается пустой строкой **''** . Мы также импортируем **include** во второй строке, чтобы лаконично добавить приложение **pages** в наш основной файл **urls.py**.

Code

```
# config/urls.py
from django.contrib import admin
from django.urls import path, include # new

urlpatterns = [
    path('admin/', admin.site.urls),
    path('', include('pages.urls')), # new
]
```

Далее мы создаем файл **urls.py** в приложении **pages**.

Command Line

```
$ touch pages/urls.py
```

Этот файл импортирует **HomePageView** и устанавливает путь, опять же, в пустую строку **''** . Обратите внимание, что мы предоставляем необязательный, но рекомендуемый именованный **URL** **'home'** в конце. Это пригодится в ближайшее время.

Code

```
# pages/urls.py
from django.urls import path
from .views import HomePageView
urlpatterns = [
    path('', HomePageView.as_view(), name='home'),
]
```

Наконец, нам нужен файл **views.py.** Мы можем использовать встроенный в **Django TemplateView,** так что единственная необходимая настройка **-** это указать наш желаемый шаблон, **home.html.**

Code

```
# pages/views.py
from django.views.generic import TemplateView

class HomePageView(TemplateView):
    template_name = 'home.html'
```

Мы почти закончили. Если вы сейчас перейдете на домашнюю страницу **http://127.0.0.1:8000/,** то действительно увидите ошибку. Но что ее вызывает? Поскольку мы запускаем контейнер в фоновом отделенном режиме **-** этот флаг **-d -** мы должны явно проверить журналы, чтобы увидеть вывод консоли.

Поэтому наберите **docker-compose logs,** что приведет к ошибке **"ModuleNotFoundError: No module named 'pages.urls' ".** Происходит то, что **Django** не обновляет автоматически файл **settings.py** для нас, основываясь на изменениях. В мире без **Docker** остановка и перезапуск сервера делают свое дело. Мы должны сделать то же самое здесь, что означает ввод **docker-compose down,** а затем **docker-compose up -d,** чтобы правильно загрузить новое приложение **books.**

Command Line

```
$ docker-compose down
$ docker-compose up -d
```

Обновите домашнюю страницу, и она заработает.

Homepage

Тесты

Время для тестов. Для нашей домашней страницы мы можем использовать **Django's SimpleTestCase**, который является специальным подразделом **Django's TestCase**, предназначенным для веб-страниц, которые не имеют включенной модели.

Сначала тестирование может показаться подавляющим, но оно быстро становится немного скучным. Вы будете использовать одну и ту же структуру и приемы снова и снова. В текстовом редакторе обновите существующий файл **pages/tests.py**. Мы начнем с тестирования шаблона.

Code

```
# pages/tests.py
from django.test import SimpleTestCase
from django.urls import reverse

class HomepageTests(SimpleTestCase):
        def test_homepage_status_code(self):
                response = self.client.get('/')
                self.assertEqual(response.status_code, 200)
        def test_homepage_url_name(self):
                response = self.client.get(reverse('home'))
                self.assertEqual(response.status_code, 200)
```

В верхней части мы импортируем **SimpleTestCase**, а также **reverse**, который полезен для тестирования наших **URL**. Затем мы создаем класс **HomepageTests**, который расширяет **SimpleTestCase**, и внутри него добавляем метод для каждого модульного теста. Обратите внимание, что мы добавляем **self** в качестве первого аргумента каждого юнит-теста. Это соглашение **Python**, которое стоит повторить.

Лучше всего быть слишком описательным в названиях ваших модульных тестов, но помните, что каждый метод должен начинаться с **test**, чтобы быть запущенным набором тестов **Django**.

Оба приведенных здесь теста проверяют, что код состояния **HTTP** для домашней страницы равен **200**, что означает, что она существует. Это еще не говорит нам ничего конкретного о содержимом страницы. Для **test_homepageview_status_code** мы создаем переменную **response**, которая обращается к домашней странице **(/)** и затем использует **Python's assertEqual** для проверки того, что код состояния равен **200**. Аналогичная схема существует для **test_homepage_url_name**, за исключением того, что мы вызываем **URL-**имя **home** через обратный метод. Вспомните, что мы добавили это в файл **pages/urls.py** в качестве лучшей практики. Даже если в будущем мы изменим фактический маршрут этой страницы, мы все равно сможем обращаться к ней по тому же **URL-**имени **home**.

Для запуска наших тестов выполните команду, предваряемую **docker-compose exec web**, чтобы она выполнялась в самом **Docker.**

Command Line

```
$ docker-compose exec web python manage.py test
Creating test database for alias 'default'...
System check identified no issues (0 silenced).
..
----------------------------------------------------------------------
Ran 4 tests in 0.277s
OK
Destroying test database for alias 'default'...
```

Почему здесь говорится о **4** тестах, хотя мы создали только **2?** Потому что мы тестируем весь проект **Django**, и в предыдущей главе в файле **users/tests.py** мы добавили два теста для собственной модели пользователя**.** Если мы хотим запустить тесты только для приложения **pages**, мы просто добавим это имя в команду, так **docker-compose exec web python manage.py test pages .**

Тестирование шаблонов

До сих пор мы проверяли, что домашняя страница существует, но мы также должны подтвердить, что она использует правильный шаблон**. SimpleTestCase** поставляется с методом **assertTemplateUsed** как раз для этой цели**!** Давайте воспользуемся им**.**

Code

```
# pages/tests.py
from django.test import SimpleTestCase
from django.urls import reverse

class HomepageTests(SimpleTestCase):
    def test_homepage_status_code(self):
        response = self.client.get('/')
        self.assertEqual(response.status_code, 200)
    def test_homepage_url_name(self):
        response = self.client.get(reverse('home'))
        self.assertEqual(response.status_code, 200)
    def test_homepage_template(self): # new
        response = self.client.get('/')
        self.assertTemplateUsed(response, 'home.html')
```

Мы снова создали переменную ответа, а затем проверили, что используется шаблон **home.html.** Давайте снова запустим тесты**.**

Command Line

```
$ docker-compose exec web python manage.py test pages
Creating test database for alias 'default'...
System check identified no issues (0 silenced).
...
----------------------------------------------------------------------
Ran 3 tests in 0.023s
OK
Destroying test database for alias 'default'...
```

Заметили ли вы что-то другое в этой команде? Мы добавили имя страницы нашего приложения, чтобы запускались только тесты в этом приложении. На этом раннем этапе вполне можно запускать все тесты, но в больших проектах, если вы знаете, что добавили тесты только в конкретное приложение, можно сэкономить время, запустив только обновленные/новые тесты, а не весь набор.

Тестирование HTML

Теперь давайте убедимся, что наша домашняя страница имеет правильный **HTML-**код, а также не содержит неправильного текста. Всегда полезно проверять как то, что тесты проходят, так и то, что тесты, которые, как мы ожидаем, не пройдут, на самом деле не проходят**!**

Code

```
# pages/tests.py
from django.test import SimpleTestCase
from django.urls import reverse

class HomepageTests(SimpleTestCase):
    def test_homepage_status_code(self):
        response = self.client.get('/')
        self.assertEqual(response.status_code, 200)
    def test_homepage_url_name(self):
        response = self.client.get(reverse('home'))
        self.assertEqual(response.status_code, 200)
    def test_homepage_template(self): # new
        response = self.client.get('/')
        self.assertTemplateUsed(response, 'home.html')
    def test_homepage_contains_correct_html(self): # new
        response = self.client.get('/')
        self.assertContains(response, 'Homepage')
    def test_homepage_does_not_contain_incorrect_html(self): # new
        response = self.client.get('/')
        self.assertNotContains(
            response, 'Hi there! I should not be on the page.')
```

Запустите тесты снова**.**

Command Line

```
$ docker-compose exec web python manage.py test
Creating test database for alias 'default'...
System check identified no issues (0 silenced).
.....
----------------------------------------------------------------------
Ran 7 tests in 0.279s
OK
Destroying test database for alias 'default'...
```

Метод setUp

Вы заметили**,** что мы**,** кажется**,** повторяемся в этих модульных тестах**?** Для каждого из них мы загружаем переменную ответа**.** Это кажется расточительным и чреватым возникновением ошибок**.** Лучше было бы придерживаться чего**-**то более **DRY (Don't Repeat Yourself).**

Поскольку модульные тесты выполняются сверху вниз, мы можем добавить метод **setUp**, который будет выполняться перед каждым тестом. Он установит **self.response** на нашу домашнюю страницу, поэтому нам больше не нужно определять переменную **response** для каждого теста. Это также означает, что мы можем удалить тест **test_homepage_url_name**, поскольку каждый раз в **setUp** мы используем обратное значение **home.**

Code

```
# pages/tests.py
from django.test import SimpleTestCase
from django.urls import reverse

class HomepageTests(SimpleTestCase): # new
        def setUp(self):
                url = reverse('home')
                self.response = self.client.get(url)
        def test_homepage_status_code(self):
                self.assertEqual(self.response.status_code, 200)
        def test_homepage_template(self):
                self.assertTemplateUsed(self.response, 'home.html')
        def test_homepage_contains_correct_html(self):
                self.assertContains(self.response, 'Homepage')
        def test_homepage_does_not_contain_incorrect_html(self):
                self.assertNotContains(
                        self.response, 'Hi there! I should not be on the page.')
```

Теперь запустите тесты снова. Поскольку **setUp** является вспомогательным методом и не начинается с **test,** он не будет считаться модульным тестом в окончательном подсчете. Поэтому будет выполнено только **4** теста.

Command Line

```
$ docker-compose exec web python manage.py test pages
Creating test database for alias 'default'...
System check identified no issues (0 silenced).
....
----------------------------------------------------------------------
Ran 4 tests in 0.278s
OK
Destroying test database for alias 'default'...
```

Разрешение

Последняя проверка представления, которую мы можем сделать, заключается в том, что наш **HomePageView** "разрешает" заданный путь **URL. Django** содержит функцию **resolve**, предназначенную именно для этой цели. Нам нужно будет импортировать как **resolve**, так и **HomePageView** в верхней части файла.

Наш фактический тест, **test_homepage_url_resolves_homepageview** , проверяет, что имя представления, используемого для **resolve /** совпадает с **HomePageView** .

Code

```
# pages/tests.py
from django.test import SimpleTestCase
from django.urls import reverse, resolve # new
from .views import HomePageView # new

class HomepageTests(SimpleTestCase):
        def setUp(self):
                url = reverse('home')
                self.response = self.client.get(url)

        def test_homepage_status_code(self):
                self.assertEqual(self.response.status_code, 200)

        def test_homepage_template(self):
                self.assertTemplateUsed(self.response, 'home.html')

        def test_homepage_contains_correct_html(self):
                self.assertContains(self.response, 'Homepage')

        def test_homepage_does_not_contain_incorrect_html(self):
                self.assertNotContains(
                        self.response, 'Hi there! I should not be on the page.')

        def test_homepage_url_resolves_homepageview(self): # new
                view = resolve('/')
                self.assertEqual(
                        view.func.__name__,
                        HomePageView.as_view().__name__
                )
```

Фух. Это наш последний тест. Давайте подтвердим, что все прошло.

Command Line

```
$ docker-compose exec web python manage.py test
Creating test database for alias 'default'...
System check identified no issues (0 silenced).
.....
----------------------------------------------------------------------
Ran 7 tests in 0.282s
OK
Destroying test database for alias 'default'...
```

Git

Пришло время добавить наши новые изменения в систему контроля исходного кода с помощью **Git.**

Command Line

```
$ git status
$ git add .
$ git commit -m 'ch4'
```

Вы можете сравнить с официальным исходным кодом на **Github** для этой главы**.**

Заключение

Мы настроили наши шаблоны и добавили первую страницу в наш проект**,** статическую домашнюю страницу**.** Мы также добавили тесты**,** которые всегда следует включать в новые изменения кода**.** Некоторые разработчики предпочитают метод**,** называемый **Test-Driven Development,** когда сначала пишутся тесты**,** а затем код**.** Лично я предпочитаю писать тесты сразу после кода**,** что мы и сделаем здесь**.**

Оба подхода работают**,** главное **-** быть строгим в тестировании**.** Проекты **Django** быстро увеличиваются в размерах**,** когда невозможно запомнить все рабочие части в голове**.** А если вы работаете в команде**,** то работать над непроверенной кодовой базой **-** просто кошмар**.** Кто знает**,** что сломается**?**

В следующей главе мы добавим в наш проект регистрацию пользователей**:** вход**,** выход и регистрацию**.**

Глава 5: Регистрация пользователей

Регистрация пользователей является основной функцией любого динамического веб-сайта. И в нашем проекте **Bookstore** она тоже будет. В этой главе мы реализуем функции входа, выхода и регистрации. Первые две функции относительно просты, так как **Django** предоставляет нам необходимые представления и урлы для них, однако регистрация является более сложной задачей, так как для нее нет встроенного решения.

Приложение Auth

Давайте начнем с реализации входа в систему и выхода из нее с помощью собственного приложения аутентификации **Django**. **Django** предоставляет нам необходимые представления и ссылки, что означает, что нам нужно только обновить шаблон, чтобы все заработало. Это экономит нам много времени как разработчикам и гарантирует, что мы не совершим ошибку, поскольку основной код уже протестирован и используется миллионами разработчиков.

Однако за эту простоту приходится расплачиваться тем, что новичкам **Django** кажется "магическим". Некоторые из этих шагов мы уже рассматривали ранее в моей книге "**Django** для начинающих", но мы не стали останавливаться и рассматривать исходный код. Намерение новичка состояло в том, чтобы широко объяснить и продемонстрировать "как" правильно реализовать регистрацию пользователей, но это обошлось без настоящего погружения в "почему" мы использовали тот код, который использовали.

Так как это более продвинутая книга, мы углубились, чтобы лучше понять исходный код. Приведенный здесь подход можно также использовать для самостоятельного изучения любой другой встроенной функциональности **Django**.

Первое, что нам нужно сделать, это убедиться, что приложение **auth** включено в наш параметр **INSTALLED_APPS**. Мы уже добавляли сюда свои собственные приложения, но вы когда-нибудь внимательно смотрели на встроенные приложения, которые **Django** добавляет автоматически для нас? Скорее всего, ответ - нет. Давайте сделаем это сейчас!

Code

```
# config/settings.py
INSTALLED_APPS = [
        'django.contrib.admin',
        'django.contrib.auth', # Yoohoo!!!!
        'django.contrib.contenttypes',
        'django.contrib.sessions',
        'django.contrib.messages',
        'django.contrib.staticfiles',
# Local
        'accounts',
         'pages',
]
```

На самом деле, уже есть **6** приложений, которые **Django** предоставляет нам для работы с сайтом. Первое из них **- admin**, а второе **- auth.** Вот как мы узнаем, что приложение **auth** уже присутствует в нашем проекте **Django.**

Когда мы в первый раз выполнили команду **migrate**, все эти приложения были связаны вместе в исходной базе данных. И помните, что мы использовали параметр **AUTH_USER_MODEL**, чтобы указать **Django** использовать нашу пользовательскую модель пользователя, а не модель **User** по умолчанию. Вот почему мы должны были дождаться завершения этой настройки, прежде чем запускать **migrate** в первый раз.

Auth URLs и представления (views)

Чтобы использовать встроенное в **Django** приложение **auth**, мы должны явно добавить его в наш файл **config/urls.py.** Самый простой подход **-** использовать **accounts/** в качестве префикса, так как это широко используется в сообществе **Django.** Внесите изменения в одну строку ниже. Обратите внимание, что по мере увеличения длины нашего файла **urls.py**, добавление комментариев для каждого типа **URL - admin**, управление пользователями, локальные приложения и т.д. **-** помогает улучшить читаемость.

Code

```
# config/urls.py
from django.contrib import admin
from django.urls import path, include

urlpatterns = [
# Django admin
    path('admin/', admin.site.urls),
# User management
    path('accounts/', include('django.contrib.auth.urls')), # new
# Local apps
    path('', include('pages.urls')),
]
```

Что входит в приложение **auth?** Оказывается, очень многое. Во-первых, есть несколько связанных с ним адресов.

Code

```
accounts/login/ [name='login']
accounts/logout/ [name='logout']
accounts/password_change/ [name='password_change']
accounts/password_change/done/ [name='password_change_done']
accounts/password_reset/ [name='password_reset']
accounts/password_reset/done/ [name='password_reset_done']
accounts/reset/<uidb64>/<token>/ [name='password_reset_confirm']
accounts/reset/done/ [name='password_reset_complete']
```

Откуда я это знаю**?** Двумя способами**.** Первый **-** официальная документация по аутентификации говорит нам об этом**!** Но второй**,** более глубокий подход **-** посмотреть на исходный код **Django,** который доступен на **Github.** Если мы будем перемещаться или искать**,** мы найдем путь к самому приложению **auth.** И в нем мы можем найти файл **urls.py** по этой ссылке**,** который показывает полный исходный код**.**

Чтобы понять исходный код **Django,** требуется практика**,** но это стоит потраченного времени**.**

Домашняя страница

Что дальше**?** Давайте обновим нашу существующую домашнюю страницу так**,** чтобы она уведомляла нас о том**,** вошел ли пользователь в систему или нет**,** что в настоящее время возможно только через администратора**.** Вот новый код для файла **templates/home.html.** Он использует теги **if/else** шаблонизатора **Django** для базовой логики**.**

Code

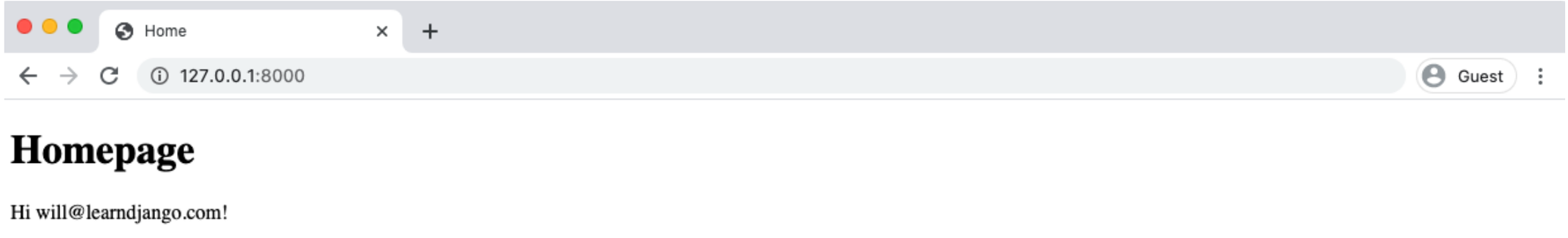
```
<!-- templates/home.html -->
{% extends '_base.html' %}
{% block title %}Home{% endblock title %}
{% block content %}
    <h1>Homepage</h1>
    {% if user.is_authenticated %}
      Hi {{ user.email }}!
    {% else %}
      <p>You are not logged in</p>
      <a href="{% url 'login' %}">Log In</a>
    {% endif %}
{% endblock content %}
```

Если пользователь вошел в систему **(**аутентифицирован**),** мы выводим приветствие**,** в котором говорится "Привет" и указывается адрес его электронной почты**.** Это обе переменные**,** которые мы можем использовать в шаблонизаторе **Django** с помощью двойных открывающих **{{** и закрывающих **}}** скобок**.**

Пользователь по умолчанию содержит множество полей**,** включая **is_authenticated** и **email,** на которые мы ссылаемся здесь**.**

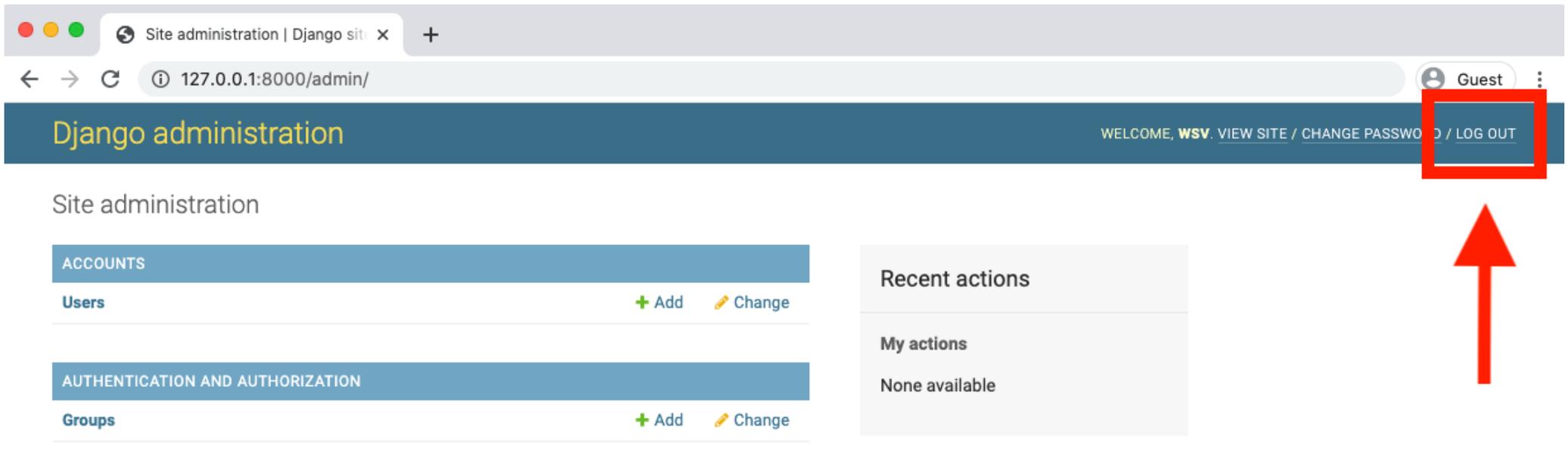
А **logout** и **login -** это имена **URL.** Тег шаблона **url** означает**,** что если мы укажем имя **URL,** то ссылка будет автоматически ссылаться на этот путь **URL.** Например**,** в предыдущей главе мы задали имя **URL** нашей домашней страницы **- home,** поэтому ссылка на домашнюю страницу будет иметь формат **{% url 'home' %}. .** Подробнее об этом в ближайшее время**.**

Если вы сейчас посмотрите на домашнюю страницу **http://127.0.0.1:8000/**, она, скорее всего, покажет адрес электронной почты вашей учетной записи суперпользователя, поскольку ранее мы использовали его для входа в систему.



Домашняя страница с приветствием

В админке на сайте **http://127.0.0.1:8000/admin/** , если вы нажмете на кнопку **"Log out"** в правом верхнем углу, мы можем выйти из админки и, соответственно, из проекта **Django.**



Ссылка для выхода администратора из системы

Вернитесь на домашнюю страницу по адресу **http://127.0.0.1:8000/** и обновите страницу.

Исходный код Django

Возможно, вы смогли бы самостоятельно собрать эти шаги из чтения **официальной документации.** Но более глубокий и лучший подход - это научиться самостоятельно читать исходный код **Django.**

Возникает вопрос, как пользователь и связанные с ним переменные волшебным образом оказались в нашем шаблоне? Ответ заключается в том, что в **Django** есть концепция, называемая **контекстом шаблона**, которая означает, что каждый шаблон загружается данными из соответствующего файла **views.py.** Мы можем использовать **user** внутри тегов шаблона для доступа к атрибутам **User.** Другими словами, **Django** просто предоставляет нам эти данные автоматически.

Так, чтобы проверить, вошел ли пользователь в систему или нет, мы обращаемся к **user** и затем можем использовать атрибут **boolean is_authenticated.** Если пользователь вошел в систему, он вернет значение **True**, и мы сможем сделать такие вещи, как отображение электронной почты пользователя. Если же пользователь не вошел в систему, результат будет **False.**

Далее у нас есть **URL-**адрес **login .** Откуда оно взялось**?** Ответ**,** конечно же**,** из самого **Django!** Давайте распакуем фрагмент кода **{% url 'login' %}** по частям**.**

Во**-**первых**,** мы используем тег шаблона **url,** который принимает в качестве первого аргумента именованный шаблон **URL.** Это необязательный параметр имени**,** который мы добавляем во все наши **URL-**пути в качестве лучшей практики**.** Поэтому к **URL,** используемому **Django** для входа в систему**,** должно быть прикреплено имя **'login',** не так ли**!**

Есть два способа узнать это**.** Другими словами**,** если бы я только что не сказал вам**,** что мы хотим использовать **{% url 'login' %} ,** как бы вы могли это понять**?**

Сначала посмотрите на официальную документацию**.** Лично я часто пользуюсь функцией поиска**,** поэтому я бы набрал что**-**то вроде **"login",** а затем щелкал бы вокруг**,** пока не нашел бы описание входа в систему**.** То**,** что нам нужно**,** на самом деле называется **authentication views** и перечисляет для нас соответствующие шаблоны **URL.**

Code

```
accounts/login/ [name='login']
accounts/logout/ [name='logout']
accounts/password_change/ [name='password_change']
accounts/password_change/done/ [name='password_change_done']
accounts/password_reset/ [name='password_reset']
accounts/password_reset/done/ [name='password_reset_done']
accounts/reset/<uidb64>/<token>/ [name='password_reset_confirm']
accounts/reset/done/ [name='password_reset_complete']
```

Это говорит нам**,** что по пути **accounts/login/** находится **"login",** а его имя **- 'login'.** Немного запутанно поначалу**,** но вот информация**,** которая нам нужна**.**

Углубляясь во вторую фазу**,** мы можем исследовать исходный код **Django,** чтобы увидеть **"logout"** в действии**.** Если вы выполните поиск на **Github,** то в конечном итоге найдете само приложение **auth.** Итак**,** давайте начнем с изучения файла **urls.py.** Вот ссылка на полный код**:**

Code

```
# django/contrib/auth/urls.py
from django.contrib.auth import views
from django.urls import path

urlpatterns = [
    path('login/', views.LoginView.as_view(), name='login'),
    path('logout/', views.LogoutView.as_view(), name='logout'),
    path('password_change/', views.PasswordChangeView.as_view(),
        name='password_change'),
    path('password_change/done/', views.PasswordChangeDoneView.as_view(),
        name='password_change_done'),
    path('password_reset/', views.PasswordResetView.as_view(),
        name='password_reset'),
    path('password_reset/done/', views.PasswordResetDoneView.as_view(),
        name='password_reset_done'),
    path('reset/<uidb64>/<token>/', views.PasswordResetConfirmView.as_view(),
        name='password_reset_confirm'),
    path('reset/done/', views.PasswordResetCompleteView.as_view(),
        name='password_reset_complete'),
]
```

Вот основной код, который **Django** использует сам для приложения **auth.** Надеюсь, вы видите, что маршрут **"logout"** не является магическим. Он прямо здесь, на виду, он использует представление **LogoutView** и имеет **URL** имя **'logout'.** Совсем не волшебство**!** Просто его немного сложно найти с первого раза.

Этот трехэтапный процесс **-** отличный способ обучения**:** либо запомнить комбинацию клавиш **Django**, либо поискать ее в документации, либо в отдельных случаях погрузиться в исходный код и по-настоящему понять, откуда берется все это добро.

Вход в систему

Вернувшись на нашу основную домашнюю страницу, нажмите на ссылку "Войти" и... получите ошибку**!**

Ошибка отсутствия шаблона входа в систему

Django выдает ошибку **TemplateDoesNotExist.** В частности, похоже, что он ожидает шаблон входа в систему по адресу **registration/login.html .** Помимо того, что **Django** говорит нам об этом, мы можем посмотреть в документации и увидеть, что нужное **template_name** имеет такое расположение.

Но давайте действительно убедимся и проверим исходный код, чтобы исключить здесь любую мнимую магию. В конце концов, это всего лишь **Django.**

Вернувшись в файл **auth/views.py**, мы можем увидеть в строке **47** для **LoginView**, что **template_name -** это **'registration/login.html' .** Поэтому, если бы мы хотели изменить местоположение по умолчанию, мы могли бы это сделать, но это означало бы переопределение **LoginView**, что кажется излишним. Давайте просто воспользуемся тем, что дает нам **Django.**

Создадим новую папку **registration** в существующей директории **templates**, а затем добавим туда наш файл **login.html.**

Command Line

```
$ mkdir templates/registration
$ touch templates/registration/login.html
```

Фактический код выглядит следующим образом. Мы расширяем наш базовый шаблон, добавляем заголовок, а затем указываем, что мы хотим использовать форму, которая будет " отправлять" или посылать данные.

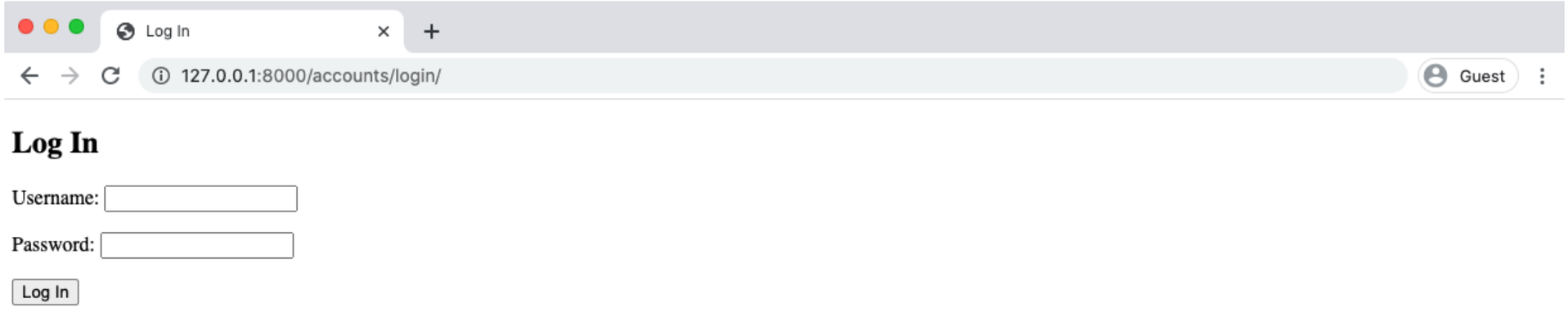
Code

```
<!-- templates/registration/login.html -->
{% extends '_base.html' %}
{% block title %}Log In{% endblock title %}
{% block content %}
    <h2>Log In</h2>
    <form method="post">
        {% csrf_token %}
        {{ form.as_p }}
        <button type="submit">Log In</button>
    </form>
{% endblock content %}
```

Вы всегда должны добавлять **CSRF-**защиту на любую отправляемую форму. В противном случае вредоносный сайт может изменить ссылку и атаковать сайт и пользователя. В **Django** есть промежуточное программное обеспечение **CSRF**, которое справится с этим за нас; все, что нам нужно сделать, это добавить теги **{% csrf_token %}** в начало формы.

Далее мы можем управлять внешним видом содержимого формы. Пока что мы будем использовать **as_p(),** чтобы каждое поле формы отображалось в теге абзаца **p.**

Покончив с этим объяснением, давайте проверим, правильно ли работает наш новый шаблон. Обновите веб-страницу по адресу **http://127.0.0.1:8000/accounts/login/ .**

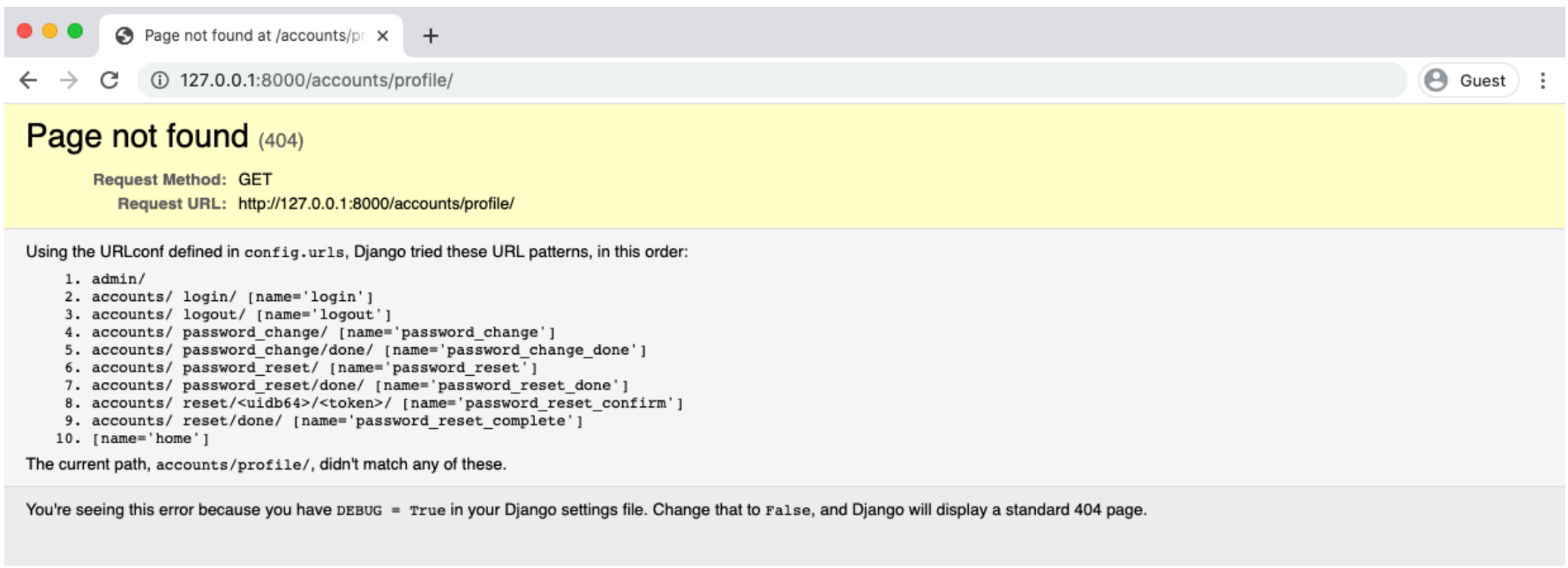


Страница входа в систему

А вот и наша страница! С любовью. Вы можете вернуться на главную страницу и убедиться, что ссылка "Войти" работает, если хотите. В качестве последнего шага попробуйте войти в систему под своей учетной записью суперпользователя на странице входа.

Переадресация(Redirects)

Вы заметили, что в последнем предложении я сказал "попытаться"? Если вы нажмете на ссылку "Войти", появится ошибка **Page not found (404).**



Ошибка "Страница не найдена" ("Page not found")

Django перенаправил нас на **127.0.0.1:8000/accounts/profile/,** но такой страницы не существует. Почему **Django** сделал это? Ну, если подумать, откуда **Django** знает, куда мы хотим перенаправить пользователя после входа в систему? Возможно, это домашняя страница. Но может быть, это страница профиля пользователя. Или любое количество вариантов.

Последняя часть головоломки входа в систему - это установка правильной конфигурации для **LOGIN_REDIRECT_URL,** потому что по умолчанию он перенаправляет на **accounts/profile .**

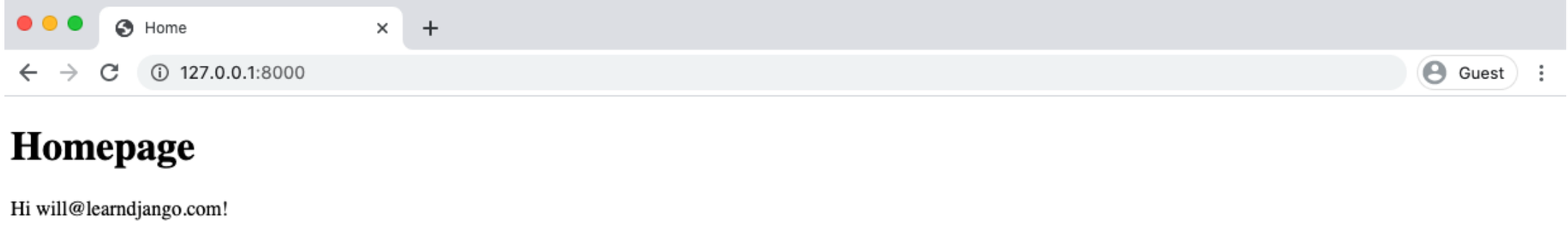
К счастью, это можно быстро исправить. Мы отправим пользователя на нашу домашнюю страницу. И так как мы указали **URL** имя **home,** это все, что нам нужно для перенаправления вошедших пользователей на домашнюю страницу.

В нижней части файла **config/settings.py** добавьте одну строку.

Code

```
# config/settings.py
LOGIN_REDIRECT_URL = 'home'
```

Повторите попытку входа в систему по адресу **http://127.0.0.1:8000/accounts/login/ .** В случае удачной попытки он перенаправляет пользователя на домашнюю страницу, приветствуя учетную запись суперпользователя, с которой вы только что вошли в систему**!**



Выход с домашней страницы

Выход из системы

Теперь давайте добавим опцию выхода из системы на нашу домашнюю страницу**,** поскольку только суперпользователь будет иметь доступ к админке**.** Как нам это сделать**?**

Если вы посмотрите на приведенные выше представления **auth,** то увидите**,** что **logout** использует **LogoutView ,** который мы можем изучить в исходном коде**,** и имеет **URL** имя **logout .** Это означает**,** что мы можем ссылаться на него с помощью тега шаблона как на просто **logout .**

Но при желании мы можем задать это сами**,** используя **LOGOUT_REDIRECT_URL,** который можно добавить в нижнюю часть нашего файла **config/settings.py.** Давайте сделаем так**,** чтобы вышедший из системы пользователь перенаправлялся на домашнюю страницу**.**

Code

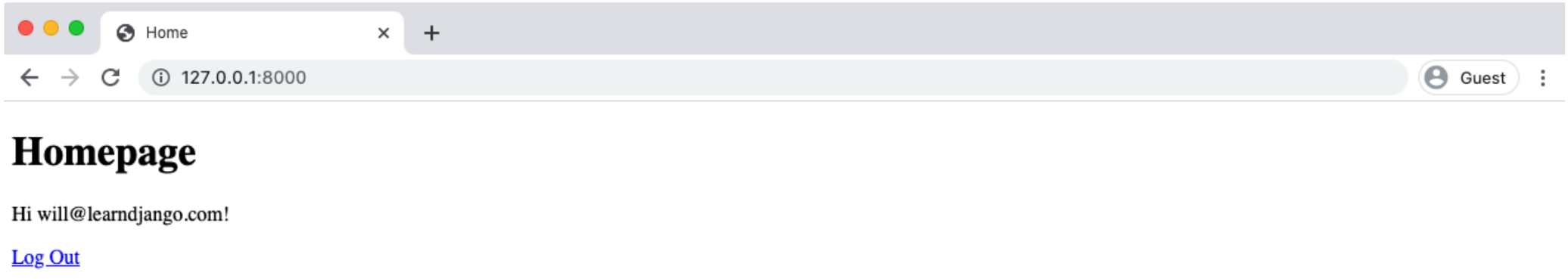
```
# config/settings.py
LOGIN_REDIRECT_URL = 'home'
LOGOUT_REDIRECT_URL = 'home' # new
```

Затем добавьте ссылку выхода из системы в **templates/home.html .**

Code

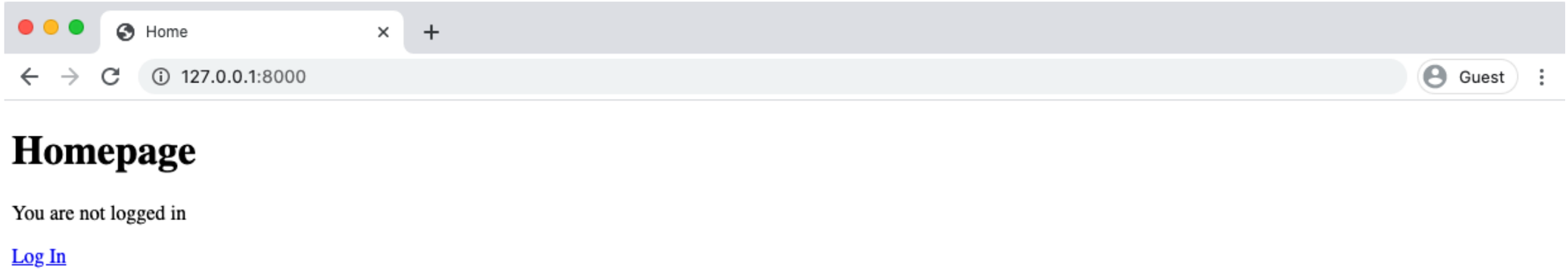
```
<!-- templates/home.html -->
{% extends '_base.html' %}
{% block title %}Home{% endblock title %}
{% block content %}
    <h1>Homepage</h1>
    {% if user.is_authenticated %}
      Hi {{ user.email }}!
      <p><a href="{% url 'logout' %}">Log Out</a></p>
    {% else %}
      <p>You are not logged in</p>
      <a href="{% url 'login' %}">Log In</a>
    {% endif %}
{% endblock content %}
```

Обновите домашнюю страницу на сайте **http://127.0.0.1:8000/,** и ссылка **" Log out"** теперь видна**.**



Домашняя страница со ссылкой для выхода из системы

Если вы нажмете на нее**,** вы выйдете из системы и будете перенаправлены на главную страницу**,** на которой видна ссылка **" Log In".**



Домашняя страница со ссылкой для входа

Регистрация

Реализация страницы регистрации для регистрации пользователей полностью зависит от нас**.** Мы пройдем через стандартные шаги для любой новой страницы**:**

- создадим файл **accounts/urls.py** на уровне приложения
- обновите файл **config/urls.py** на уровне проекта**,** чтобы он указывал на приложение **accounts**
- добавим представление под названием **SignupPageView**
- создайте шаблон **signup.html**
- обновите файл **home.html** для отображения страницы регистрации

Часто задают вопрос**:** каков правильный порядок выполнения этих шагов**?** Честно говоря**,** это не имеет значения**,** поскольку все они нужны для правильной работы страницы регистрации**.** Как правило**,** я предпочитаю начинать с **urls,** затем переходить к представлениям и**,** наконец**,** к шаблонам**,** но это вопрос личных предпочтений**.**

Для начала создайте файл **urls.py** в приложении **accounts.** До этого момента оно содержит только нашего **CustomUser** в файле **models.py;** мы не настроили никаких маршрутов или представлений**.**

Command Line

```
$ touch accounts/urls.py
```

URL путь для страницы регистрации будет принимать представление под названием **SignupPageView (**которое мы создадим далее**),** по маршруту **signup/ ,** и иметь имя **signup,** которое мы можем позже использовать для ссылки на страницу с помощью тега шаблона **url.** Существующие имена **url** для **login** и **signup** записаны во встроенном файле приложения **Django django/contrib/auth/urls.py,** который мы рассмотрели выше**.**

Code

```
# accounts/urls.py
from django.urls import path
from .views import SignupPageView

urlpatterns = [
    path('signup/', SignupPageView.as_view(), name='signup'),
]
```

Затем обновите файл **config/urls.py,** чтобы включить в него приложение **accounts.** Мы можем создать любой маршрут**,** который нам нравится**,** но обычно используется тот же маршрут **accounts/,** который используется приложением **auth** по умолчанию**.** Обратите внимание**,** что важно включить путь для **accounts.urls** ниже**:** Пути **URL** загружаются сверху вниз**,** поэтому это гарантирует**,** что все пути **URL auth** будут загружены первыми**.**

Code

```
# config/urls.py
from django.contrib import admin
from django.urls import path, include

urlpatterns = [
# Django admin
     path('admin/', admin.site.urls),
# User management
    path('accounts/', include('django.contrib.auth.urls')),
# Local apps
    path('accounts/', include('accounts.urls')), # new
    path('', include('pages.urls')),
]
```

Теперь создайте представление **SignupPageView .** Оно ссылается на **CustomUserCreationForm** и имеет **success_url,** который указывает на страницу входа**,** то есть после отправки формы пользователь будет перенаправлен туда**.** **template_name** будет **signup.html .**

Code

```
# accounts/views.py
from django.urls import reverse_lazy
from django.views import generic
from .forms import CustomUserCreationForm

class SignupPageView(generic.CreateView):
        form_class = CustomUserCreationForm
        success_url = reverse_lazy('login')
        template_name = 'registration/signup.html'
```

В качестве последнего шага создайте файл под названием **signup.html** в существующем каталоге **registration/.**

Command Line

```
$ touch templates/registration/signup.html
```

Код в основном идентичен коду страницы входа в систему**.**

Code

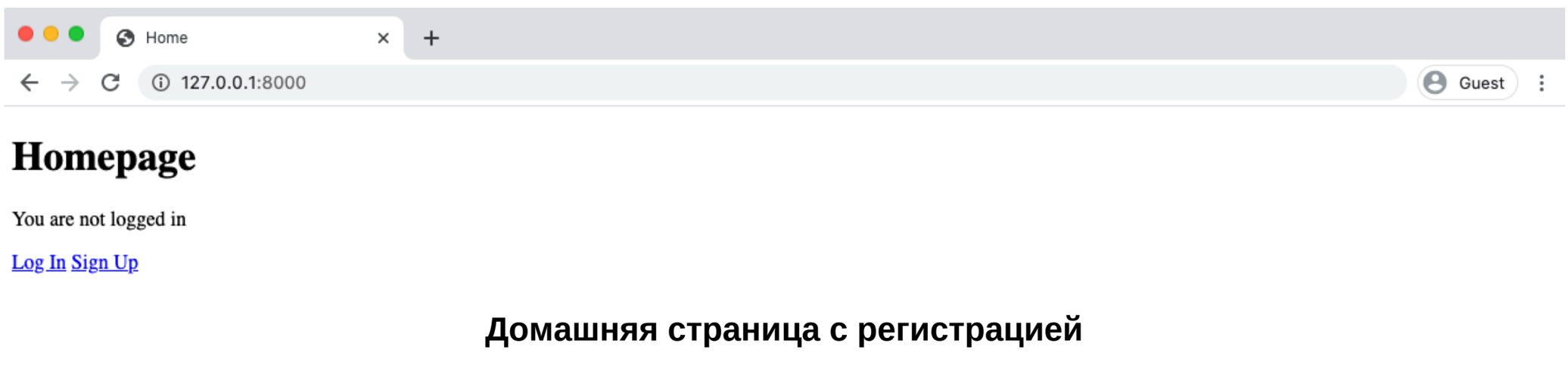
```
<!-- templates/registration/signup.html -->
{% extends '_base.html' %}
{% block title %}Sign Up{% endblock title %}
{% block content %}
    <h2>Sign Up</h2>
    <form method="post">
        {% csrf_token %}
        {{ form.as_p }}
        <button type="submit">Sign Up</button>
    </form>
{% endblock content %}
```

В качестве последнего шага мы можем добавить строку для **"Sign Up"** в наш шаблон **home.html** прямо под ссылкой для **"Log In".** Это изменение состоит из одной строки**.**

Code

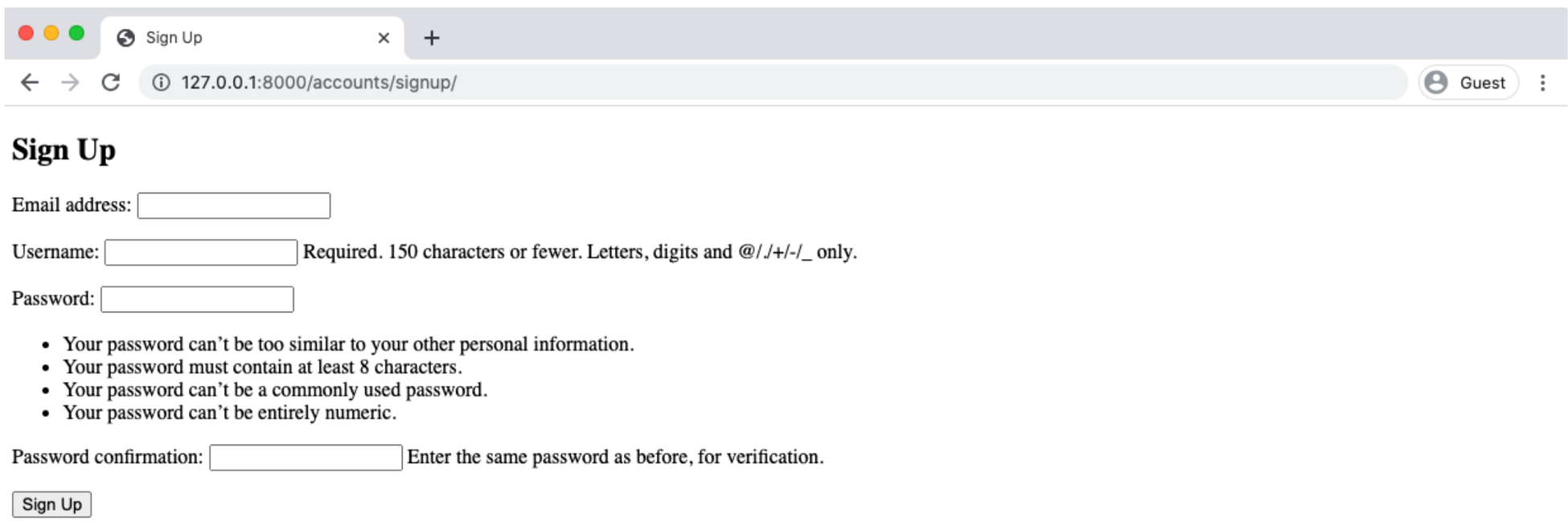
```
<!-- templates/home.html -->
{% extends '_base.html' %}
{% block title %}Home{% endblock title %}
{% block content %}
    <h1>Homepage</h1>
    {% if user.is_authenticated %}
      Hi {{ user.email }}!
      <p><a href="{% url 'logout' %}">Log Out</a></p>
    {% else %}
      <p>You are not logged in</p>
      <a href="{% url 'login' %}">Log In</a>
      <a href="{% url 'signup' %}">Sign Up</a>
    {% endif %}
{% endblock content %}
```

Все готово! Перезагрузите домашнюю страницу, чтобы увидеть нашу работу.



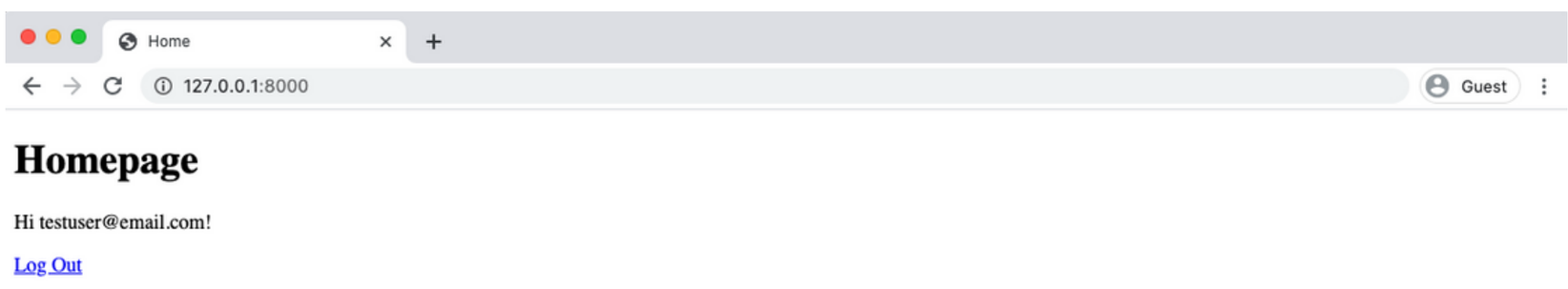
Домашняя страница с регистрацией

Ссылка **" Sign Up"** перенаправит нас на сайт **http://127.0.0.1:8000/accounts/signup/ .**



Страница регистрации

Создайте нового пользователя с адресом электронной почты **testuser@email.com ,** именем пользователя **testuser** и паролем **testpass123.** После ввода данных произойдет перенаправление на страницу **Log In.** Войдите в систему с новой учетной записью, и она перенаправит нас на главную страницу с персональным приветствием.



Домашняя страница с приветствием тестового пользователя

Тесты

Для тестов нам не нужно тестировать функции входа и выхода из системы, поскольку они встроены в **Django** и уже имеют тесты. Однако нам нужно проверить функциональность регистрации! Давайте начнем с создания метода **setUp**, который загрузит нашу страницу. Затем мы заполним шаблон **test_signup_-** тестами на код состояния, используемый шаблон, включенный и исключенный текст, аналогично тому, как мы делали это в прошлой главе для главной страницы. В текстовом редакторе обновите файл **accounts/tests.py** с этими изменениями.

Code

```
# accounts/tests.py
from django.contrib.auth import get_user_model
from django.test import TestCase
from django.urls import reverse # new

class CustomUserTests(TestCase):
...
class SignupPageTests(TestCase): # new
        def setUp(self):
                url = reverse('signup')
                self.response = self.client.get(url)
        def test_signup_template(self):
                self.assertEqual(self.response.status_code, 200)
                self.assertTemplateUsed(self.response, 'registration/signup.html')
                self.assertContains(self.response, 'Sign Up')
                self.assertNotContains(
                        self.response, 'Hi there! I should not be on the page.')
```

Затем запустите наши тесты.

Command Line

```
$ docker-compose exec web python manage.py test
Creating test database for alias 'default'...
System check identified no issues (0 silenced).
........
----------------------------------------------------------------------
Ran 8 tests in 0.329s
OK
Destroying test database for alias 'default'...
```

Далее мы можем проверить, что наша **CustomUserCreationForm** используется и что страница преобразуется в **SignupPageView .**

Code

```
# accounts/tests.py
from django.contrib.auth import get_user_model
from django.test import TestCase
from django.urls import reverse, resolve # new
from .forms import CustomUserCreationForm # new
from .views import SignupPageView # new

class CustomUserTests(TestCase):
...
class SignupPageTests(TestCase):
    def setUp(self):
        url = reverse('signup')
        self.response = self.client.get(url)

    def test_signup_template(self):
        self.assertEqual(self.response.status_code, 200)
        self.assertTemplateUsed(self.response, 'signup.html')
        self.assertContains(self.response, 'Sign Up')
        self.assertNotContains(
            self.response, 'Hi there! I should not be on the page.')

    def test_signup_form(self): # new
        form = self.response.context.get('form')
        self.assertIsInstance(form, CustomUserCreationForm)
        self.assertContains(self.response, 'csrfmiddlewaretoken')

    def test_signup_view(self): # new
        view = resolve('/accounts/signup/')
        self.assertEqual(
            view.func.__name__,
            SignupPageView.as_view().__name__
)
```

Запустите наши тесты снова.

Command Line

```
$ docker-compose exec web python manage.py test
Creating test database for alias 'default'...
System check identified no issues (0 silenced).
.........
----------------------------------------------------------------------
Ran 10 tests in 0.328s
OK
Destroying test database for alias 'default'...
```

Все готово.

setUpTestData()

В **Django 1.8** было представлено крупное обновление **TestCase**, которое добавило возможность запускать тесты как в рамках всего класса, так и для каждого отдельного теста. В частности, **setUpTestData()** позволяет создавать начальные данные на уровне класса, которые могут быть применены ко всему **TestCase**. Это позволяет создавать гораздо более быстрые тесты, чем использование **setUp()**, однако необходимо следить за тем, чтобы не модифицировать объекты, созданные в **setUpTestData()**, в ваших тестовых методах.

В этой книге мы будем использовать **setUp()**, но имейте в виду, что если ваш набор тестов кажется медленным, то потенциальной оптимизацией может быть использование **setUpTestData()**.

Git

Как всегда, не забудьте сохранить нашу работу, добавив изменения в **Git**.

Command Line

```
$ git status
$ git add .
$ git commit -m 'ch5'
```

Официальный исходный код находится на **Github**, если вы хотите сравнить свой код.

Заключение

Наш проект **Bookstore -** не самый красивый сайт в мире, но на данный момент он очень функционален. В следующей главе мы настроим наши статические параметры и добавим **Bootstrap** для улучшения стиля.

Глава 6: Статические файлы

Статические ресурсы, такие как **CSS**, **JavaScript** и изображения, являются основным компонентом любого сайта, и **Django** предоставляет нам большую степень гибкости в их конфигурации и хранении. В этой главе мы настроим наши начальные статические ресурсы и добавим **Bootstrap** в наш проект для улучшения стиля.

приложение staticfiles

Django полагается на приложение **staticfiles** для управления статическими файлами всего проекта, делает их доступными для быстрой локальной разработки на файловой системе, а также объединяет их в одно место, которое может быть использовано в продакшене с лучшей производительностью. Этот процесс и различие между локальными и производственными статическими файлами сбивает с толку многих новичков **Django.** Для начала мы обновим конфигурацию приложения **staticfiles** в **settings.py .**

STATIC_URL

Первая настройка статических файлов, **STATIC_URL**, уже включена для нас в файл **config/settings.py.**

Code

```
# config/settings.py
STATIC_URL = '/static/'
```

Это задает **URL**, который мы можем использовать для ссылки на статические файлы. Обратите внимание, что в конце имени директории важно включить косую черту **/.**

STATICFILES_DIRS

Далее идет **STATICFILES_DIRS**, который определяет расположение статических файлов в локальной разработке. В нашем проекте все они будут находиться в статической директории верхнего уровня.

Code

```
# config/settings.py
STATIC_URL = '/static/'
STATICFILES_DIRS = (str(BASE_DIR.joinpath('static')),) # new
```

Часто бывает так, что в проекте существует несколько директорий со статическими файлами, поэтому здесь обычно добавляются скобки **Python [],** которые обозначают список, чтобы учесть будущие добавления.

STATIC_ROOT

STATIC_ROOT - это местоположение статических файлов для производства, поэтому ему должно быть присвоено другое имя, обычно **staticfiles .** Когда приходит время развертывать проект **Django**, команда **collectstatic** автоматически компилирует все доступные статические файлы по всему проекту в один каталог. Это намного быстрее, чем если бы статические файлы были разбросаны по всему проекту, как это бывает при локальной разработке.

Code

```
# config/settings.py
STATIC_URL = '/static/'
STATICFILES_DIRS = (str(BASE_DIR.joinpath('static')),)
STATIC_ROOT = str(BASE_DIR.joinpath('staticfiles')) # new
```

STATICFILES_FINDERS

Последняя настройка **- STATICFILES_FINDERS**, которая указывает **Django**, как искать каталоги статических файлов. Он установлен для нас неявно, и хотя это необязательный шаг, я предпочитаю сделать его явным во всех проектах.

Code

```
# config/settings.py
STATICFILES_FINDERS = [
    "django.contrib.staticfiles.finders.FileSystemFinder",
    "django.contrib.staticfiles.finders.AppDirectoriesFinder",
]
```

FileSystemFinder ищет любые статические файлы в параметре **STATICFILES_DIRS**, который мы установили на **static.** Затем **AppDirectoriesFinder** ищет любые директории с именем **static**, расположенные внутри приложения, в отличие от директории **static** на уровне проекта. Эта настройка считывается сверху вниз, то есть если файл с именем **static/img.png** будет найден **FileSystemFinder**'ом первым, он будет заменен на файл **img.png**, расположенный, скажем, в приложении **pages** по адресу **pages/static/img.png .**

Таким образом, наша конечная группа настроек должна выглядеть следующим образом:

Code

```
# config/settings.py
STATIC_URL = '/static/'
STATICFILES_DIRS = (str(BASE_DIR.joinpath('static')),)
STATIC_ROOT = str(BASE_DIR.joinpath('staticfiles')) # new
STATICFILES_FINDERS = [ # new
    "django.contrib.staticfiles.finders.FileSystemFinder",
    "django.contrib.staticfiles.finders.AppDirectoriesFinder",
]
```

Директория статических файлов

Теперь давайте добавим несколько статических файлов и включим их в наш проект. Несмотря на то, что мы ссылаемся на статический директорий для наших файлов, мы сами должны создать его, поэтому сделаем это сейчас, вместе с новыми поддиректориями для **CSS**, **JavaScript** и изображений.

Command Line

```
$ mkdir static
$ mkdir static/css
$ mkdir static/js
$ mkdir static/images
```

Затем создайте файл **base.css.**

Command Line

```
$ touch static/css/base.css
```

Мы будем придерживаться базовых принципов и сделаем заголовок **h1** красным. Мы хотим показать, как можно добавить **CSS** в наш проект, а не углубляться в сам **CSS.**

Code

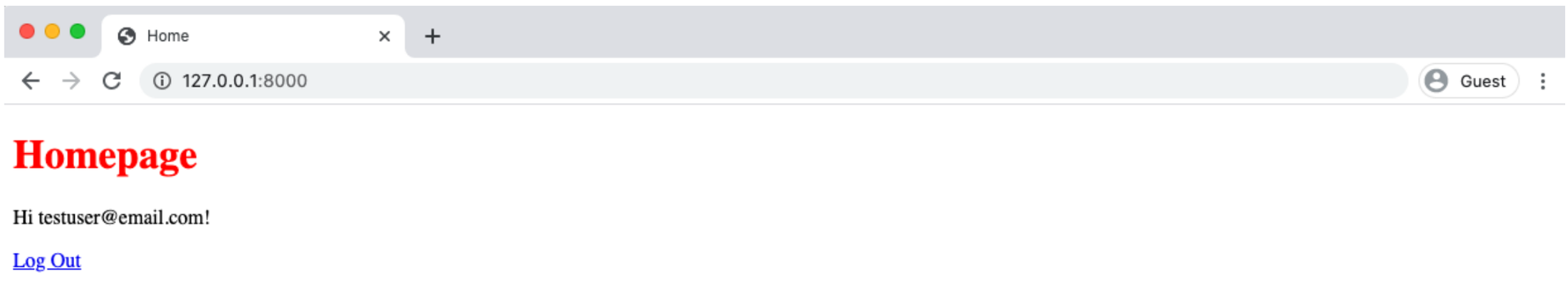
```
/* static/css/base.css */
h1 {
    color: red;
}
```

Если вы сейчас обновите главную страницу, то увидите, что ничего не изменилось. Это потому, что статические файлы должны быть явно загружены в шаблоны. Сначала загрузите все статические файлы в верхней части страницы с помощью **{% load static %}**, а затем включите ссылку на файл **base.css.** Тег шаблона **static** использует **STATIC_URL**, который мы установили в **/static/**, поэтому вместо того, чтобы прописывать **static/css/base.css**, мы можем просто сослаться на **css/base.css.**

Code

```
<!-- templates/_base.html -->
{% load static %}
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8">
    <title>{% block title %}Bookstore{% endblock %}</title>
    <!-- CSS -->
    <link rel="stylesheet" href="{% static 'css/base.css' %}">
</head>
...
```

Обновите главную страницу, чтобы увидеть нашу работу. Вот наш **CSS** в действии!



Домашняя страница с красным текстом

Если вместо этого вы видите экран ошибки, говорящий **Invalid block tag on line 7: 'static'.** Вы забыли зарегистрировать или загрузить этот тег**?** тогда вы забыли включить строку **{% load static %}** в верхней части файла**.** Я сам постоянно так делаю**.**

Изображения

Как насчет изображения**?** Вы можете скачать обложку книги **"Django** для профессионалов" по этой ссылке**.** Сохраните ее в директории **books/static/images** как **dfp.png .** Чтобы отобразить его на главной странице**,** обновите **templates/home.html .** Добавьте оба тега **{% load static %}** вверху и в предпоследней строке ссылку **<img>** для файла**.**

Code

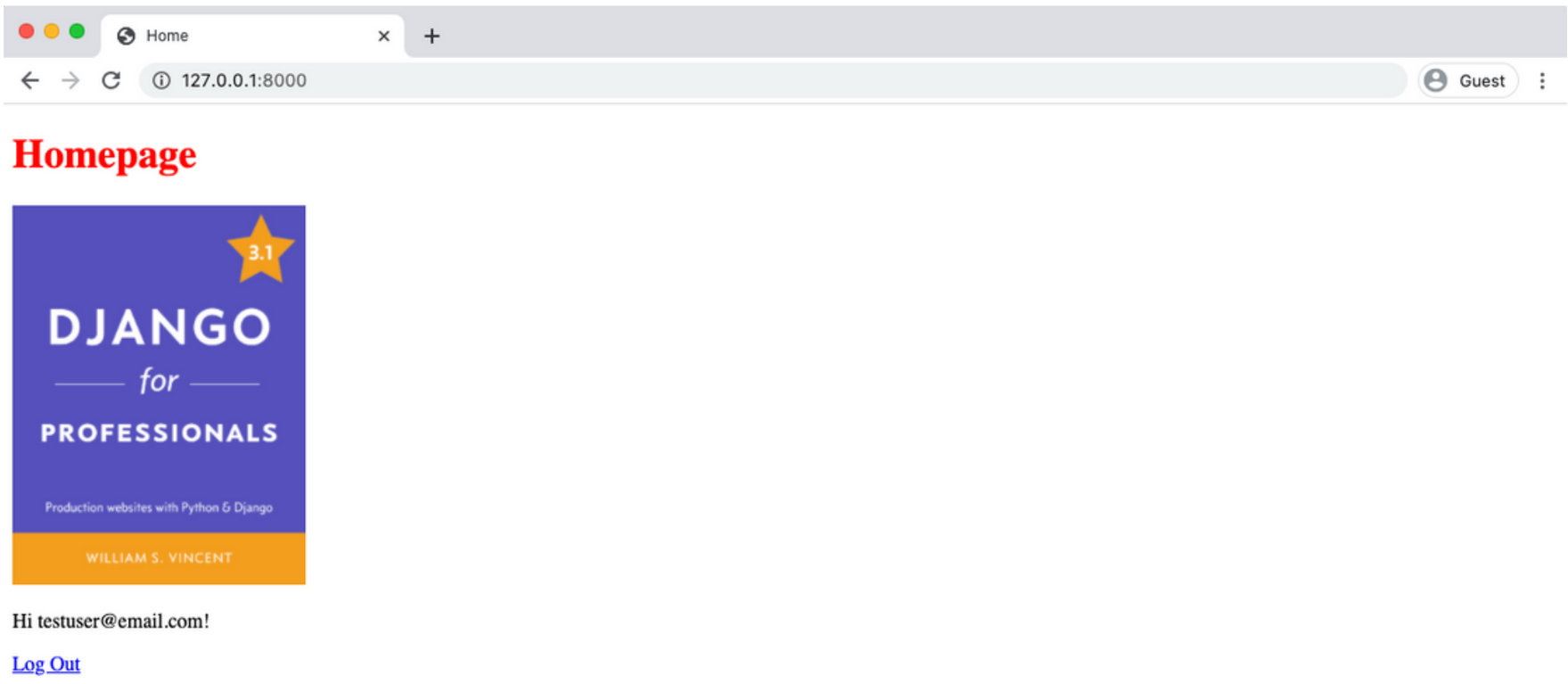
```
<!-- templates/home.html -->
{% extends '_base.html' %}
{% load static %}
{% block title %}Home{% endblock title %}
{% block content %}
    <h1>Homepage</h1>
    <img class="bookcover" src="{% static 'images/dfp.png' %}">
     {% if user.is_authenticated %}
        <p>Hi {{ user.email }}!</p>
        <p><a href="{% url 'logout' %}">Log Out</a></p>
     {% else %}
         <p>You are not logged in</p>
          <p><a href="{% url 'login' %}">Log In</a> |
               <a href="{% url 'signup' %}">Sign Up</a></p>
     {% endif %}
{% endblock content %}
```

Обновив домашнюю страницу**,** вы увидите**,** что необработанный файл довольно большой**!** Давайте разберемся с этим с помощью дополнительного **CSS.**

Code

```
/* static/css/base.css */
h1 {
    color: red;
}
.bookcover {
    height: 300px;
    width: auto;
}
```

Теперь обновите домашнюю страницу, и изображение обложки книги прекрасно впишется.



Домашняя страница с обложкой книги

JavaScript

Чтобы добавить **JavaScript,** мы проделаем аналогичный процесс**.** Создайте новый файл под названием **base.js .**

Command Line

```
$ touch static/js/base.js
```

Часто я помещаю здесь код отслеживания**,** например**,** для **Google Analytics,** но для демонстрации мы добавим оператор **console.log,** чтобы убедиться**,** что **JavaScript** загрузился правильно**.**

Code

```
// static/js/base.js
console.log('JavaScript here!')
```

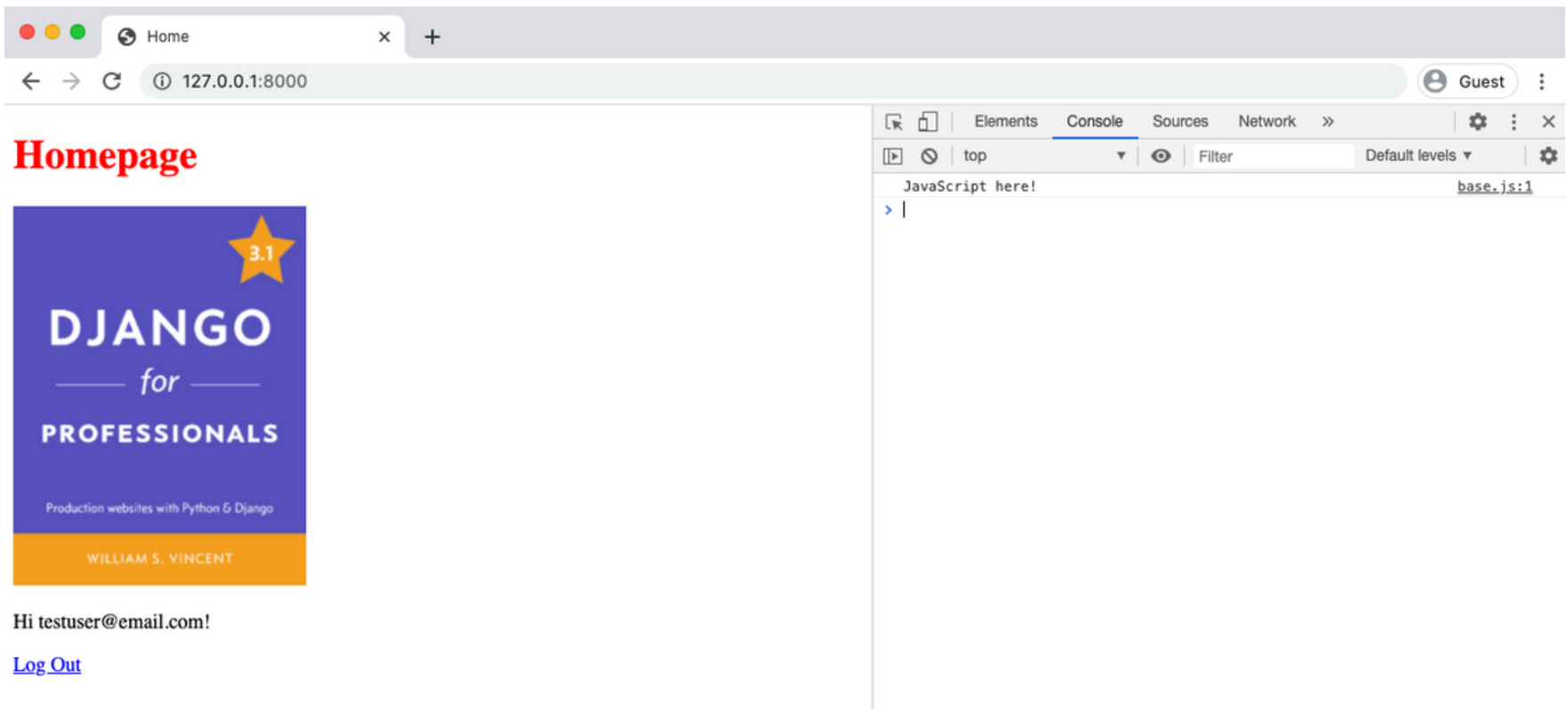
Теперь добавьте его в наш шаблон **_base.html. JavaScript** следует добавлять в самый низ файла**,** чтобы он загружался последним**,** после **HTML, CSS** и других компонентов**,** которые первыми появляются на экране при отображении в веб-браузере**.** Это создает видимость того**,** что вся веб-страница загружается быстрее**.**

Code

```
<!-- templates/_base.html -->
{% load static %}
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8">
    <title>{% block title %}Bookstore{% endblock title %}</title>
     <!-- CSS -->
    <link rel="stylesheet" href="{% static 'css/base.css' %}">
</head>
<body>
    <div class="container">
        {% block content %}
        {% endblock content %}
    </div>
<!-- JavaScript -->
     <script src="{% static 'js/base.js' %}"></script>
</body>
</html>
```

В веб-браузере сделайте доступной консоль **JavaScript.** Для этого откройте **Developer Tools** и убедитесь**,** что вы находитесь в разделе **"Console".** В браузере **Chrome,** который используется для изображений в этой книге**,** перейдите в верхнем меню в раздел "Вид"**,** затем "Разработчик" **->** "Инструменты разработчика"**,** что откроет боковую панель**.**

Убедитесь**,** что в списке опций выбрана "Консоль"**.** Если вы обновите страницу**,** вы должны увидеть следующее**:**



Представление консоли JavaScript на домашней странице

collectstatic

Представьте**,** что мы хотим сразу же развернуть наш сайт**.** Среди прочих шагов**,** нам нужно будет запустить **collectstatic,** чтобы создать единый**,** готовый к производству директорий всех статических файлов нашего проекта**.**

Command Line

```
$ docker-compose exec web python manage.py collectstatic
135 static files copied to '/code/staticfiles'.
```

Если вы посмотрите в свой текстовый редактор, то там теперь есть каталог **staticfiles** с четырьмя подкаталогами: **admin , css , images** и **js .** Первый **-** это статические файлы приложения **Django admin,** а остальные три мы указали. Вот почему скопировано **122** файла.

Bootstrap

Написание собственных **CSS** для вашего сайта **-** достойная цель, и я советую всем разработчикам программного обеспечения, даже **back-end**, когда-нибудь попробовать. Но с практической точки зрения существует причина существования таких фреймворков, как **Bootstrap:** они экономят вам кучу времени при запуске нового проекта. Если у вас нет специального дизайнера, с которым можно сотрудничать, придерживайтесь фреймворка для ранних итераций вашего сайта.

В этом разделе мы добавим **Bootstrap** в наш проект вместе с существующим файлом **base.css.** Написание всего этого вручную займет много времени и чревато ошибками, поэтому это тот редкий случай, когда я советую просто скопировать/вставить из официального исходного кода.

Обратите внимание, что порядок имеет значение как для **CSS,** так и для **JavaScript.** Файл будет загружаться сверху вниз, поэтому наш файл **base.css** идет после **CSS Bootstrap**, поэтому наш стиль **h1** отменяет стиль **Bootstrap** по умолчанию. В нижней части файла также важно загрузить сначала **jQuery**, затем **PopperJS**, и только потом файл **Bootstrap JavaScript.**

Наконец, обратите внимание, что в проект был добавлен навигационный заголовок с базовой логикой, так что если пользователь вошел в систему, то будет видна только ссылка **"Log Out"**, в то время как пользователь, вышедший из системы, увидит обе ссылки **"Log In"** и **"Sign Up".**

Code

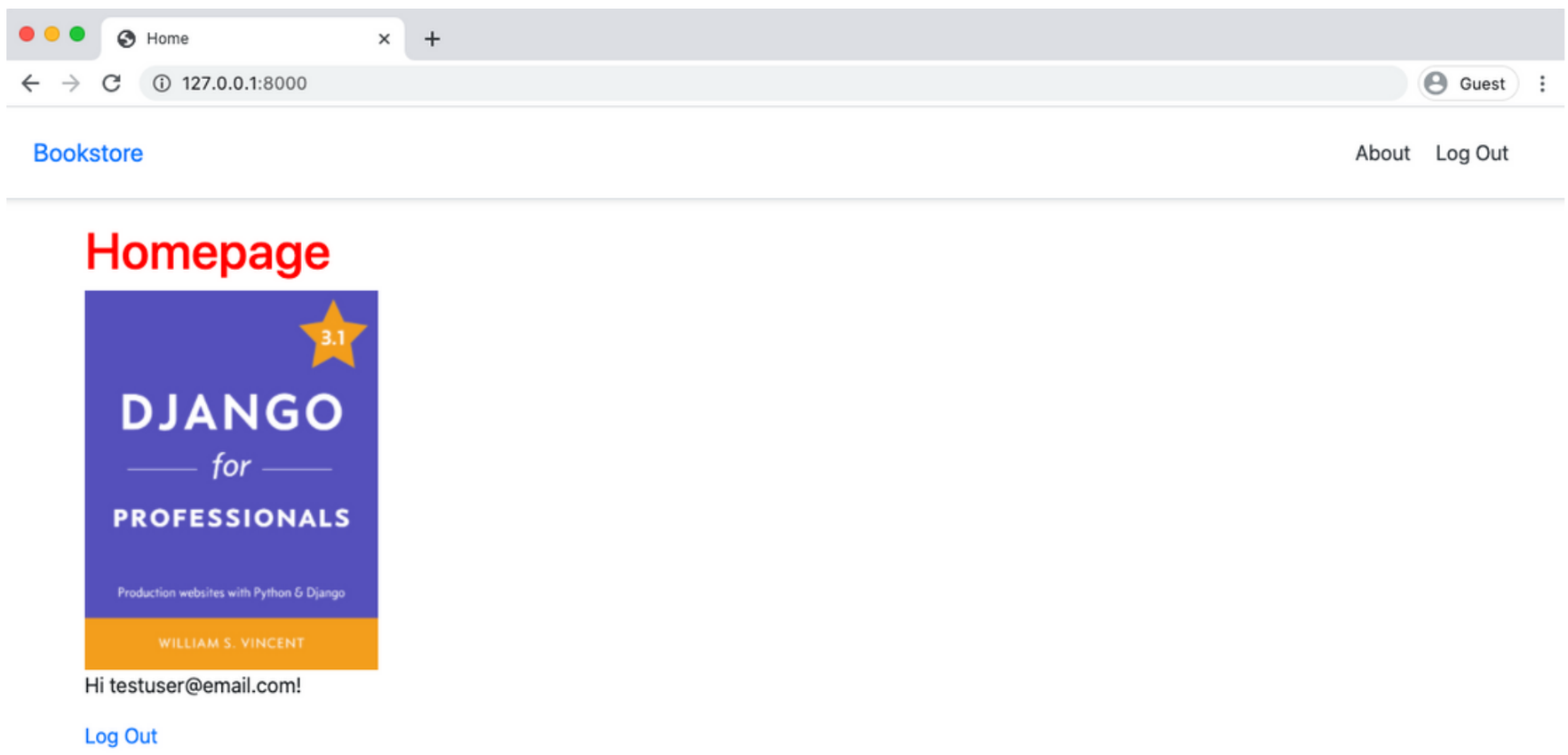
```
<!-- templates/_base.html -->
{% load static %}
<!DOCTYPE html>
<html>

<head>
    <meta charset="utf-8">
    <title>{% block title %}Bookstore{% endblock title %}</title>
    <meta name="viewport" content="width=device-width, initial-scale=1,
shrink-to-fit=no">
    <!-- CSS -->
    <link rel="stylesheet" href="https://stackpath.bootstrapcdn.com/bootstrap/\
4.5.0/css/bootstrap.min.css" integrity="sha384-9aIt2nRpC12Uk9gS9baDl411\
NQApFmC26EwAOH8WgZl5MYYxFfc+NcPb1dKGj7Sk" crossorigin="anonymous">
    <link rel="stylesheet" href="{% static 'css/base.css' %}">
</head>

<body>
    <header>
        <!-- Fixed navbar -->
        <div class="d-flex flex-column flex-md-row align-items-center p-3 px-md-4
mb-3 bg-white border-bottom shadow-sm">
            <a href="{% url 'home' %}" class="navbar-brand my-0 mr-md-auto
font-weight-normal">Bookstore</a>
            <nav class="my-2 my-md-0 mr-md-3">
                <a class="p-2 text-dark" href="#">About</a>
                {% if user.is_authenticated %}
                <a class="p-2 text-dark" href="{% url 'logout' %}">Log Out</a>
                {% else %}
                <a class="p-2 text-dark" href="{% url 'login' %}">Log In</a>
                <a class="btn btn-outline-primary" href="{% url 'signup' %}">Sign Up</a>
                {% endif %}
            </nav>
        </div>
    </header>
    <div class="container">
        {% block content %}
        {% endblock content %}
    </div>
    <!-- JavaScript -->
    <!-- jQuery first, then Popper.js, then Bootstrap JS -->
    <script src="https://code.jquery.com/jquery-3.5.1.slim.min.js" integrity="sha384-
VCmXjywReHh4PwowAiWNagnWcLhlEJLA5buUprzK8rxF\
geH0kww/aWY76TfkUoSX" crossorigin="anonymous"></script>
    <script src="https://cdn.jsdelivr.net/npm/popper.js@1.16.1/dist/umd/\
popper.min.js" integrity="sha384-9/reFTGAW83EW2RDu2S0VKaIzap3H66lZ\
H81PoYlFhbGU+6BZp6G7niu735Sk7lN" crossorigin="anonymous"></script>
    <script src="https://stackpath.bootstrapcdn.com/bootstrap/4.5.1/js/\
bootstrap.min.js" integrity="sha384-1CmrxMRARb6aLqgBO7yyAxTOQE2AKb\
9GfXnEo760AUcUmFx3ibVJJAzGytlQcNXd" crossorigin="anonymous"></script>
</body>

</html>
```

Лучше не пытаться набирать этот код. Вместо этого скопируйте и вставьте его из официального репозитория с одним заметным изменением: в строке **18** исходного кода обязательно измените тег **href** на **# ,** а не на **{% url 'about' %}. .** Другими словами, код должен соответствовать приведенному выше и выглядеть следующим образом:

Code

```
<!-- templates/_base.html -->
<a class="p-2 text-dark" href="#">About</a>
```

В следующем разделе мы добавим **URL-**маршрут страницы о сайте. Если вы обновите главную страницу после внесения этих изменений, она должна выглядеть следующим образом:



Домашняя страница с помощью Bootstrap

Страница о сайте

Вы заметили на панели навигации ссылку на страницу "О сайте"**?** Проблема в том, что ни страницы, ни ссылки пока не существует. Но поскольку у нас уже есть удобное приложение для создания страниц, его можно сделать довольно быстро.

Поскольку это будет статическая страница, нам не нужна модель базы данных. Однако нам понадобятся шаблон, представление и **url.** Начнем с шаблона под названием **about.html .**

Command Line

```
$ touch templates/about.html
```

Страница будет буквально пока просто называться **"About Page",** наследуясь от **_base.html .**

Code

```
<!-- templates/about.html -->
{% extends '_base.html' %}
{% block title %}About{% endblock title %}
{% block content %}
  <h1>About Page</h1>
{% endblock content %}
```

Представление может опираться на встроенный в **Django TemplateView,** как и наша домашняя страница**.**

Code

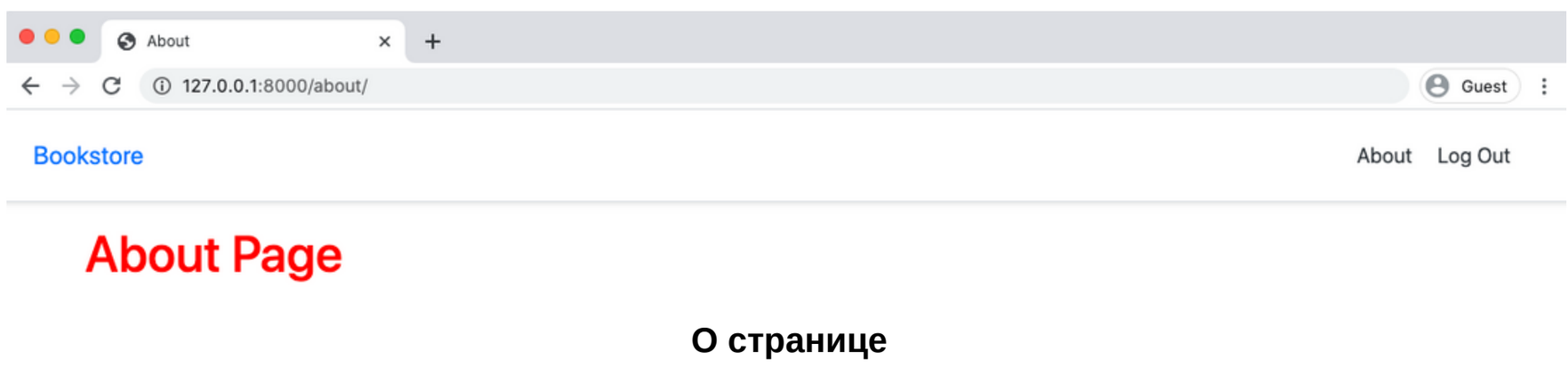
```
# pages/views.py
from django.views.generic import TemplateView

class HomePageView(TemplateView):
        template_name = 'home.html'
class AboutPageView(TemplateView): # new
        template_name = 'about.html'
```

Путь **URL** также будет довольно похожим**:** укажите его в **about/ ,** импортируйте соответствующее представление и укажите имя **URL about .**

Code

```
# pages/urls.py
from django.urls import path
from .views import HomePageView, AboutPageView # new

urlpatterns = [
      path('about/', AboutPageView.as_view(), name='about'), # new
      path('', HomePageView.as_view(), name='home'),
]
```

Если теперь перейти на сайт **http://127.0.0.1:8000/about/,** то можно увидеть страницу "О сайте**".**



О странице

В качестве последнего шага обновите ссылку в навигационной панели на данную страницу**.** Поскольку мы указали имя в **URL-**пути **about,** именно его мы и будем использовать**.**

В строке **18** файла **_base.html** измените строку со ссылкой на страницу **About** на следующую**:**

Code

```
<!-- templates/_base.html -->
<a class="p-2 text-dark" href="{% url 'about' %}">About</a>
```

Django Crispy Forms

Последнее обновление касается наших форм**.** Популярный сторонний пакет **django-crispy-forms** предоставляет множество приятных обновлений**.**

Для его установки мы будем следовать обычной схеме**:** установить в **Docker,** остановить наш контейнер **Docker,** а затем собрать его заново**.**

Command Line

```
$ docker-compose exec web pipenv install django-crispy-forms==1.9.2
$ docker-compose down
$ docker-compose up -d --build
```

Теперь добавьте **crispy forms** в параметр **INSTALLED_APPS.** Обратите внимание**,** что здесь его имя должно быть **crispy_forms.** Приятной дополнительной возможностью является указание **bootstrap4** в **CRISPY_TEMPLATE_PACK,** который предоставит нам предварительно стилизованные формы**.**

Code

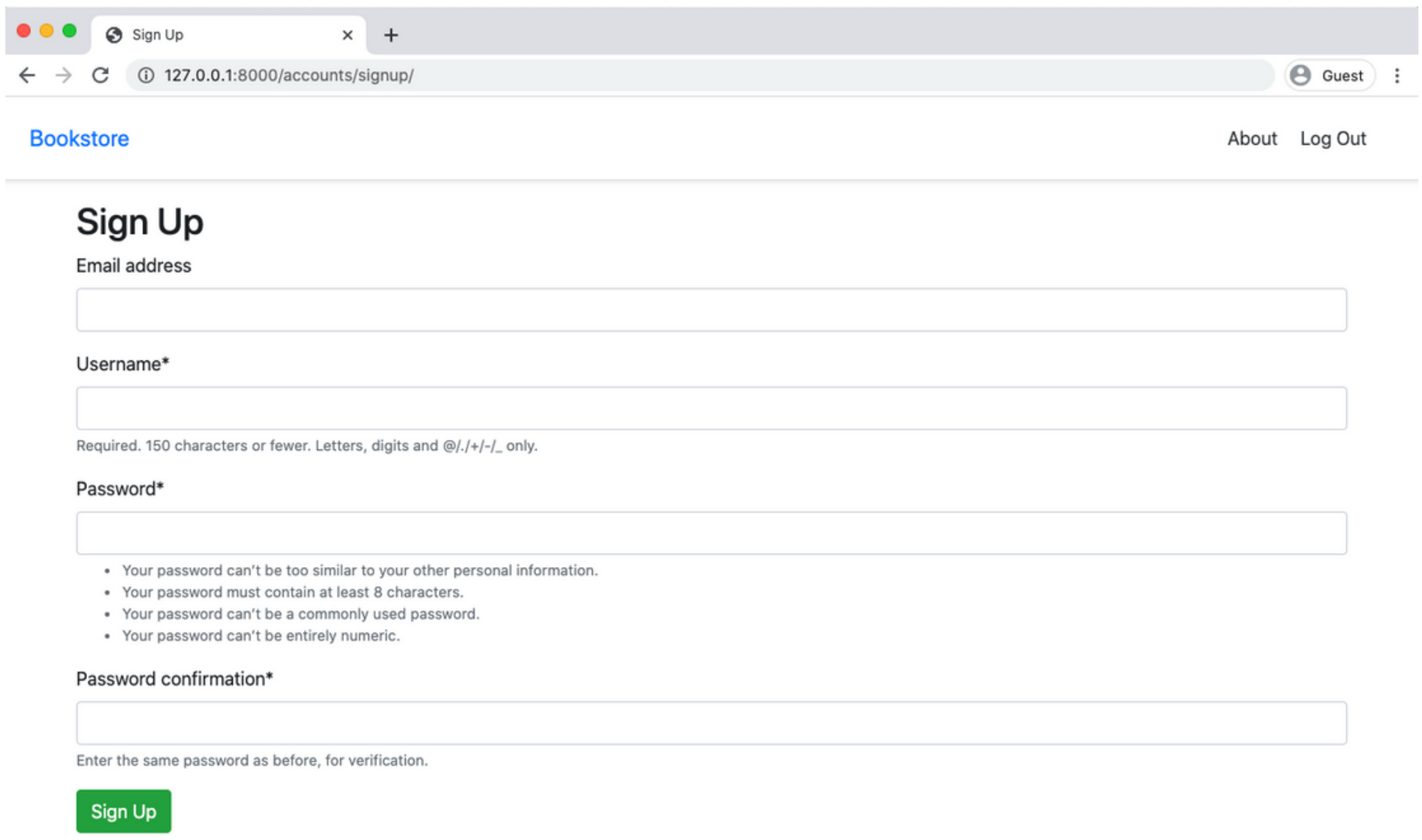
```
# config/settings.py
INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
# Third-party
    'crispy_forms', # new
# Local
    'accounts',
    'pages',
]

# django-crispy-forms
CRISPY_TEMPLATE_PACK = 'bootstrap4' # new
```

Для использования **Crispy Forms** мы загружаем **crispy_forms_tags** в верхней части шаблона и добавляем **{{ form|crispy }}** вместо **{{ form.as_p}}** для отображения полей формы**.** В это время мы также добавим стилизацию **Bootstrap** для кнопки **Submit.**

Начните с файла **signup.html .** Сделайте обновления**,** указанные ниже**.**

Code

```
<!-- templates/registration/signup.html -->
{% extends '_base.html' %}
{% load crispy_forms_tags %}
{% block title %}Sign Up{% endblock title %}
{% block content %}
<h2>Sign Up</h2>
<form method="post">
  {% csrf_token %}
  {{ form|crispy }}
  <button class="btn btn-success" type="submit">Sign Up</button>
</form>
{% endblock content %}
```

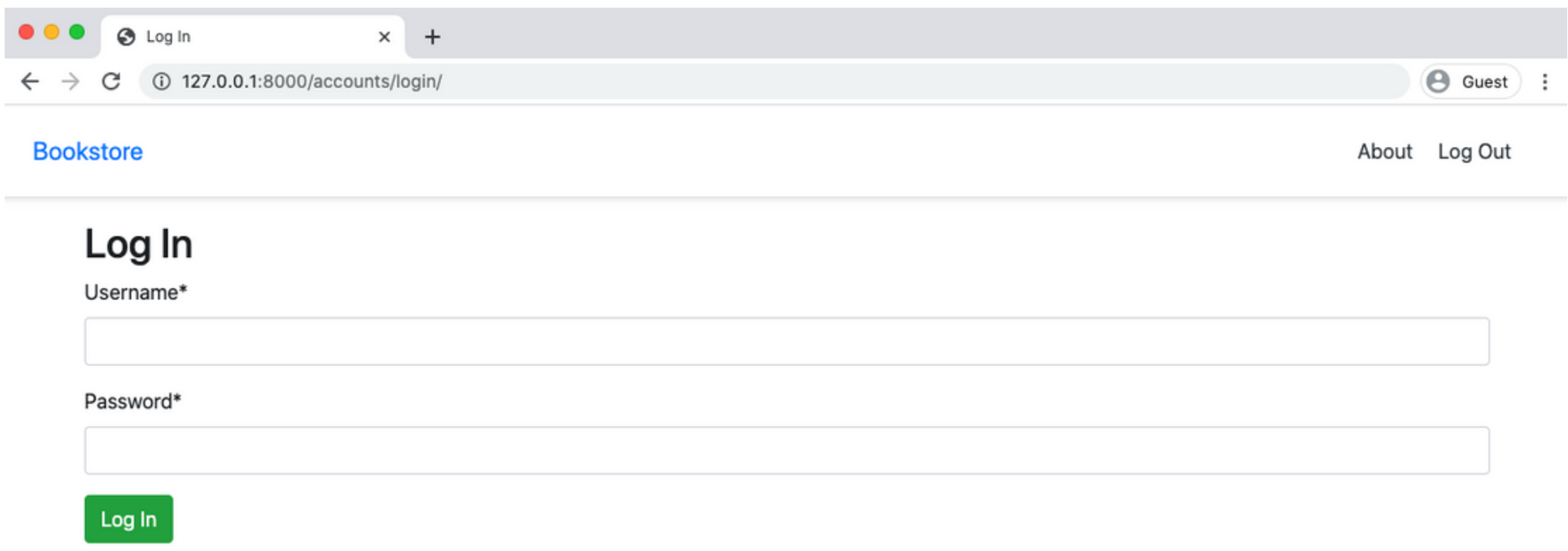


Страница регистрации с Crispy Forms

Обновите также **login.html** с тегами **crispy_forms_tags** в верхней части и **{{ form|crispy }}** в форме.

Code

```
<!-- templates/registration/login.html -->
{% extends '_base.html' %}
{% load crispy_forms_tags %}
{% block title %}Log In{% endblock title %}
{% block content %}
<h2>Log In</h2>
<form method="post">
 {% csrf_token %}
 {{ form|crispy }}
 <button class="btn btn-success" type="submit">Log In</button>
</form>
{% endblock content %}
```



Страница входа в систему с Crispy Forms

Тесты

Время для тестов, которые будут очень похожи на те, которые мы добавили ранее для нашей домашней страницы.

Code

```
# pages/tests.py
from django.test import SimpleTestCase
from django.urls import reverse, resolve
from .views import HomePageView, AboutPageView # new

class HomepageTests(SimpleTestCase):
...
class AboutPageTests(SimpleTestCase): # new
        def setUp(self):
            url = reverse('about')
            self.response = self.client.get(url)
    def test_aboutpage_status_code(self):
        self.assertEqual(self.response.status_code, 200)
    def test_aboutpage_template(self):
        self.assertTemplateUsed(self.response, 'about.html')
    def test_aboutpage_contains_correct_html(self):
        self.assertContains(self.response, 'About Page')
    def test_aboutpage_does_not_contain_incorrect_html(self):
        self.assertNotContains(
            self.response, 'Hi there! I should not be on the page.')
    def test_aboutpage_url_resolves_aboutpageview(self):
        view = resolve('/about/')
        self.assertEqual(
            view.func.__name__,
            AboutPageView.as_view().__name__
        )
```

Запустите тесты.

Command Line

```
$ docker-compose exec web python manage.py test
Creating test database for alias 'default'...
System check identified no issues (0 silenced).
...............
----------------------------------------------------------------------
Ran 15 tests in 0.433s
OK
Destroying test database for alias 'default'...
```

Git

Проверьте статус наших изменений в этой главе, добавьте их все, а затем отправьте сообщение о фиксации.

Command Line

```
$ git status
$ git add .
$ git commit -m 'ch6'
```

Как всегда, вы можете сравнить свой код с официальным кодом на **Github**, если возникнут какие-либо проблемы.

Заключение

Статические файлы являются основной частью каждого веб-сайта, и в **Django** мы должны предпринять ряд дополнительных шагов, чтобы они компилировались и эффективно размещались в продакшене. Позже в книге мы узнаем, как использовать выделенную сеть доставки контента **(CDN)** для размещения и отображения статических файлов нашего проекта.

Глава 7: Усовершенствованная регистрация пользователей

На данный момент у нас реализована стандартная регистрация пользователей **Django.** Но часто это только отправная точка в профессиональных проектах. Как насчет того, чтобы немного настроить все под себя? Например, стандартный шаблон **Django username/email/password** несколько устарел в наши дни. Гораздо чаще для регистрации и входа в систему требуется просто ввести **email/**пароль. И действительно, каждая часть потока аутентификации - формы, электронные письма, страницы - может быть настроена при желании.

Еще одним важным фактором во многих проектах является необходимость социальной аутентификации, то есть обработка регистрации и входа через сторонние сервисы, такие как **Google**, **Facebook** и так далее.

Мы могли бы реализовать собственные решения с нуля, но здесь есть определенные риски: регистрация пользователей - это сложная область со многими движущимися частями и одна из областей, где мы действительно не хотим допустить ошибку в безопасности.

По этой причине многие профессиональные разработчики **Django** полагаются на популярный сторонний пакет **django- allauth**. Добавление любого стороннего пакета должно сопровождаться определенной осторожностью, поскольку вы добавляете еще одну зависимость в свой технический стек. Важно убедиться, что любой пакет является актуальным и хорошо протестированным. К счастью, **django-allauth** является и тем, и другим.

Ценой небольшой магии он решает все эти проблемы и делает настройку намного, намного проще.

django-allauth

Начнем с установки **django-allauth.** Поскольку мы используем **Pipenv,** мы хотим избежать конфликтов с **Pipfile.lock,** поэтому сначала мы установим его в **Docker**, затем остановим **Docker** и перестроим наш образ с флагом **--build,** который предотвращает кэширование образа по умолчанию и гарантирует, что весь наш образ будет собран с нуля.

Command Line

```
$ docker-compose exec web pipenv install django-allauth==0.42.0
$ docker-compose down
$ docker-compose up -d --build
```

Наш сайт будет работать так же, как и раньше, поскольку мы не сообщили **Django** о новом пакете **django-allauth** в явном виде. Для этого нам нужно обновить конфигурацию **INSTALLED_APPS** в нашем файле **settings.py**, добавив встроенный, но необязательный фреймворк сайтов **Django**, а также **allauth** и его функцию учетных записей **allauth.account .**

Фреймворк сайтов **Django -** это мощная функция, которая позволяет одному проекту **Django** управлять несколькими сайтами. Учитывая, что в нашем проекте только один сайт, мы установим **SITE_ID** равным **1.** Если мы добавим второй сайт, его **ID** будет **2,** третий сайт будет иметь **ID 3,** и так далее.

Code

```
# config/settings.py
INSTALLED_APPS = [
        'django.contrib.admin',
        'django.contrib.auth',
        'django.contrib.contenttypes',
        'django.contrib.sessions',
        'django.contrib.messages',
        'django.contrib.staticfiles',
        'django.contrib.sites', # new
# Third-party
        'crispy_forms',
         'allauth', # new
         'allauth.account', # new
# Local
        'accounts',
         'pages',
]
# django-allauth config
SITE_ID = 1 # new
```

AUTHENTICATION_BACKENDS

Файл **settings.py**, создаваемый **Django** для любого нового проекта, содержит ряд явных настроек - те, которые мы уже видим в файле - а также длинный дополнительный список неявных настроек, которые существуют, но не видны. Поначалу это может сбить с толку. Полный список настроек можно найти здесь.

Примером может служить настройка **AUTHENTICATION_BACKENDS**. Под капотом **Django** устанавливает его в **'django.contrib.auth.backends.ModelBackend'**, который используется, когда **Django** пытается аутентифицировать пользователя. Мы можем добавить следующую строку в **settings.py** и текущее поведение останется неизменным:

Code

```
AUTHENTICATION_BACKENDS = (
    'django.contrib.auth.backends.ModelBackend',
)
```

Однако для **django-allauth** нам необходимо добавить его специфические опции аутентификации, что позволит нам в скором времени перейти к использованию входа через электронную почту. Поэтому в нижней части вашего файла **settings.py** добавьте следующую секцию:

Code

```
# config/settings.py
# django-allauth config
SITE_ID = 1
AUTHENTICATION_BACKENDS = (
    'django.contrib.auth.backends.ModelBackend',
    'allauth.account.auth_backends.AuthenticationBackend', # new
)
```

EMAIL_BACKEND

Еще одна неявно заданная конфигурация **- EMAIL_BACKEND.** По умолчанию **Django** будет искать настроенный **SMTP** сервер для отправки писем**. django-allauth** будет отправлять такое письмо при успешной регистрации пользователя**,** что мы можем и будем настраивать позже**,** но поскольку у нас еще нет правильно настроенного **SMTP** сервера**,** это приведет к ошибке**.**

Решение на данный момент заключается в том**,** чтобы **Django** выводил любые письма в консоль командной строки**.** Таким образом**,** мы можем отменить неявную конфигурацию по умолчанию**,** используя **console** вместо **smtp.** Добавьте это в нижней части файла **settings.py.**

Code

```
# config/settings.py
# django-allauth config
SITE_ID = 1
AUTHENTICATION_BACKENDS = (
    'django.contrib.auth.backends.ModelBackend',
    'allauth.account.auth_backends.AuthenticationBackend',
)
EMAIL_BACKEND = 'django.core.mail.backends.console.EmailBackend' # new
```

ACCOUNT_LOGOUT_REDIRECT

Есть еще одно тонкое изменение**,** которое необходимо внести в наши конфигурации**.** Если вы снова посмотрите на страницу конфигураций**,** то увидите**,** что там есть параметр **ACCOUNT_LOGOUT_REDIRECT,** который по умолчанию указывает на путь к домашней странице по адресу **/ .**

В нашем текущем файле **settings.py** есть следующие две строки для перенаправления**,** которые указывают на домашнюю страницу через ее **URL** имя **'home' .**

Code

```
# config/settings.py
LOGIN_REDIRECT_URL = 'home'
LOGOUT_REDIRECT_URL = 'home'
```

Проблема в том**,** что **django-allauth 'ACCOUNT_LOGOUT_REDIRECT** фактически переопределяет встроенный **LOGOUT_REDIRECT_URL ,** однако**,** поскольку они оба указывают на домашнюю страницу**,** это изменение может быть незаметным**.** Чтобы защитить наше приложение на будущее**,** поскольку**,** возможно**,** мы не хотим всегда перенаправлять на домашнюю страницу при выходе из системы**,** мы должны явно указать перенаправление при выходе из системы**.**

Мы также можем перенести две строки перенаправления в раздел **django-allauth config.** Вот как должен выглядеть весь раздел **django-allauth config** на данный момент**.**

Code

```
# config/settings.py
# django-allauth config
LOGIN_REDIRECT_URL = 'home'
ACCOUNT_LOGOUT_REDIRECT = 'home' # new
SITE_ID = 1
AUTHENTICATION_BACKENDS = (
    'django.contrib.auth.backends.ModelBackend',
    'allauth.account.auth_backends.AuthenticationBackend',
)
EMAIL_BACKEND = 'django.core.mail.backends.console.EmailBackend'
```

Учитывая, что мы внесли множество изменений в наш файл **config/settings.py**, давайте теперь запустим **migrate** для обновления нашей базы данных.

Command Line

```
$ docker-compose exec web python manage.py migrate
```

URLs

Нам также нужно поменять **URL** встроенного приложения **auth** на **URL** собственного приложения **allauth** от **django-allauth.** Мы по-прежнему будем использовать тот же путь **URL accounts/**, однако, поскольку мы будем использовать шаблоны и маршруты **django allauth** для регистрации, мы можем удалить путь **URL** для нашего приложения **accounts.**

Code

```
# config/urls.py
from django.contrib import admin
from django.urls import path, include

urlpatterns = [
    # Django admin
    path('admin/', admin.site.urls),
     # User management
    path('accounts/', include('allauth.urls')), # new
    # Local apps
     path('', include('pages.urls')),
]
```

На этом этапе мы можем удалить **accounts/urls.py** и **accounts/views.py**, которые были созданы исключительно для нашей рукописной страницы регистрации и больше не используются.

Templates

Приложение **auth** в **Django** ищет шаблоны в каталоге **templates/registration**, но **allauth** предпочитает, чтобы они находились в каталоге **templates/account.** Поэтому мы создадим новую директорию **templates/account** и скопируем в нее наши существующие шаблоны **login.html** и **signup.html.**

Command Line

```
$ mkdir templates/account
$ mv templates/registration/login.html templates/account/login.html
$ mv templates/registration/signup.html templates/account/signup.html
```

Здесь легко случайно добавить букву **s** к **account,** но не делайте этого**,** иначе вы получите ошибку**.** Правильный адрес директории **- templates/account/**

На этом этапе мы можем удалить каталог **templates/registration,** поскольку он больше не нужен**.**

Command Line

```
$ rm -r templates/registration
```

rm означает удалить**,** а **-r** означает сделать это рекурсивно**,** что необходимо всякий раз**,** когда вы работаете с директорией**.** Если вам нужна дополнительная информация об этой команде**,** вы можете набрать **man rm,** чтобы прочитать руководство**.**

Последний шаг **-** обновление **URL** ссылок в **templates/_base.html** и **templates/home.html** для использования имен **URL django-allauth,** а не **Django.** Мы делаем это**,** добавляя префикс **account_-,** так что **Django 'logout'** теперь будет **'account_logout', 'login'** будет **'account_login',** а **signup** будет **account_signup.**

Code

```
<!-- templates/_base.html -->
...
<nav class="my-2 my-md-0 mr-md-3">
    <a class="p-2 text-dark" href="{% url 'about' %}">About</a>
    {% if user.is_authenticated %}
    <a class="p-2 text-dark" href="{% url 'account_logout' %}">Log Out</a>
    {% else %}
    <a class="p-2 text-dark" href="{% url 'account_login' %}">Log In</a>
    <a class="btn btn-outline-primary" href="{% url 'account_signup' %}">Sign
Up</a>
    {% endif %}
</nav>
...
```

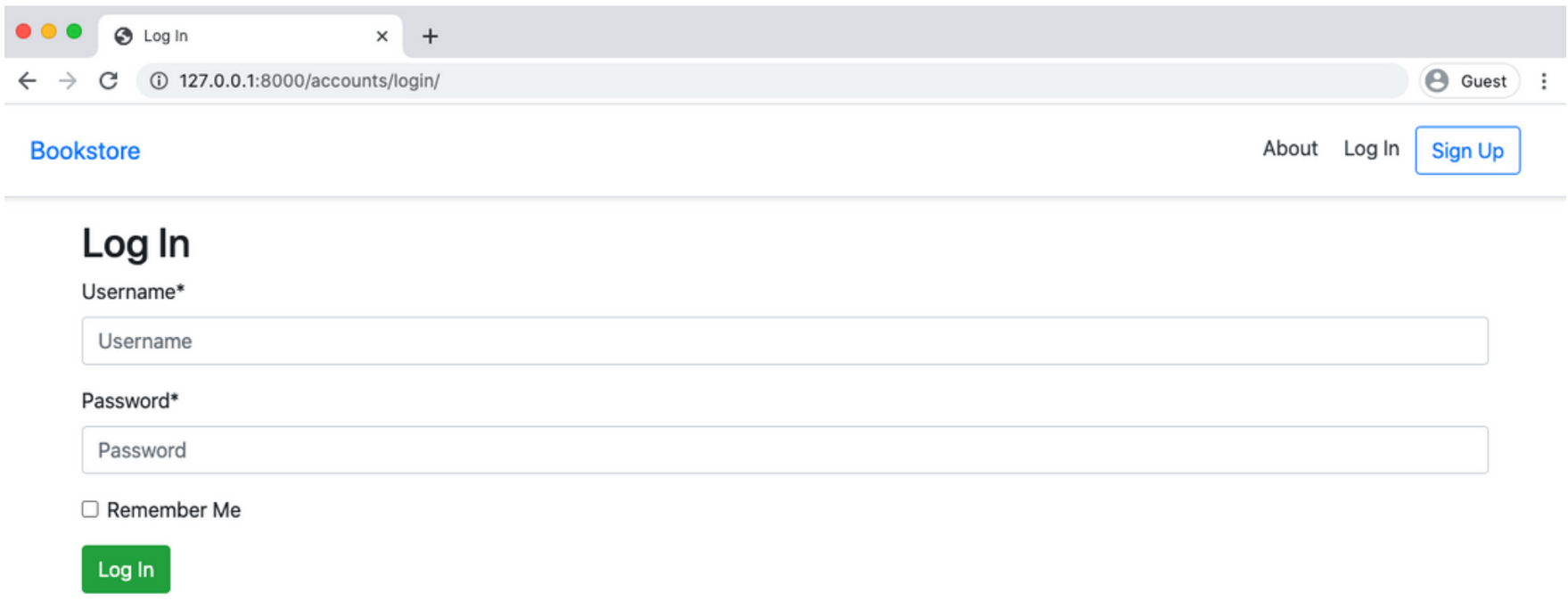
Code

```
<!-- templates/home.html -->
{% extends '_base.html' %}
{% load static %}
{% block title %}Home{% endblock title %}
{% block content %}
<h1>Homepage</h1>
<img class="bookcover" src="{% static 'images/dfp.png' %}">
{% if user.is_authenticated %}
<p>Hi {{ user.email }}!</p>
<p><a href="{% url 'account_logout' %}">Log Out</a></p>
{% else %}
<p>You are not logged in</p>
<p><a href="{% url 'account_login' %}">Log In</a> |
   <a href="{% url 'account_signup' %}">Sign Up</a>
</p>
{% endif %}
{% endblock content %}
```

И мы закончили**!**

Log In(вход в систему)

Обновите главную страницу сайта **http://127.0.0.1:8000 ,** выйдите из системы**,** если вы уже вошли в нее**,** затем нажмите на ссылку "Войти"**.** Страница "Войти" теперь обновлена**.**



Страница входа в систему

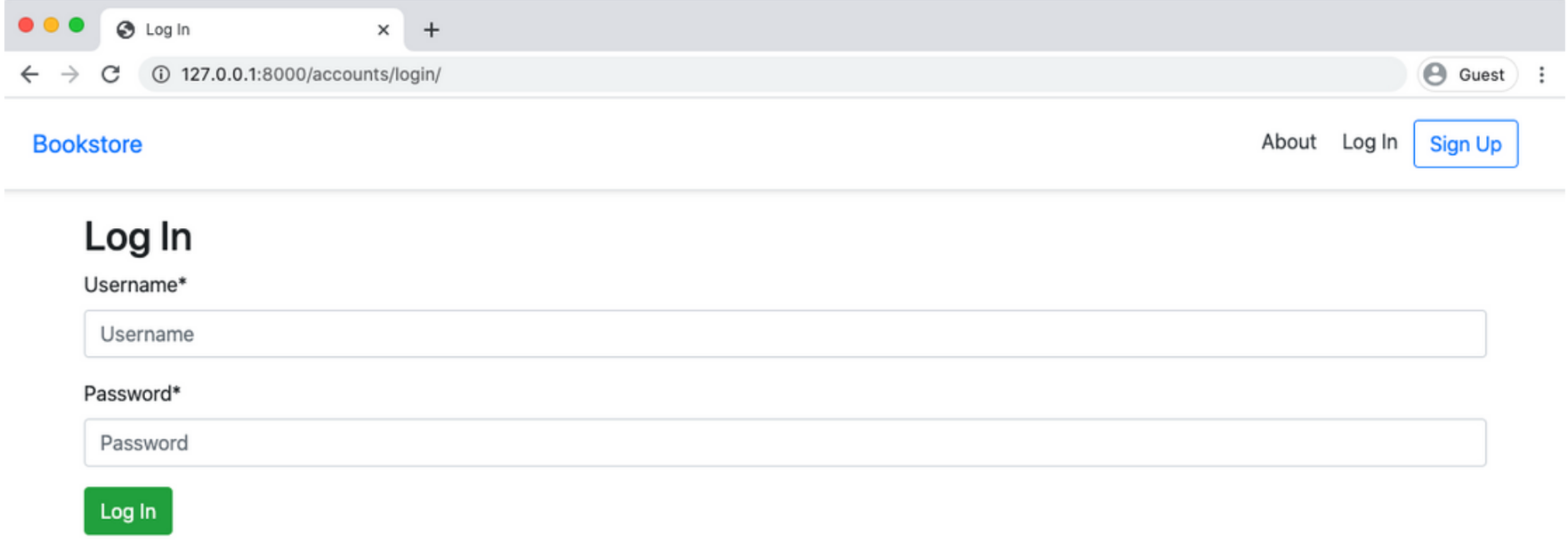
Обратите внимание на новую опцию "Запомнить меня"**.** Это первая из многих конфигураций**,** которые предоставляет **django-allauth.** По умолчанию **None** спрашивает пользователя**,** хочет ли он запомнить свою сессию**,** чтобы не входить в систему снова**.** Можно также установить значение **False,** чтобы не запоминать**,** или **True,** чтобы всегда помнить**.** Мы выберем **True -** именно так будет работать традиционная страница входа в **Django.**

В разделе **# django-allauth config** файла **config/settings.py** добавьте новую строку для этого**.**

Code

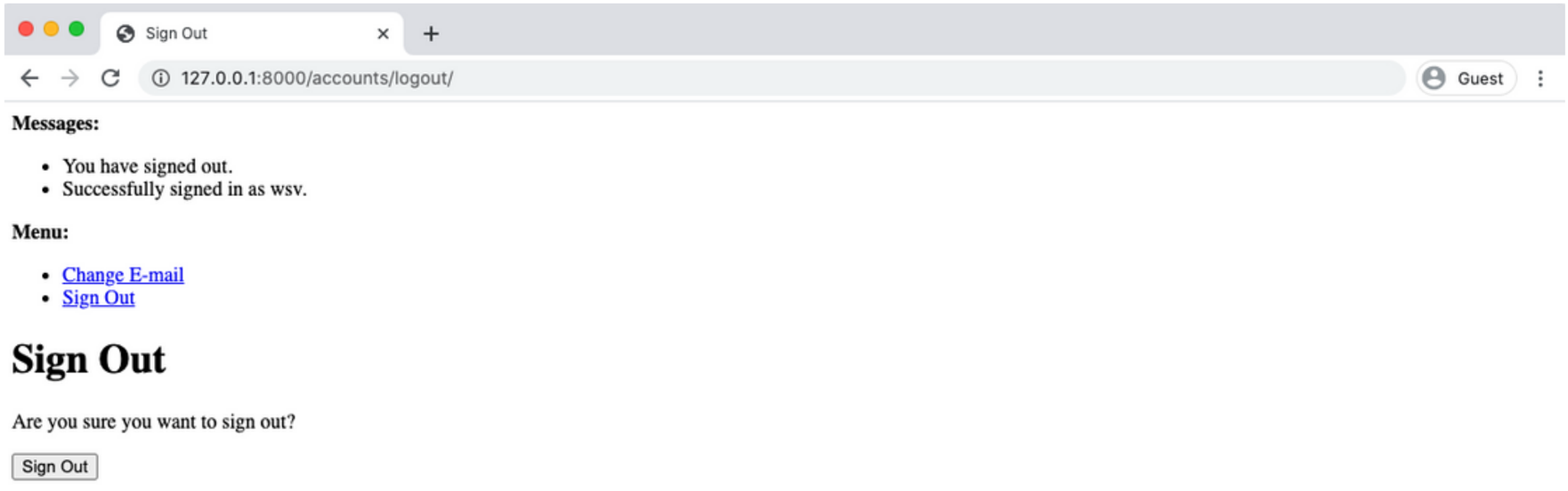
```
# config/settings.py
# django-allauth config
...
ACCOUNT_SESSION_REMEMBER = True # new
```

Обновите страницу **" Log In",** и поле исчезнет**!**



Страница входа в систему Нет блока

Если вы попробуете войти в форму входа с учетной записью суперпользователя**,** она перенаправит вас обратно на главную страницу с приветственным сообщением**.** Нажмите на ссылку "Выйти"**.**



Страница выхода из системы

Вместо прямого выхода из системы**, django-allauth** имеет промежуточную страницу **"Log Out",** которую мы можем настроить в соответствии с остальной частью нашего проекта**.**

Log Out (Выход из системы)

Обновите стандартный шаблон **Log Out,** создав файл **templates/account/logout.html** для его переопределения**.**

Command Line

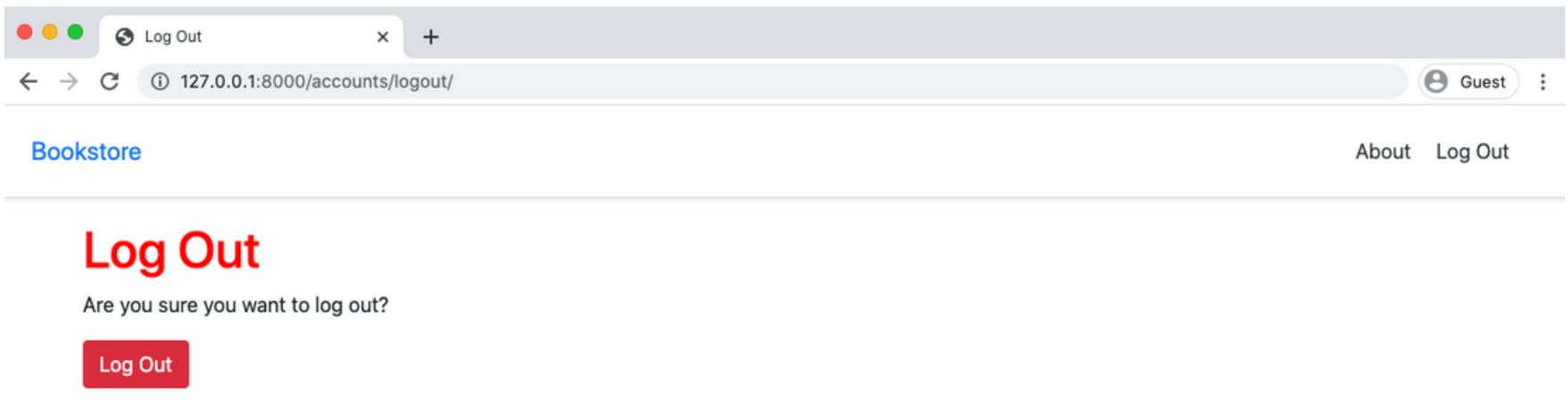
```
$ touch templates/account/logout.html
```

Как и другие наши шаблоны**,** он будет расширять **_base.html** и включать стилизацию **Bootstrap** для кнопки отправки**.**

Code

```
<!-- templates/account/logout.html -->
{% extends '_base.html' %}
{% load crispy_forms_tags %}
{% block title %}Log Out{% endblock %}
{% block content %}
<h1>Log Out</h1>
<p>Are you sure you want to log out?</p>
<form method="post" action="{% url 'account_logout' %}">
  {% csrf_token %}
  {{ form|crispy }}
  <button class="btn btn-danger" type="submit">Log Out</button>
</form>
{% endblock content %}
```

Обновите страницу.

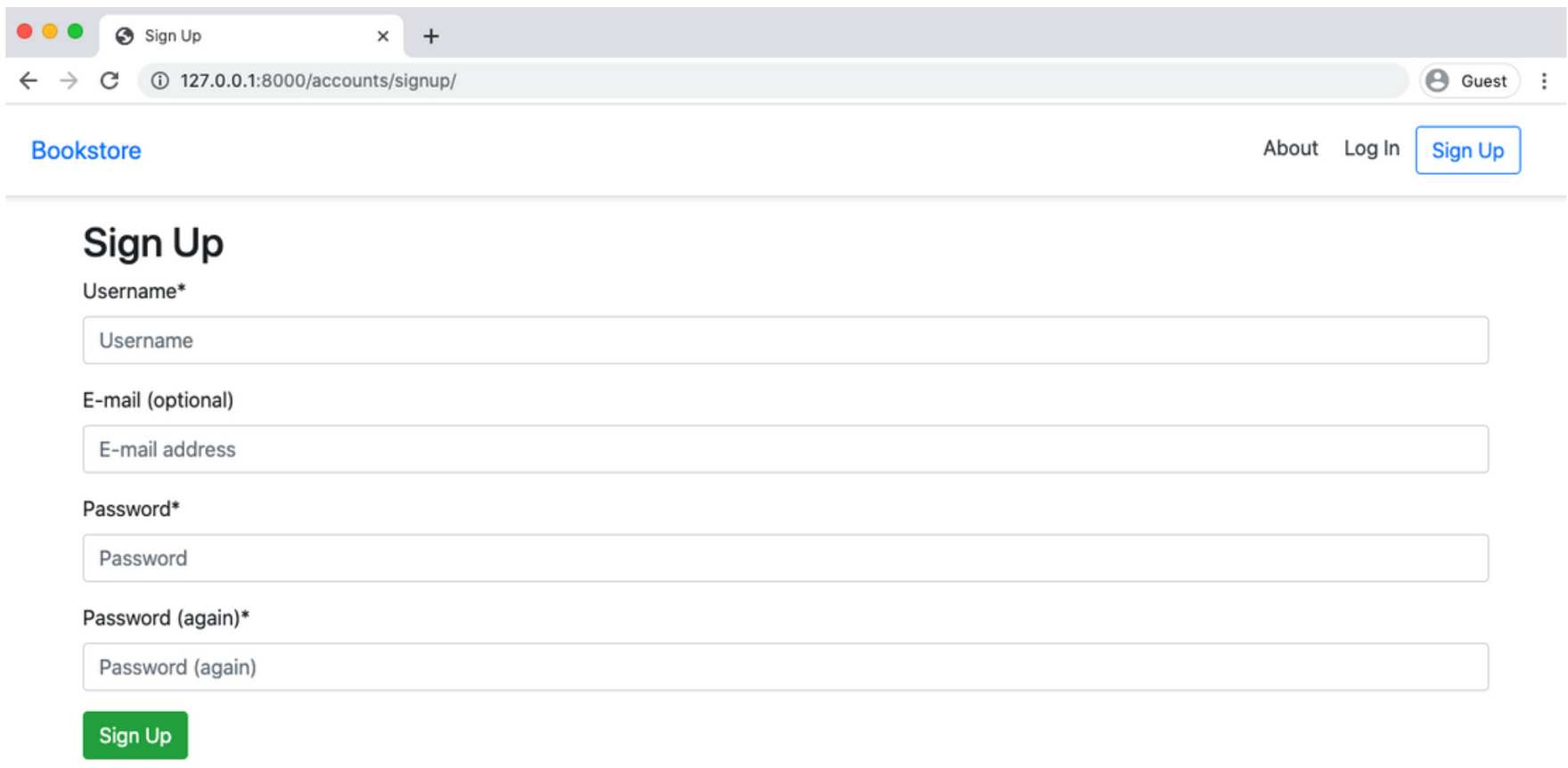


Собственная страница выхода из системы

Затем нажмите на ссылку "Выход", чтобы завершить процесс.

Регистрация

В верхней части нашего сайта, в навигационной панели, нажмите на ссылку "Регистрация", которая оформлена в стиле **Bootstrap** и **django-crispy-forms.**



Страница регистрации

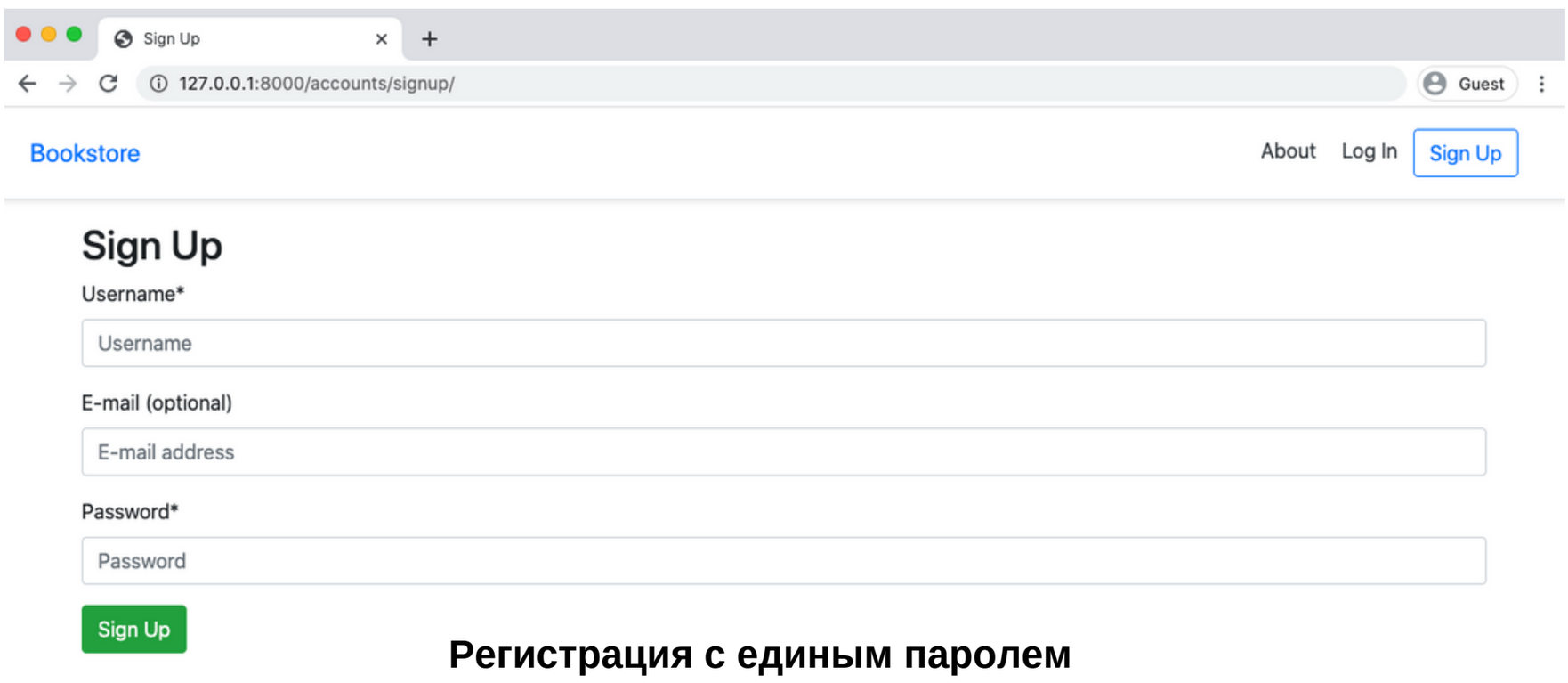
Дополнительная настройка, которую мы можем сделать с помощью **django-allauth**, заключается в том, чтобы запрашивать пароль только один раз. Поскольку опции смены и сброса пароля мы настроим позже, меньше риск того, что пользователь, неправильно набравший пароль, будет заблокирован.

Если вы посмотрите на параметры конфигурации **django-allauth**, это изменение можно выполнить одним движением.

Code

```
# config/settings.py
# django-allauth config
...
ACCOUNT_SIGNUP_PASSWORD_ENTER_TWICE = False # new
```

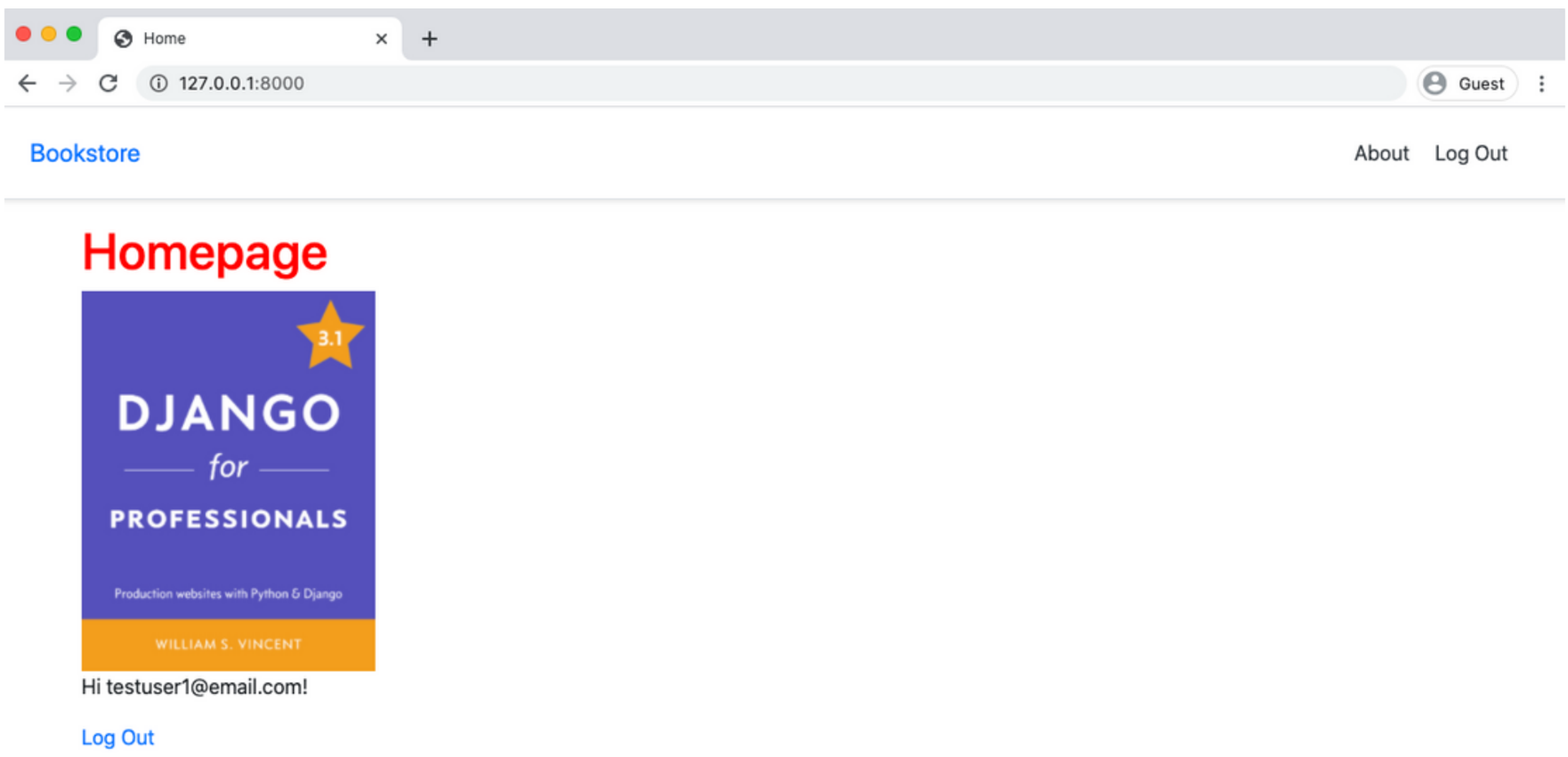
Обновите страницу, и форма обновится, удалив дополнительную строку пароля.



Регистрация с единым паролем

Теперь создайте нового пользователя, чтобы убедиться, что все работает. Мы можем назвать пользователя **testuser1**, использовать **testuser1@email.com** в качестве электронной почты и **testpass123** в качестве пароля.

После отправки он перенаправит вас на домашнюю страницу.



Домашняя страница testuser

Помните, как мы настроили вывод **email** в консоль**? django-allauth** автоматически отправляет **email** при регистрации, который мы можем просмотреть, набрав **docker-compose logs .**

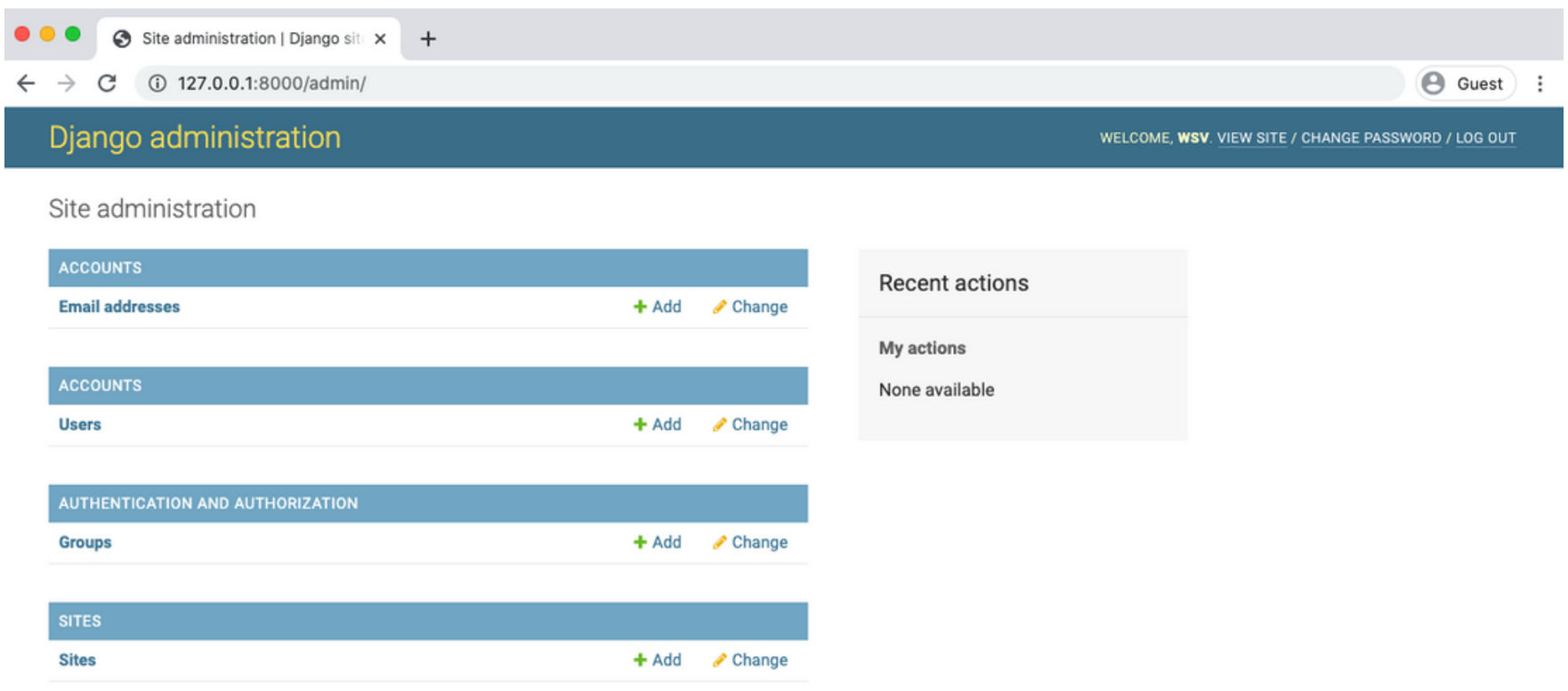
Command Line

```
$ docker-compose logs
...
web_1 | Content-Type: text/plain; charset="utf-8"
web_1 | MIME-Version: 1.0
web_1 | Content-Transfer-Encoding: 7bit
web_1 | Subject: [example.com] Please Confirm Your E-mail Address
web_1 | From: webmaster@localhost
web_1 | To: testuser@email.com
web_1 | Date: Mon, 03 Aug 2020 14:04:15 -0000
web_1 | Message-ID: <155266195771.15.17095643701553564393@cdab877c4af3>
web_1 |
web_1 | Hello from example.com!
web_1 |
web_1 | You're receiving this e-mail because user testuser1 has given yours as
an e-mail address to connect their account.
web_1 |
web_1 | To confirm this is correct, go to http://127.0.0.1:8000/accounts/
confirm-emailMQ:1h4oIn:GYETeK5dRClGjcgA8NbuOoyvafA/
web_1 |
web_1 | Thank you from example.com!
web_1 | example.com
web_1 | --------------------------------------------------------------------
...
```

Вот оно. Позже мы настроим это сообщение и сконфигурируем соответствующую почтовую службу для отправки его реальным пользователям.

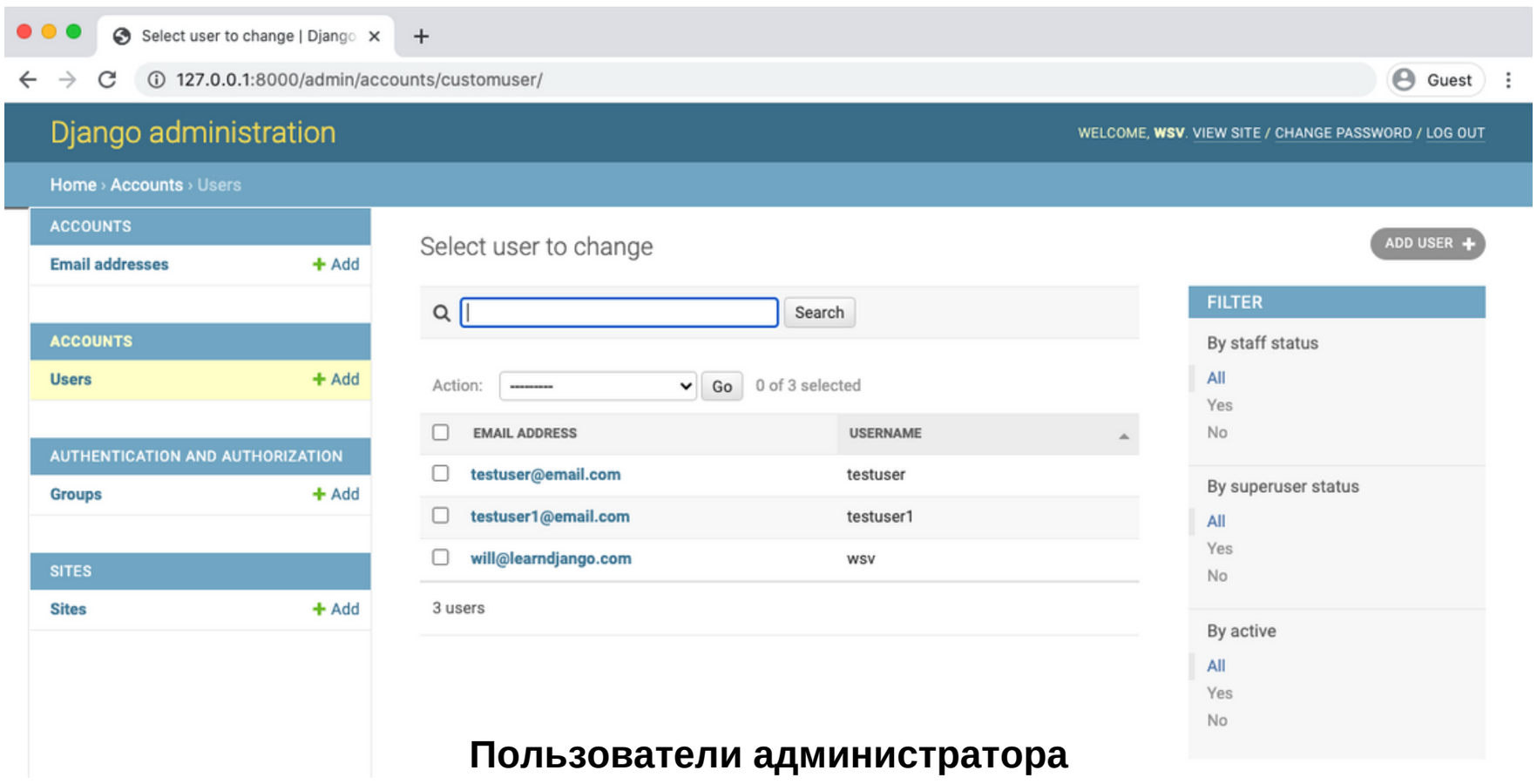
Admin

Зайдите в админку под учетной записью суперпользователя по адресу **http://127.0.0.1:8000/admin/**, и мы увидим, что она тоже изменилась теперь, когда задействован **django-allauth**.



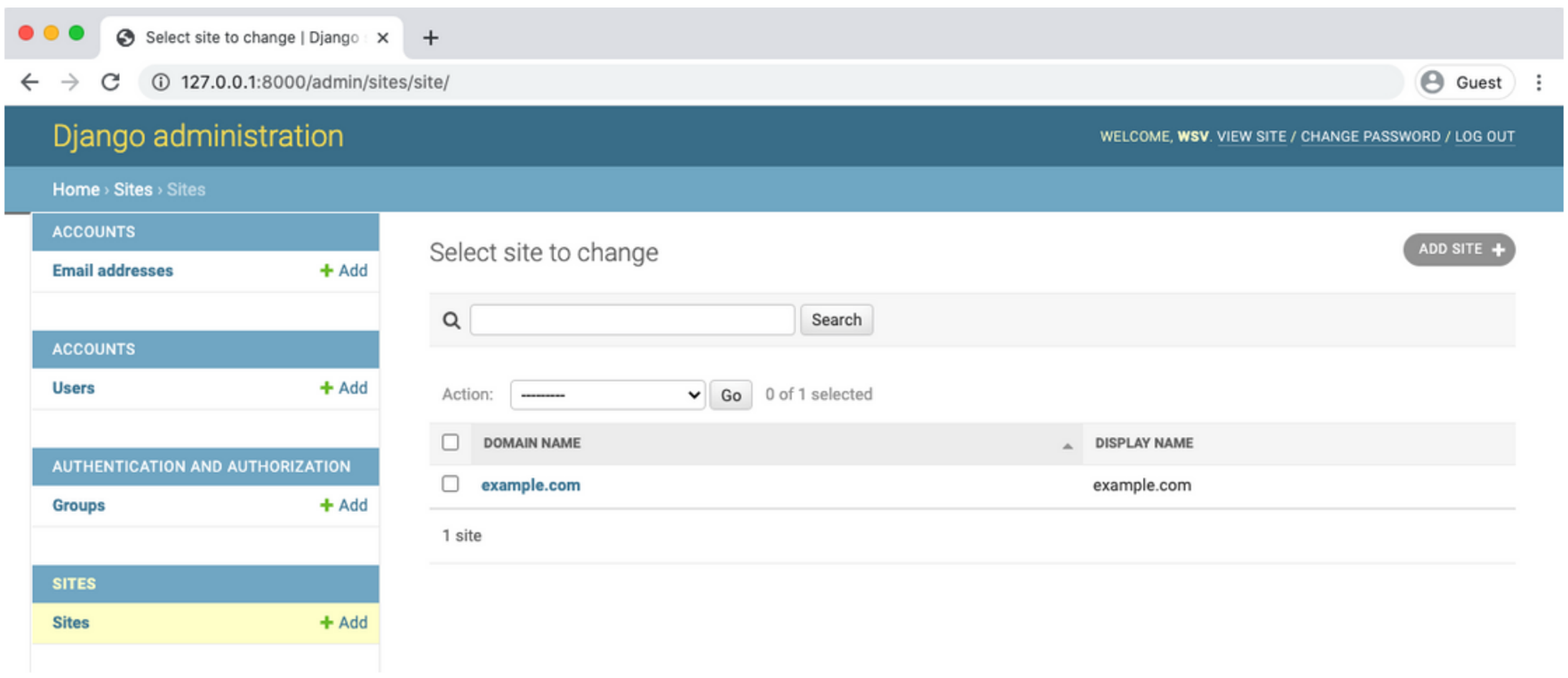
Домашняя страница администратора

Есть два новых раздела: Учетные записи и Сайты, появившиеся благодаря нашей недавней работе. Если вы нажмете на раздел "Пользователи", то увидите традиционный вид, в котором представлены три текущие учетные записи пользователей.



Пользователи администратора

Новое в **Django 3.1 -** это боковая панель слева**,** которая означает**,** что мы можем перейти непосредственно к разделу Сайты**,** чтобы увидеть**,** что предоставляет фреймворк **Django sites.** Мы обновим и Доменное имя**,** и Отображаемое имя в более поздней главе о настройке электронной почты**.**



Admin Sites

Вход только по электронной почте

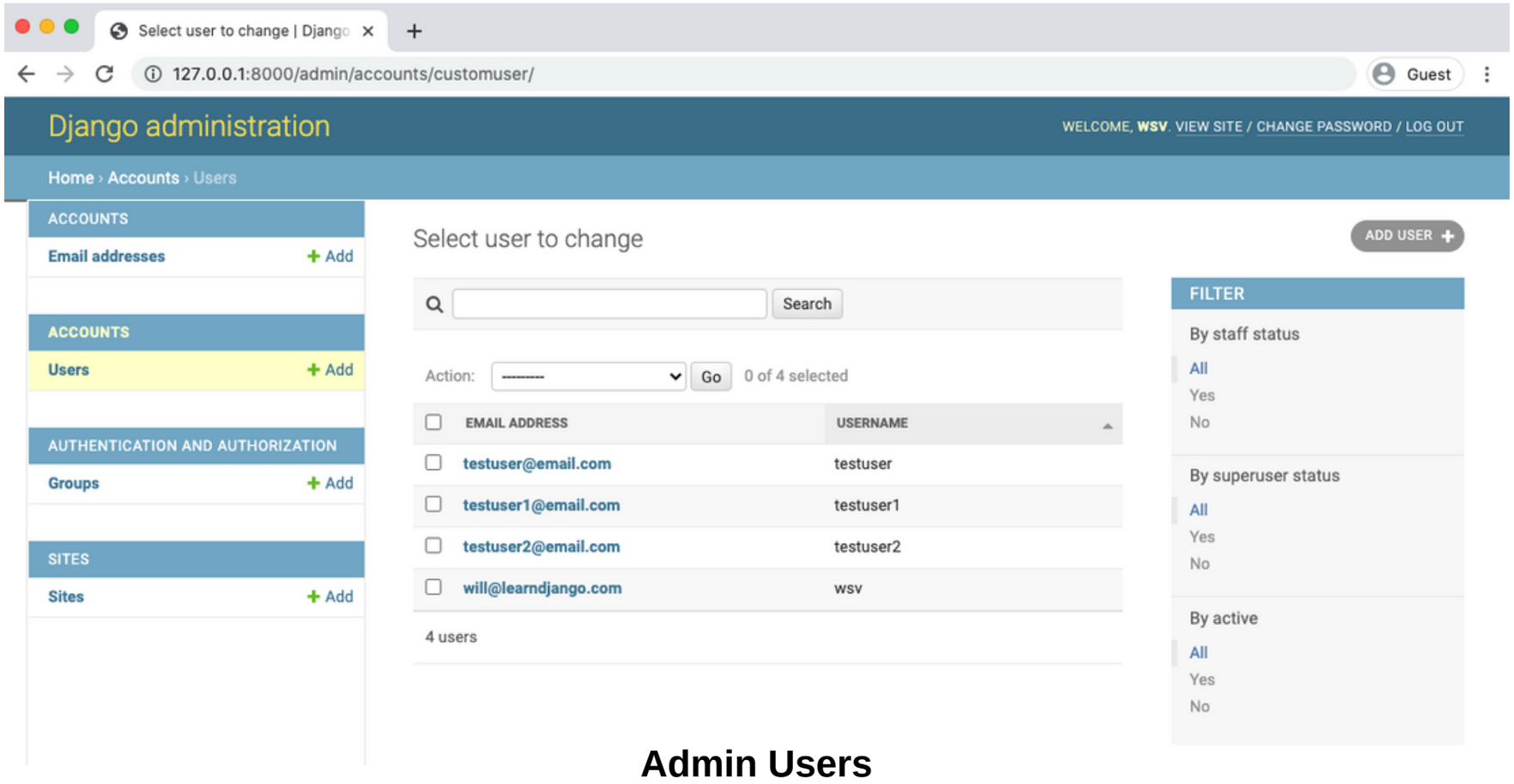
Пришло время по**-**настоящему использовать обширный список конфигураций **django-allauth,** перейдя на использование только **email** для входа**,** а не **username.** Это требует нескольких изменений**.** Сначала мы сделаем имя пользователя необязательным**,** а вместо него установим обязательным **email.** Затем мы потребуем**,** чтобы **email** был уникальным и выбранным методом аутентификации**.**

Code

```
# config/settings.py
# django-allauth config
...
ACCOUNT_USERNAME_REQUIRED = False # new
ACCOUNT_AUTHENTICATION_METHOD = 'email' # new
ACCOUNT_EMAIL_REQUIRED = True # new
ACCOUNT_UNIQUE_EMAIL = True # new
```

Вернитесь на главную страницу и нажмите на **"Log Out",** поскольку вы вошли в систему под учетной записью суперпользователя**.** Затем нажмите на ссылку **"Sign Up"** на панели навигации и создайте учетную запись для **testuser2@email.com** с **testpass123** в качестве пароля**.**

После успешного перенаправления на главную страницу перейдите в админку**,** чтобы проверить**,** что произошло на самом деле**.** Войдите в систему под учетной записью суперпользователя и перейдите в раздел "Пользователи"**.**



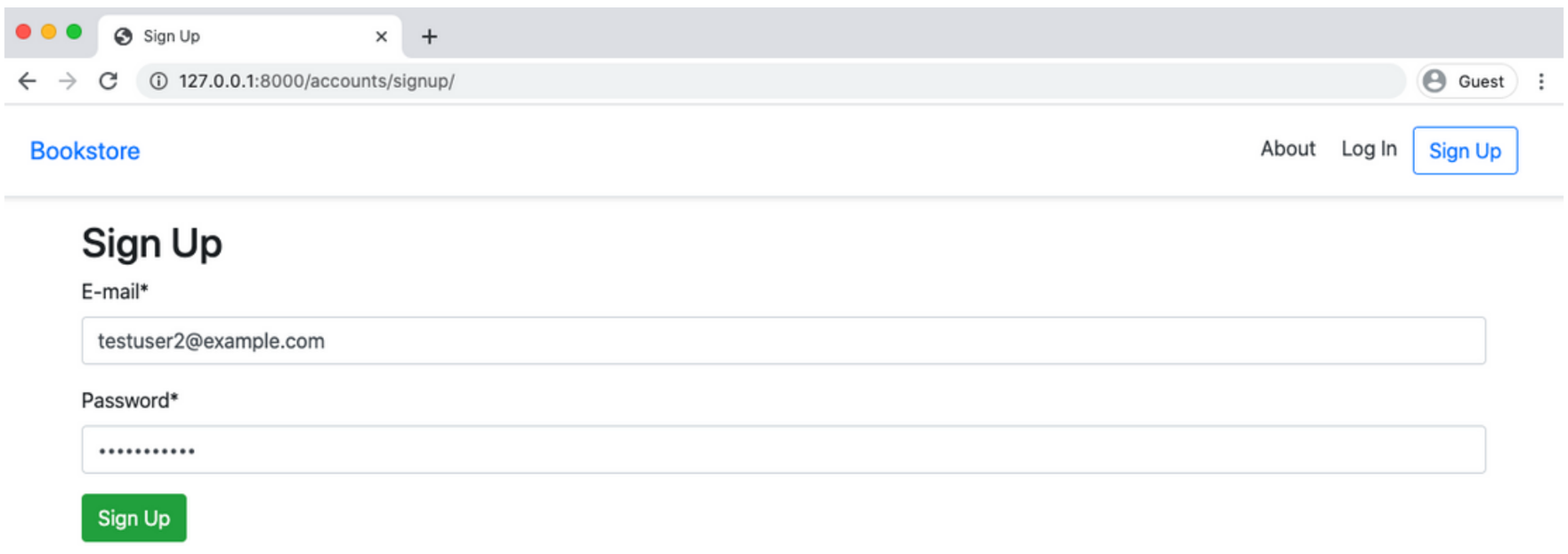
Admin Users

Мы видим**,** что **django-allauth** автоматически заполнил для нас поле **username,** основываясь на части **email** перед **@ .** Это происходит потому**,** что наша базовая модель **CustomUser** все еще имеет поле **username.** Мы не удалили его**.**

Хотя такой подход может показаться немного хакерским**,** на самом деле он работает просто отлично**.** Полное удаление имени пользователя из пользовательской модели пользователя требует использования **AbstractBaseUser,** что является дополнительным**,** необязательным шагом**,** на который идут некоторые разработчики**.** Это требует гораздо больше кода и понимания**,** поэтому не рекомендуется**,** если вы не знаете толк в системе аутентификации **Django!**

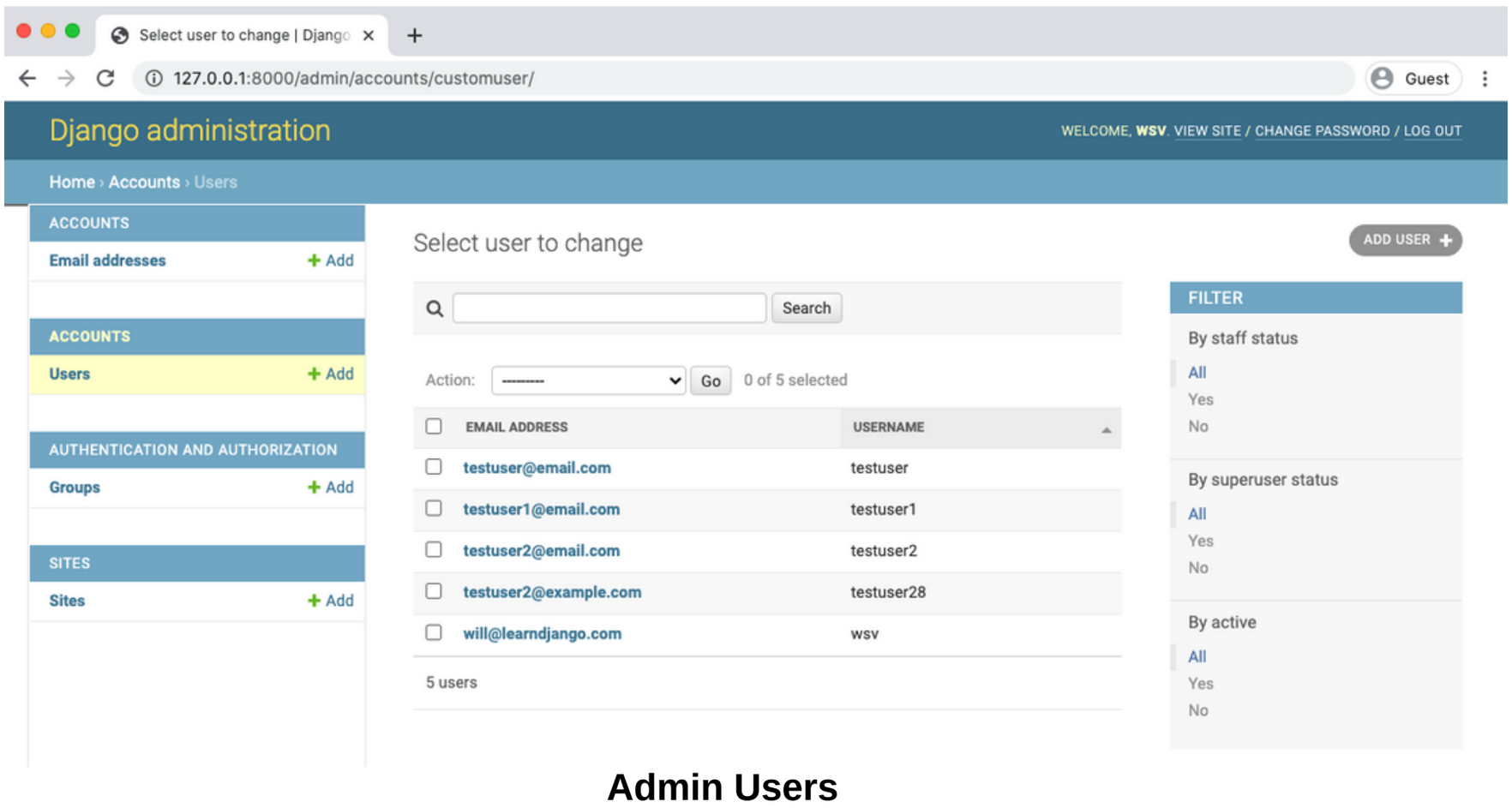
Однако**,** здесь есть крайний случай**,** который мы должны подтвердить**:** что произойдет**,** если у нас есть **testuser2@email.com,** а затем мы зарегистрируемся на **testuser2@example.com?** Не приведет ли это к имени пользователя **testuser2** для обоих**,** что вызовет конфликт**?**

Давайте попробуем**!** Выйдите из админки**,** на странице регистрации создайте аккаунт для **testuser2@example.com .**



Sign Up Form(Форма для регистации

Теперь снова войдите в админку и перейдите в раздел Пользователи.



Admin Users

django-allauth автоматически добавляет двухзначную строку к имени пользователя. В данном случае это **28**, поэтому **testuser2** становится **testuser28 .** Эта двузначная строка будет сгенерирована случайным образом.

Тесты

Пришло время для тестов. Как любой хороший сторонний пакет **django-allauth** поставляется со своими собственными тестами, поэтому нам не нужно заново тестировать его основную функциональность, просто подтвердите, что наш проект работает так, как ожидается.

Если вы запустите наш текущий набор тестов, то обнаружите **3** ошибки, связанные с **SignupPageTests**, поскольку мы используем **django-allauth** для этого, а не наши собственные представления, формы и **url.**

Command Line

```
$ docker-compose exec web python manage.py test
...
Ran 15 tests in 0.363s
FAILED (errors=3)
```

Давайте обновим тесты. Первая проблема заключается в том, что **signup** больше не является правильным именем **URL,** вместо этого мы используем **account_signup,** которое предоставляет **django-allauth.** Как я узнал об этом**?** Я посмотрел на исходный код и нашел имя **URL.**

Шаблон **signup.html** также теперь расположен по адресу **account/signup.html .** И мы больше не используем **CustomUserCreationForm,** поэтому мы можем удалить этот тест**.** Удалите также импорты для **CustomUserCreationForm** и **SignupPageView** в верхней части файла**.**

Code

```
# accounts/tests.py
from django.contrib.auth import get_user_model
from django.test import TestCase
from django.urls import reverse, resolve

class CustomUserTests(TestCase):
...
class SignupTests(TestCase): # new
    username = 'newuser'
    email = 'newuser@email.com'
def setUp(self):
    url = reverse('account_signup')
    self.response = self.client.get(url)
def test_signup_template(self):
    self.assertEqual(self.response.status_code, 200)
    self.assertTemplateUsed(self.response, 'account/signup.html')
    self.assertContains(self.response, 'Sign Up')
    self.assertNotContains(
        self.response, 'Hi there! I should not be on the page.')
def test_signup_form(self):
    new_user = get_user_model().objects.create_user(
        self.username, self.email)
    self.assertEqual(get_user_model().objects.all().count(), 1)
    self.assertEqual(get_user_model().objects.all()
        [0].username, self.username)
    self.assertEqual(get_user_model().objects.all()
        [0].email, self.email)
```

Запустите тесты снова.

Command Line

```
$ docker-compose exec web python manage.py test
Creating test database for alias 'default'...
System check identified no issues (0 silenced).
..............
----------------------------------------------------------------------
Ran 14 tests in 0.410s
OK
Destroying test database for alias 'default'...
```

Social

Если вы хотите добавить социальную аутентификацию, это всего лишь несколько настроек. У меня есть полное руководство по интеграции **Github**. Процесс аналогичен для **Google**, **Facebook** и всех остальных провайдеров, поддерживаемых **django-allauth**. Вот полный список провайдеров.

Git

Как обычно, зафиксируйте изменения в коде с помощью **Git.**

Command Line

```
$ git status
$ git add .
$ git commit -m 'ch7'
```

А если возникнут проблемы, сравните с официальным исходным кодом на **Github.**

Заключение

Теперь у нас есть работающий процесс регистрации пользователей, который при необходимости можно быстро расширить до социальной аутентификации. В следующей главе мы добавим переменные окружения в наш проект для большей безопасности и гибкости.

Глава 8: Environment Variables(Переменные окружения)

Переменные окружения - это переменные, которые могут быть загружены в операционную среду проекта во время выполнения, в отличие от жесткого кода в самой базе кода. Они считаются неотъемлемой частью популярной методологии **Twelve-Factor App Design** и лучшей практикой **Django**, поскольку позволяют повысить уровень безопасности и упростить локальную/ производственную конфигурацию.

Почему больший уровень безопасности? Потому что мы можем хранить действительно секретную информацию - учетные данные базы данных, ключи **API** и так далее - отдельно от фактической базы кода. Это хорошая идея, потому что использование системы контроля версий, например **git**, означает, что достаточно одного неудачного коммита, чтобы учетные данные были добавлены туда навсегда. Это означает, что любой, кто имеет доступ к базе кода, имеет полный контроль над проектом. Это очень, очень опасно. Гораздо лучше ограничить круг лиц, имеющих доступ к приложению, и переменные окружения предоставляют элегантный способ сделать это.

Второе преимущество заключается в том, что переменные окружения значительно облегчают переключение между локальной и рабочей средами кода. Как мы увидим, существует ряд конфигураций настроек, которые **Django** использует по умолчанию, чтобы облегчить локальную разработку, но которые должны быть изменены, когда тот же проект готов к производству.

]

environs[django]

Существует множество различных способов работы с переменными окружения в **Python**, но для этого проекта мы будем использовать пакет **environs**, который имеет специфическую для **Django** опцию, устанавливающую ряд дополнительных пакетов, помогающих в настройке. В командной строке установите **environs[django]** . Обратите внимание, что вам, вероятно, потребуется добавить одинарные кавычки '' вокруг пакета, если вы используете **Zsh** в качестве терминальной оболочки, поэтому выполните **pipenv install 'environs[django]==8.0.0'** . Нам также нужно будет развернуть наш контейнер **Docker** и пересобрать его с новым пакетом.

Command Line

```
$ docker-compose exec web pipenv install 'environs[django]==8.0.0'
$ docker-compose down
$ docker-compose up -d --build
```

В файле **config/settings.py** есть три строки импорта, которые нужно добавить в верхней части файла, сразу под импортом **Path** .

Code

```
# config/settings.py
from pathlib import Path
from environs import Env # new
env = Env() # new
env.read_env() # new
```

Все готово.

SECRET_KEY

Для нашей первой переменной окружения мы зададим **SECRET_KEY**, случайно сгенерированную строку, используемую для криптографической подписи и создаваемую каждый раз, когда выполняется команда **startproject.** Очень важно, чтобы **SECRET_KEY** действительно хранился в секрете.

В моем файле **config/settings.py** он имеет следующее значение:

Code

```
# config/settings.py
SECRET_KEY = ')*_s#exg*#w+#-xt=vu8b010%%a&p@4edwyj0=(nqq90b9a8*n'
```

Обратите внимание на одинарные кавычки **('')** вокруг **SECRET_KEY,** которые делают его строкой **Python.**

На самом деле они не являются частью самого значения **SECRET_KEY,** что является простой ошибкой.

Переход на переменные окружения состоит из двух этапов:

- добавить переменную окружения в файл **docker-compose.yml**
- обновить **config/settings.py,** чтобы указать на переменную.

В файле **docker-compose.yml** добавьте секцию **environment** под веб-сервисом. Это будет переменная, которую мы назовем **DJANGO_SECRET_KEY** со значением нашего существующего **SECRET_KEY .** Вот как выглядит обновленный файл:

docker-compose.yml

```
# config/settings.py
version: '3.8'
services:
    web:
        build: .
        command: python /code/manage.py runserver 0.0.0.0:8000
        volumes:
            - .:/code
        ports:
            - 8000:8000
        depends_on:
          - db
        environment:
          - "DJANGO_SECRET_KEY=)*_s#exg*#w+#-xt=vu8b010%%a&p@4edwyj0=(nqq90b9a8*n"
    db:
        image: postgres:11
        volumes:
            - postgres_data:/var/lib/postgresql/data/
        environment:
            - "POSTGRES_HOST_AUTH_METHOD=trust"
volumes:
    postgres_data:
```

Обратите внимание, что если ваш **SECRET_KEY** включает знак доллара, **$**, то вам нужно добавить дополнительный знак доллара, **$$**. Это связано с тем, как **docker-compose** обрабатывает подстановку переменных. В противном случае вы увидите ошибку!

Code

```
# config/settings.py
SECRET_KEY = env("DJANGO_SECRET_KEY")
```

Если вы обновите сайт, то увидите, что все работает как прежде, чего мы и хотим. Если бы по какой-то причине **SECRET_KEY** не был загружен должным образом, мы бы увидели сообщение об ошибке, так как **Django** требует его для правильной работы.

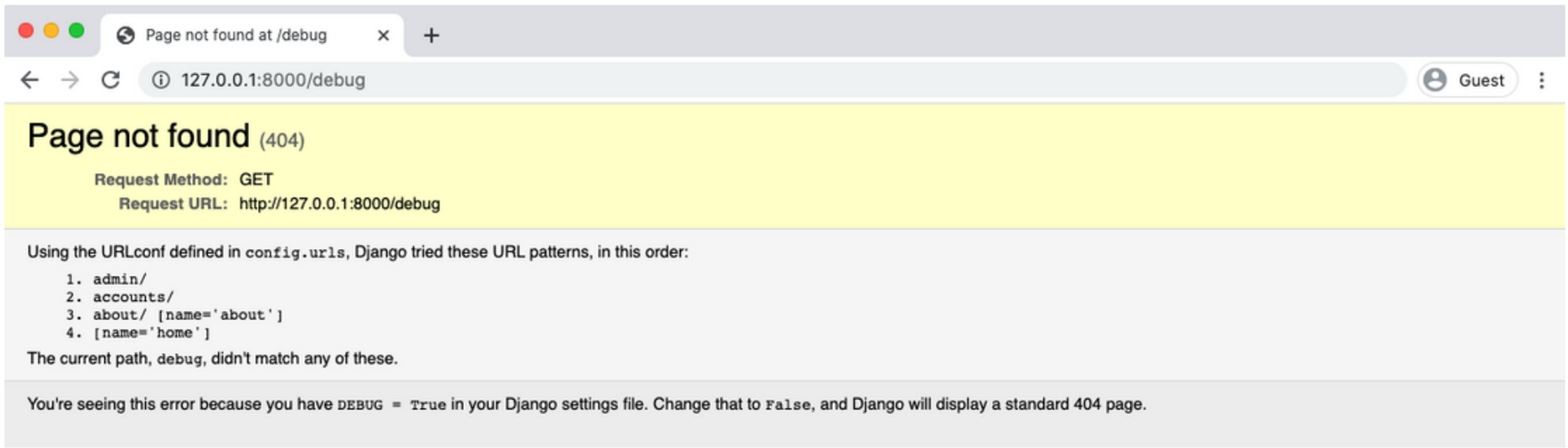
Внимательные читатели могут заметить, что хотя мы теперь используем переменную окружения, фактическое значение **SECRET_KEY** по-прежнему видно в нашем исходном коде, поскольку оно просто перенесено в **docker-compose.yml** . Это правда! Однако, когда мы настроим наш сайт для производства, мы создадим отдельный файл для производственных целей - **docker-compose-production.yml -** и загрузим переменные окружения производства через **.env** файл, который не отслеживается **Git**'ом.

Пока же цель этой главы - начать использовать переменные окружения локально для значений, которые должны быть либо действительно секретными, либо измененными в контексте производства.

DEBUG и ALLOWED_HOSTS

Как отмечается в контрольном списке развертывания **Django**, существует ряд параметров, которые необходимо обновить, прежде чем веб-сайт будет безопасно развернут в производстве. Главными из них являются **DEBUG** и **ALLOWED_HOSTS**.

Когда **DEBUG** установлен в **True**, **Django** выводит длинное сообщение и подробный отчет об ошибке при возникновении ошибки. Например, попробуйте зайти на несуществующую страницу, такую как **/debug** .



Debug Page

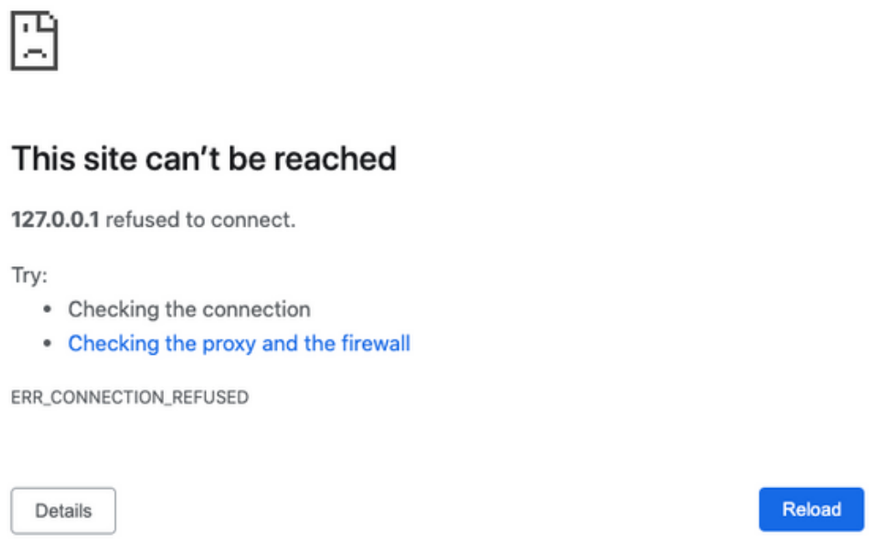
Это отлично подходит для наших целей как разработчиков, но это также является дорожной картой для хакера в производственных условиях. Когда для параметра **DEBUG** установлено значение **False**, необходимо добавить параметр **ALLOWED_HOSTS** , который контролирует определенные узлы или домены, которые могут получить доступ к веб-сайту. Мы добавим два локальных порта - **localhost** и **127.0.0.1**, а также **.herokuapp.com**, который будет использоваться **Heroku** для нашего производственного сайта.

Обновите файл **config/settings.py** с двумя новыми настройками:

Code

```
# config/settings.py
DEBUG = False # new
ALLOWED_HOSTS = ['.herokuapp.com', 'localhost', '127.0.0.1'] # new
```

Затем обновите веб-страницу.



Debug Page Not Found

Именно такое поведение мы хотим видеть на нашем живом сайте: никакой информации, только общее сообщение. Когда мы развернем сайт, мы будем использовать элегантный способ переключения между двумя настройками, а пока измените **DEBUG** на переменную окружения **DJANGO_DEBUG** .

Code

```
# config/settings.py
DEBUG = env.bool("DJANGO_DEBUG")
```

Затем обновите **docker-compose.yml,** чтобы **DJANGO_DEBUG** имел значение **True.**

docker-compose.yml

```
version: '3.8'
services:
  web:
      build: .
      command: python /code/manage.py runserver 0.0.0.0:8000
      volumes:
          - .:/code
      ports:
         - 8000:8000
      depends_on:
          - db
      environment:
          - "DJANGO_SECRET_KEY=)*_s#exg*#w+#-xt=vu8b010%%a&p@4edwyj0=(nqq90b9a8*n"
         - "DJANGO_DEBUG=True"
   db:
      image: postgres:11
      volumes:
          - postgres_data:/var/lib/postgresql/data/
      environment:
          - "POSTGRES_HOST_AUTH_METHOD=trust"
volumes:
  postgres_data:
```

После внесения изменений обновите сайт, и он будет работать как прежде.

DATABASES

Когда мы устанавливали **environs[django]** ранее, " плюшки" **Django** включали элегантный пакет **dj- database-url**, который принимает все конфигурации базы данных, необходимые для нашей базы данных, **SQLite** или **PostgreSQL.** Это будет очень полезно позже в продакшене.

На данный момент мы можем настроить его на локальное использование **PostgreSQL,** добавив значение по умолчанию. Обновите существующую конфигурацию **DATABASES** следующим образом:

Code

```
# config/settings.py
DATABASES = {
      "default": env.dj_db_url("DATABASE_URL",
       default="postgres://postgres@db/postgres")
}
```

Переменная окружения **DATABASE_URL** будет создана **Heroku** при развертывании. Подробнее об этом позже. Обновите сайт, чтобы убедиться, что все по-прежнему работает правильно.

Git

В этой главе мы внесли ряд важных изменений, поэтому не забудьте зафиксировать обновления кода в **Git.**

Command Line

```
$ git status
$ git add .
$ git commit -m 'ch8'
```

Если возникнут какие-либо проблемы, сравните свои файлы с официальным исходным кодом на **Github**.

Заключение

Добавление переменных окружения - необходимый шаг для любого действительно профессионального **Django** проекта. К ним нужно привыкнуть, но они неоценимы для переключения между локальным и рабочим окружением, как мы будем делать позже в этой книге. В следующей главе мы полностью настроим параметры электронной почты и добавим функцию сброса пароля.

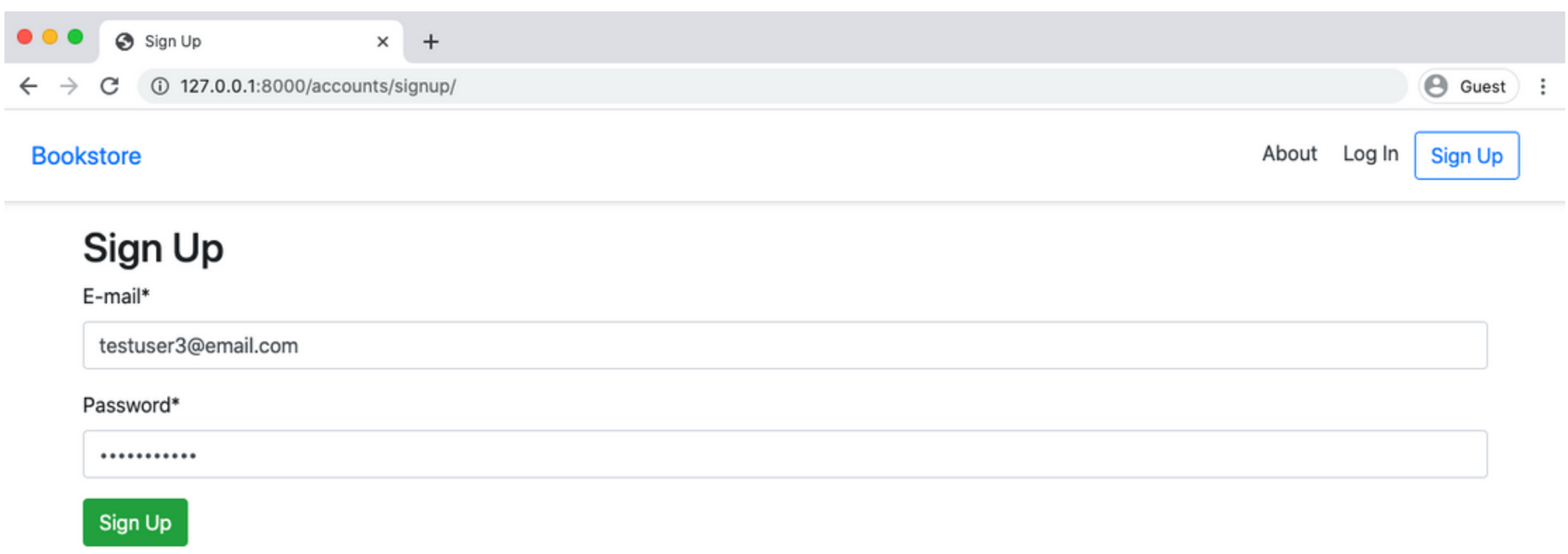
Глава 9: Электронная почта

В этой главе мы полностью настроим электронную почту и добавим функции изменения и сброса пароля. В настоящее время электронные письма фактически не отправляются пользователям. Они просто выводятся в консоль командной строки. Мы изменим это, подписавшись на сторонний почтовый сервис, получив ключи **API** и обновив наш файл **settings.py**. **Django** позаботится обо всем остальном.

Пока что вся наша работа - пользовательская модель, страницы приложения, статические активы, аутентификация с помощью **allauth** и переменные окружения - может быть применена практически к любому новому проекту. После этой главы мы приступим к созданию самого сайта книжного магазина, а не к фундаментальным шагам.

Собственные письма-подтверждения

Давайте зарегистрируем новую учетную запись пользователя, чтобы рассмотреть текущий поток регистрации пользователей. Затем мы настроим его. Убедитесь, что вы вышли из системы, а затем перейдите на страницу **Sign Up**. Я решил использовать **testuser3@email.com** и **testpass123** в качестве пароля.



testuser3 Sign Up

После отправки мы перенаправляемся на домашнюю страницу с пользовательским приветствием, и нам отправляется письмо в консоли командной строки. Вы можете увидеть это, проверив журналы с помощью **docker-compose logs .**

Чтобы настроить это письмо, нам сначала нужно найти существующие шаблоны. Перейдите к исходному коду **django-allauth** на **Github** и выполните поиск по части сгенерированного текста. В результате мы обнаружим, что на самом деле используются два файла: один для темы письма, **email_confirmation_subject.txt ,** а другой для тела письма**, email_confirmation_-message.txt .**

Чтобы обновить оба файла, мы переопределим их, воссоздав структуру **django-allauth**, что означает создание собственной директории **email** в **templates/account** и добавление туда собственных версий файлов.

Command Line

```
$ mkdir templates/account/email
$ touch templates/account/email/email_confirmation_subject.txt
$ touch templates/account/email/email_confirmation_message.txt
```

Давайте начнем со строки темы, поскольку она короче. Вот текст по умолчанию из **django-allauth .**

email_confirmation_subject.txt

```
{% load i18n %}
{% autoescape off %}
{% blocktrans %}Please Confirm Your E-mail Address{% endblocktrans %}
{% endautoescape %}
```

Первая строка, **{% load i18n %}** , предназначена для поддержки функции интернационализации **Django**, возможности поддержки нескольких языков. Затем идет тег шаблона **Django** для автоскрытия. По умолчанию он включен и защищает от проблем безопасности, таких как **cross site scripting**. Но поскольку здесь мы можем доверять содержанию текста, он отключен.

Наконец, мы переходим к самому тексту, который обернут в теги шаблона **blocktrans** для поддержки переводов. Давайте изменим текст, чтобы продемонстрировать, что мы можем это сделать.

email_confirmation_subject.txt

```
{% load i18n %}
{% autoescape off %}
{% blocktrans %}Confirm Your Sign Up{% endblocktrans %}
{% endautoescape %}
```

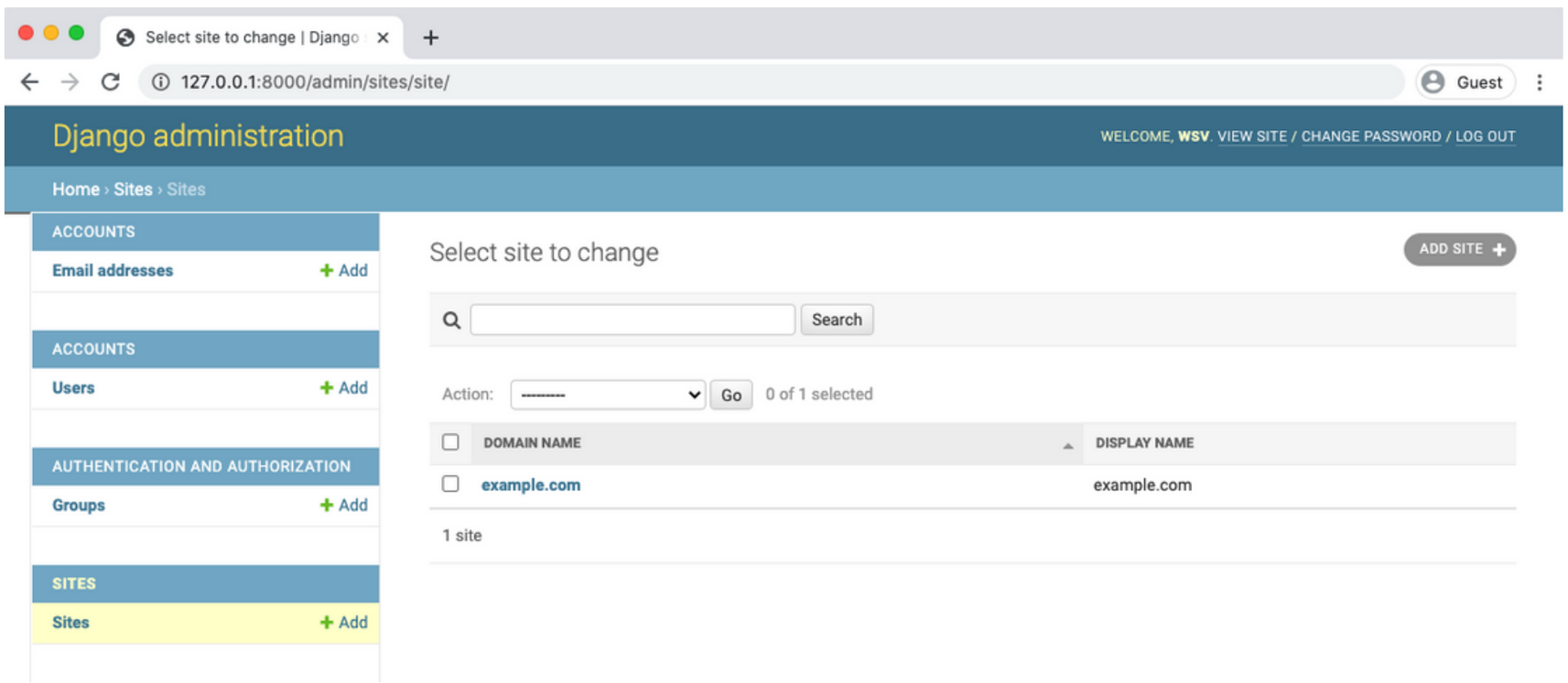
Теперь перейдем к самому сообщению о подтверждении электронной почты. Вот текущее значение по умолчанию:

email_confirmation_message.txt

```
{% load account %}{% user_display user as user_display %}{% load i18n %}\
{% autoescape off %}{% blocktrans with site_name=current_site.name\
    site_domain=current_site.domain %}\
Hello from {{ site_name }}!

You're receiving this e-mail because user {{ user_display }} has given\
yours as an e-mail address to connect their account.

To confirm this is correct, go to {{ activate_url }}
{% endblocktrans %}{% endautoescape %}
{% blocktrans with site_name=current_site.name\
     site_domain=current_site.domain %}
Thank you from {{ site_name }}!
{{ site_domain }}{% endblocktrans %}
```

Обратите внимание, что обратные косые черты **** включены для форматирования, но не являются необходимыми в исходном коде. Другими словами, вы можете удалить их из приведенного ниже кода и других примеров кода по мере необходимости.

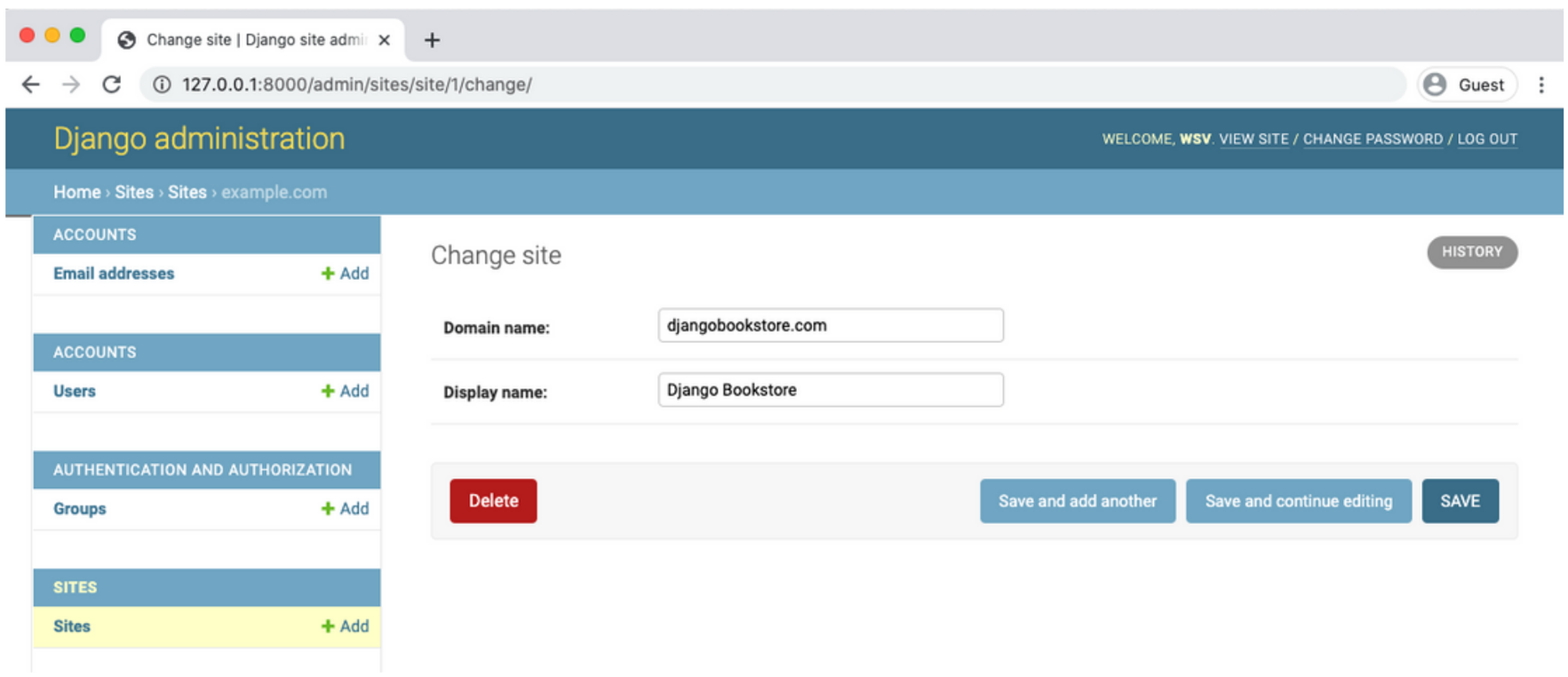
Вы, вероятно, заметили, что отправленное по умолчанию письмо ссылалось на наш сайт **example.com**, который здесь отображается как **{{** имя_сайта **}}. .** Откуда это взялось**?** Ответ **-** из раздела **sites** админки **Django**, который используется **django-allauth .** Итак, зайдите в админку по адресу **http://127.0.0.1:8000/admin/** и нажмите на ссылку **Sites** на главной странице.



Admin Sites

Здесь есть **" Domain Name"** и **"Display Name".** Нажмите на **example.com** в разделе **" Domain Name ",** чтобы мы могли его отредактировать**.** Доменное имя **-** это полное доменное имя сайта**,** например**, djangobookstore.com,** а отображаемое имя **-** это человекочитаемое название сайта**,** например**, Django Bookstore.**

Внесите эти изменения и нажмите кнопку "Сохранить" в правом нижнем углу**,** когда все будет готово**.**



Admin Sites - DjangoBookstore.com

Итак**,** вернемся к нашему электронному письму**.** Давайте немного изменим его**,** изменив приветствие с **"Hello"** на **"Hi".**

email_confirmation_message.txt

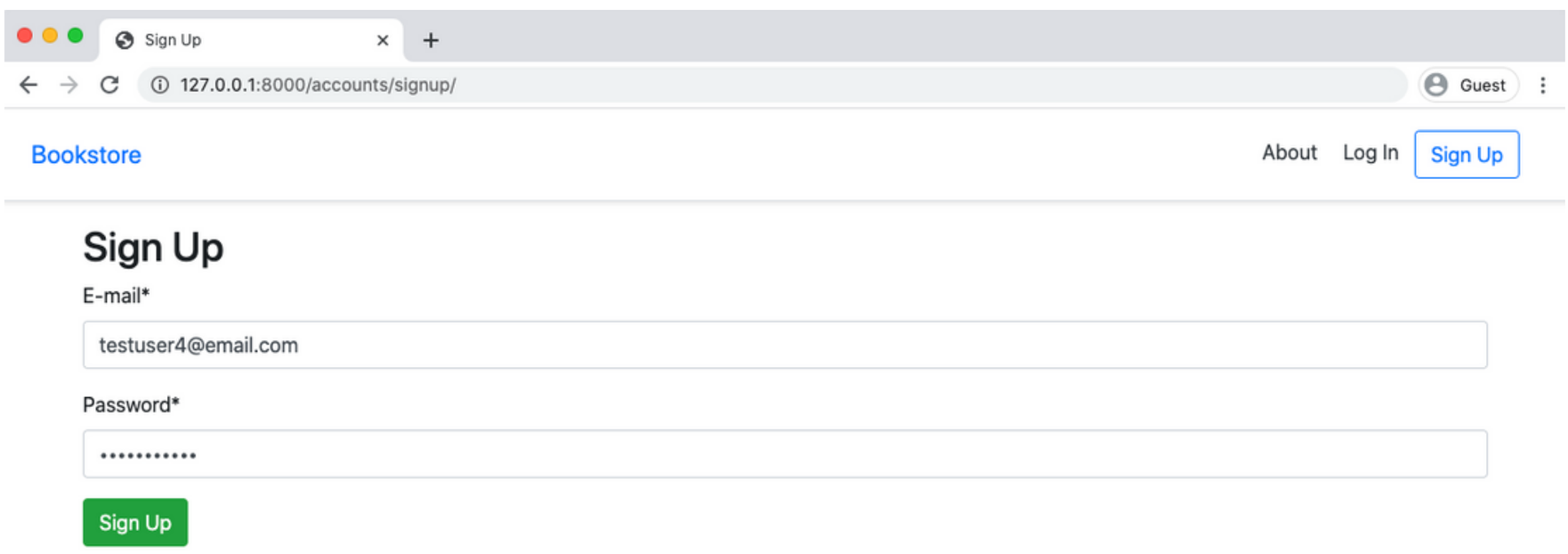
```
{% load account %}{% user_display user as user_display %}{% load i18n %}\
{% autoescape off %}{% blocktrans with site_name=current_site.name
      \site_domain=current_site.domain %}
Hi from {{ site_name }}!

You're receiving this e-mail because user {{ user_display }} has given\
yours as an e-mail address to connect their account.

To confirm this is correct, go to {{ activate_url }}
{% endblocktrans %}{% endautoescape %}
{% blocktrans with site_name=current_site.name\
      site_domain=current_site.domain %}
Thank you from {{ site_name }}!
{{ site_domain }}{% endblocktrans %}
```

И последнее, что нужно изменить. Вы заметили, что письмо было отправлено с **webmaster@localhost?** Это настройка по умолчанию, которую мы также можем изменить с помощью **DEFAULT_FROM_EMAIL.** Давайте сделаем это сейчас, добавив следующую строку в нижней части файла **config/settings.py.**

Code

```
# config/settings.py
DEFAULT_FROM_EMAIL = 'admin@djangobookstore.com'
```

Убедитесь, что вы вышли из сайта, и снова перейдите на страницу регистрации, чтобы создать нового пользователя. Для удобства я использовал **testuser4@email.com.**



Sign Up testuser4

Войдите в систему и после перенаправления на домашнюю страницу проверьте командную строку, чтобы увидеть сообщение, набрав **docker-compose logs .**

Command Line

```
...
web_1 | Content-Transfer-Encoding: 7bit
web_1 | Subject: [Django Bookstore] Confirm Your Sign Up
web_1 | From: admin@djangobookstore.com
web_1 | To: testuser4@email.com
web_1 | Date: Mon, 03 Aug 2020 18:34:50 -0000
web_1 | Message-ID: <156312929025.27.23320962393978337 69@87d045aff8f7>
web_1 |
web_1 | Hi from Django Bookstore!
web_1 |
web_1 | You're receiving this e-mail because user testuser4 has given yours\
as an e-mail address to connect their account.
web_1 |
web_1 | To confirm this is correct, go to http://127.0.0.1:8000/accounts/\
confirm-email/NA:1hmjKk:6MiDB5XoLW3HAhePuZ5WucR0Fiw/
web_1 |
web_1 | Thank you from Django Bookstore!
web_1 | djangobookstore.com
```

И вот он новый параметр **From,** новый домен **djangobookstore.com** и новое сообщение в письме**.**

Страница подтверждения электронной почты

Нажмите на уникальную ссылку **URL** в письме**,** которая перенаправляет на страницу подтверждения электронной почты**.**

Confirm Email Page

Не очень привлекательно**.** Давайте обновим его**,** чтобы он соответствовал внешнему виду остальной части нашего сайта**.** Повторный поиск в исходном коде **django-allauth** на **Github** показывает**,** что имя и расположение этого файла **- templates/account/email_confirm.html .** Итак**,** давайте создадим наш собственный шаблон**.**

Command Line

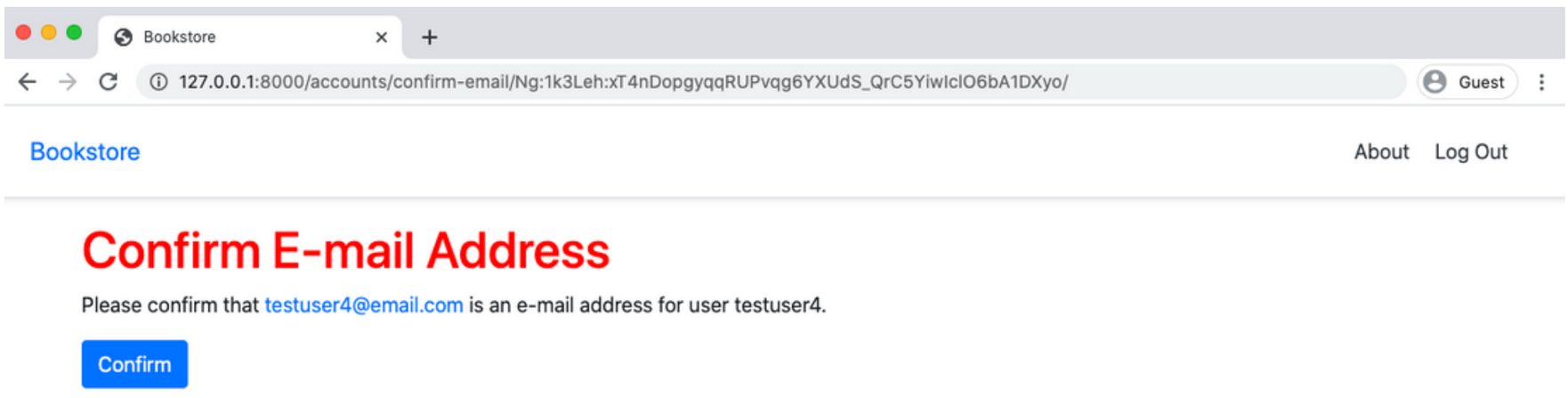
```
$ touch templates/account/email_confirm.html
```

А затем обновите его**,** чтобы расширить **_base.html** и использовать **Bootstrap** для кнопки**.**

Code

```
<!-- templates/account/email_confirm.html -->
{% extends '_base.html' %}
{% load i18n %}
{% load account %}
{% block head_title %}{% trans "Confirm E-mail Address" %}{% endblock %}
{% block content %}
<h1>{% trans "Confirm E-mail Address" %}</h1>
{% if confirmation %}
{% user_display confirmation.email_address.user as user_display %}
<p>{% blocktrans with confirmation.email_address.email as email %}Please confirm
   that <a href="mailto:{{ email }}">{{ email }}</a> is an e-mail address for user
   {{ user_display }}.{% endblocktrans %}</p>
<form method="post" action="{% url 'account_confirm_email' confirmation.key %}">
   {% csrf_token %}
   <button class="btn btn-primary" type="submit">{% trans 'Confirm' %}</button>
</form>
{% else %}
{% url 'account_email' as email_url %}
<p>{% blocktrans %}This e-mail confirmation link expired or is invalid. Please
   <a href="{{ email_url }}">issue a new e-mail confirmation request</a>.\
   {% endblocktrans %}
</p>
{% endif %}
{% endblock %}
```

Обновите страницу, чтобы увидеть наше обновление.



Обновлена страница подтверждения электронной почты

Сброс пароля и смена пароля

Django и **django-allauth** также поставляются с поддержкой дополнительных функций учетной записи пользователя, таких как возможность сбросить забытый пароль и изменить существующий пароль, если вы уже вошли в систему.

Расположение страниц сброса и изменения пароля по умолчанию следующее:

- **http://127.0.0.1:8000/accounts/password/reset/**
- **http://127.0.0.1:8000/accounts/password/change/**

Если вы пройдете через поток каждого из них, вы можете найти соответствующие шаблоны и почтовые сообщения в исходном коде **django-allauth**.

Email Service(Служба электронной почты)

Электронные письма, которые мы настроили до сих пор, обычно называют " **Transactional Emails ",** поскольку они основаны на каком-либо действии пользователя. Это отличается от " **Marketing Emails ",** таких как, например, ежемесячный информационный бюллетень.

Существует множество провайдеров транзакционной электронной почты, включая **SendGrid**, **MailGun**, **Amazon's Simple Email Service. Django** не зависит от того, какой провайдер используется; шаги похожи для всех, и у многих есть бесплатный уровень.

После регистрации аккаунта в выбранном вами почтовом сервисе у вас часто будет выбор между использованием **SMTP** и **Web API. SMTP** проще в настройке, но веб-**API** является более настраиваемым и надежным. Начните с **SMTP** и продвигайтесь дальше: конфигурации электронной почты могут быть довольно сложными сами по себе.

После получения имени пользователя и пароля у почтового провайдера, несколько настроек позволят **Django** использовать их для отправки писем.

Первым шагом будет обновление конфигурации **EMAIL_BACKEND,** которая должна находиться в самом низу файла **config/settings.py,** поскольку мы уже обновляли ее ранее.
Code

```
# config/settings.py
EMAIL_BACKEND = 'django.core.mail.backends.smtp.EmailBackend' # new
```

А затем настроить **EMAIL_HOST , EMAIL_HOST_USER , EMAIL_HOST_PASSWORD , EMAIL_PORT ,** и **EMAIL_USE_TLS** в соответствии с инструкциями вашего почтового провайдера в качестве переменных окружения.

В официальном исходном коде **EMAIL_BACKEND** останется консолью, но предыдущие шаги - это то, как добавить почтовый сервис. Если вы столкнулись с проблемой правильной настройки электронной почты, что ж, вы не одиноки! **Django**, по крайней мере, делает это намного, намного проще, чем реализация без преимуществ фреймворка, включающего встроенные батареи.

Git

Чтобы зафиксировать обновления кода этой главы, обязательно проверьте статус изменений, добавьте их все и включите сообщение о фиксации.

Command Line

```
$ git status
$ git add .
$ git commit -m 'ch9'
```

Если у вас возникли проблемы, сравните свой код с официальным исходным кодом на **Github.**

Заключение

Правильная настройка электронной почты - это в основном одноразовая проблема. Но это необходимая часть любого производственного веб-сайта. На этом мы завершаем базовые главы нашего проекта **Bookstore.** В следующей главе мы, наконец, приступим к созданию самого книжного магазина.

Глава 10: Приложение " Books"

В этой главе мы создадим приложение **Books** для нашего проекта**,** которое будет отображать все доступные книги и иметь отдельную страницу для каждой**.** Мы также рассмотрим различные подходы к **URL,** начиная с использования **id,** затем перейдем к **slug** и**,** наконец**,** к **UUID.**

Для начала мы должны создать новое приложение**,** которое мы назовем **books.**

Command Line

```
$ docker-compose exec web python manage.py startapp books
```

А чтобы убедиться**,** что **Django** знает о нашем новом приложении**,** откройте текстовый редактор и добавьте новое приложение в раздел **INSTALLED_APPS** в наш файл **config/settings.py:**

Code

```
# config/settings.py
INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    'django.contrib.sites',
# Third-party
    'allauth',
    'allauth.account',
    'crispy_forms',
# Local
     'accounts',
     'pages',
     'books', # new
]
```

Итак**,** начальное создание завершено**!**

Models(модели)

В конечном итоге нам понадобятся модель**,** представление**, url** и шаблон для каждой страницы**,** поэтому часто возникают споры о том**,** с чего начать**.** Модель **-** это хорошее место для начала**,** поскольку она задает структуру**.** Давайте подумаем**,** какие поля мы захотим включить**.** Для простоты мы начнем с названия**,** автора и цены**.**

Обновите файл **books/models.py**, чтобы включить в него нашу новую модель **Books.**

Code

```
# books/models.py
from django.db import models

class Book(models.Model):
    title = models.CharField(max_length=200)
    author = models.CharField(max_length=200)
    price = models.DecimalField(max_digits=6, decimal_places=2)
    def __str__(self):
        return self.title
```

Сверху мы импортируем класс **Django models**, а затем создаем модель **Book**, которая является его подклассом, что означает, что мы автоматически получаем доступ ко всему в **django.db.models.Model** и можем добавлять дополнительные поля и методы по желанию.

Для **title** и **author** мы ограничиваем длину до **200** символов, а для **price** используем **DecimalField**, что является хорошим выбором при работе с валютой.

Ниже мы указали метод **__str__**, чтобы контролировать, как объект будет выводиться в оболочке **Admin** и **Django**.

Теперь, когда наша новая модель базы данных создана, нам нужно создать для нее новую запись миграции.

Command Line

```
$ docker-compose exec web python manage.py makemigrations books
Migrations for 'books':
    books/migrations/0001_initial.py
        - Create model Book
```

А затем примените миграцию к нашей базе данных.

Command Line

```
$ docker-compose exec web python manage.py migrate
```

Добавление имени книги приложений к каждой команде необязательно, но является хорошей привычкой, поскольку это позволяет сфокусировать и файл миграций, и команду **migrate** только на этом приложении. Если бы мы не указывали имя приложения, то все изменения были бы включены в файл миграций и базу данных **migrate**, что может быть сложнее для отладки в дальнейшем.

Наша база данных настроена. Давайте добавим некоторые данные в админку.

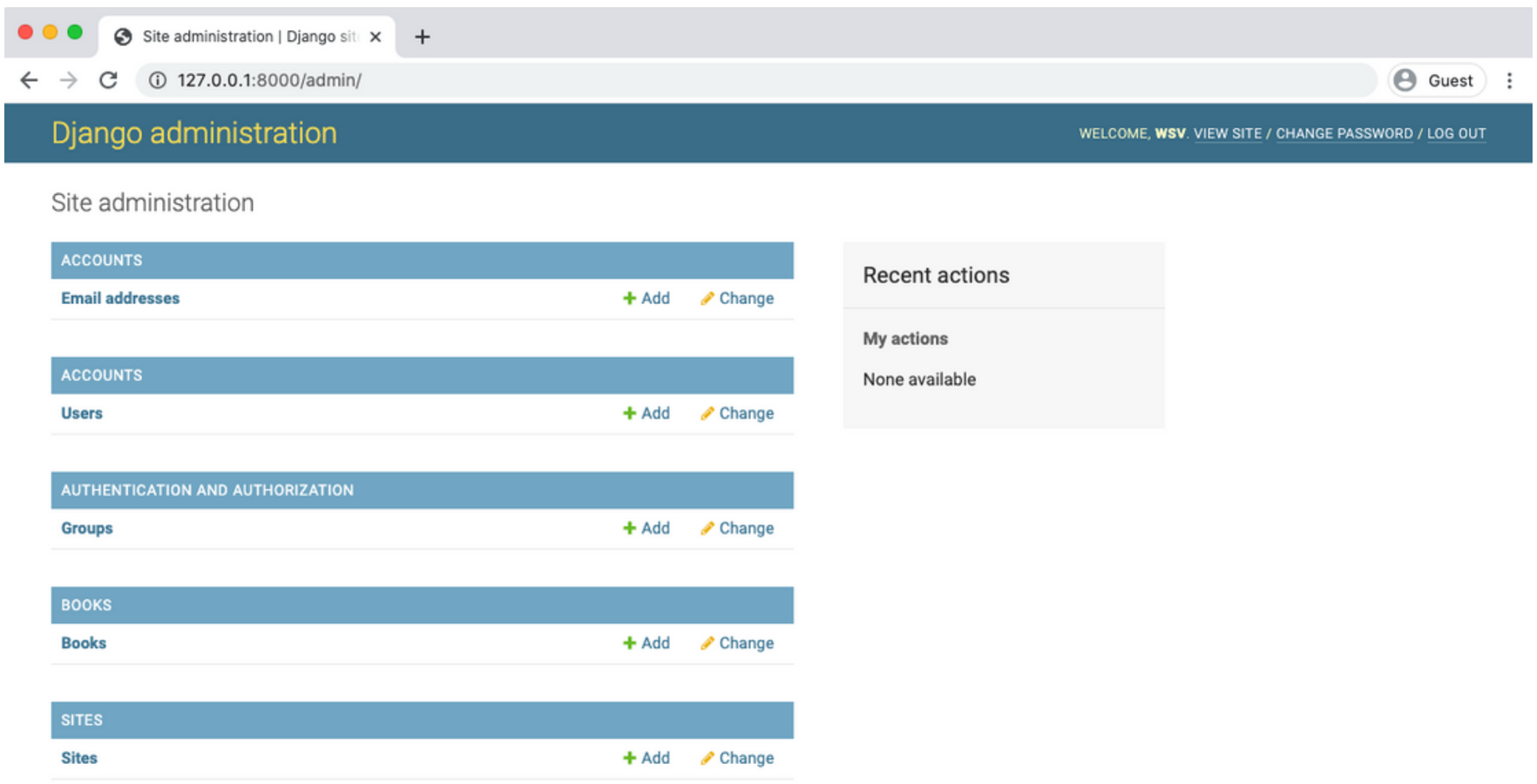
Admin

Нам нужен способ доступа к нашим данным, для чего отлично подходит **Django admin.** Не забудьте обновить файл **books/admin.py**, иначе приложение не появится**!** Я забываю этот шаг почти каждый раз, даже после использования **Django** в течение многих лет**.**

Code

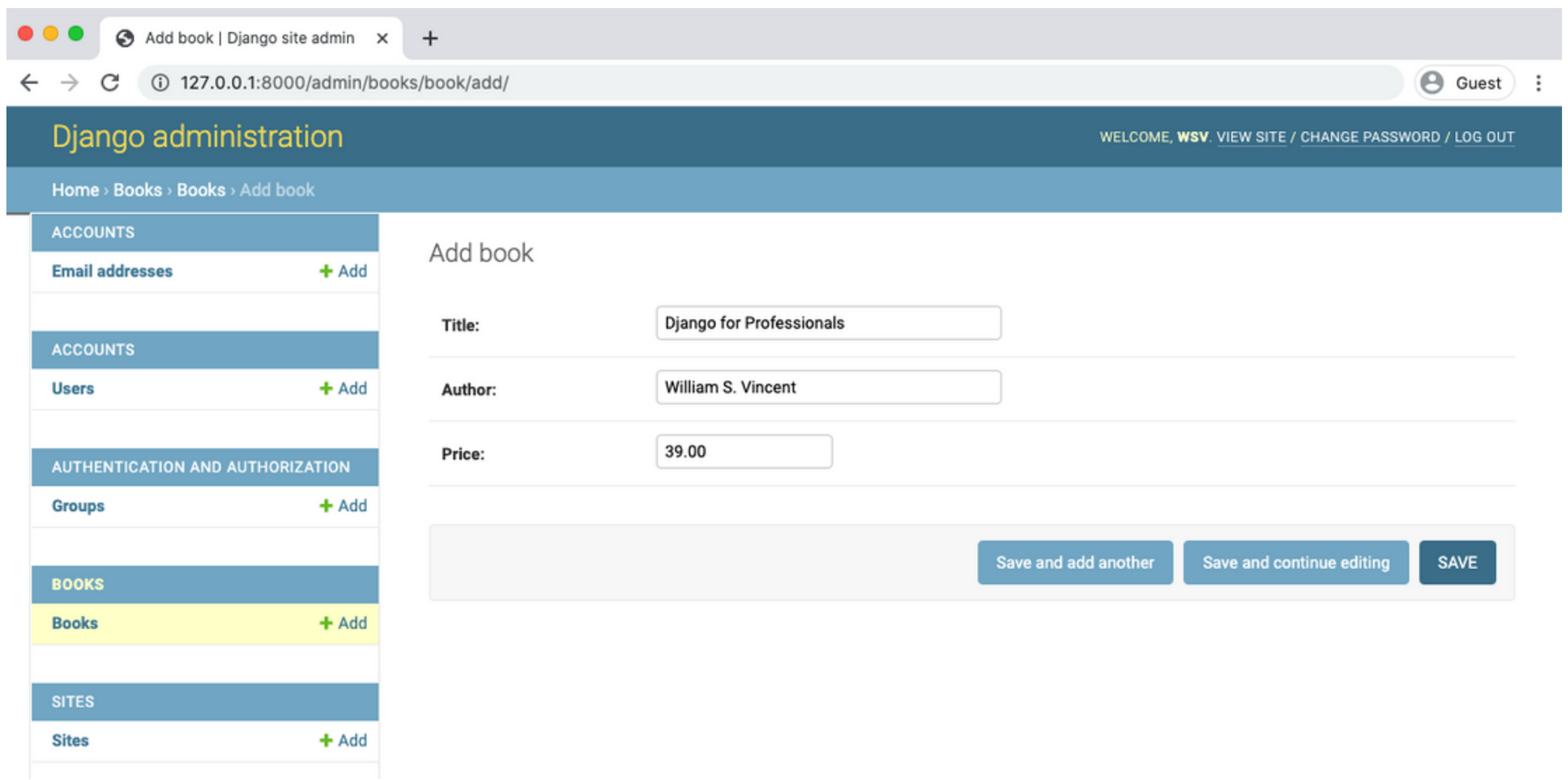
```
# books/admin.py
from django.contrib import admin
from .models import Book

admin.site.register(Book)
```

Если вы заглянете в админку на сайте **http://127.0.0.1:8000/admin/,** приложение **" Books"** теперь там**.**



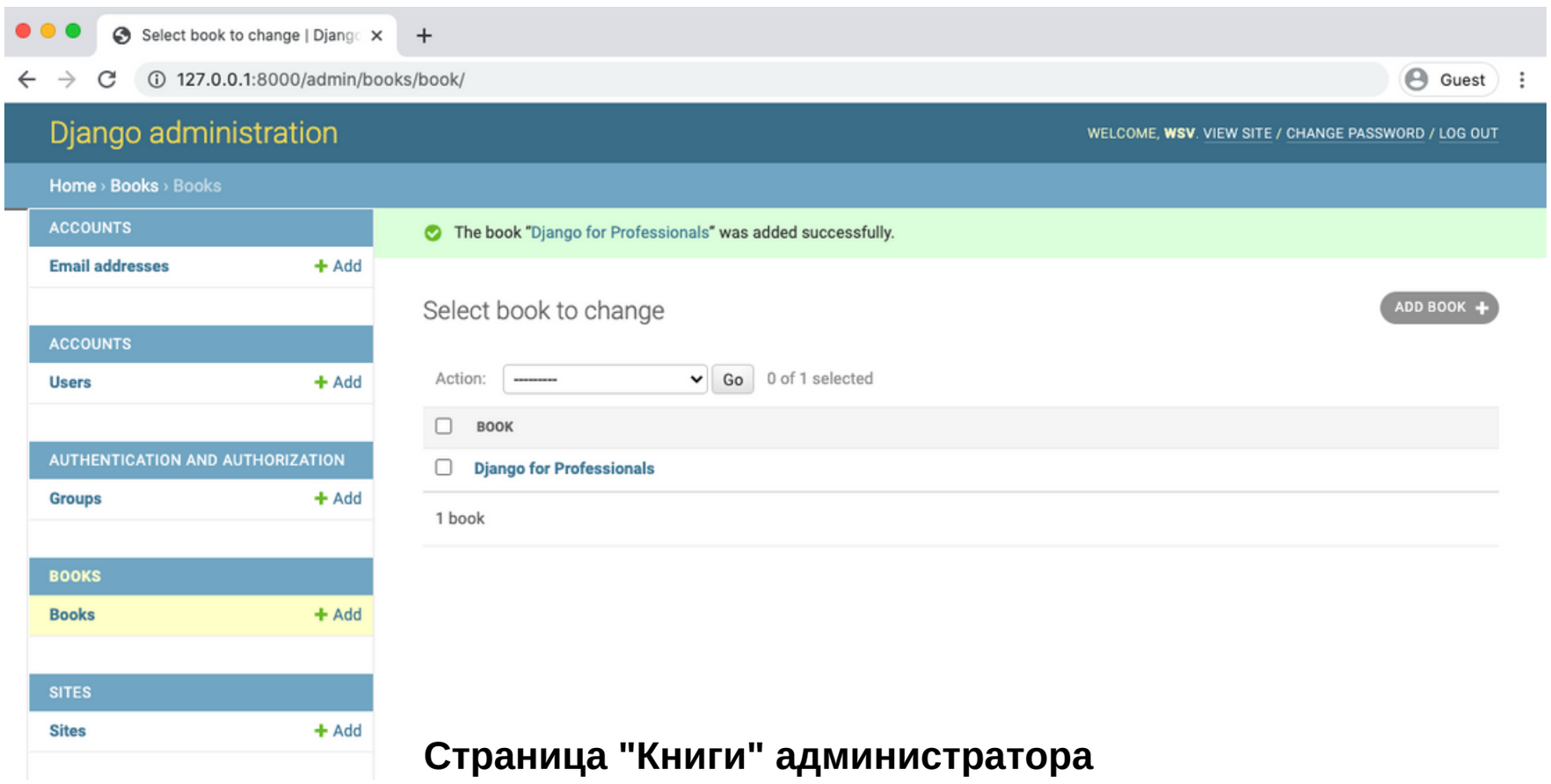
Домашняя страница администратора

Давайте добавим запись о книге **"Django** для профессионалов"**.** Нажмите на кнопку **+** Добавить рядом с **Books,** чтобы создать новую запись**.** Название книги **- "Django** для профессионалов"**,** автор **-** "Уильям С**.** Винсент"**,** цена **- $39.00.** Нет необходимости включать знак доллара **$** в сумму**,** так как мы добавим его в наш будущий шаблон**.**



Admin - Django for Professionals book

После нажатия на кнопку "Сохранить" происходит перенаправление на главную страницу **Books,** на которой отображается только название**.**



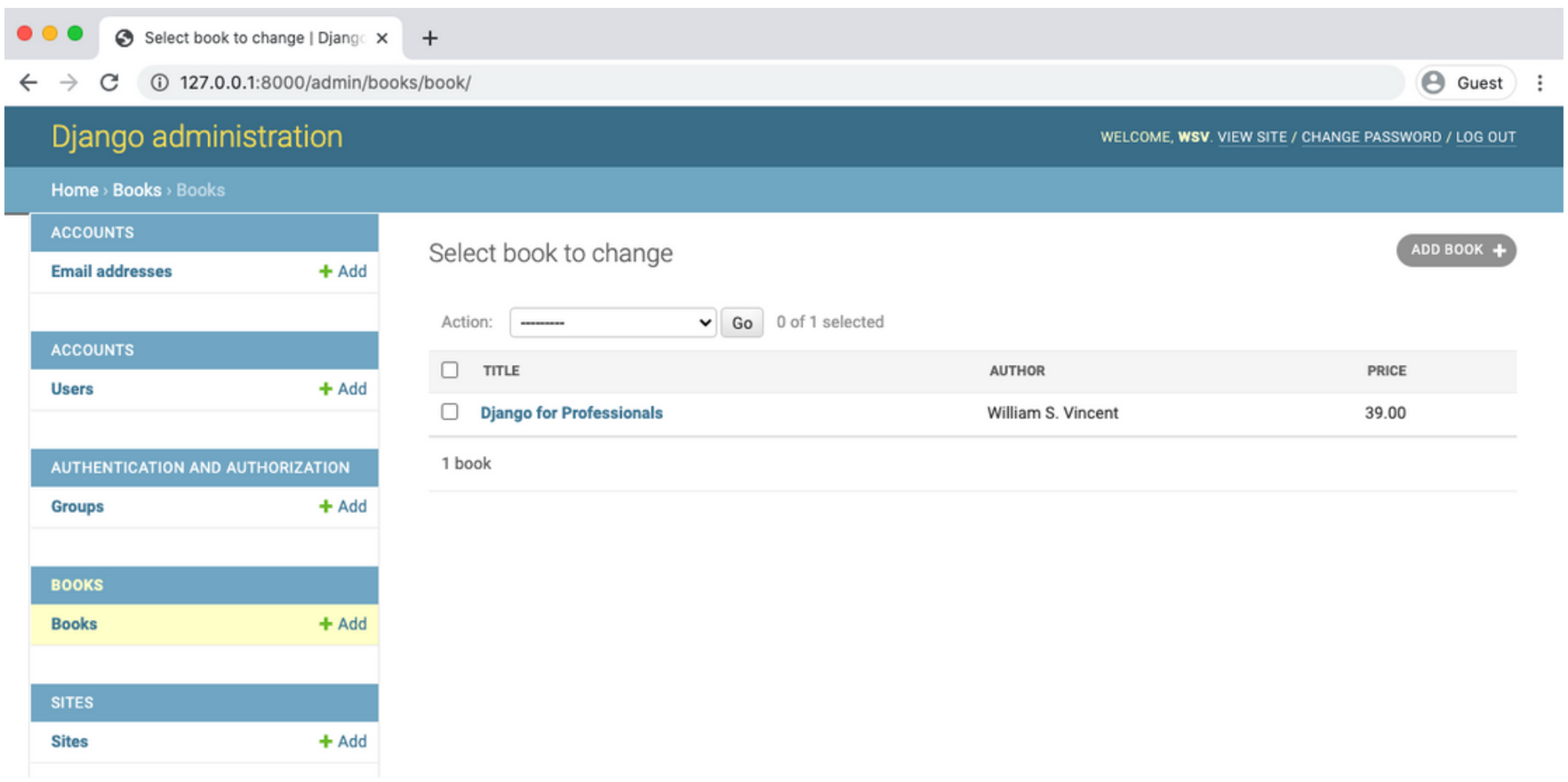
Страница "Книги" администратора

Давайте обновим файл **books/admin.py,** чтобы указать, какие поля мы также хотим отображать**.**

Code

```
# books/admin.py
from django.contrib import admin
from .models import Book

class BookAdmin(admin.ModelAdmin):
        list_display = ("title", "author", "price",)

admin.site.register(Book, BookAdmin)
```

Затем обновите страницу**.**



Страница списка книг администратора

Теперь**,** когда наша модель базы данных готова**,** нам нужно создать необходимые представления**, URL** и шаблоны**,** чтобы мы могли отображать информацию в нашем веб-приложении**.** Вопрос о том**,** с чего начать**,** всегда вызывает недоумение у разработчиков**.**

Лично я часто начинаю с **URL,** затем с представлений и шаблонов**.**

URLs

Нам нужно обновить два файла **urls.py.** Первый из них **- config/urls.py .**
Добавьте новый путь для приложения **books.**

Code

```
# config/urls.py
from django.contrib import admin
from django.urls import path, include

urlpatterns = [
# Django admin
    path('admin/', admin.site.urls),
# User management
    path('accounts/', include('allauth.urls')),
# Local apps
    path('', include('pages.urls')),
    path('books/', include('books.urls')), # new
]
```

Теперь создайте пути **URL** нашего приложения для книг**.** Сначала мы должны создать этот файл**.**

Command Line

```
$ touch books/urls.py
```

Мы будем использовать пустую строку **''**, поэтому в сочетании с тем фактом**,** что все **URL-**адреса приложения **books** будут начинаться с **books/**, это также будет маршрут для нашего основного представления списка каждой книги**.** Представление**,** на которое он ссылается**, BookListView,** еще не создано**.**

Code

```
# books/urls.py
from django.urls import path
from .views import BookListView

urlpatterns = [
    path('', BookListView.as_view(), name='book_list'),
]
```

Обратите внимание**,** что шаблон **book_list.html** еще не существует**.**

Templates

Создавать папку для конкретного приложения в шаблонах необязательно**,** но это может помочь**,** особенно по мере роста числа**,** поэтому мы создадим папку под названием **books .**

Command Line

```
$ mkdir templates/books/
$ touch templates/books/book_list.html
```

Code

```
<!-- templates/books/book_list.html -->
{% extends '_base.html' %}
{% block title %}Books{% endblock title %}
{% block content %}
{% for book in object_list %}
<div>
    <h2><a href="">{{ book.title }}</a></h2>
</div>
{% endfor %}
{% endblock content %}
```

В верхней части мы отмечаем, что этот шаблон расширяет **_base.html**, а затем обертывает наш нужный код блоками контента. Мы используем **Django Templating Language** для установки простого цикла **for** для каждой книги. Обратите внимание, что **object_list** происходит от **ListView** и содержит все объекты в нашем представлении.

Последний шаг - раскрутить и затем выключить наши контейнеры, чтобы перезагрузить файл **Django settings.py**. В противном случае он не поймет, что мы внесли изменения, и поэтому появится страница с ошибкой, а в логах появится сообщение **"ModuleNotFoundError:** Нет модуля с именем **'books.urls'".**

Разверните вниз, а затем снова вверх наши контейнеры.

Command Line

```
$ docker-compose down
$ docker-compose up -d
```

Если вы зайдете на сайт **http://127.0.0.1:8000/books/**, страница книг будет работать.

Books Page

object_list

ListView полагается на **object_list ,** как мы только что видели, но это далеко не самое понятное имя. Лучше переименовать его в более дружественное имя с помощью **context_object_name .**

Обновите **books/views.py** следующим образом.

Code

```
# books/views.py
from django.views.generic import ListView
from .models import Book

class BookListView(ListView):
        model = Book
        context_object_name = 'book_list' # new
        template_name = 'books/book_list.html'
```

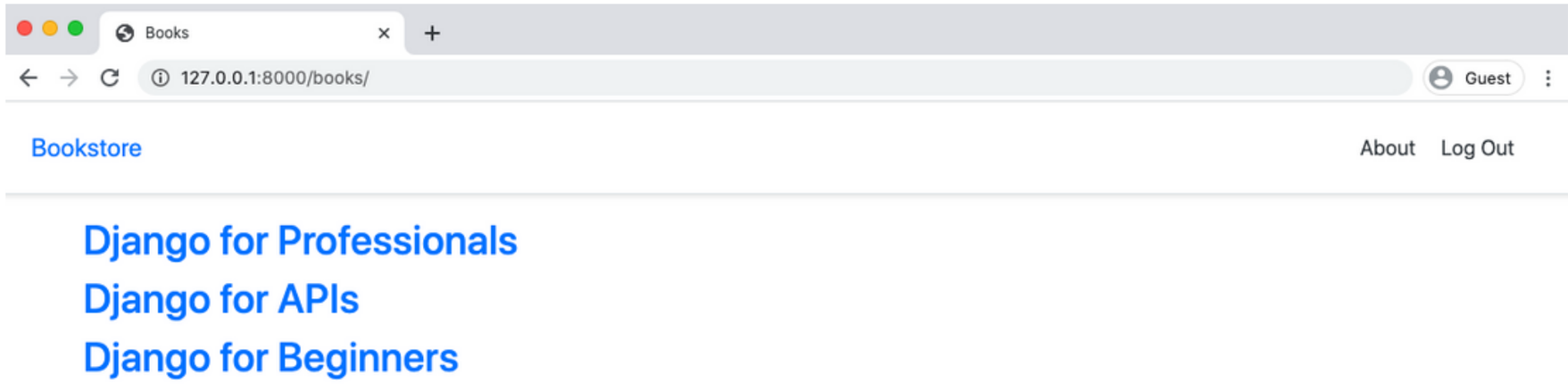
А затем поменяйте в нашем шаблоне **object_list** на **book_list .**

Code

```
<!-- templates/books/book_list.html -->
{% extends '_base.html' %}
{% block title %}Books{% endblock title %}
{% block content %}
{% for book in book_list %}
<div>
   <h2><a href="">{{ book.title }}</a></h2>
</div>
{% endfor %}
{% endblock content %}
```

Обновите страницу, и она будет работать как прежде! Эта техника особенно полезна в больших проектах, где над проектом работает несколько разработчиков. Очень сложно для **front-end** инженера правильно угадать, что означает **object_list!**

Чтобы доказать, что представление списка работает для нескольких элементов, добавьте еще две книги на сайт через администратора. Я добавил две другие мои книги по **Django - "Django** для начинающих" и **"Django** для **API",** у которых в качестве автора указан **"William S. Vincent",** а в качестве цены **- "39.00".**



Three Books

Страница отдельной книги

Теперь мы можем добавить отдельные страницы для каждой книги с помощью другого представления на основе класса **Generic** под названием **DetailView.**

Наш процесс аналогичен странице **Books** и начинается с **URL,** импортируя **BookDetailView** во второй строке, а затем устанавливая путь к первичному ключу каждой книги, который будет представлен как целое число **<int:pk> .**

Code

```
# books/urls.py
from django.urls import path
from .views import BookListView, BookDetailView # new

urlpatterns = [
    path('', BookListView.as_view(), name='book_list'),
    path('<int:pk>/', BookDetailView.as_view(), name='book_detail'), # new
]
```

Django автоматически добавляет автоинкрементный первичный ключ к нашим моделям баз данных**.** Поэтому**,** хотя мы объявили только поля **title, author** и **body** в нашей модели **Book,** под капотом **Django** добавил еще одно поле **id,** которое является нашим первичным ключом**.** Мы можем обращаться к нему либо как к **id,** либо как к **pk.**

Для нашей первой книги **pk** будет **1.** Для второй **- 2.** И так далее**.** Поэтому**,** когда мы переходим на страницу индивидуальной записи для нашей первой книги**,** мы можем ожидать**,** что ее **URL-**маршрут будет **books/1.**

Теперь перейдем к файлу **books/views.py,** где мы импортируем **DetailView** и создадим класс **BookDetailView,** который также определяет поля **model** и **template_name.**

Code

```
# books/views.py
from django.views.generic import ListView, DetailView # new
from .models import Book

class BookListView(ListView):
        model = Book
        context_object_name = 'book_list'
        template_name = 'books/book_list.html'

class BookDetailView(DetailView): # new
        model = Book
        template_name = 'books/book_detail.html'
```

И**,** наконец**,** шаблон **book_detail.html .**

Command Line

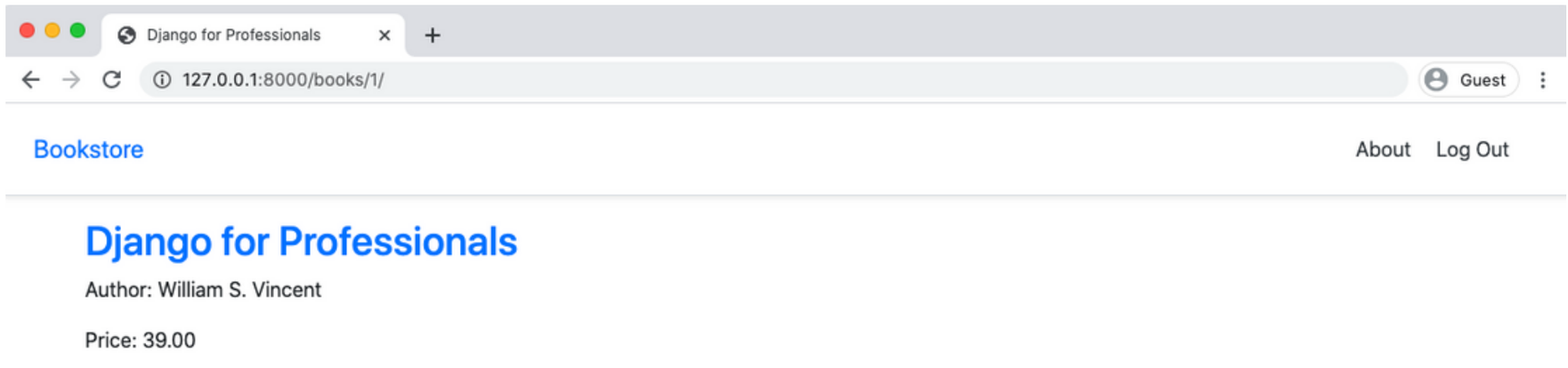
```
$ touch templates/books/book_detail.html
```

Затем пусть он отобразит все текущие поля**.** Мы также можем отобразить заголовок в тегах **title,** чтобы он отображался на вкладке браузера**.**

Code

```
<!-- templates/books/book_detail.html -->
{% extends '_base.html' %}
{% block title %}{{ object.title }}{% endblock title %}
{% block content %}
<div class="book-detail">
    <h2><a href="">{{ object.title }}</a></h2>
    <p>Author: {{ object.author }}</p>
    <p>Price: {{ object.price }}</p>
</div>
{% endblock content %}
```

Если вы сейчас перейдете на сайт **http://127.0.0.1:8000/books/1/,** то увидите специальную страницу для нашей первой книги**.**



Book Detail Page

context_object_name

Так же как **ListView** по умолчанию использует **object_list,** который мы обновили**,** чтобы сделать его более конкретным**,** так и **DetailView** по умолчанию использует **object,** который мы можем сделать более описательным с помощью **context_object_name.** Мы установим его на **book .**

Code

```
# books/views.py
...
class BookDetailView(DetailView):
        model = Book
        context_object_name = 'book' # new
        template_name = 'books/book_detail.html'
```

Не забудьте также обновить наш шаблон**,** заменив в нем объект на **book** для наших трех полей**.**

Code

```
<!-- templates/books/book_detail.html -->
{% extends '_base.html' %}
{% block title %}{{ book.title }}{% endblock title %}
{% block content %}
<div class="book-detail">
    <h2><a href="">{{ book.title }}</a></h2>
    <p>Author: {{ book.author }}</p>
    <p>Price: {{ book.price }}</p>
</div>
{% endblock content %}
```

В качестве последнего шага мы хотим, чтобы ссылка на странице списка книг указывала на отдельную страницу. С помощью тега шаблона **url** мы можем указать на **book_detail -** имя **URL,** заданное в **books/urls.py -** и затем передать **pk**

Code

```
<!-- templates/books/book_list.html -->
{% extends '_base.html' %}
{% block title %}Books{% endblock title %}
{% block content %}
{% for book in book_list %}
<div>
  <h2><a href="{% url 'book_detail' book.pk %}">{{ book.title }}</a></h2>
</div>
{% endfor %}
{% endblock content %}
```

Обновите страницу со списком книг на **http://127.0.0.1:8000/books/,** и теперь все ссылки кликабельны и ведут на нужную страницу отдельной книги.

get_absolute_url

Один дополнительный шаг, который мы еще не сделали, **-** это добавление метода **get_absolute_url(),** который устанавливает радикальный **URL** для модели. Он также необходим при использовании функции **reverse(),** которая часто применяется.

Вот как добавить его в наш файл **books/models.py.** Импортируйте **reverse** в верхней части. Затем добавьте метод **get_absolute_url,** который будет обратным имени нашего **URL, book_detail,** и передает **id** в виде строки.

Code

```
# books/models.py
from django.db import models
from django.urls import reverse # new

class Book(models.Model):
        title = models.CharField(max_length=200)
        author = models.CharField(max_length=200)
        price = models.DecimalField(max_digits=6, decimal_places=2)
        def __str__(self):
                return self.title
        def get_absolute_url(self): # new
                return reverse('book_detail', args=[str(self.id)])
```

Затем мы можем обновить шаблоны. В настоящее время наша ссылка **a href** использует **{% url 'book_detail' book.pk %} .** Однако вместо этого мы можем использовать **get_absolute_url,** который уже имеет переданный **pk.**

Code

```
<!-- templates/books/book_list.html -->
{% extends '_base.html' %}
{% block title %}Books{% endblock title %}
{% block content %}
{% for book in book_list %}
<div>
    <h2><a href="{{ book.get_absolute_url }}">{{ book.title }}</a></h2>
</div>
{% endfor %}
{% endblock content %}
```

Также нет необходимости использовать тег шаблона **url**, достаточно одной единственной канонической ссылки, которая может быть изменена при необходимости в файле **books/models.py** и оттуда будет распространяться по всему проекту. Это более чистый подход, и его следует использовать, когда вам нужны отдельные страницы для объекта.

Primary Keys vs. IDs

Можно запутаться, что использовать в проекте - первичный ключ **(PK)** или идентификатор, особенно учитывая, что в **Django DetailView** они рассматриваются как взаимозаменяемые. Однако есть тонкое различие.

id - это поле модели, которое **Django** автоматически устанавливает на автоинкремент. Таким образом, первая книга имеет **id** равный **1**, вторая запись **- 2** и так далее. По умолчанию оно также рассматривается как первичный ключ **pk** модели.

Однако можно вручную изменить первичный ключ модели. Это не обязательно должно быть **id**, но может быть что-то вроде **object_id**, в зависимости от конкретного случая использования. Кроме того, в **Python** есть встроенный объект **id()**, который иногда может вызывать путаницу и/или ошибки.

В отличие от этого, первичный ключ **pk** относится к полю первичного ключа модели, поэтому в случае сомнений безопаснее использовать **pk.** И на самом деле, в следующем разделе мы обновим **id** нашей модели!

Slugs vs. UUIDs

Использование поля **pk** в **URL** нашего **DetailView** быстро и просто, но не идеально для реального проекта. В настоящее время **pk -** это то же самое, что и наш автоинкрементный **id.** Среди прочих проблем, это сообщает потенциальному хакеру точное количество записей в вашей базе данных; это сообщает им точный идентификатор, который может быть использован в потенциальной атаке; и могут возникнуть проблемы с синхронизацией, если у вас есть несколько внешних интерфейсов.

Существует два альтернативных подхода. Первый называется **"slug" -** это газетный термин, обозначающий короткую метку для чего-то, которая часто используется в **URL-**адресах. Например, в нашем примере **"Django** для профессионалов" его **slug** может быть **django-for-professionals .** Существует даже поле модели **SlugField,** которое может быть использовано и либо добавлено при создании поля заголовка вручную, либо автоматически заполнено при сохранении. Основной проблемой при работе со **slug** является обработка дубликатов, хотя это можно решить, добавляя случайные строки или числа в данное поле **slug.** Однако проблема синхронизации остается.

Лучшим подходом является использование **UUID (Universally Unique IDentifier),** который **Django** теперь поддерживает с помощью специального поля **UUIDField.**

Давайте реализуем **UUID,** добавив новое поле в нашу модель, а затем обновим путь **URL.**

Импортируйте **uuid** в верхней части, а затем обновите поле **id,** чтобы оно стало **UUIDField,** которое теперь является первичным ключом. Мы также используем **uuid4** для шифрования. Это позволяет нам использовать **DetailView,** который требует либо поля **slug,** либо поля **pk;** он не будет работать с полем **UUID** без существенной модификации.

Code

```
# books/models.py
import uuid # new
from django.db import models
from django.urls import reverse

class Book(models.Model):
        id = models.UUIDField(primary_key=True,default=uuid.uuid4,
                editable=False)
        title = models.CharField(max_length=200)
        author = models.CharField(max_length=200)
        price = models.DecimalField(max_digits=6, decimal_places=2)
        def __str__(self):
                return self.title
        def get_absolute_url(self):
                return reverse('book_detail', args=[str(self.id)])
```

В пути **URL** поменяйте **int** на **uuid** в подробном виде.

Code

```
# books/urls.py
from django.urls import path
from .views import BookListView, BookDetailView

urlpatterns = [
    path('', BookListView.as_view(), name='book_list'),
    path('<uuid:pk>', BookDetailView.as_view(), name='book_detail'), # new
]
```

Но теперь мы столкнулись с проблемой: уже есть существующие записи в книге, фактически три, со своими собственными идентификаторами, а также связанные файлы миграции, которые их используют. Создание новой миграции в таком случае вызывает реальные проблемы. Самый простой подход, который мы будем использовать, является самым разрушительным: просто удалить старые миграции книг и начать все сначала.

Command Line

```
$ docker-compose exec web rm -r books/migrations
$ docker-compose down
```

Последняя проблема заключается в том, что мы также сохраняем нашу базу данных **PostgreSQL** через монтируемый том, который все еще содержит записи в старых полях **id.** Вы можете увидеть это с помощью команды **docker volume ls.**

Command Line

```
$ docker volume ls
DRIVER          VOLUME NAME
local           books_postgres_data
```

Самый простой подход **-** просто удалить том и начать работу с **Docker** заново**.** Поскольку мы находимся на ранней стадии проекта**,** мы пойдем этим путем**;** более зрелый проект потребует более сложного подхода**.**

Шаги включают запуск наших контейнеров **web** и **db;** добавление нового начального файла миграции для приложения **books,** применение всех обновлений с помощью **migrate ,** а затем повторное создание учетной записи суперпользователя**.**

Command Line

```
$ docker volume rm books_postgres_data
$ docker-compose up -d
$ docker-compose exec web python manage.py makemigrations books
$ docker-compose exec web python manage.py migrate
$ docker-compose exec web python manage.py createsuperuser
```

Теперь войдите в админку и снова добавьте три книги**.** Если вы перейдете на главную страницу книг и нажмете на отдельную книгу**,** вы попадете на новую страницу с подробной информацией**,** в **URL** которой будет указан **UUID.**

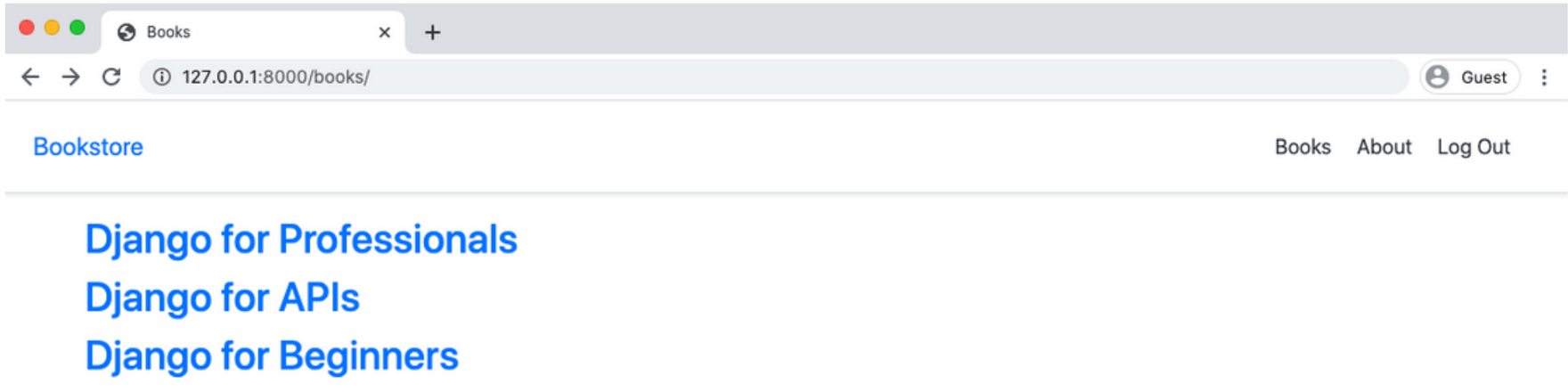
Django for Professionals book UUID

Navbar

Давайте добавим ссылку на страницу книг в нашу панель навигации**.** Мы можем использовать тег шаблона **url** и **URL-**имя страницы**: book_list .**

Code

```
<!--templates/_base.html -->
<nav class="my-2 my-md-0 mr-md-3">
    <a class="p-2 text-dark" href="{% url 'book_list' %}">Books</a>
    <a class="p-2 text-dark" href="{% url 'about' %}">About</a>
```



Обновленная панель навигации

Тесты

Теперь нам нужно протестировать нашу модель и представления**.** Мы хотим убедиться**,** что модель **Books** работает так**,** как ожидалось**,** включая ее строковое представление**.** И мы хотим протестировать **ListView** и **DetailView.**

Вот как выглядят примеры тестов в файле **books/tests.py.**

Code

```
# books/tests.py
from django.test import TestCase
from django.urls import reverse
from .models import Book

class BookTests(TestCase):
        def setUp(self):
                self.book = Book.objects.create(title='Harry Potter',
          author='JK Rowling',price='25.00',)

 def test_book_listing(self):
                self.assertEqual(f'{self.book.title}', 'Harry Potter')
                self.assertEqual(f'{self.book.author}', 'JK Rowling')
                self.assertEqual(f'{self.book.price}', '25.00')

 def test_book_list_view(self):
                response = self.client.get(reverse('book_list'))
                self.assertEqual(response.status_code, 200)
                self.assertContains(response, 'Harry Potter')
                self.assertTemplateUsed(response, 'books/book_list.html')

def test_book_detail_view(self):
                response = self.client.get(self.book.get_absolute_url())
                no_response = self.client.get('/books/12345/')
                self.assertEqual(response.status_code, 200)
                self.assertEqual(no_response.status_code, 404)
                self.assertContains(response, 'Harry Potter')
                self.assertTemplateUsed(response, 'books/book_detail.html')
```

Мы импортируем **TestCase** и в нашем методе **setUp** добавляем образец книги для тестирования**. test_book_listing** проверяет правильность строкового представления и содержания**.** Затем мы используем **test_book_- list_view** для подтверждения того**,** что наша главная страница возвращает код состояния **200 HTTP,** содержит наш основной текст и использует правильный шаблон **books/book_list.html.** Наконец**, test_book_detail_view** проверяет**,** что наша страница с подробной информацией работает так**,** как ожидалось**,** и что ошибочная страница возвращает **404.** В тестах всегда полезно проверять как то**,** что что**-**то существует**,** так и то**,** что чего**-**то неправильного не существует**.**

Теперь запустите эти тесты**.** Все они должны пройти**.**

Command Line

```
$ docker-compose exec web python manage.py test
Creating test database for alias 'default'...
System check identified no issues (0 silenced).
.................
----------------------------------------------------------------------
Ran 17 tests in 0.369s
OK
Destroying test database for alias 'default'...
```

Git

Мы проделали много работы в этой главе, поэтому теперь добавьте все это в систему контроля версий с помощью **Git**, добавив новые файлы и добавив сообщение о фиксации.

Command Line

```
$ git status
$ git add .
$ git commit -m 'ch10'
```

Официальный исходный код этой главы доступен на **Github** для ознакомления.

Заключение

Мы подошли к концу довольно длинной главы, но теперь архитектура нашего проекта **Bookstore** стала намного понятнее. Мы добавили модель книг, научились изменять структуру **URL** и перешли на гораздо более безопасный шаблон **UUID**.

В следующей главе мы узнаем об отношениях внешних ключей и добавим в наш проект опцию отзывов.

Глава 11: Приложение для отзывов

В этой главе мы добавим приложение для отзывов, чтобы читатели могли оставлять отзывы о своих любимых книгах. Это даст нам возможность обсудить внешние ключи, структуру приложения и погрузиться в формы.

Foreign Keys(внешние ключи)

Мы уже использовали внешний ключ в нашей модели пользователя, но не задумывались об этом. Теперь придется! По сути, таблицу базы данных можно представить себе как электронную таблицу со строками и столбцами. В ней должно быть поле первичного ключа, которое является уникальным и относится к каждой записи. В прошлой главе мы заменили **id** на **UUID**, но один ключ все равно существует!

Это важно, когда мы хотим связать две таблицы вместе. Например, наша модель **Books** будет связана с моделью **Reviews**, поскольку каждый отзыв должен быть связан с соответствующей книгой. Это подразумевает связь по внешнему ключу.

Существует три возможных типа отношений внешнего ключа:

- **One-to-one**
- **One-to-many**
- **Many-to-many**

Отношения "один к одному" - это самый простой вид отношений. Примером может служить таблица имен людей и таблица номеров социального страхования. Каждый человек имеет только один номер социального страхования, и каждый номер социального страхования связан только с одним человеком.

На практике отношения "один к одному" встречаются редко. Необычно, что обе стороны отношений могут быть сопоставлены только с одним экземпляром. Другими примерами могут быть отношения "страна - флаг" или "человек - паспорт".

Отношения "один ко многим" встречаются гораздо чаще, и в **Django** они используются по умолчанию в качестве внешних ключей. Например, один студент может записаться на множество занятий. Или сотрудник имеет одно название должности, может быть "инженер-программист", но в данной компании может быть много инженеров-программистов.

Также возможны отношения **ManyToManyField**. Рассмотрим список книг и список авторов: у каждой книги может быть более одного автора, а каждый автор может написать более одной книги. Это отношение "многие-ко-многим". Как и в предыдущих двух примерах, для соединения двух списков необходимо связанное поле **Foreign Key**. Дополнительные примеры включают отношения "врачи и пациенты" (каждый врач принимает несколько пациентов и наоборот) или "сотрудники и задачи" (у каждого сотрудника есть несколько задач, а над каждой задачей работают несколько сотрудников).

Проектирование баз данных - это увлекательная, глубокая тема, которая является одновременно искусством и наукой. По мере роста числа таблиц в проекте со временем почти неизбежно возникает необходимость в рефакторинге для решения проблем, связанных с неэффективностью, раздутостью и откровенными ошибками. Нормализация - это процесс структурирования реляционной базы данных, хотя он выходит далеко за рамки этой книги.

Модель отзывов

Возвращаясь к нашему базовому приложению отзывов, первое, о чем следует подумать, это какой тип отношения внешнего ключа будет использоваться. Если мы собираемся связать пользователя с отзывом, то это будет простое отношение "один ко многим". Однако можно также связать книги с отзывами, что будет представлять собой отношение "многие-ко-многим". "Правильный" выбор быстро становится несколько субъективным и, конечно, зависит от конкретных потребностей проекта.

В этом проекте мы будем рассматривать приложение отзывов как один ко многим между авторами и отзывами, поскольку это более простой подход.

Здесь мы снова сталкиваемся с выбором, как разработать наш проект. Добавим ли мы модель **Reviews** в наш существующий файл **books/models.py** или создадим специальное приложение для отзывов, на которое затем будем ссылаться? Давайте начнем с добавления модели **Reviews** в приложение **books.**

Code

```
# books/models.py
import uuid
from django.contrib.auth import get_user_model # new
from django.db import models
from django.urls import reverse

class Book(models.Model):
...
class Review(models.Model): # new
        book = models.ForeignKey(Book,on_delete=models.CASCADE,
            related_name='reviews',)
        review = models.CharField(max_length=255)
        author = models.ForeignKey(get_user_model(),on_delete=models.CASCADE,)
        def __str__(self):
            return self.review
```

Вверху в разделе импорта включите **get_user_model ,** который необходим для ссылки на нашу модель **CustomUser**, затем создайте выделенную модель **Review.** Поле **book -** это внешний ключ "один ко многим", который связывает **Book** с **Review**, и мы следуем стандартной практике, называя его так же, как и связанную модель. Поле отзыва содержит фактическое содержимое, которое, возможно, может быть текстовым полем, в зависимости от того, сколько места вы хотите предоставить для длины отзыва! На данный момент мы заставим отзывы быть короткими - не более **255** символов. А затем мы также свяжемся с полем автора для автоматического заполнения отзыва текущим пользователем.

Для всех отношений "многие-к-одному", таких как **ForeignKey**, мы также должны указать опцию **on_delete**. И мы используем **get_user_model** для ссылки на нашу собственную модель пользователя.

Создайте новый файл миграций для наших изменений, а затем запустите **migrate**, чтобы применить их.

Command Line

```
$ docker-compose exec web python manage.py makemigrations books
$ docker-compose exec web python manage.py migrate
```

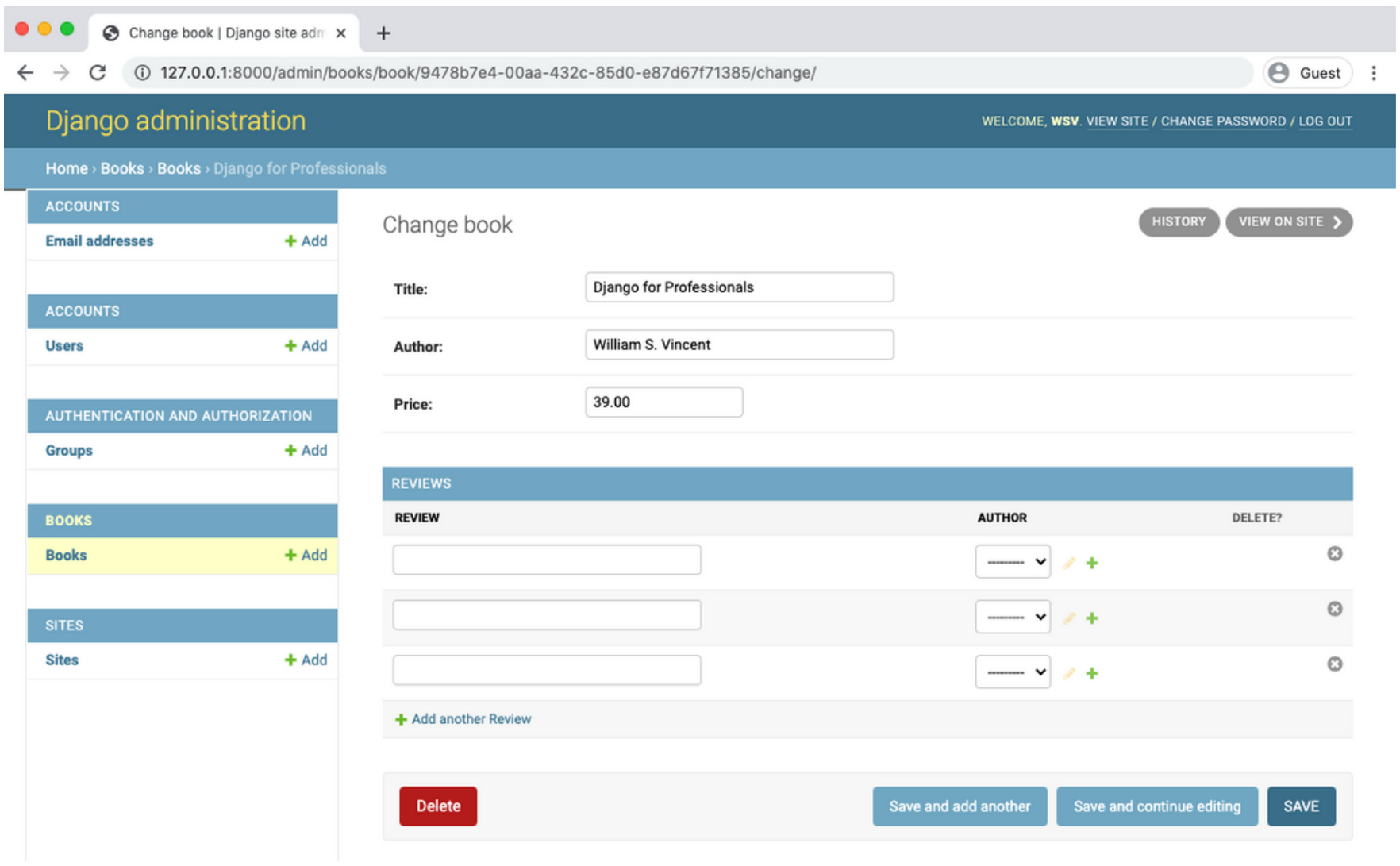
Admin

Чтобы приложение отзывов появилось в админке, нам нужно обновить **books/admin.py**, добавив модель **Review** и указав отображение **TabularInline.**

Code

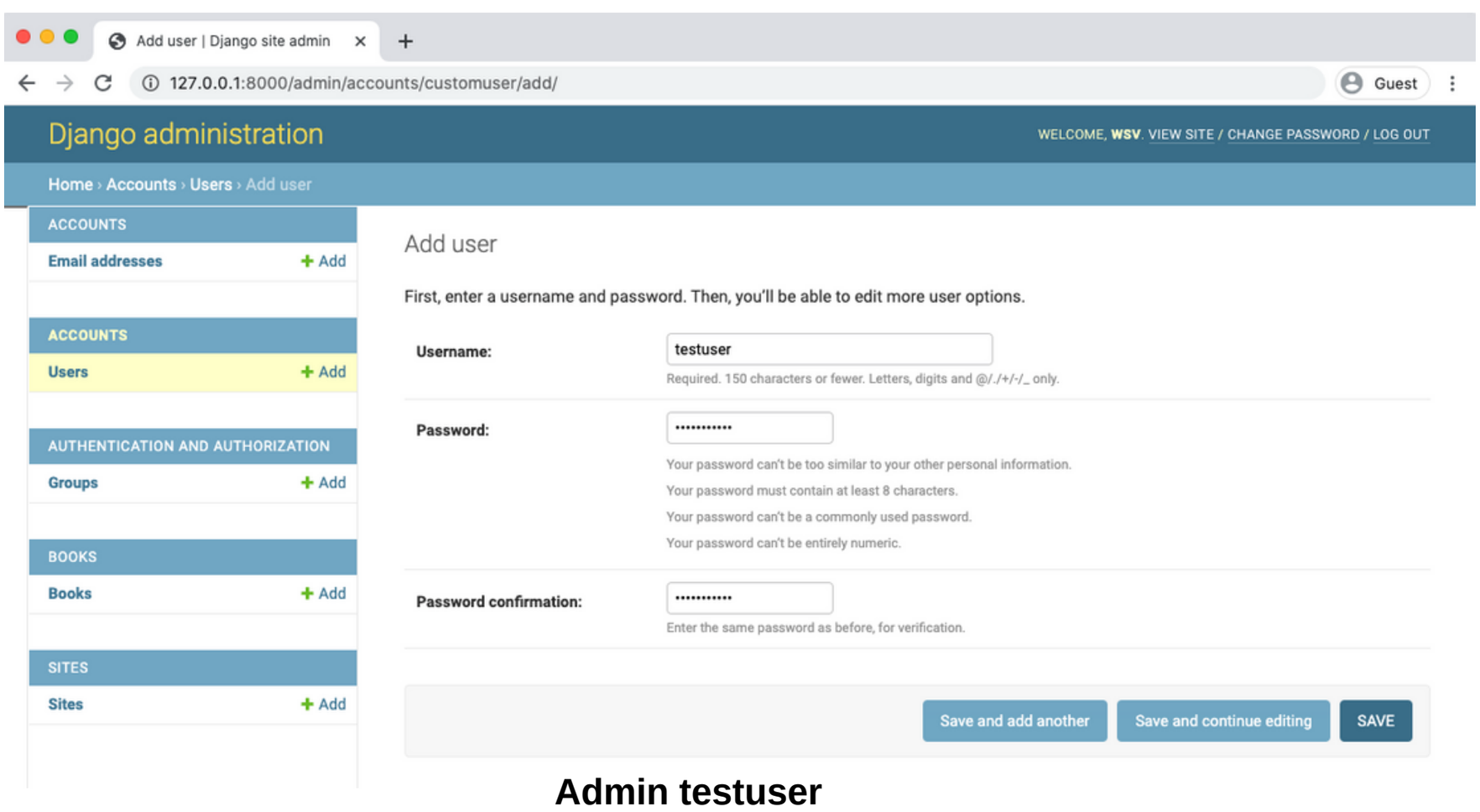
```
# books/admin.py
from django.contrib import admin
from .models import Book, Review

class ReviewInline(admin.TabularInline):
    model = Review
class BookAdmin(admin.ModelAdmin):
    inlines = [ReviewInline, ]
    list_display = ("title", "author", "price",)

admin.site.register(Book, BookAdmin)
```

Теперь перейдите в раздел книг на сайте **http://127.0.0.1:8000/admin/books/book/**, а затем нажмите на любую из книг, чтобы увидеть отзывы, отображаемые на странице отдельной книги.



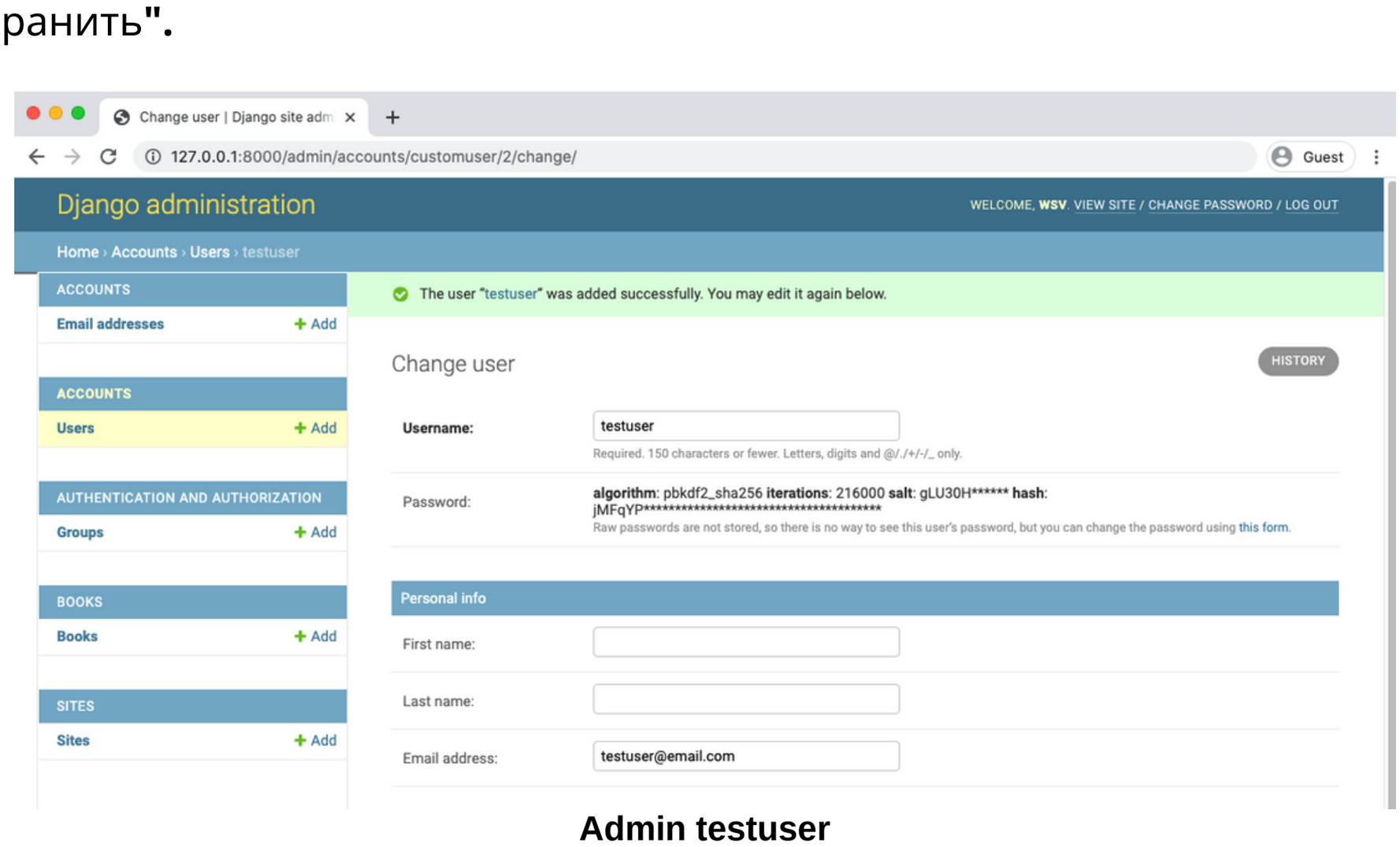
Django for Professionals Admin Reviews

На данный момент мы ограничены отзывами существующих пользователей, хотя ранее мы уже создавали **testuser@email.com**, который был удален, когда мы удалили монтирование тома базы данных в предыдущей главе. Есть два варианта добавления этой учетной записи: мы можем перейти на главный сайт и воспользоваться ссылкой **"Sign Up"**, или мы можем добавить ее непосредственно из админки. Давайте сделаем последнее. В разделе "Пользователи" на главной странице администратора нажмите на кнопку **"+** Добавить". Добавьте нового пользователя под именем **testuser.**



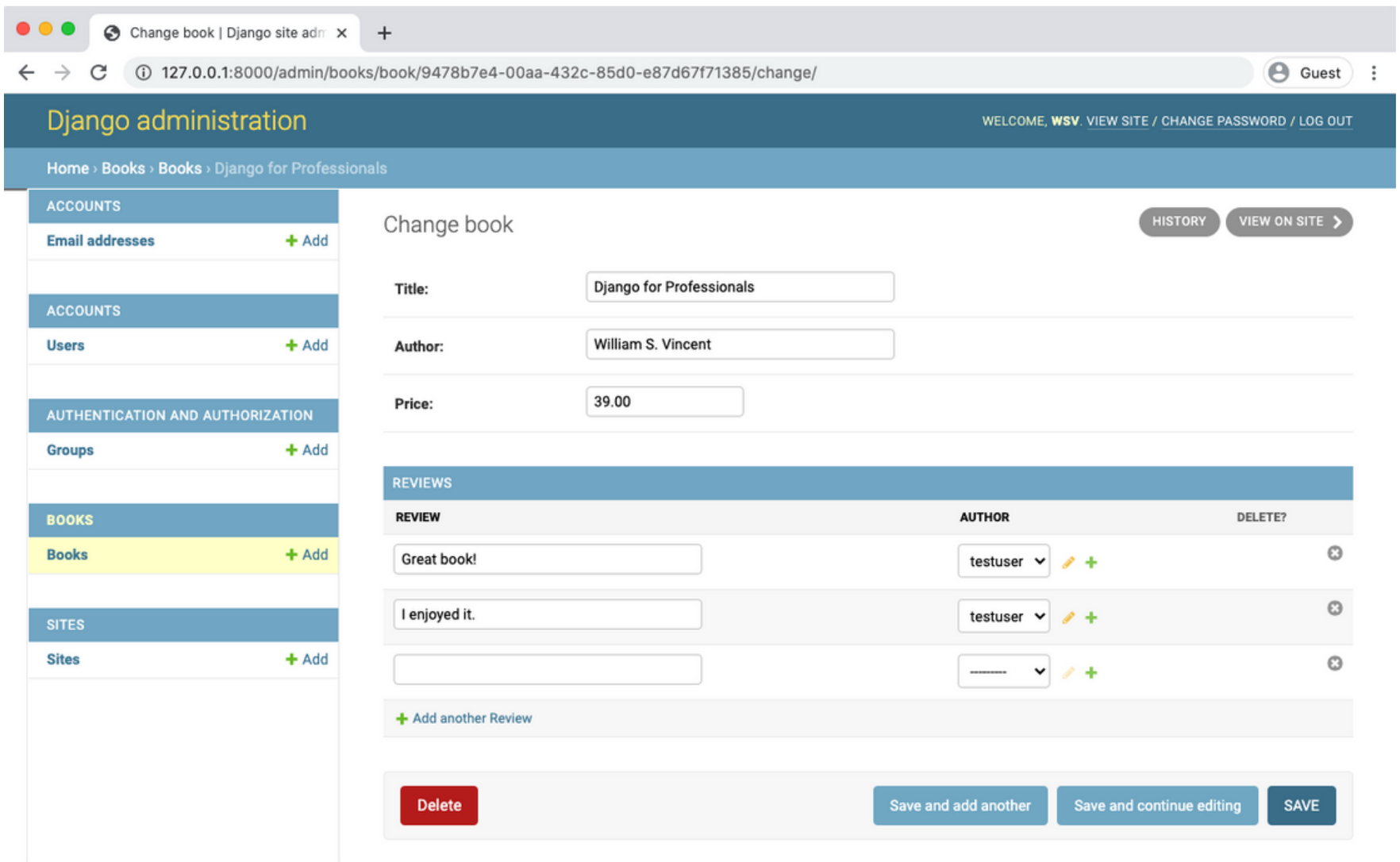
Admin testuser

Затем на следующей странице добавьте **testuser@email.com** в качестве адреса электронной почты**.** Прокрутите страницу до самого низа и нажмите кнопку "Сохранить"**.**



Admin testuser

Итак**,** наконец**,** мы можем добавить отзывы к книге **"Django** для профессионалов" с помощью **testuser .** Перейдите обратно в раздел **" Books "** и нажмите на нужную книгу**.** Напишите два отзыва и в качестве **AUTHOR** обязательно выберите **testuser .**



Добавить два отзыва

Templates

После создания модели отзывов пришло время обновить наши шаблоны для отображения отзывов на отдельной странице для каждой книги**.** Добавьте базовый раздел "Отзывы"**,** а затем переберите все существующие отзывы**.** Поскольку это отношение внешнего ключа**,** мы используем для этого **book.reviews.all.** Затем отобразим поле отзыва с помощью **review.review** и автора с помощью **review.author .**

Code

```
# templates/books/book_detail.html
{% extends '_base.html' %}
{% block title %}{{ book.title }}{% endblock title %}
{% block content %}
<div class="book-detail">
    <h2><a href="">{{ book.title }}</a></h2>
    <p>Author: {{ book.author }}</p>
    <p>Price: {{ book.price }}</p>
    <div>
        <h3>Reviews</h3>
        <ul>
            {% for review in book.reviews.all %}
            <li>{{ review.review }} ({{ review.author }})</li>
            {% endfor %}
        </ul>
    </div>
</div>
{% endblock content %}
```

Вот и все**!** Перейдите на индивидуальную страницу **"Django** для профессионалов"**,** чтобы увидеть результат**.** Ваш **url** будет отличаться от приведенного здесь**,** потому что мы используем **UUID.**

Reviews on Detail Page

Тесты

Пришло время для тестов. Нам нужно создать нового пользователя для нашего отзыва и добавить отзыв в метод **setUp** в нашем наборе тестов. Затем мы можем проверить, что объект книги содержит правильный отзыв.

Для этого нужно импортировать **get_user_model**, а также добавить модель **Review** сверху. Мы можем использовать **create_user** для создания нового пользователя под названием **reviewuser**, а затем объекта отзыва, который связан с нашим единственным объектом книги. Наконец, в разделе **test_book_detail_view** мы можем добавить дополнительный тест **assertContains** к объекту ответа.

Code

```
# books/tests.py
from django.contrib.auth import get_user_model # new
from django.test import TestCase
from django.urls import reverse
from .models import Book, Review # new

class BookTests(TestCase):
    def setUp(self):
        self.user = get_user_model().objects.create_user(username='reviewuser',
            email='reviewuser@email.com',password='testpass123')
        self.book = Book.objects.create(title='Harry Potter',author='JK Rowling',
            price='25.00',)
        self.review = Review.objects.create(book = self.book,author = self.user,
            review = 'An excellent review',)

    def test_book_listing(self):
        self.assertEqual(f'{self.book.title}', 'Harry Potter')
        self.assertEqual(f'{self.book.author}', 'JK Rowling')
        self.assertEqual(f'{self.book.price}', '25.00')

    def test_book_list_view(self):
        response = self.client.get(reverse('book_list'))
        self.assertEqual(response.status_code, 200)
        self.assertContains(response, 'Harry Potter')
        self.assertTemplateUsed(response, 'books/book_list.html')

    def test_book_detail_view(self):
        response = self.client.get(self.book.get_absolute_url())
        no_response = self.client.get('/books/12345/')
        self.assertEqual(response.status_code, 200)
        self.assertEqual(no_response.status_code, 404)
        self.assertContains(response, 'Harry Potter')
        self.assertContains(response, 'An excellent review') # new
        self.assertTemplateUsed(response, 'books/book_detail.html')
```

Если вы запустите тесты сейчас, все они должны пройти.

Command Line

```
$ docker-compose exec web python manage.py test
Creating test database for alias 'default'...
System check identified no issues (0 silenced).
.................
----------------------------------------------------------------------
Ran 17 tests in 0.675s
OK
Destroying test database for alias 'default'...
```

Git

Добавьте наши новые изменения кода в **Git** и включите сообщение о фиксации для главы.

Command Line

```
$ git status
$ git add .
$ git commit -m 'ch11'
```

Код для этой главы можно найти в официальном репозитории **Github**.

Заключение

Со временем мы могли бы обновить функциональность отзывов с помощью формы на самой странице, однако это означает **AJAX** вызовы с использованием **jQuery**, **React**, **Vue** или другого специализированного **JavaScript** фреймворка. К сожалению, полное освещение этого вопроса выходит за рамки данной книги.

По мере роста проекта может также иметь смысл разделить отзывы на отдельное приложение. Это очень субъективный выбор. В целом, придерживаться максимально простого подхода - добавлять внешние ключи в существующее приложение до тех пор, пока оно не станет слишком большим, чтобы в нем можно было легко разобраться, - вполне разумный подход.

В следующей главе мы добавим загрузку изображений на наш сайт, чтобы можно было создавать обложки для каждой книги.

Глава 12: Загрузка файлов/изображений

В главе **6** мы уже настраивали статические файлы, такие как изображения, но загружаемые пользователем файлы, например, обложки книг, несколько отличаются. Начнем с того, что **Django** называет первые статичными, в то время как все, что загружается пользователем, будь то файл или изображение, называется медиа.

Процесс добавления этой функции для файлов или изображений аналогичен, но для изображений необходимо установить **Python** библиотеку обработки изображений **Pillow**, которая включает дополнительные возможности, такие как базовая валидация.

Давайте установим **pillow** по уже знакомой схеме: установим его в **Docker**, остановим наши контейнеры и заставим собрать новый образ.

Command Line

```
$ docker-compose exec web pipenv install pillow==7.2.0
$ docker-compose down
$ docker-compose up -d --build
```

Медиафайлы

Принципиальная разница между статическими и медиафайлами заключается в том, что первым можно доверять, а вторым по умолчанию доверять нельзя. При работе с загружаемым пользователем контентом всегда возникают проблемы с безопасностью. В частности, важно проверять все загруженные файлы, чтобы убедиться, что они соответствуют заявленным характеристикам. Существует несколько неприятных способов, с помощью которых злоумышленник может атаковать сайт, слепо принимающий загруженные пользователем файлы.

Для начала давайте добавим две новые конфигурации в файл **config/settings.py**. По умолчанию **MEDIA_URL** и **MEDIA_ROOT** пустые и не отображаются, поэтому нам нужно их настроить:

- **MEDIA_ROOT -** абсолютный путь файловой системы к директории для загружаемых пользователем файлов.
- **MEDIA_URL -** это **URL**, который мы можем использовать в наших шаблонах для файлов

Мы можем добавить обе эти настройки после **STATICFILES_FINDERS** в нижней части файла **config/settings.py.** Мы будем использовать общепринятое соглашение, называя оба параметра **media .** Не забудьте включить косую черту **/** для **MEDIA_URL!**

Code

```
# config/settings.py
MEDIA_URL = '/media/' # new
MEDIA_ROOT = str(BASE_DIR.joinpath('media')) # new
```

Затем добавьте новый директорий под названием **media** и поддиректорию **covers** внутри него.

Command Line

```
$ mkdir media
$ mkdir media/covers
```

И, наконец, поскольку предполагается, что загруженный пользователем контент существует в производственном контексте, для локального отображения медиа элементов нам необходимо обновить **config/urls.py**, чтобы он показывал файлы локально. Для этого нужно импортировать **settings** и **static** в верхней части, а затем добавить дополнительную строку в нижней части.

Code

```
# config/urls.py
from django.conf import settings # new
from django.conf.urls.static import static # new
from django.contrib import admin
from django.urls import path, include

urlpatterns = [
        # Django admin
       path('admin/', admin.site.urls),
        # User management
        path('accounts/', include('allauth.urls')),
       # Local apps
        path('', include('pages.urls')),
        path('books/', include('books.urls')),

] + static(settings.MEDIA_URL, document_root=settings.MEDIA_ROOT) # new
```

Модели

С нашей общей конфигурацией медиа мы разобрались, теперь мы можем перейти к нашим моделям. Для хранения изображений мы будем использовать поле **ImageField** от **Django**, которое поставляется с некоторыми базовыми проверками обработки изображений.

Имя поля **- cover**, и мы указываем, что местоположение загруженного изображения будет в **MEDIA_ROOT/covers (**часть **MEDIA_ROOT** подразумевается на основе нашей предыдущей конфигурации **settings.py).**

Code

```
# books/models.py
class Book(models.Model):
        id = models.UUIDField( primary_key=True, default=uuid.uuid4,
                editable=False)
        title = models.CharField(max_length=200)
        author = models.CharField(max_length=200)
        price = models.DecimalField(max_digits=6, decimal_places=2)
        cover = models.ImageField(upload_to='covers/') # new
        def __str__(self):
                return self.title
        def get_absolute_url(self):
                return reverse('book_detail', kwargs={'pk': str(self.pk)})
```

> Если бы мы хотели разрешить загрузку обычного файла, а не файла изображения, то единственным отличием было бы изменение **ImageField** на **FileField.**

Поскольку мы обновили модель, пришло время создать файл миграций.

Command Line

```
$ docker-compose exec web python manage.py makemigrations books
You are trying to add a non-nullable field 'cover_image' to book
without a default; we can't do that (the database needs something to populate
existing rows).
Please select a fix:
  1. Provide a one-off default now (will be set on all existing rows with a null
     value for this column)
  2. Quit, and let me add a default in models.py
Select an option:
```

Ой! Что случилось? Мы добавляем новое поле базы данных, но у нас уже есть три записи в базе данных для каждой книги. Тем не менее, мы не смогли установить значение по умолчанию для обложки.
Чтобы исправить это, введите **2** для выхода, и мы добавим пустое поле, установленное в **True** для существующих изображений.

Code

```
# books/models.py
cover = models.ImageField(upload_to='covers/', blank=True) # new
```

> Часто можно увидеть, как **blank** и **null** используются вместе, чтобы установить значение по умолчанию для поля. Загвоздка в том, что тип поля **- ImageField** против **CharField** и так далее **-** диктует, как правильно их использовать, поэтому внимательно читайте документацию для дальнейшего использования.

Теперь мы можем создать файл миграции без ошибок.

Command Line

```
$ docker-compose exec web python manage.py makemigrations books
```

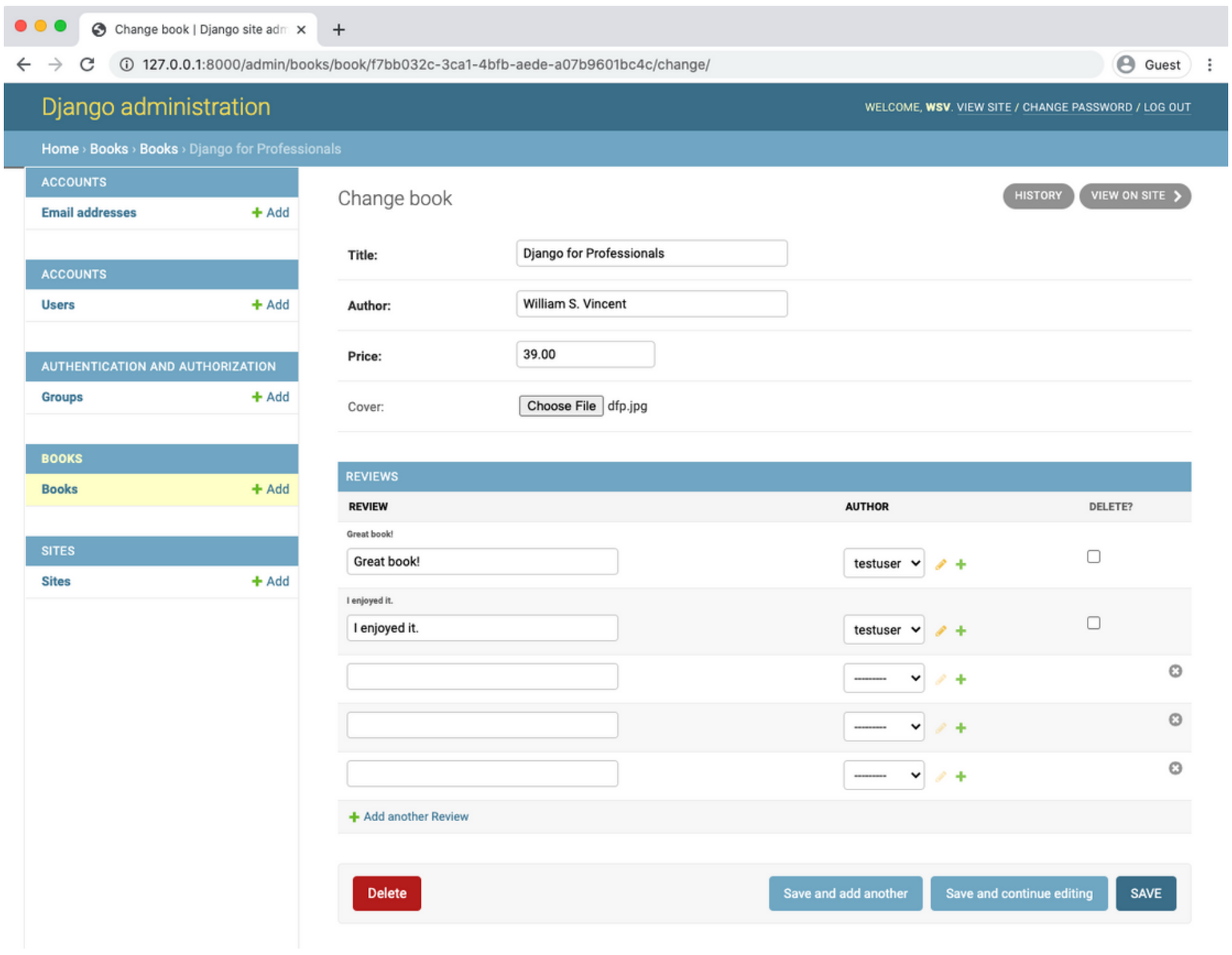
А затем примените миграцию к нашей базе данных.

Command Line

```
$ docker-compose exec web python manage.py migrate
```

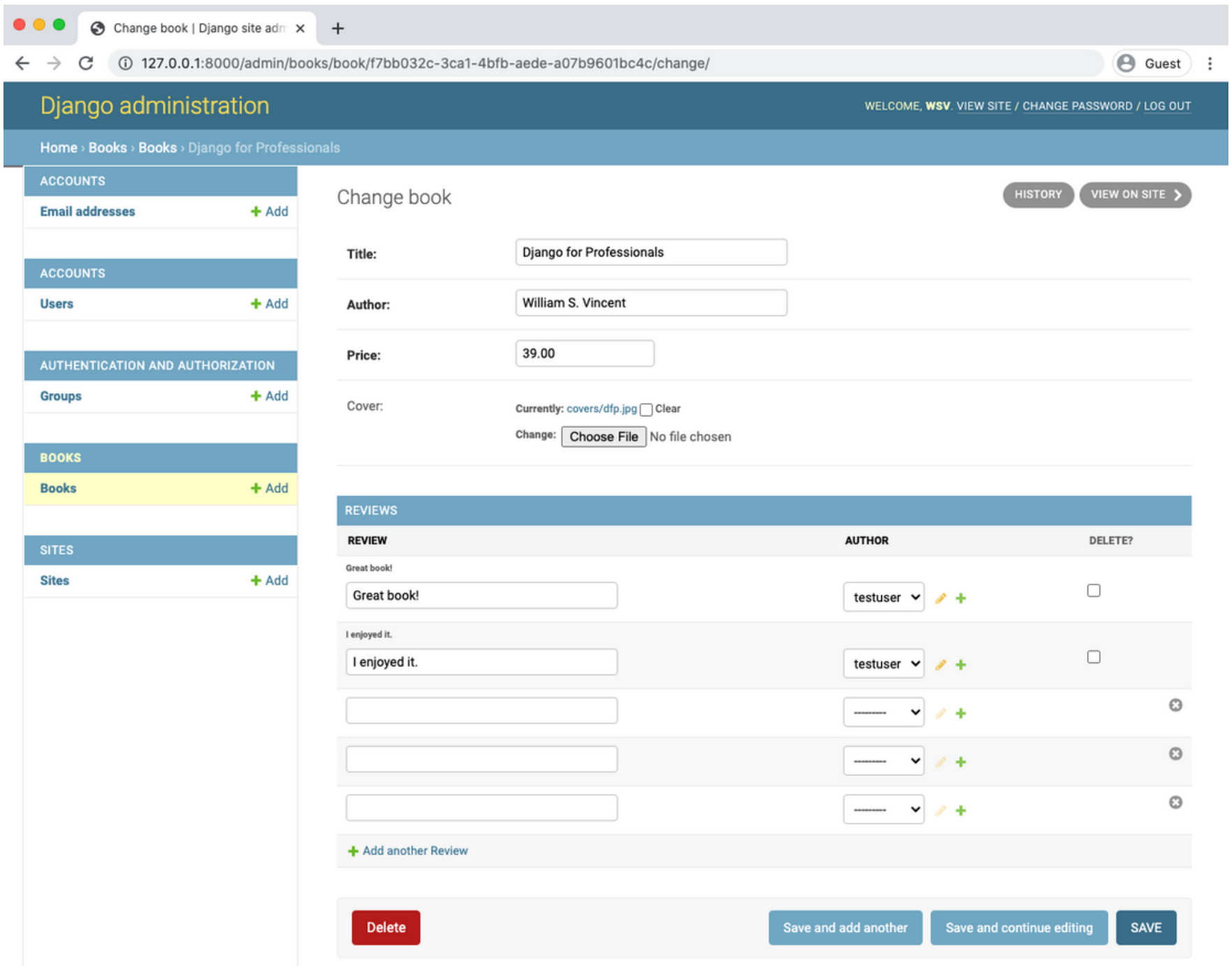
Admin

Мы в самом разгаре**!** Перейдите в админку и выберите запись для книги **"Django** для профессионалов**".** Поле обложки уже видно**,** и у нас уже есть его копия локально в **static/images/dfp.png,** поэтому используйте этот файл для загрузки**,** а затем нажмите кнопку "Сохранить" внизу справа**.**



Администратор добавляет обложку

Это приведет к перенаправлению в основной раздел "Книги"**.** Снова щелкните по ссылке для **"Django for Profesionals",** и мы увидим**,** что она уже существует в нужном нам месте **- covers/ .**



Администратор с обложкой

Шаблон

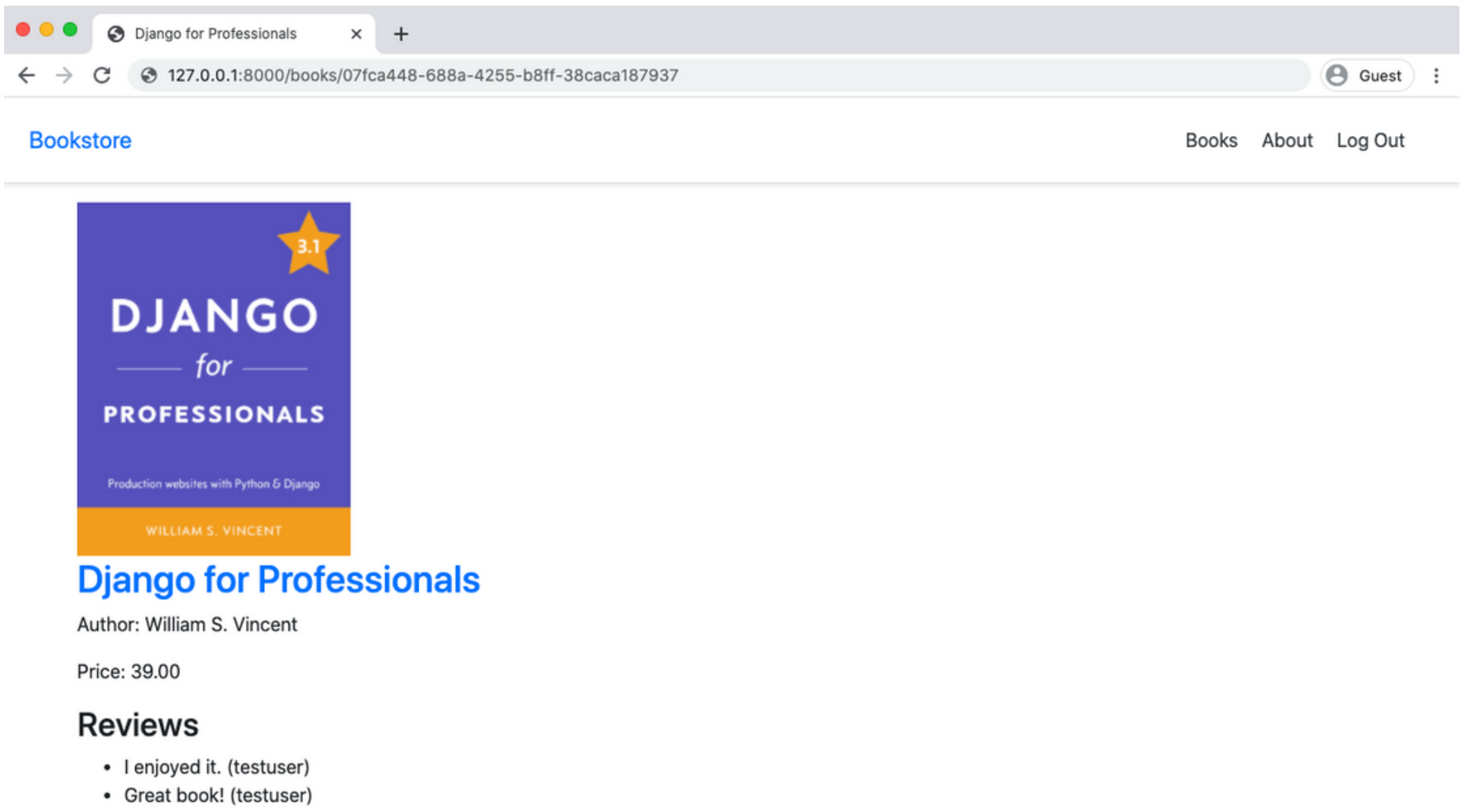
Итак, последний шаг. Давайте обновим наш шаблон, чтобы отображать обложку книги на отдельной странице. Маршрут будет **book.cover.url**, указывающий на местоположение обложки в нашей файловой системе.

Вот как выглядит обновленный файл **book_detail.html** с этим изменением в одну строку над заголовком

Code

```
# templates/books/book_detail.html
{% extends '_base.html' %}
{% block title %}{{ book.title }}{% endblock title %}
{% block content %}
<div class="book-detail">
    <img class="bookcover" src="{{ book.cover.url}}" alt="{{ book.title }}">
    <h2><a href="">{{ book.title }}</a></h2>
    <p>Author: {{ book.author }}</p>
    <p>Price: {{ book.price }}</p>
    <div>
      <h3>Reviews</h3>
      <ul>
        {% for review in book.reviews.all %}
        <li>{{ review.review }} ({{ review.author }})</li>
        {% endfor %}
      </ul>
    </div>
</div>
{% endblock content %}
```

Если вы сейчас зайдете на страницу **"Django** для профессионалов**",** то увидите там изображение обложки**!**



Cover image

Одна из возможных проблем заключается в том**,** что наш шаблон теперь ожидает наличия обложки**.** Если вы перейдете к одной из двух других книг**,** для которых мы не добавили обложку**,** вы увидите следующее сообщение об ошибке**.**

Ошибка изображения обложки

Мы должны добавить некоторую базовую логику в наш шаблон**,** чтобы в случае отсутствия обложки шаблон не искал ее**!** Это можно сделать с помощью оператора **if,** который проверяет наличие **book.cover** и отображает ее**,** если она существует**.**

Code

```
# templates/books/book_detail.html
{% extends '_base.html' %}
{% block title %}{{ book.title }}{% endblock title %}
{% block content %}
<div class="book-detail">
  {% if book.cover %}
  <img class="bookcover" src="{{ book.cover.url}}" alt="{{ book.title }}">
  {% endif %}
  <h2><a href="">{{ book.title }}</a></h2>
  ...
```

Если вы обновите страницу любой из книг, вы увидите, что они отображают правильную страницу, хотя и без обложки.

Следующие шаги

Существует несколько дополнительных шагов, которые было бы неплохо предпринять в рамках проекта, но они выходят за рамки данной книги. К ним относится добавление специальных форм создания/редактирования/удаления для создания книг и изображения обложки. Довольно длинный список дополнительных проверок может и должен быть размещен на форме загрузки изображения, чтобы гарантировать, что в базу данных будет добавлено только нормальное изображение.

Следующим шагом может быть хранение медиафайлов в специальной сети **CDN (Content Delivery Network)** для дополнительной безопасности. Это также может быть полезно для производительности на очень больших сайтах для статических файлов, но для медиафайлов это хорошая идея независимо от размера.

Наконец, здесь не помешали бы тесты, хотя они в первую очередь должны быть направлены на раздел проверки форм, а не на основную загрузку изображений через админку. Опять же, это область, которая может стать довольно сложной, но заслуживает дальнейшего изучения.

Git

Обязательно создайте новый **Git-**коммит для изменений в этой главе.

Command Line

```
$ git status
$ git add .
$ git commit -m 'ch12'
```

Как всегда, вы можете сравнить свой код с официальным исходным кодом на **Github.**

Заключение

В этой главе было показано, как добавлять пользовательские файлы в проект. На практике это просто, но дополнительный слой проблем безопасности делает эту область заслуживающей внимания при масштабировании.

В следующей главе мы добавим разрешения на наш сайт, чтобы заблокировать его.

Глава 13: Разрешения (Permissions)

В настоящее время для нашего проекта **Bookstore** не установлено никаких прав доступа. Любой пользователь, даже не вошедший в систему, может посетить любую страницу и выполнить любое доступное действие. Хотя это и хорошо для прототипирования, внедрение надежной структуры разрешений является обязательным условием перед развертыванием сайта в производство.

Django поставляется со встроенными опциями авторизации для блокировки страниц либо для зарегистрированных пользователей, либо для определенных групп, либо для пользователей с соответствующим индивидуальным разрешением.

Только для зарегистрированных пользователей

Смущает то, что существует несколько способов добавить даже самое базовое разрешение: ограничение доступа только для вошедших в систему пользователей. Это можно сделать необработанным способом с помощью декоратора **login_required()** или, поскольку мы пока используем представления на основе классов, с помощью миксина **LoginRequired**.

Давайте начнем с ограничения доступа к страницам Книги только для зарегистрированных пользователей. Для этого есть ссылка в навигационной панели, так что это не случай, когда пользователь случайно находит **URL** (что тоже может случиться); в данном случае **URL** является достаточно публичным.

Сначала импортируйте **LoginRequiredMixin** сверху, а затем добавьте его перед **ListView**, поскольку миксины загружаются слева направо. Таким образом, первое, что проверяется, это вошел ли пользователь в систему; если нет, то нет необходимости загружать **ListView**. Другая часть - это установка **login_url** для пользователя, на который он будет перенаправлен. Это имя **URL** для входа в систему, которое, поскольку мы используем **django-allauth**, будет **account_login** . Если бы мы использовали традиционную систему аутентификации **Django**, то эта ссылка называлась бы просто **login** .

Структура для **BookDetailView** такая же: добавьте **LoginRequiredMixin** и маршрут **login_url**.

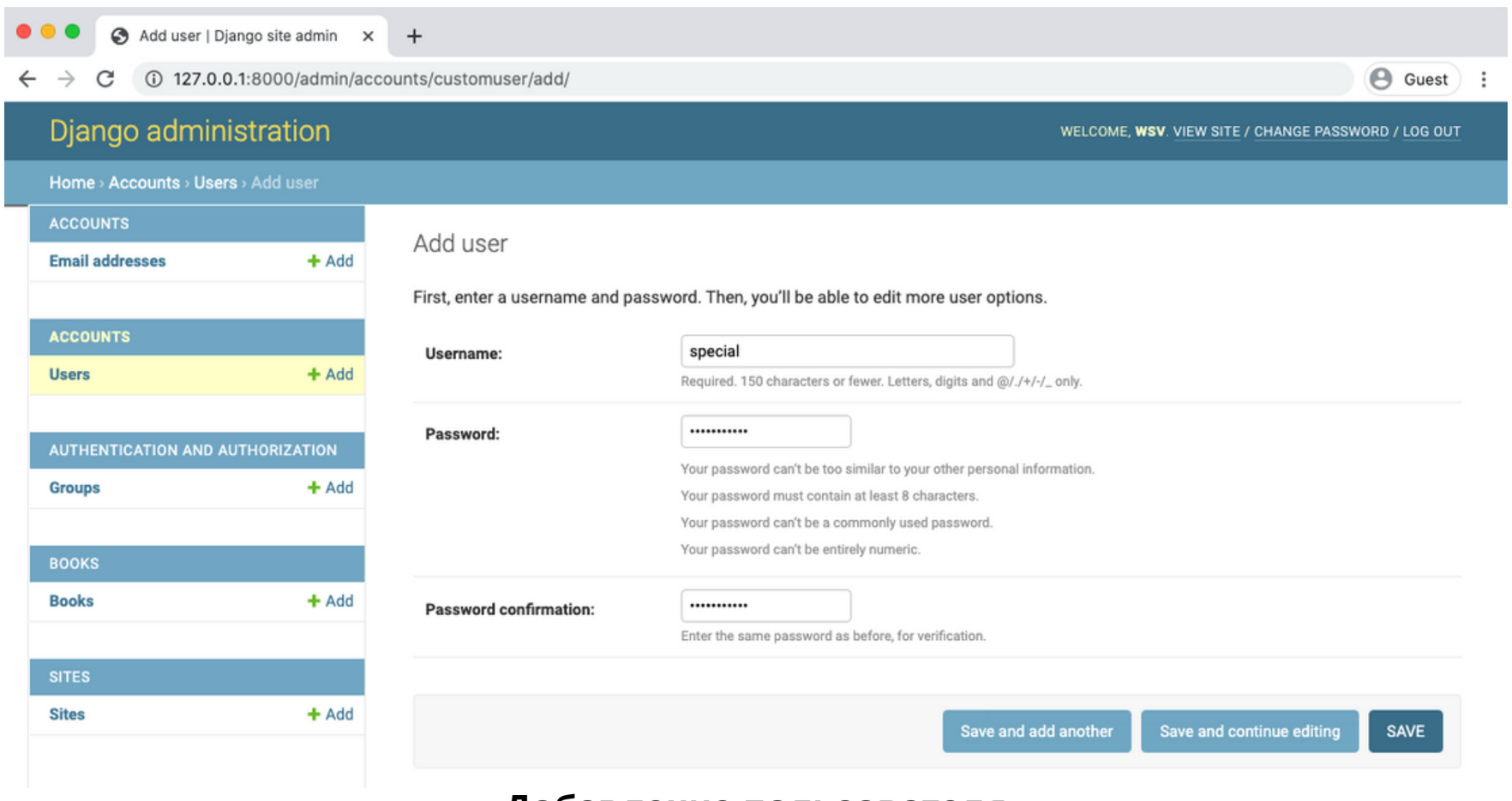
Code

```
# books/views.py
from django.contrib.auth.mixins import LoginRequiredMixin # new
from django.views.generic import ListView, DetailView
from .models import Book

class BookListView(LoginRequiredMixin, ListView): # new
        model = Book
        context_object_name = 'book_list'
        template_name = 'books/book_list.html'
        login_url = 'account_login' # new

class BookDetailView(LoginRequiredMixin, DetailView): # new
        model = Book
        context_object_name = 'book'
        template_name = 'books/book_detail.html'
        login_url = 'account_login' # new
```

И это все! Если вы сейчас выйдете из своей учетной записи и нажмете на ссылку "Книги", она автоматически перенаправит вас на страницу входа в систему. Однако если вы вошли в систему, страница "Книги" загрузится нормально. Даже если вы каким-то образом узнали **UUID** конкретной страницы книги, вы также будете перенаправлены на страницу **Log In!**

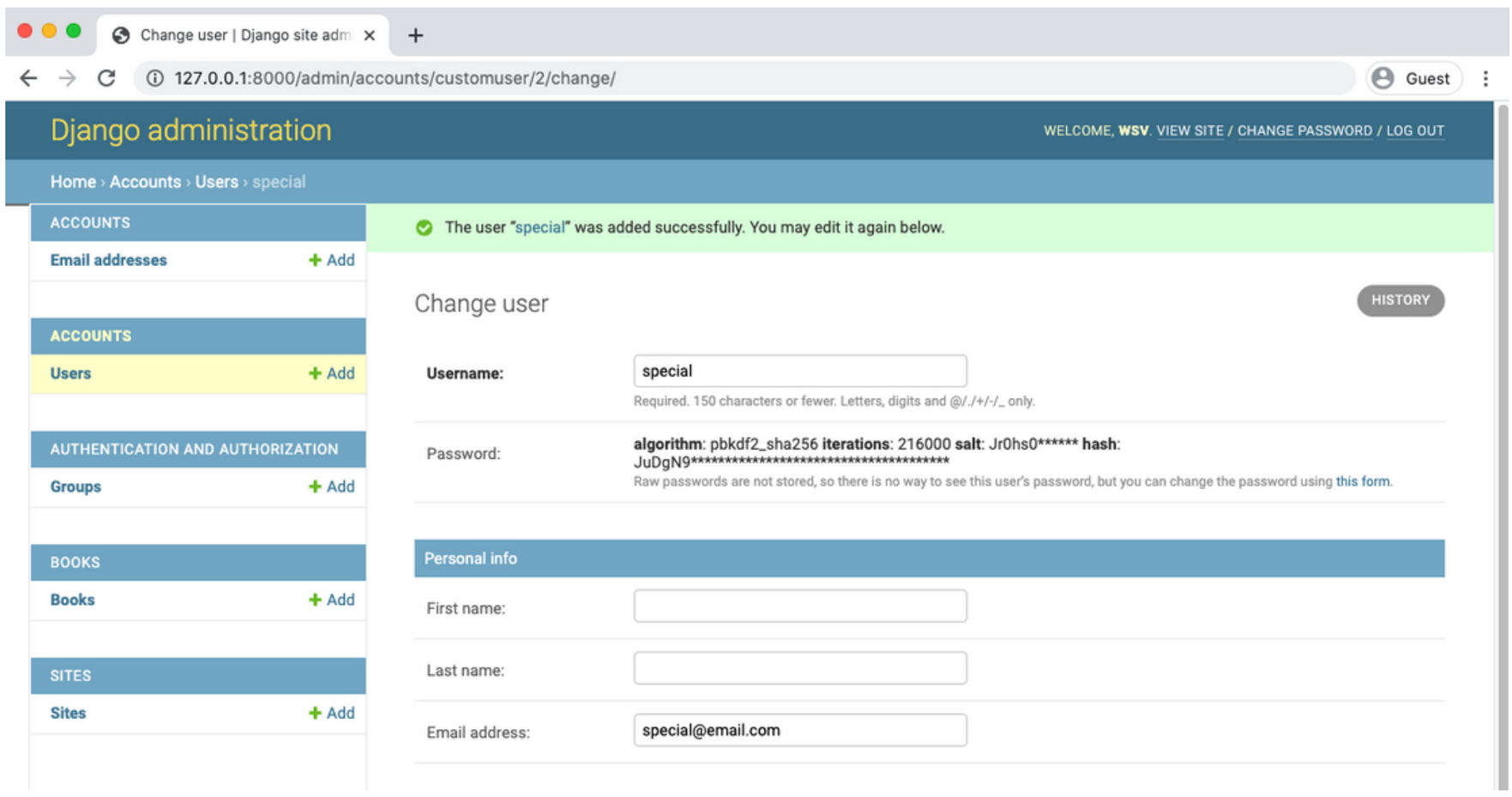
Разрешения (permissions)

Django поставляется с базовой системой разрешений, которая управляется через **Django admin.** Чтобы продемонстрировать ее, нам нужно создать новую учетную запись пользователя. Вернитесь на главную страницу администратора и нажмите на **"+** Добавить" рядом с разделом Пользователи **.**
Мы назовем этого нового пользователя **special** и зададим пароль **testpass123 .** Нажмите на кнопку "Сохранить".



Добавление пользователя

Вторая страница позволяет нам установить "Адрес электронной почты" на **special@email.com .** Мы используем **django-allauth,** так что наша страница входа требует только **email,** и страница регистрации также использует только **email,** но так как мы не настроили администратора, он все еще ожидает **username**



User Email

Если бы мы хотели полностью отказаться от стандартной системы пользователей, это означало бы использование **AbstractBaseUser** вместо **AbstractUser** в главе **3,** когда мы настраивали нашу пользовательскую модель пользователя.

Прокручивая страницу дальше в самый низ, вы найдете опции для установки прав доступа групп и пользователей. Это длинный список настроек по умолчанию, которые предоставляет **Django.** Сейчас мы не будем их использовать, поскольку в следующем разделе мы создадим пользовательское разрешение, поэтому просто нажмите на кнопку "Сохранить" в правом нижнем углу, чтобы наш адрес электронной почты был обновлен для учетной записи пользователя.

Собственные разрешения

Установка собственных разрешений - гораздо более распространенное явление в проекте **Django.** Мы можем установить их через класс **Meta** в наших моделях баз данных.

Например, давайте добавим специальный статус, чтобы автор мог читать все книги. Другими словами, они имеют доступ к **DetailView.** Мы можем быть гораздо более конкретными с разрешениями, ограничивая их для каждой книги, но это хороший первый шаг.

В файле **books/models.py** мы добавим класс **Meta** и зададим имя разрешения и описание, которое будет видно в админке.

Code

```
# books/models.py
...
class Book(models.Model):
        id = models.UUIDField(primary_key=True, default=uuid.uuid4,
             editable=False)
        title = models.CharField(max_length=200)
        author = models.CharField(max_length=200)
        price = models.DecimalField(max_digits=6, decimal_places=2)
        cover = models.ImageField(upload_to='covers/', blank=True)

        class Meta: # new
            permissions = [('special_status', 'Can read all books'), ]
        def __str__(self):
            return self.title
        def get_absolute_url(self):
            return reverse('book_detail', args=[str(self.id)])
...
```

> Порядок внутренних классов и методов здесь преднамеренный. Он следует разделу **Model style** из документации **Django**.

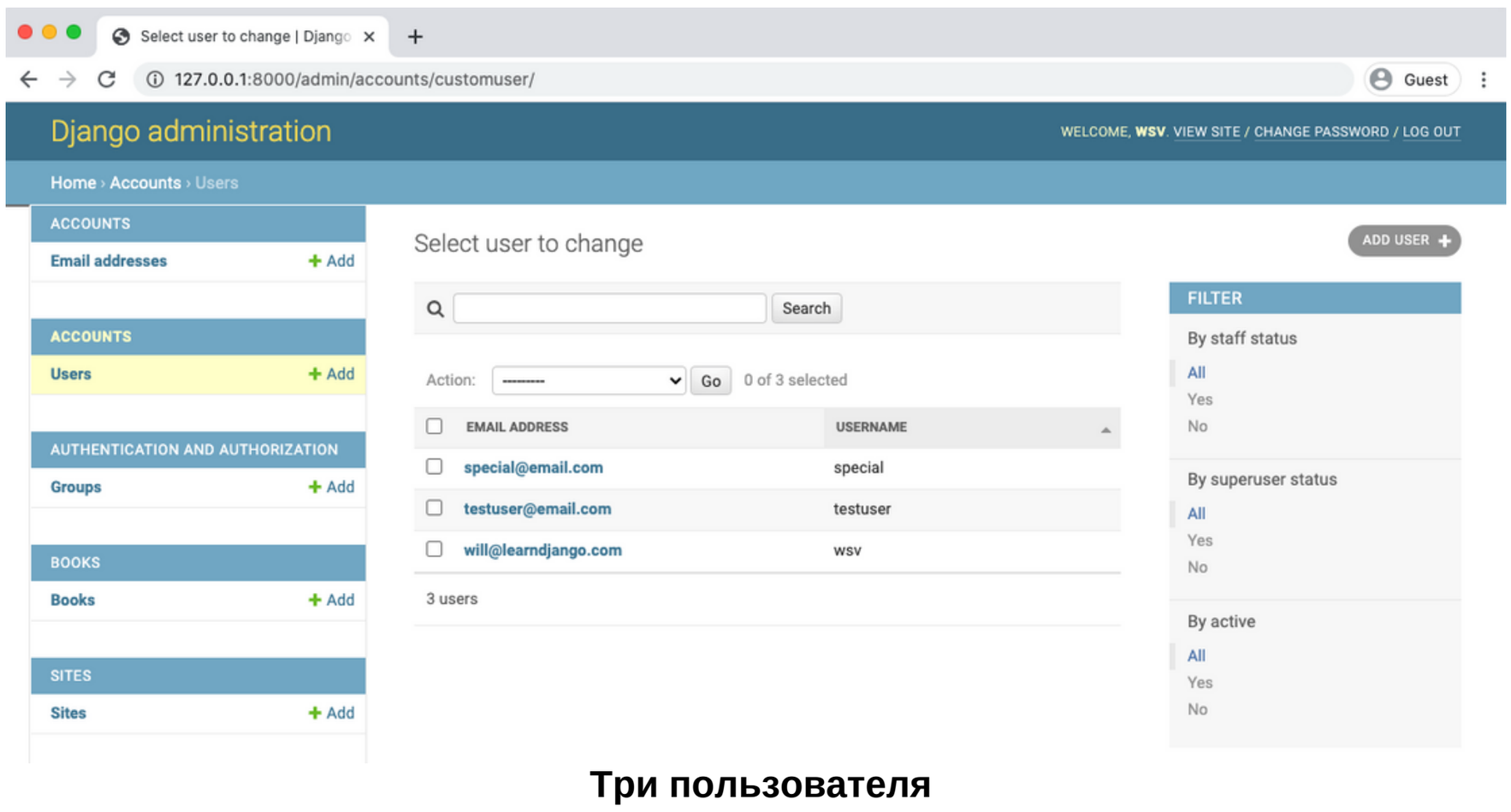
Поскольку мы обновили модель нашей базы данных, мы должны создать новый файл миграций и затем применить его.

Command Line

```
$ docker-compose exec web python manage.py makemigrations books
$ docker-compose exec web python manage.py migrate
```

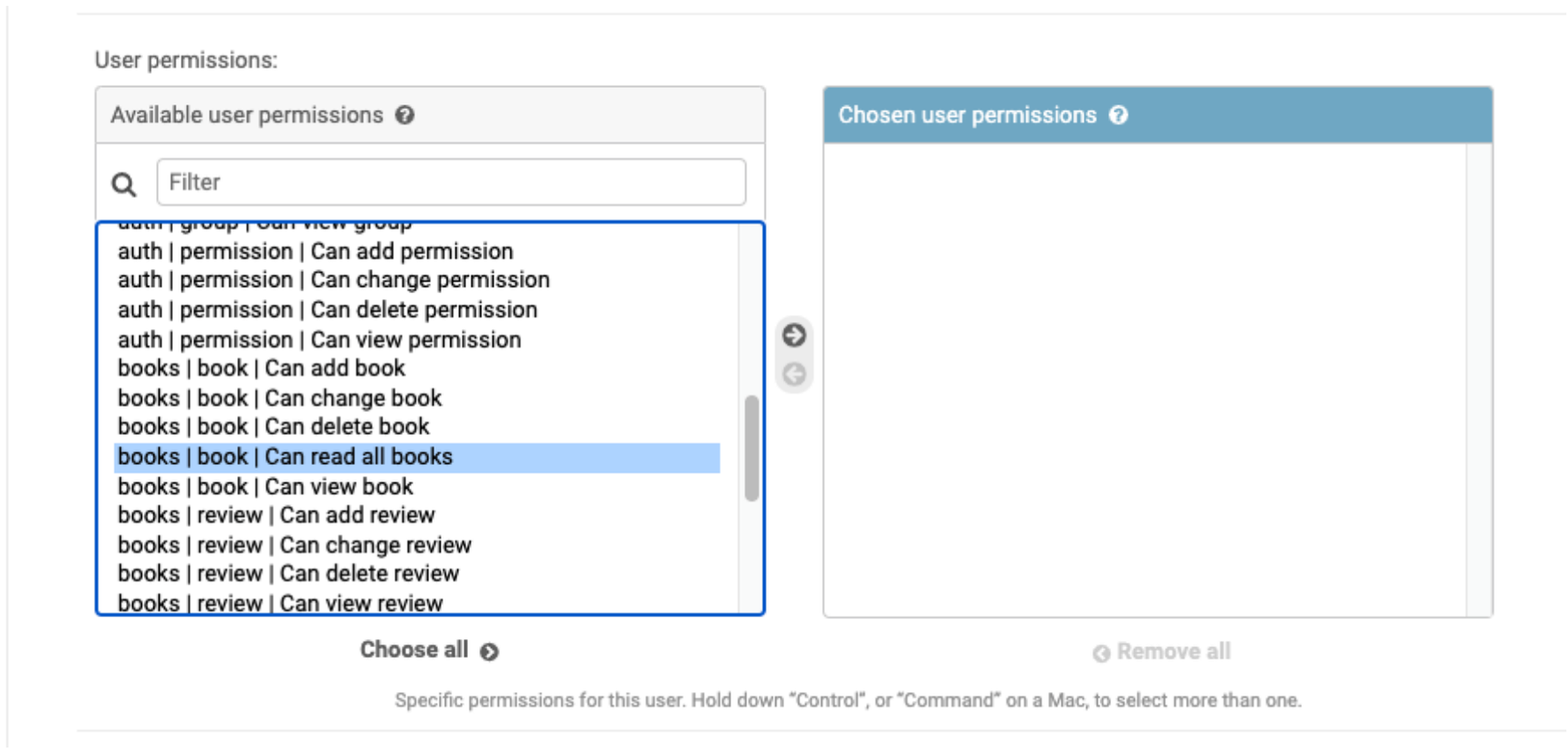
Разрешения пользователя

Теперь нам нужно применить это пользовательское разрешение к нашему новому специальному пользователю. Благодаря администратору это не является сложной задачей. Перейдите в раздел Пользователи, где перечислены три существующих пользователя: **special@email.com , testuser@email.com** и **will@learndjango.com,** который является моей учетной записью суперпользователя.



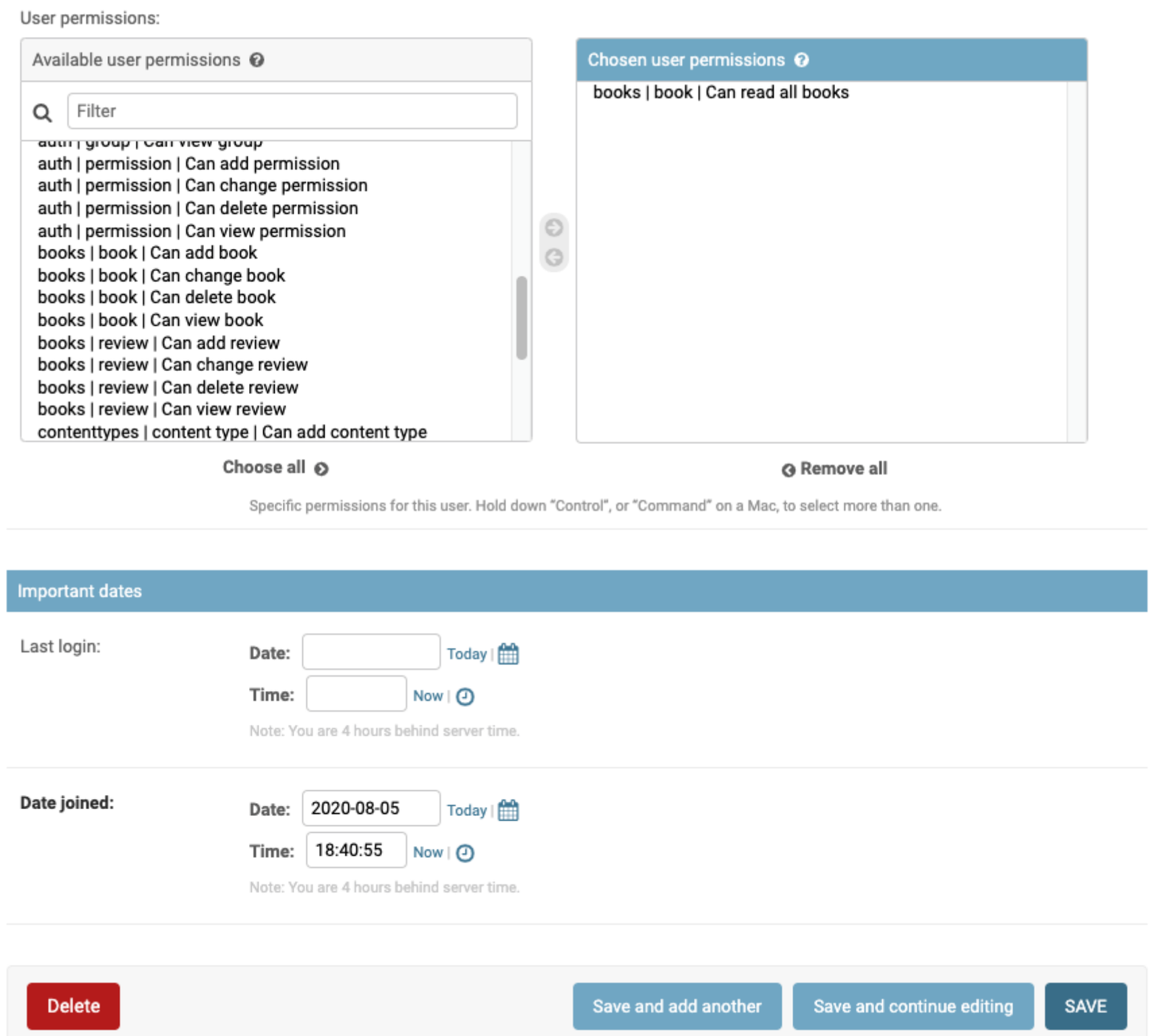
Три пользователя

Нажмите на пользователя **special@email.com,** а затем прокрутите вниз до раздела **" User permissions" (**Разрешения пользователя**)** в нижней части страницы**.** В нем найдите книги **| book | Can read all books** и выберите его**.**



Может читать все книги

Нажмите на стрелку **->,** чтобы добавить его в **" Select user permissions".** Не забудьте нажать кнопку "Сохранить" в нижней части страницы**.**



Добавление разрешения

PermissionRequiredMixin

Последний шаг - применение пользовательского разрешения с помощью **PermissionRequiredMixin**. Одна из многих замечательных особенностей представлений на основе классов заключается в том, что мы можем реализовать расширенную функциональность с очень небольшим количеством кода с нашей стороны, и этот конкретный миксин является хорошим примером этого.

Добавьте **PermissionRequiredMixin** в наш список импорта в верхней строке. Затем добавьте его в **DetailView** после **LoginRequiredMixin**, но перед **DetailView**. Порядок должен иметь смысл: если пользователь еще не вошел в систему, нет смысла делать дополнительную проверку наличия у него разрешения. Наконец, добавьте поле **permission_required**, которое определяет желаемое разрешение. В нашем случае оно называется **special_status** и существует в модели **books**.

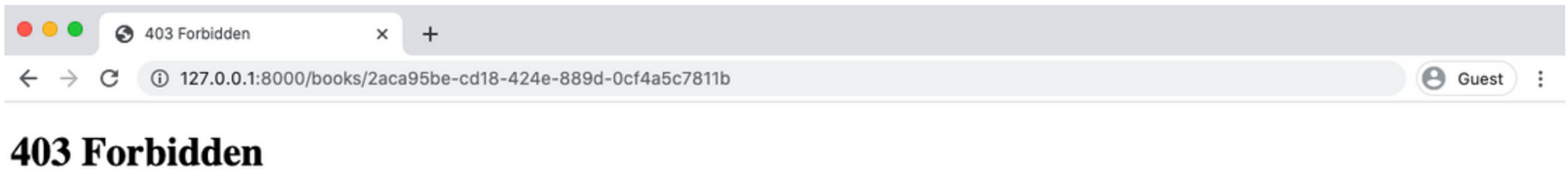
Code

```
# books/views.py
from django.contrib.auth.mixins import ( LoginRequiredMixin,
PermissionRequiredMixin) # new

from django.views.generic import ListView, DetailView
from .models import Book

class BookListView(LoginRequiredMixin, ListView):
    model = Book
    context_object_name = 'book_list'
    template_name = 'books/book_list.html'
    login_url = 'account_login'

class BookDetailView(LoginRequiredMixin,PermissionRequiredMixin, # new
    DetailView):
    model = Book
    context_object_name = 'book'
    template_name = 'books/book_detail.html'
    login_url = 'account_login'
    permission_required = 'books.special_status' # new
```

Можно добавить несколько разрешений через поле **permission_required**, хотя в данном случае мы этого не делаем.

Чтобы опробовать нашу работу, выйдите из админки. Это необходимо, потому что учетная запись суперпользователя используется для администратора и по умолчанию имеет доступ ко всему. Не самая лучшая учетная запись для тестирования!

Войдите на сайт книжного магазина, используя учетную запись **testuser@email.com**, а затем перейдите на страницу **Books**, где перечислены три доступных наименования. Если вы нажмете на любую из книг, вы увидите ошибку **"403 Forbidden"**, потому что разрешение было отклонено.



403 Error Page

Теперь вернитесь на главную страницу **http://127.0.0.1:8000/** и выйдите из системы**.** Затем войдите в систему**,** используя учетную запись **special@email.com.** Снова перейдите на страницу "Книги" и откройте страницу каждой отдельной книги**.**

Groups & UserPassesTestMixin

Третий доступный миксин разрешений **-** это **UserPassesTestMixin,** который ограничивает доступ к представлению только для пользователей**,** прошедших определенный тест**.**

В больших проектах большое значение приобретают группы**,** которые являются способом **Django** применять разрешения к категории пользователей**.** Если вы посмотрите на домашнюю страницу администратора**,** там есть специальный раздел **Groups,** где их можно добавить и установить права доступа**.** Это намного эффективнее**,** чем добавлять разрешения для каждого пользователя**.**

Например**,** если на вашем сайте есть премиум**-**раздел**,** то при обновлении пользователя можно перевести его в премиум**-**группу**,** и тогда он получит доступ к любому количеству специфических дополнительных разрешений**.**

Тесты

Хорошей идеей является проведение тестов всякий раз**,** когда вносятся изменения в код**.** В конце концов**,** весь смысл тестирования заключается в том**,** чтобы проверить**,** не вызвали ли мы непреднамеренно сбой в работе другой части приложения**.**

Command Line

```
$ docker-compose exec web python manage.py test
...
Ran 17 tests in 0.519s
FAILED (failures=2)
```

Оказалось**,** что у нас есть несколько неудачных тестов**!** В частности**,** **test_book_list_view** и **test_book_- detail_view** оба жалуются на код состояния **302,** означающий перенаправление**,** а не **200** для успеха**.** Это происходит потому**,** что мы только что добавили требование**,** что для просмотра списка книг требуется вход в систему**,** а для страницы с подробной информацией пользователь должен иметь разрешение **special_status.**

Первый шаг **-** импортировать **Permission** из встроенных моделей **auth.** Затем в наших **BookTests** в **books/tests.py** добавим разрешение **special_status** в метод **setUp,** чтобы оно было доступно для всех наших тестов**.** Мы перенесем существующий единственный тест **test_book_list_view** в один для вошедших пользователей и один для вышедших**.** И мы обновим тест просмотра подробностей**,** чтобы проверить**,** есть ли у пользователя правильное разрешение**.**

Code

```
# books/tests.py
from django.contrib.auth import get_user_model
from django.contrib.auth.models import Permission # new
from django.test import Client, TestCase
from django.urls import reverse
from .models import Book, Review

class BookTests(TestCase):
        def setUp(self):
                self.user = get_user_model().objects.create_user(
                        username='reviewuser',
                        email='reviewuser@email.com',
                        password='testpass123')
                self.special_permission = Permission.objects.get(
                        codename='special_status') # new
                self.book = Book.objects.create(title='Harry Potter',author='JK
                        Rowling',price='25.00',)
                self.review = Review.objects.create(book = self.book, author = self.user,
                        review = 'An excellent review',)

        def test_book_listing(self):
            ...

        def test_book_list_view_for_logged_in_user(self): # new
                self.client.login(email='reviewuser@email.com', password='testpass123')
                response = self.client.get(reverse('book_list'))
                self.assertEqual(response.status_code, 200)
                self.assertContains(response, 'Harry Potter')
                self.assertTemplateUsed(response, 'books/book_list.html')

        def test_book_list_view_for_logged_out_user(self): # new
                self.client.logout()
                response = self.client.get(reverse('book_list'))
                self.assertEqual(response.status_code, 302)
                self.assertRedirects(
                        response, '%s?next=/books/' % (reverse('account_login')))
                response = self.client.get('%s?next=/books/' % (reverse('account_login')))
                self.assertContains(response, 'Log 0In')

        def test_book_detail_view_with_permissions(self): # new
                self.client.login(email='reviewuser@email.com', password='testpass123')
                self.user.user_permissions.add(self.special_permission)
                response = self.client.get(self.book.get_absolute_url())
                no_response = self.client.get('/books/12345/')
                self.assertEqual(response.status_code, 200)
                self.assertEqual(no_response.status_code, 404)
                self.assertContains(response, 'Harry Potter')
                self.assertContains(response, 'An excellent review')
                self.assertTemplateUsed(response, 'books/book_detail.html')
```

Code

```
# books/tests.py
from django.contrib.auth import get_user_model
from django.contrib.auth.models import Permission # new
from django.test import Client, TestCase
from django.urls import reverse
from .models import Book, Review

class BookTests(TestCase):
        def setUp(self):
                self.user = get_user_model().objects.create_user(
                        username='reviewuser',
                        email='reviewuser@email.com',
                        password='testpass123')
                self.special_permission = Permission.objects.get(
                        codename='special_status') # new
                self.book = Book.objects.create(title='Harry Potter',author='JK
                        Rowling',price='25.00',)
                self.review = Review.objects.create(book = self.book, author = self.user,
                        review = 'An excellent review',)

        def test_book_listing(self):
            ...

        def test_book_list_view_for_logged_in_user(self): # new
                self.client.login(email='reviewuser@email.com', password='testpass123')
                response = self.client.get(reverse('book_list'))
                self.assertEqual(response.status_code, 200)
                self.assertContains(response, 'Harry Potter')
                self.assertTemplateUsed(response, 'books/book_list.html')

        def test_book_list_view_for_logged_out_user(self): # new
                self.client.logout()
                response = self.client.get(reverse('book_list'))
                self.assertEqual(response.status_code, 302)
                self.assertRedirects(
                        response, '%s?next=/books/' % (reverse('account_login')))
                response = self.client.get('%s?next=/books/' % (reverse('account_login')))
                self.assertContains(response, 'Log 0In')

        def test_book_detail_view_with_permissions(self): # new
                self.client.login(email='reviewuser@email.com', password='testpass123')
                self.user.user_permissions.add(self.special_permission)
                response = self.client.get(self.book.get_absolute_url())
                no_response = self.client.get('/books/12345/')
                self.assertEqual(response.status_code, 200)
                self.assertEqual(no_response.status_code, 404)
                self.assertContains(response, 'Harry Potter')
                self.assertContains(response, 'An excellent review')
                self.assertTemplateUsed(response, 'books/book_detail.html')
```

Если вы запустите набор тестов снова, все тесты должны пройти.

Command Line

```
$ docker-compose exec web python manage.py test
...
Ran 18 tests in 0.944s
OK
```

Git

Обязательно создайте новый **Git-**коммит для изменений в этой главе**.**

Command Line

```
$ git status
$ git add .
$ git commit -m 'ch13'
```

Как всегда, вы можете сравнить свой код с официальным исходным кодом на **Github.**

Заключение

Разрешения и группы **-** это очень субъективная область, которая сильно варьируется от проекта к проекту**.** Однако основы остаются неизменными и повторяют то, что мы рассмотрели здесь**.** Первым шагом обычно является ограничение доступа только для вошедших в систему пользователей**,** а затем добавление дополнительных пользовательских разрешений вокруг групп или пользователей**.** В следующей главе мы добавим функцию поиска на сайт книжного магазина**.**

Глава 14: Поиск

Поиск - это фундаментальная функция большинства веб-сайтов и, конечно, всех, связанных с электронной коммерцией, как наш книжный магазин. В этой главе мы узнаем, как реализовать базовый поиск с помощью форм и фильтров. Затем мы улучшим его с помощью дополнительной логики и рассмотрим способы более глубокого использования возможностей поиска в **Django.** Сейчас в нашей базе данных всего три книги, но приведенный здесь код можно масштабировать до любого количества книг.

Функциональность поиска состоит из двух частей: формы для передачи поискового запроса пользователя и страницы результатов, которая выполняет фильтр на основе этого запроса. Определение "правильного" типа фильтра - вот где поиск становится интересным и трудным. Но сначала нам нужно создать и форму, и страницу результатов поиска.

Мы можем начать с любой из них, но сначала настроим фильтрацию, а затем форму.

Страница результатов поиска

Мы начнем со страницы результатов поиска. Как и для всех страниц **Django**, это означает добавление специального **URL**, представления и шаблона. Порядок реализации не имеет особого значения, но мы добавим их именно в таком порядке.

В **books/urls.py** добавьте путь **search/** с именем **URL search_results,** который использует представление **SearchResultsListView.**

Code

```
# books/urls.py
from django.urls import path
from .views import BookListView, BookDetailView, SearchResultsListView # new

urlpatterns = [
    path('', BookListView.as_view(), name='book_list'),
    path('<uuid:pk>', BookDetailView.as_view(), name='book_detail'),
    path('search/', SearchResultsListView.as_view(), name='search_results'),
]
```

Следующим является представление **SearchResultsListView,** которое на данный момент представляет собой список всех доступных книг. Это главный кандидат на использование **ListView .** Его шаблон будет называться **search_results.html** и находиться в каталоге **templates/books/.** Единственный новый код **-** для **SearchResultsListView,** поскольку мы уже импортировали и **ListView,** и модель **Book** в верхней части файла.

Code

```
# books/views.py
...
class SearchResultsListView(ListView): # new
        model = Book
        context_object_name = 'book_list'
        template_name = 'books/search_results.html'
```

И последнее **-** наш шаблон **search_results.html,** который необходимо создать**.**

Command Line

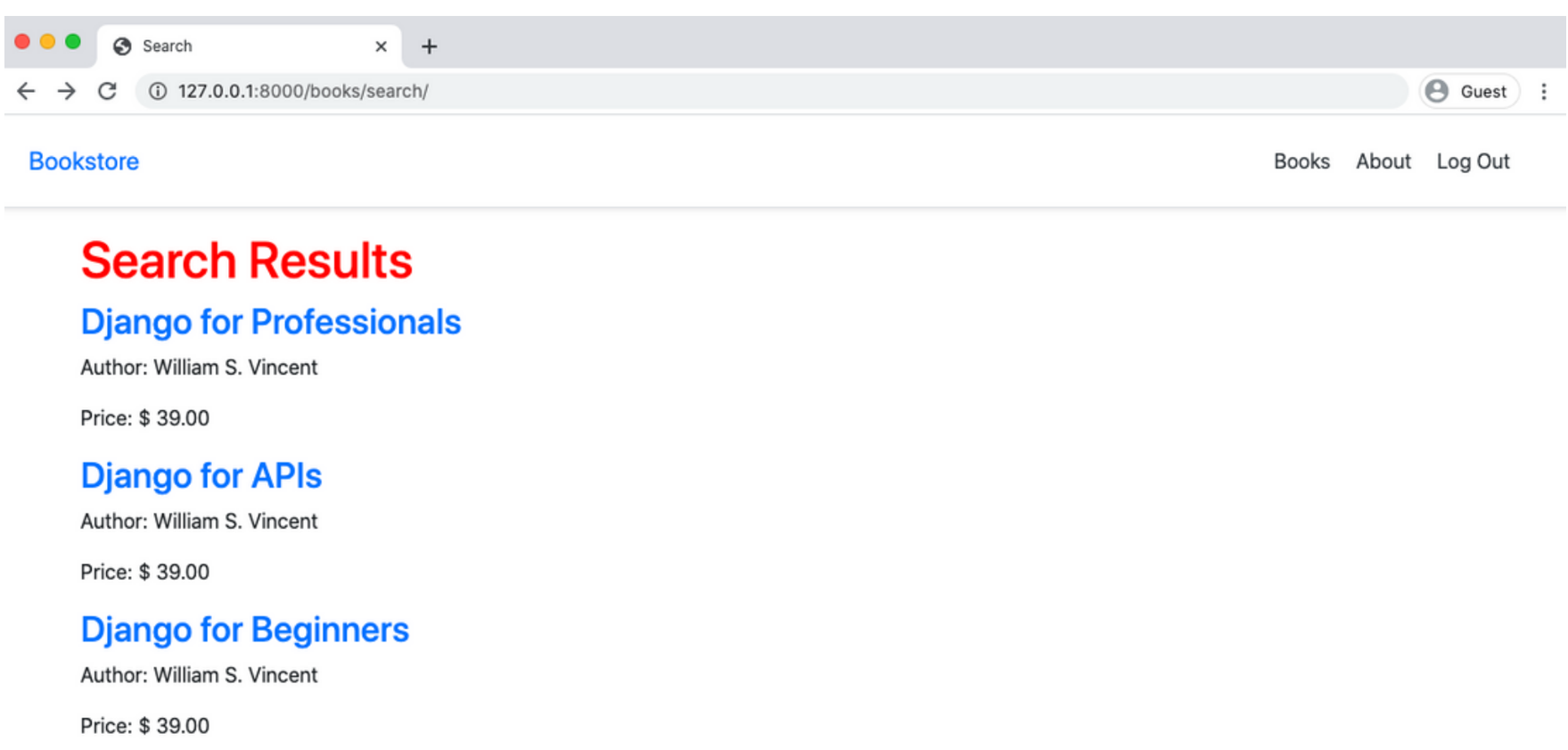
```
$ touch templates/books/search_results.html
```

На данный момент он выводит список всех доступных книг по названию**,** автору и цене**.**

Code

```
<!-- templates/books/search_results.html -->
{% extends '_base.html' %}
{% block title %}Search{% endblock title %}
{% block content %}
<h1>Search Results</h1>
{% for book in book_list %}
<div>
   <h3><a href="{{ book.get_absolute_url }}">{{ book.title }}</a></h3>
   <p>Author: {{ book.author }}</p>
   <p>Price: $ {{ book.price }}</p>
</div>
{% endfor %}
{% endblock content %}
```

Если вы все еще вошли в учетную запись пользователя**,** выйдите из нее**.**
Страница результатов поиска теперь доступна по адресу
http://127.0.0.1:8000/books/search/ .



Страница поиска

А вот и он**!**

Базовая фильтрация

В **Django QuerySet** используется для фильтрации результатов из модели базы данных. В настоящее время наша страница результатов поиска не похожа на такую страницу, потому что она выводит все результаты из модели **Book**. В конечном итоге мы хотим запустить фильтр на основе поискового запроса пользователя, но сначала мы рассмотрим несколько вариантов фильтрации.

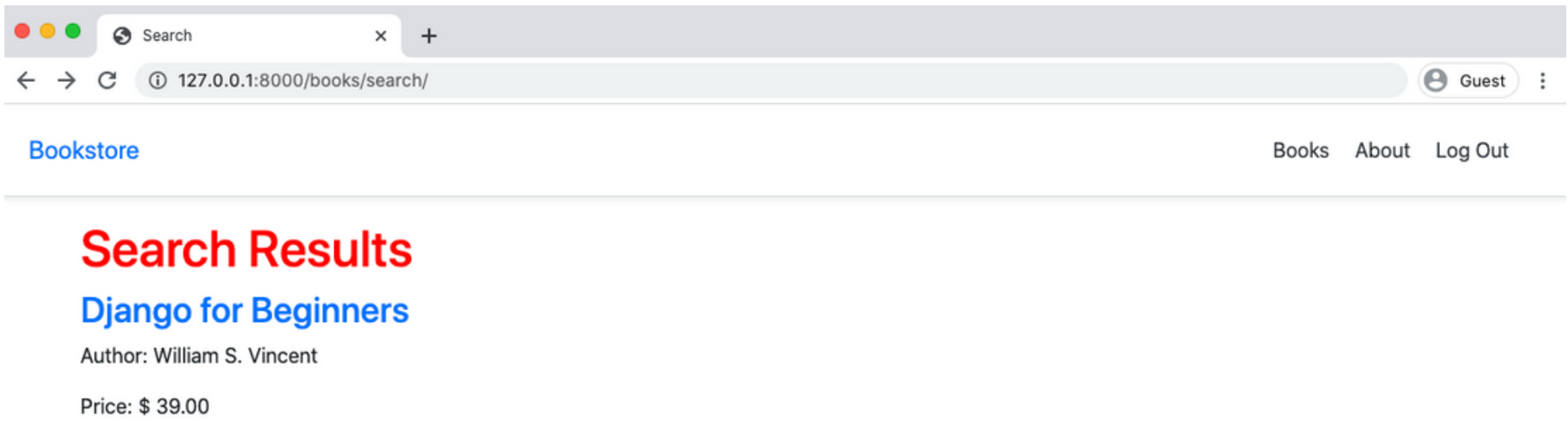
Оказывается, существует множество способов настройки набора запросов, в том числе через менеджер в самой модели, но для простоты мы можем добавить фильтр всего одной строкой. Так что давайте сделаем это!

Мы можем переопределить атрибут **queryset** по умолчанию для **ListView**, который по умолчанию показывает все результаты. Документация по **queryset** довольно надежна и подробна, но часто использование **contains** (который чувствителен к регистру) или **icontains** (который не чувствителен к регистру) является хорошей отправной точкой. Мы будем реализовывать фильтр на основе заголовков, которые "содержат" название "начинающие".

Code

```
# books/views.py
class SearchResultsListView(ListView):
        model = Book
        context_object_name = 'book_list'
        template_name = 'books/search_results.html'
        queryset = Book.objects.filter(title__icontains='beginners') # new
```

Обновите страницу результатов поиска, и теперь отображается только книга с названием, содержащим "**beginners**". Успешно!



Страница поиска для "beginners"

Для базовой фильтрации в большинстве случаев достаточно встроенных методов **queryset filter() , all() , get()** или **exclude().** Однако существует также очень надежный и подробный **API QuerySet,** который заслуживает дальнейшего изучения**.**

Q Objects

Использование **filter()** является мощным средством**,** и можно даже объединять фильтры в цепочки**,** например**,** искать все названия**,** содержащие **"beginners"** и **"django".** Однако часто вам нужны более сложные фильтры**,** которые могут использовать **"OR",** а не только **"AND";** вот тогда**-**то и приходит время обратиться к **Q-**объектам**.**

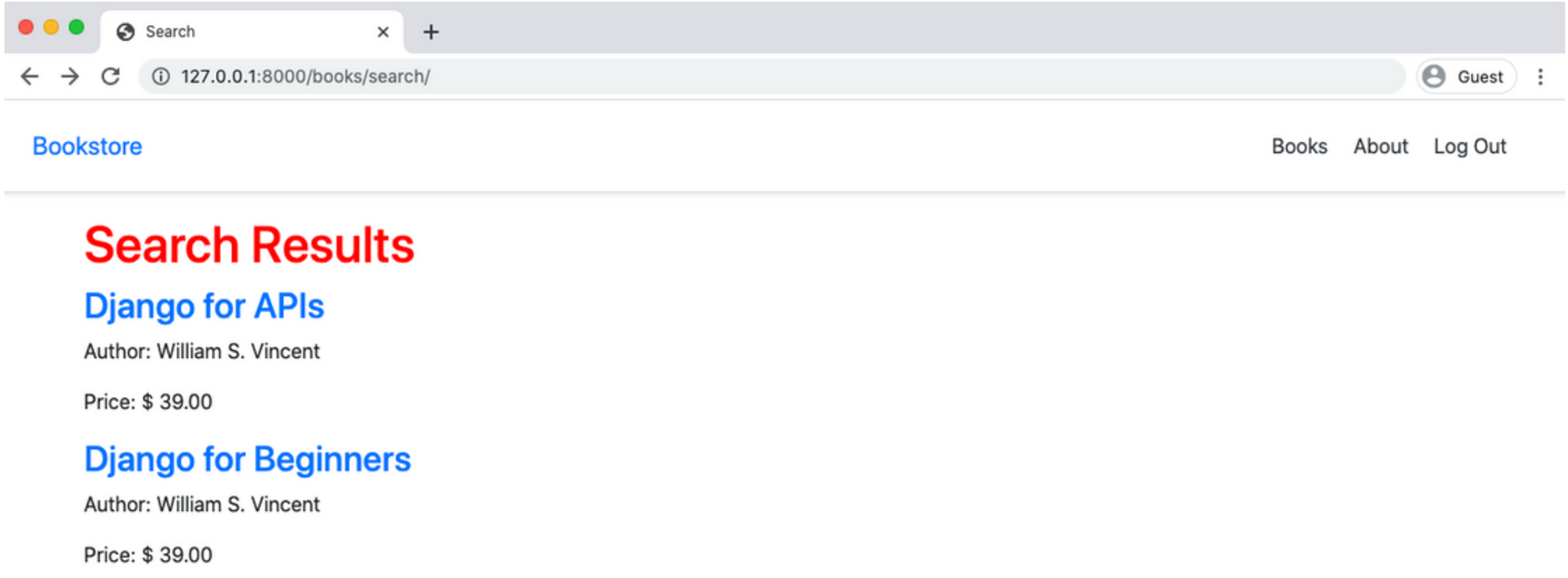
Вот пример**,** в котором мы задаем фильтр для поиска результатов**,** соответствующих заголовкам **"beginners"** или **"api".** Это так же просто**,** как импортировать **Q** в верхней части файла**,** а затем слегка подправить наш существующий запрос**.** Символ | обозначает оператор "или"**.** Мы можем фильтровать по любому доступному полю**:** не только по названию**,** но и по автору или цене**.**

При увеличении количества фильтров может оказаться полезным разделить переопределение **queryset** с помощью **get_- queryset() .** Именно так мы и поступим**,** но обратите внимание**,** что этот выбор необязателен**.**

Code

```
# books/views.py
from django.db.models import Q # new
...
class SearchResultsListView(ListView):
        model = Book
        context_object_name = 'book_list'
        template_name = 'books/book_list.html'
        def get_queryset(self): # new
                return Book.objects.filter( Q(title__icontains='beginners') |
                        Q(title__icontains='api')
```

Обновите страницу с результатами поиска**,** чтобы увидеть новый результат**.**



Поиск с помощью Q-объектов

Обновите страницу с результатами поиска, чтобы увидеть новый результат.

Теперь перейдем к соответствующей форме поиска, чтобы не вводить фильтры в жестком виде, а заполнять их на основе поискового запроса пользователя.

Формы

В основе своей веб-форма проста: она принимает пользовательский ввод и отправляет его на **URL** с помощью метода **GET** или **POST.** Однако на практике это фундаментальное поведение в Интернете может быть чудовищно сложным.

Первая проблема - отправка данных формы: куда на самом деле попадают данные и как их обрабатывать? Не говоря уже о том, что существует множество проблем безопасности, когда мы позволяем пользователям отправлять данные на сайт.

Есть только два варианта того, как отправлять форму: с помощью методов **GET** или **POST HTTP**.

POST объединяет данные формы, кодирует их для передачи, отправляет их на сервер, а затем получает ответ. Любой запрос, который изменяет состояние базы данных - создает, редактирует или удаляет данные - должен использовать **POST**.

GET объединяет данные формы в строку, которая добавляется к **URL**-адресу назначения. **GET** следует использовать только для запросов, которые не влияют на состояние приложения, например, при поиске, когда ничего в базе данных не меняется, в основном мы просто выполняем просмотр отфильтрованного списка.

Если вы посмотрите на **URL** после посещения **Google.com**, то увидите свой поисковый запрос в самом **URL** страницы результатов поиска.

Для получения дополнительной информации в **Mozilla** есть подробные руководства по отправке данных формы и проверке данных формы, которые стоит просмотреть, если вы еще не знакомы с основами работы с формами.

Форма поиска

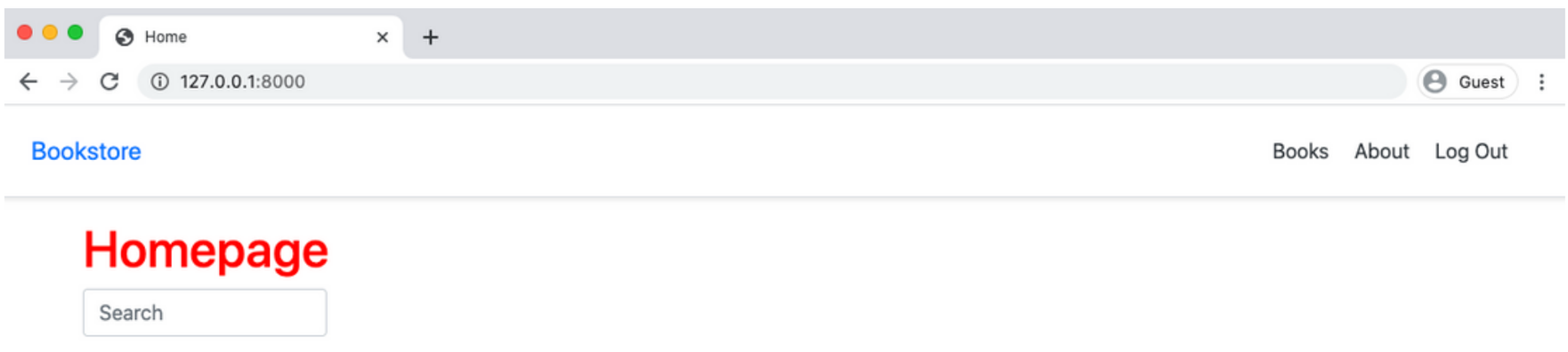
Давайте добавим основную форму поиска на текущую домашнюю страницу прямо сейчас. В будущем ее можно будет легко разместить в навигационной панели или на специальной странице поиска.

Мы начнем с **HTML**-тегов **<form>** и используем стилизацию **Bootstrap**, чтобы они выглядели красиво. Действие определяет, куда перенаправить пользователя после отправки формы, и это будет страница **search_results**. Как и в случае со всеми ссылками **URL**, это имя **URL** для страницы. Затем мы указываем желаемый метод **get**, а не **post**.

Вторая часть формы - это вход, который содержит поисковый запрос пользователя. Мы присваиваем ему имя переменной **q**, которая впоследствии будет видна в **URL**, а также доступна в файле **views**. Мы добавляем стилизацию **Bootstrap** с классом , указываем тип ввода - текст, добавляем **Placeholder**, который является текстом по умолчанию, который подсказывает пользователю. Последняя часть, **aria-label**, - это имя, предоставляемое пользователям программы чтения с экрана. Доступность является важной частью веб-разработки и должна быть учтена с самого начала: включайте **aria-label** во все ваши формы!

```
Code
<!-- templates/home.html -->
{% extends '_base.html' %}
{% load static %}
{% block title %}Home{% endblock title %}
{% block content %}
<h1>Homepage</h1>
<form class="form-inline mt-2 mt-md-0" action="{% url 'search_results' %}"
method="get">
    <input name="q" class="form-control mr-sm-2" type="text"
                placeholder="Search" aria-label="Search">
</form>
{% endblock content %}
```

Перейдите на главную страницу, и новая поисковая строка появится.



Домашняя страница с поисковой формой

Попробуйте ввести поиск, например, **"hello".** После нажатия кнопки **Return** вы будете перенаправлены на страницу результатов поиска. Обратите внимание, в частности, что **URL** содержит поисковый запрос **/search/?q=hello .**

URL-адрес со строкой запроса

Однако результаты не изменились**!** А все потому, что наш **SearchResultsListView** все еще имеет жестко закодированные значения**.** Последний шаг **-** взять поисковый запрос пользователя, представленный **q** в **URL**, и передать его фактическим фильтрам поиска**.**

Code

```
# books/views.py
class SearchResultsListView(ListView):
        model = Book
        context_object_name = 'book_list'
        template_name = 'books/search_results.html'
        def get_queryset(self): # new
                query = self.request.GET.get('q')
                return Book.objects.filter(
                        Q(title__icontains=query) | Q(author__icontains=query)
)
```

Что изменилось**?** Мы добавили переменную **query**, которая принимает значение **q** из формы отправки**.** Затем мы обновили наш фильтр**,** чтобы использовать запрос либо для поля названия**,** либо для поля автора**.** Вот и все**!** Обновите страницу результатов поиска **-** она по**-**прежнему имеет тот же **URL** с нашим запросом **-** и результат будет ожидаемым**:** никаких результатов ни по названию**,** ни по автору для **"hello".**

Вернитесь на главную страницу и попробуйте новый поиск**,** например**, "api"** или **"beginners",** чтобы увидеть полную функциональность поиска в действии**.**

Git

Не забудьте сохранить нашу текущую работу в этой главе**,** зафиксировав новый код в **Git.**

Command Line

```
$ git status
$ git add .
$ git commit -m 'ch14'
```

Официальный исходный код этой главы доступен на **Github.**

Заключение

Наш базовый поиск завершен, но мы только поцарапали поверхность потенциальных оптимизаций поиска. Например, может быть, мы хотим добавить в форму поиска кнопку, которую можно нажать в дополнение к нажатию клавиши **Return?** Или, что еще лучше, включить валидацию формы. Помимо фильтрации с помощью **AND** и **OR** есть и другие факторы, если мы хотим получить качественный поиск **Google**, такие как релевантность и многое другое.

Существует несколько сторонних пакетов с расширенными возможностями, например, **django-watson** или **django-haystack**, однако, учитывая, что мы используем **PostgreSQL** в качестве базы данных, мы можем воспользоваться его полнотекстовым поиском и другими возможностями, встроенными в сам **Django**.

Последний вариант **-** использовать решение корпоративного уровня, такое как **ElasticSearch**, которое должно работать на отдельном сервере **(**что не так сложно с **Docker)**, или положиться на размещенное решение, такое как **Swiftype** или **Algolia**.

В следующей главе мы изучим множество оптимизаций производительности, доступных в **Django**, и подготовим наш проект **Bookstore** к возможному развертыванию.

Глава 15: Производительность

Первоочередной задачей любого веб-сайта является его правильная работа и наличие соответствующих тестов. Но если вашему проекту посчастливилось получить большое количество трафика, внимание быстро переключается на производительность и обеспечение максимальной эффективности. Это увлекательное и сложное занятие для многих инженеров, но оно также может стать ловушкой.

У ученого-компьютерщика Дональда Кнута есть знаменитая цитата, которую стоит прочитать целиком:

"Настоящая проблема заключается в том, что программисты потратили слишком много времени, беспокоясь об эффективности в неправильных местах и в неправильное время; преждевременная оптимизация - это корень всех зол (или, по крайней мере, большинства из них) в программировании".

Хотя важно организовать надлежащий мониторинг, чтобы вы могли оптимизировать свой проект позже, не стоит уделять этому слишком много внимания на начальном этапе. Невозможно правильно имитировать производственную среду на местном уровне. И нет способа точно предсказать, как будет выглядеть посещаемость сайта. Но можно потратить слишком много времени на поиск крошечного прироста производительности на ранних стадиях, вместо того чтобы общаться с пользователями и вносить более значительные улучшения в код!

В этой главе мы сосредоточимся на общих чертах производительности, связанной с **Django**-технологиями, и выделим области, заслуживающие дальнейшего изучения в масштабе. В целом, производительность сводится к четырем основным областям: оптимизация запросов к базе данных, кэширование, индексы и сжатие внешних активов, таких как изображения, **JavaScript** и **CSS**.

django-debug-toolbar

Прежде чем мы сможем оптимизировать наши запросы к базе данных, нам нужно их увидеть. И для этого в сообществе **Django** по умолчанию используется сторонний пакет **django-debug-toolbar**. Он поставляется с настраиваемым набором панелей для проверки полного цикла запросов/ответов любой конкретной страницы.

Как обычно, мы можем установить его в **Docker** и остановить наши запущенные контейнеры.

Command Line

```
$ docker-compose exec web pipenv install django-debug-toolbar==2.2
$ docker-compose down
```

В нашем файле **config/settings.py** необходимо задать три отдельные конфигурации**:**

1. **INSTALLED_APPS**
2. **Middleware**
3. **NTERNAL_IPS**

Сначала добавьте **Debug Toolbar** в конфигурацию **INSTALLED_APPS.** Обратите внимание**,** что правильное название **- debug_toolbar,** а не **django_debug_toolbar,** как можно было бы ожидать**.**

Code

```
# config/settings.py
INSTALLED_APPS = [
        'django.contrib.admin',
        'django.contrib.auth',
        'django.contrib.contenttypes',
        'django.contrib.sessions',
        'django.contrib.messages',
        'django.contrib.staticfiles',
        'django.contrib.sites',
       # Third-party
        'crispy_forms',
        'allauth',
        'allauth.account',
        'debug_toolbar', # new
       # Local
        'accounts',
        'pages',
        'books',
]
```

Во**-**вторых**,** добавьте **Debug Toolbar** в **Middleware,** где она в основном реализована**.**

Code

```
# config/settings.py
MIDDLEWARE = [
        'django.middleware.security.SecurityMiddleware',
        'django.contrib.sessions.middleware.SessionMiddleware',
        'django.middleware.common.CommonMiddleware',
        'django.middleware.csrf.CsrfViewMiddleware',
        'django.contrib.auth.middleware.AuthenticationMiddleware',
        'django.contrib.messages.middleware.MessageMiddleware',
        'django.middleware.clickjacking.XFrameOptionsMiddleware',
        'debug_toolbar.middleware.DebugToolbarMiddleware', # new
]
```

В-третьих, установите также **INTERNAL_IPS.** Если бы мы не находились в **Docker,** этот параметр можно было бы установить на **'127.0.0.1',** однако, поскольку мы запускаем наш веб-сервер в **Docker,** требуется дополнительный шаг, чтобы он соответствовал машинному адресу **Docker.** Добавьте следующие строки в нижней части **config/settings.py .**

Code

```
# config/settings.py
...
# django-debug-toolbar
import socket
hostname, _, ips = socket.gethostbyname_ex(socket.gethostname())
INTERNAL_IPS = [ip[:-1] + "1" for ip in ips]
```

Фух. Это выглядит немного пугающе, но в основном это гарантирует, что наш **INTERNAL_IPS** совпадает с адресом нашего хоста **Docker.**
Теперь пересоберите базовый образ, чтобы он содержал пакет и обновленную конфигурацию настроек.

Command Line

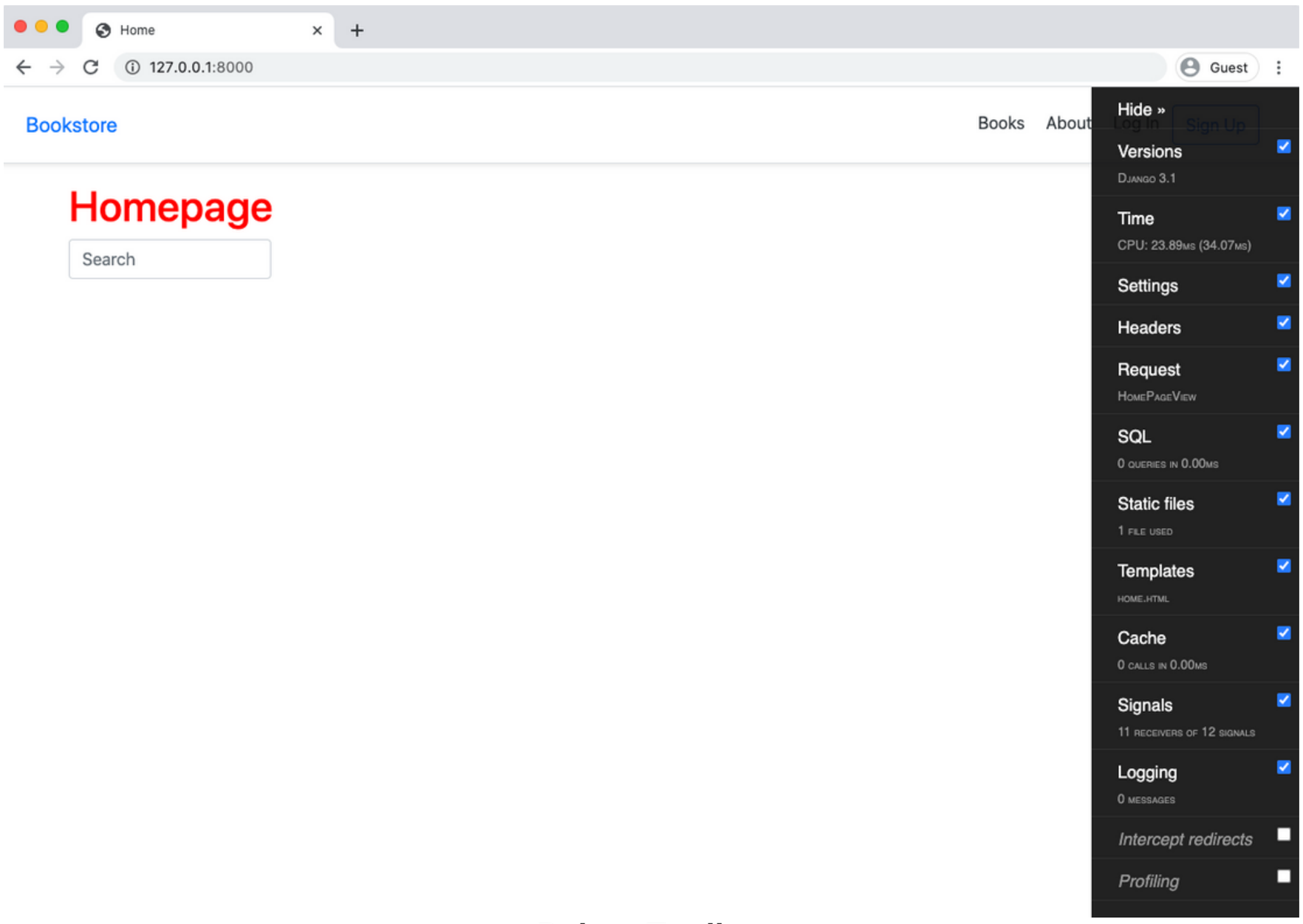
```
$ docker-compose up -d --build
```

Остался последний шаг, и он заключается в обновлении **URLconf.** Мы хотим, чтобы **Debug Toolbar** появлялась только в том случае, если **DEBUG** равен **true,** поэтому мы добавим логику для ее отображения только в этом случае в нижней части файла **config/urls.py.**

Code

```
# config/urls.py
from django.conf import settings
from django.conf.urls.static import static
from django.contrib import admin
from django.urls import path, include

urlpatterns = [
# Django admin
        path('admin/', admin.site.urls),
# User management
       path('accounts/', include('allauth.urls')),
# Local apps
        path('', include('pages.urls')),
        path('books/', include('books.urls')),

] + static(settings.MEDIA_URL, document_root=settings.MEDIA_ROOT)
if settings.DEBUG: # new
      import debug_toolbar
      urlpatterns = [
              path('__debug__/', include(debug_toolbar.urls)),
] + urlpatterns
```

Теперь, если вы обновите главную страницу, вы увидите панель инструментов **django-debug-toolbar** с правой стороны.

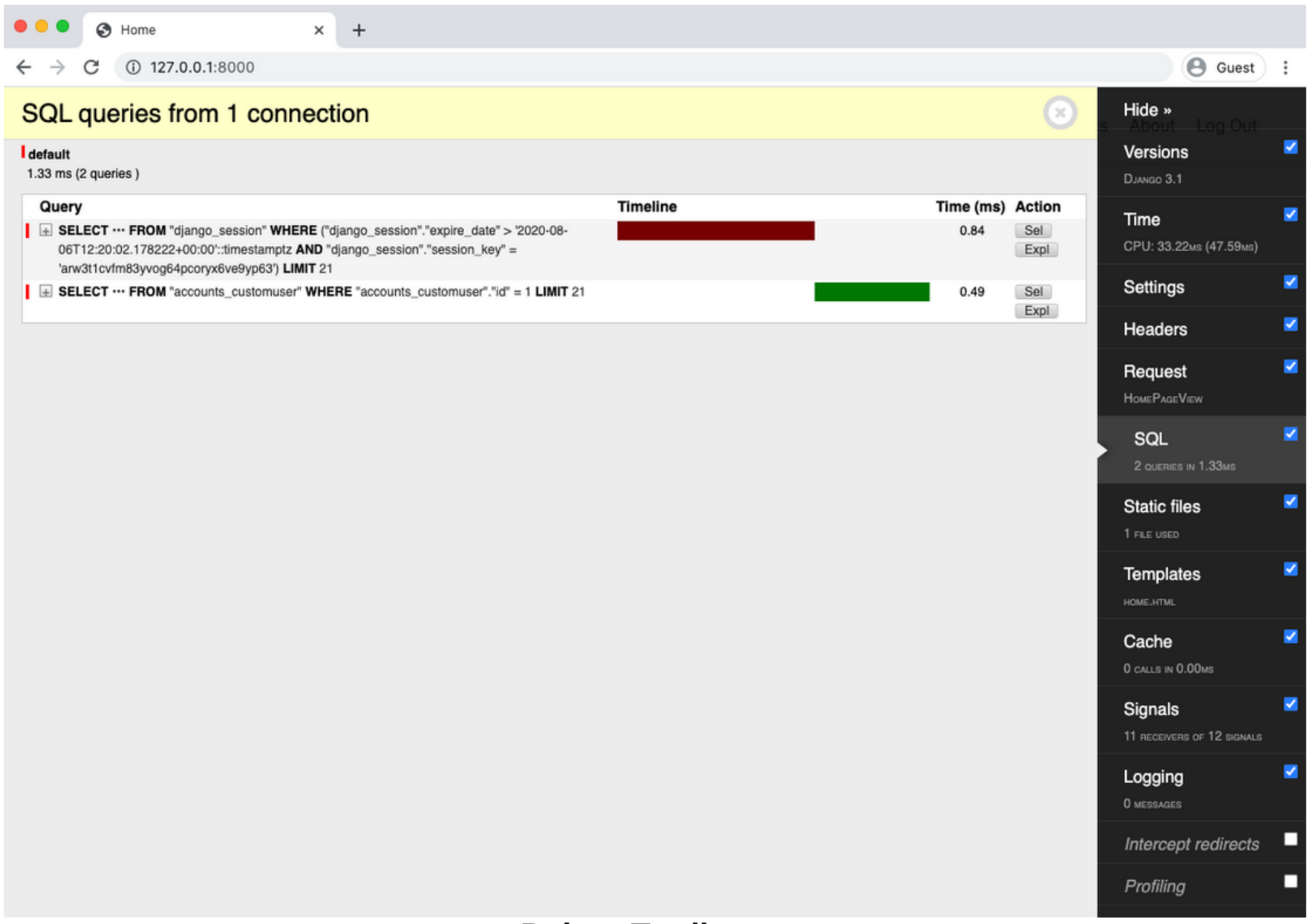


Debug Toolbar

Если нажать на ссылку **" Hide"** вверху, она станет гораздо меньшей боковой панелью в правой части страницы.

Анализ страниц

Debug Toolbar имеет множество возможных настроек, но настройки по умолчанию позволяют многое узнать о нашей домашней странице. Например, мы можем увидеть текущую используемую версию **Django**, а также время, которое потребовалось для загрузки страницы. А также конкретный запрос, который был вызван **HomePageView**. Это может показаться очевидным, но в больших кодовых базах, особенно если вы начинающий разработчик, может быть не очевидно, какое представление вызывает какую страницу. **Debug Toolbar -** это полезный быстрый старт для понимания существующих сайтов. Однако, вероятно, самым полезным элементом является **SQL**, который показывает запросы на конкретной странице. Если вы сейчас вышли из системы, то на главной странице нет **SQL-**запросов. Поэтому пройдите вперед и войдите в систему под учетной записи суперпользователя, а затем вернитесь на главную страницу. Панель инструментов отладки показывает, что выполняются два запроса и время выполнения каждого из них.



Debug Toolbar

На больших и плохо оптимизированных сайтах на одной странице могут выполняться сотни или даже тысячи запросов**!**

select_related и prefetch_related

Каковы варианты действий**,** если вы все**-**таки обнаружили**,** что работаете над **Django-**сайтом со слишком большим количеством **SQL-**запросов на странице**?** В общем случае**,** меньшее количество больших запросов будет быстрее**,** чем множество маленьких**,** хотя это можно и нужно проверять на практике**.** Две распространенные техники для этого **-** **select_related()** и **prefetch_related()**.

select_related используется для отношений с одним значением через прямые отношения "один ко многим" или "один к одному"**.** Он создает **SQL-**соединение и включает поля связанного объекта в оператор **SELECT,** в результате чего все связанные объекты включаются в один более сложный запрос к базе данных**.** Этот единый запрос обычно более производителен**,** чем несколько более мелких запросов**.**

prefetch_related используется для набора или списка объектов**,** как отношения "многие ко многим" или "многие к одному"**.** Под капотом выполняется поиск для каждого отношения**,** и "соединение" происходит на языке **Python,** а не **SQL.** Это позволяет осуществлять предварительную выборку объектов типа "многие ко многим" и "многие к одному"**,** что невозможно сделать с помощью **select_related,** в дополнение к отношениям типа "внешний ключ" и "один к одному"**,** которые поддерживаются **select_related.**

Внедрение одного или обоих методов на веб**-**сайте **-** это обычный первый шаг к сокращению запросов и времени загрузки страницы**.**

Кэширование

Рассмотрим, что наш проект **Bookstore** представляет собой динамический веб-сайт. Каждый раз, когда пользователь запрашивает страницу, нашему серверу приходится выполнять различные вычисления, включая запросы к базе данных, отрисовку шаблонов и т.д., прежде чем обслужить ее. Это занимает время и намного медленнее, чем простое чтение файла со статического сайта, где содержимое не меняется.

Однако на больших сайтах этот тип накладных расходов может быть довольно медленным, и кэширование - одно из первых решений в арсенале веб-разработчика. Реализация кэширования в нашем текущем проекте - это, безусловно, излишество, но, тем не менее, мы рассмотрим возможные варианты и реализуем базовую версию.

Кэш - это хранение в памяти дорогостоящего вычисления. Выполненный однажды, он не требует повторного выполнения! Два наиболее популярных варианта - **Memcached**, который имеет встроенную поддержку **Django**, и **Redis**, который обычно реализуется с помощью стороннего пакета **django-redis**.

Django имеет свой собственный фреймворк кэширования, который включает четыре различных варианта кэширования в порядке убывания степени детализации:

1.) Кэш для каждого сайта - самый простой в настройке и кэширует весь ваш сайт.
2.) Кэш для отдельных представлений позволяет кэшировать отдельные представления.
3.) Кэширование фрагментов шаблона позволяет указать конкретный раздел шаблона для кэширования.
4.) Низкоуровневый **API** кэша позволяет вручную устанавливать, извлекать и поддерживать определенные объекты в кэше.

Почему бы просто не кэшировать все и всегда? Одна из причин заключается в том, что кэш-память стоит дорого, поскольку она хранится в виде оперативной памяти: подумайте о стоимости перехода с **8** на **16** ГБ оперативной памяти на вашем ноутбуке по сравнению с **256** или **512** ГБ места на жестком диске. Кроме того, кэш должен быть "теплым", то есть заполненным обновленным контентом, поэтому в зависимости от потребностей сайта оптимизация кэша, чтобы он был точным, но не расточительным, требует довольно много настроек.

Если вы хотите реализовать кэширование для каждого сайта, что является самым простым подходом, вы добавите **UpdateCacheMiddleware** в самом верху конфигурации **MIDDLEWARE** в **config/settings.py** и **FetchFromCacheMiddleware** в самом низу. Также установите три дополнительных поля **CACHE_MIDDLEWARE_ALIAS**, **CACHE_MIDDLEWARE_SECONDS** и **CACHE_MIDDLEWARE_KEY_PREFIX**.

Code

```
# config/settings.py
MIDDLEWARE = [
        'django.middleware.cache.UpdateCacheMiddleware', # new
        'django.middleware.security.SecurityMiddleware',
        'django.contrib.sessions.middleware.SessionMiddleware',
        'django.middleware.common.CommonMiddleware',
        'debug_toolbar.middleware.DebugToolbarMiddleware',
        'django.middleware.csrf.CsrfViewMiddleware',
        'django.contrib.auth.middleware.AuthenticationMiddleware',
        'django.contrib.messages.middleware.MessageMiddleware',
        'django.middleware.clickjacking.XFrameOptionsMiddleware',
        'debug_toolbar.middleware.DebugToolbarMiddleware',
        'django.middleware.cache.FetchFromCacheMiddleware', # new
]

CACHE_MIDDLEWARE_ALIAS = 'default'
CACHE_MIDDLEWARE_SECONDS = 604800
CACHE_MIDDLEWARE_KEY_PREFIX = ''
```

Единственное значение по умолчанию, которое вы, возможно, захотите настроить, это **CACHE_MIDDLEWARE_SECONDS**, которое по умолчанию определяет количество секунд **(600)** для кэширования страницы. По истечении этого периода кэш истекает и становится пустым. Хорошее значение по умолчанию в начале работы **- 604800** секунд или **1** неделя **(60** секунд **x 60** минут **x 168** часов**)** для сайта с редко меняющимся содержимым. Но если вы обнаружите, что ваш кэш быстро заполняется или у вас сайт, на котором содержимое меняется часто, сокращение этого параметра будет хорошим первым шагом.

Внедрение кэширования на данном этапе не является обязательным. Как только сайт будет запущен, необходимость кэширования для каждого сайта, каждой страницы и т.д. быстро станет очевидной. Кроме того, возникает дополнительная сложность, поскольку **Memcache** должен быть запущен как отдельный экземпляр. На хостинге **Heroku**, который мы будем использовать в главе **18** для развертывания, есть бесплатный уровень, доступный через **Memcachier**.

Индексы

Индексирование **-** это распространенная техника для ускорения работы базы данных. Это отдельная структура данных, которая позволяет ускорить поиск и обычно применяется только к первичному ключу в модели. Недостатком является то, что индексы требуют дополнительного места на диске, поэтому их следует использовать с осторожностью.

Как бы ни был велик соблазн просто добавить индексы к первичным ключам с самого начала, лучше начать без них и добавлять их только позже, исходя из производственных потребностей. Общее правило гласит, что если данное поле используется часто, например, в **10-25%** всех запросов, то оно является главным кандидатом на индексирование.

Исторически индексное поле можно было создать, добавив **db_index=True** к любому полю модели. Например, если бы мы хотели добавить его к полю **id** в нашей модели **Book**, это выглядело бы следующим образом (на самом деле не стоит этого делать).

Code

```
# books/models.py
...
class Book(models.Model):
        id = models.UUIDField(
            primary_key=True, db_index=True, # new
            default=uuid.uuid4,  editable=False)
...
```

Это изменение должно быть добавлено через файл миграции и перенесено.

Начиная с **Django 1.11**, были добавлены индексы моделей на основе классов, поэтому вместо них можно включать в раздел **Meta**. Таким образом, вы можете написать предыдущий индекс следующим образом:

Code

```
# books/models.py
...
class Book(models.Model):
        id = models.UUIDField( primary_key=True, default=uuid.uuid4,
                editable=False)
...
        class Meta:
                indexes = [ models.Index(fields=['id'], name='id_index'),  ]
                permissions = [  ("special_status", "Can read all books"),
]
```

Поскольку мы изменили модель, мы должны создать файл миграций и применить его.

Command Line

```
$ docker-compose exec web python manage.py makemigrations books
$ docker-compose exec web python manage.py migrate
```

django-extensions

Еще один очень популярный пакет сторонних разработчиков для проверки проекта **Django - django-extensions,** который добавляет ряд полезных пользовательских расширений**.**

Особенно полезным является **shell_plus,** который автоматически загружает все модели в оболочку**,** что значительно упрощает работу с **Django ORM.**

Front-end ресурсы

Последним основным источником узких мест на сайте является загрузка **front-end** файлов**. CSS** и **JavaScript** могут стать довольно большими**,** поэтому такие инструменты**,** как **django-compressor,** могут помочь минимизировать их размер**.**

Изображения часто являются первым местом**,** на которое следует обратить внимание с точки зрения размера файлов**.** Имеющийся у нас набор статических**/**медиа файлов будет масштабироваться до довольно больших размеров**,** но для действительно больших сайтов стоит рассмотреть возможность использования сети доставки контента **(CDN)** для изображений вместо их хранения в файловой системе сервера**.**

Вы также можете предоставлять пользователям изображения разного размера**.** Например**,** вместо того**,** чтобы уменьшать большую обложку книги для страницы списка или поиска**,** можно хранить уменьшенную версию миниатюры и предоставлять ее по мере необходимости**.** Сторонний пакет **easy-thumbnails -** хорошее начало для этого**.**

Фантастическая бесплатная электронная книга по этой теме **- Essential Image Optimization** от **Addy Osmani,** в которой подробно рассматривается оптимизация изображений и автоматизация**.**

В качестве последней проверки существуют автоматизированные тесты для определения скорости внешнего интерфейса**,** такие как **Google PageSpeed Insights,** которые присваивают оценку на основе скорости загрузки страницы**.**

Git

В этой главе было внесено много изменений в код**,** поэтому убедитесь**,** что все зафиксировано с помощью **Git.**

Command Line

```
$ git status
$ git add .
$ git commit -m 'ch15'
```

Если у вас возникли какие-либо ошибки, обязательно посмотрите журналы **docker-compose logs** и сравните свой код с официальным исходным кодом на **Github**.

Заключение

Существует почти бесконечный список оптимизаций производительности, которые можно применить к проекту. Но не забывайте мудрый совет Дональда Кнута и не слишком увлекайтесь этим. Узкие места проявят себя в производстве и должны быть устранены в основном тогда, а не заранее. Вы должны помнить, что проблемы с производительностью - это хорошая проблема! Они устранимы и означают, что ваш проект интенсивно используется.

Глава 16: Безопасность

Всемирная паутина - опасное место. Существует множество злоумышленников и еще больше ботов, которые пытаются взломать ваш сайт и причинить вред. Поэтому понимание и внедрение средств безопасности является обязательным для любого сайта.

К счастью, **Django** имеет очень хороший послужной список, когда дело доходит до безопасности, благодаря своему многолетнему опыту решения проблем веб-безопасности, а также надежному и регулярному циклу обновления безопасности. Новые функциональные релизы выходят примерно раз в **9** месяцев, например, с **2.2** по **3.0**, но есть и выпуски патчей, устраняющих ошибки и проблемы безопасности, например, с **2.2.2** по **2.2.3**, которые выходят почти ежемесячно.

Однако, как и в случае с любым другим инструментом, важно правильно реализовать функции безопасности, и в этой главе мы рассмотрим, как это сделать в нашем проекте **Bookstore**.

Социальная инженерия

Самый большой риск для безопасности любого веб-сайта, в конечном счете, не технический: это люди. Термин "социальная инженерия" относится к технике поиска людей, имеющих доступ к системе, которые вольно или невольно поделятся своими учетными данными с плохим агентом.

В наши дни фишинг, вероятно, является наиболее вероятным виновником, если вы работаете в технической организации. Достаточно одного неудачного клика на ссылку в электронном письме, чтобы злоумышленник получил потенциальный доступ к системе или, по крайней мере, весь доступ, который есть у скомпрометированного сотрудника.

Чтобы снизить этот риск, внедрите надежную схему разрешений и предоставляйте только тот доступ к безопасности, который необходим сотруднику, не больше. Нужен ли каждому инженеру доступ к производственной базе данных? Скорее всего, нет. Нужен ли неинженерам доступ на запись? Опять же, скорее всего, нет. Эти вопросы лучше обсуждать заранее, и по умолчанию лучше добавлять разрешения только по мере необходимости, а не устанавливать по умолчанию статус суперпользователя для всех!

Обновления Django

Обновление вашего проекта до последней версии **Django -** еще один важный способ оставаться в безопасности. И я имею в виду не только обновление до последней версии **(2.2, 3.0, 3.1** и т.д.**),** которая выходит примерно каждые **9** месяцев. Существуют также ежемесячные обновления безопасности в виде патчей **2.2.1, 2.2.2, 2.2.3** и т. д.

Что касается релизов долгосрочной поддержки **(LTS)?** Определенные функциональные релизы, обозначенные как **LTS**, получают исправления безопасности и потери данных в течение гарантированного периода времени, обычно около **3** лет. Например, **Django 2.2** является **LTS** и будет поддерживаться до **2022** года, когда **Django 4.0** выйдет в качестве следующей **LTS** версии. Можете ли вы оставаться на **LTS**-версиях? Да. Должны ли вы? Нет. Лучше и надежнее оставаться в актуальном состоянии.

Не поддавайтесь соблазну и реальности многих реальных проектов, которые не посвящают часть времени разработчиков обновлению версий **Django**. Веб-сайт подобен автомобилю: он нуждается в регулярном обслуживании, чтобы работать наилучшим образом. Откладывая обновления, вы только усугубляете проблему.

Как обновлять? В **Django** есть предупреждения об износе, которые можно и нужно запускать для каждого нового релиза, набрав **python -Wa manage.py test** . Гораздо лучше обновляться с **2.0** до **2.1** и **2.2** и запускать предупреждения об устаревании каждый раз, чем пропускать несколько версий.

Контрольный список развертывания

Чтобы помочь с развертыванием и проверкой параметров безопасности, документация **Django** содержит специальный контрольный список развертывания, который дополнительно описывает параметры безопасности.

Еще лучше то, что есть команда, которую можно запустить для автоматизации рекомендаций **Django**, **python manage.py check --deploy** , которая проверит, готов ли проект к развертыванию. Она использует фреймворк проверки системы **Django**, который можно использовать для настройки подобных команд в зрелых проектах.

Поскольку мы работаем в **Docker**, к команде необходимо добавить **docker-compose exec web**.

Command Line

```
$ docker-compose exec web python manage.py check --deploy
System check identified some issues:
WARNINGS:
...
System check identified 5 issues (0 silenced).
```

Как здорово! Описательный и длинный список вопросов, которые мы можем пройти один за другим, чтобы подготовить наш проект **Bookstore** к производству.

docker-compose-prod.yml

В конечном счете, наши локальные настройки для разработки будут отличаться от настроек для производства. Мы уже начали настраивать это в главе **8:** Переменные окружения. Вспомните, что мы добавили переменные окружения для **SECRET_KEY , DEBUG** и **DATABASES .** Но мы не установили производственные значения или способ эффективного переключения между локальным и производственным.

Есть несколько способов решить эту проблему. Учитывая, что мы будем развертываться на **Heroku,** наш подход заключается в создании файла **docker-compose-prod.yml,** который мы можем использовать для тестирования производственной среды, и мы вручную добавим переменные среды в производственную среду.

Для начала создайте файл **docker-compose-prod.yml** в той же папке, что и **docker-compose.yml .**

Command Line

```
$ touch docker-compose-prod.yml
```

По умолчанию **Git** будет отслеживать каждый файл или папку в нашем проекте. Мы не хотим, чтобы это произошло с новым файлом, поскольку он будет содержать конфиденциальную информацию. Решением является создание файла **.gitignore,** который содержит файлы или папки, которые будут игнорироваться **Git'**ом.

Создайте новый файл.

Command Line

```
$ touch .gitignore
```

Добавьте к нему наш единственный файл.

.gitignore

```
docker-compose-prod.yml
__pycache__/
db.sqlite3
.DS_Store # Mac only
```

Если вам интересно, на **Github** есть официальный файл **gitignore Python,** содержащий дополнительные конфигурации, достойные дальнейшего изучения.

Запустите **git status** снова, и файл **docker-compose-prod.yml** не будет виден, хотя он все еще находится в нашем проекте. Это то, что нам нужно!

Пока что скопируйте файл **docker-compose.yml** в **docker-compose-prod.yml** .

docker-compose-prod.yml

```
version: '3.8'
services:
  web:
      build: .
      command: python /code/manage.py runserver 0.0.0.0:8000
      volumes:
        - .:/code
      ports:
        - 8000:8000
      depends_on:
        - db
      environment:
        - "DJANGO_SECRET_KEY=)*_s#exg*#w+#-xt=vu8b010%%a&p@4edwyj0=(nqq90b9a8*n"
        - "DJANGO_DEBUG=True"
db:
    image: postgres:11
    volumes:
      - postgres_data:/var/lib/postgresql/data/
    environment:
      - "POSTGRES_HOST_AUTH_METHOD=trust"
volumes:
  postgres_data:
```

Чтобы запустить наш новый файл, выключите хост **Docker** и перезапустите его с помощью флага **-f**, чтобы указать альтернативный файл **compose.** По умолчанию **Docker** предполагает наличие файла **docker-compose.yml**, поэтому добавление флага **-f** в этом случае не требуется.

Command Line

```
$ docker-compose down
$ docker-compose -f docker-compose-prod.yml up -d --build
$ docker-compose exec web python manage.py migrate
```

Флаг **--build** добавляется для первоначальной сборки образа вместе со всеми соответствующими программными пакетами для нового файла **compose.** Также запускается **migrate** для новой базы данных. Это совершенно новый экземпляр нашего проекта! Поэтому у него не будет учетной записи суперпользователя или каких-либо наших данных, таких как информация о книгах. Но это не страшно; эта информация может быть добавлена в производстве, а наше внимание сосредоточено на создании локальной среды тестирования производства.

Перейдите на сайт, и все должно работать как прежде, даже несмотря на то, что мы используем другой компонентный файл.

DEBUG

В конечном итоге, наша цель в этой главе - пройти контрольный список развертывания **Django** с помощью файла **docker-compose-prod.yml.** Давайте начнем с изменения параметра **DEBUG,** который установлен в **True,** но в **production** должен быть **False.**

docker-compose-prod.yml

```
environment:
- "DJANGO_SECRET_KEY=)*_s#exg*#w+#-xt=vu8b010%%a&p@4edwyj0=(nqq90b9a8*n"
- "DJANGO_DEBUG=False" # new
```

Выключите **Docker** и запустите его снова после внесения изменений.

Command Line

```
$ docker-compose down
$ docker-compose -f docker-compose-prod.yml up -d --build
```

Сайт должен работать так же, как и раньше, но чтобы проверить, что **DEBUG** установлен в **False,** посетите несуществующую страницу, например **http://127.0.0.1:8000/debug.**

Debug Page Not Found

И есть общее сообщение **"Not Found",** подтверждающее, что у нас **DEBUG** установлен на **False.** Потому что если бы он был **True,** то вместо него был бы подробный отчет об ошибке.

Давайте запустим контрольный список развертывания **Django** снова, теперь, когда значение **DEBUG** изменено. Вспомните, что когда мы запускали его ранее в этой главе, было **5** проблем.

Command Line

```
$ docker-compose exec web python manage.py check --deploy
System check identified some issues:
WARNINGS:
...
System check identified 4 issues (0 silenced).
```

Мы сократили до **4,** поскольку **DEBUG** установлен на **False.** Прогресс**!**

Переменные по умолчанию

Переменные окружения служат двум целям в нашем проекте **Django:** они сохраняют такие элементы как **SECRET_KEY** в тайне и служат способом переключения между локальными и производственными настройками**.** Хотя нет ничего плохого в том**,** чтобы иметь две переменные окружения для такого параметра**,** как **DEBUG,** вероятно**,** будет лучше использовать значение по умолчанию**,** когда нам не нужно держать что**-**то в секрете**.**

Например**,** мы можем переписать конфигурацию **DEBUG** следующим образом**:**

Code

```
# config/settings.py
DEBUG = env.bool("DJANGO_DEBUG", default=False)
```

Это означает значение по умолчанию **False,** если переменная окружения отсутствует**.** Если такая переменная есть**,** то используйте ее**.** Мы сохраним переменную **DJANGO_DEBUG** в локальном файле **docker-compose.yml,** но удалим ее в **docker-compose-prod.yml .** Такой подход приводит к меньшему размеру файла **docker-compose-prod.yml** и**,** возможно**,** более безопасен**,** так как если по какой**-**то причине переменные окружения не загружаются должным образом**,** мы не включим локальные настройки разработки случайно**.** Используются только производственные значения**.**

Перейдите к обновлению **docker-compose-prod.yml,** удалив **DJANGO_DEBUG .**

docker-compose-prod.yml

```
environment:
- "DJANGO_SECRET_KEY=)*_s#exg*#w+#-xt=vu8b010%%a&p@4edwyj0=(nqq90b9a8*n"
```

Если вы отключите **Docker** и перезапустите локальные или производственные настройки**,** они будут работать**.**

SECRET_KEY

Наш **SECRET_KEY** в настоящее время виден в файле **docker-compose.yml.** Для большей безопасности мы должны сгенерировать новый производственный ключ и протестировать его через **docker-compose-prod.yml . SECRET_KEY -** это **50-**символьная случайная строка**,** генерируемая заново каждый раз при выполнении команды **startproject.** Для генерации нового ключа мы можем использовать встроенный в **Python** модуль **secrets.**

Command Line

```
$ docker-compose exec web python -c 'import secrets; print(secrets.token_urlsafe(38))'
ldBHq0YGYxBzaMJnLVOiNG7hruE8WKzGG2zGpYxoTNmphB0mdBo
```

Параметр **token_urlsafe** возвращает количество байтов в текстовой строке, безопасной для **URL.** При кодировке **Base64** в среднем каждый байт содержит **1,3** символа. Поэтому использование **38** дает в данном случае **51** символ. Главное, чтобы ваш **SECRET_KEY** содержал не менее **50** символов. Каждый раз при выполнении команды выводится новое значение.

Небольшое напоминание: поскольку мы работаем с **Docker**, если ваш **SECRET_KEY** включает знак доллара, **$**, то вам нужно добавить дополнительный знак доллара, **$$.** Это связано с тем, как **docker-compose** обрабатывает подстановку переменных. В противном случае вы увидите ошибку!

Добавьте новый **SECRET_KEY** в файл **docker-compose-prod.yml**, чтобы он выглядел следующим образом:

docker-compose-prod.yml

```
# docker-compose-prod.yml
environment:
- "DJANGO_SECRET_KEY=ldBHq0YGYxBzaMJnLVOiNG7hruE8WKzGG2zGpYxoTNmphB0mdBo"
```

Перезапустите наш контейнер **Docker,** который теперь использует действительно секретный ключ **SECRET_KEY .**

Command Line

```
$ docker-compose down
$ docker-compose -f docker-compose-prod.yml up -d --build
```

Веб-сайт должен работать так же, как и раньше. Осталось решить четыре вопроса в контрольном списке развертывания, но сначала кратко о веб-безопасности, чтобы мы могли понять, почему эти параметры важны.

Web Security(Веб безопасность)

Несмотря на то, что **Django** по умолчанию решает большинство распространенных проблем безопасности, все же важно понимать часто встречающиеся методы атак и шаги, которые предпринимает **Django** для их смягчения. Вы можете найти обзор на странице безопасности **Django**, но мы углубимся в эту тему здесь.

Django по умолчанию поставляется с рядом дополнительных промежуточных программ безопасности, которые защищают от других атак по циклу запрос/ответ.

Полное объяснение каждого из них выходит за рамки этой книги, но стоит прочитать о защите, которую команда безопасности **Django** обеспечивала на протяжении многих лет. Не изменяйте значения по умолчанию без веских причин.

SQL-инъекция

Начнем с атаки **SQL injection**, которая возникает, когда злоумышленник может выполнить произвольный **SQL-**код в базе данных. Рассмотрим форму входа на сайт. Что произойдет, если злоумышленник вместо этого напечатает **DELETE from users WHERE user_id=user_id?** Если это будет выполнено в базе данных без надлежащей защиты, это может привести к удалению всех записей пользователей! Нехорошо. В этом комиксе **XKCD** приводится юмористический, хотя и потенциально точный пример того, как это может произойти.

К счастью, **Django ORM** по умолчанию автоматически очищает пользовательский ввод при построении наборов запросов, чтобы предотвратить этот тип атаки. Но здесь нужно быть осторожным: **Django** предоставляет возможность выполнения пользовательских **sql** или **raw** запросов. Их следует использовать с особой осторожностью, поскольку они могут открыть уязвимость для **SQL-**инъекций.

Некоммерческий проект **Open Web Application Security Project (OWASP)** имеет фантастическую и очень подробную шпаргалку по **SQL-**инъекциям, которую рекомендуется прочитать.

XSS (Cross Site Scripting)

Межсайтовый скриптинг **(XSS) -** это еще одна классическая атака, которая происходит, когда злоумышленник может внедрить небольшие фрагменты кода на веб-страницы, просматриваемые другими людьми. Этот код, обычно **JavaScript**, если он хранится в базе данных, затем извлекается и отображается другим пользователям.

Например, рассмотрим форму, используемую для написания отзывов о книгах на нашем сайте. Что если вместо того, чтобы напечатать: "Эта книга была замечательной", пользователь напечатает что-нибудь с помощью **JavaScript?** Например, **<script>alert('hello');</script>** . Если бы этот скрипт хранился в базе данных, то на странице каждого будущего пользователя появлялось бы всплывающее окно с надписью "привет". Хотя этот конкретный пример скорее раздражает, чем опасен, сайт, уязвимый к **XSS**, очень опасен, потому что злоумышленник может вставить в страницу любой **JavaScript**, включая **JavaScript**, который украдет практически все, что угодно у ничего не подозревающего пользователя.

Для предотвращения **XSS** атак шаблоны **Django** автоматически экранируют определенные символы, которые потенциально опасны, включая скобки **(< и >)**, одинарные кавычки **'**, двойные кавычки **"**, и амперсанд **&.** Есть некоторые крайние случаи, когда вы можете захотеть отключить автоэскейп, но это должно использоваться с крайней осторожностью.

Для дальнейшего прочтения рекомендуется "Шпаргалка по **XSS**" от **OWASP.**

Cross-Site Request Forgery (CSRF)

CSRF (Cross-Site Request Forgery) - это третий основной тип атаки, но, как правило, менее известный, чем **SQL Injection** или **XSS.** По сути, она использует то доверие, которое сайт имеет к браузеру пользователя.

Когда пользователь входит на сайт, назовем его для примера банковским сайтом, сервер отправляет обратно токен сессии для этого пользователя. Он включается в **HTTP-**заголовки всех последующих запросов и удостоверяет подлинность пользователя. Но что произойдет, если злоумышленник каким-то образом получит доступ к этому токену сессии?

Например, рассмотрим пользователя, который входит в свой банк на одной вкладке браузера. Затем на другой вкладке он открывает свою электронную почту и нажимает на ссылку, полученную от злоумышленника. Эта ссылка выглядит легитимной, но на самом деле она указывает на банк пользователя, в который он все еще входит! Таким образом, вместо того чтобы оставить комментарий в блоге на этом поддельном сайте, за кулисами учетные данные пользователя используются для перевода денег с его счета на счет хакера.

На практике существует множество способов получить учетные данные пользователя с помощью **CSRF-**атаки: не только ссылки, но и скрытые формы, специальные теги изображений и даже **AJAX-**запросы.

Django обеспечивает защиту от **CSRF** путем включения случайного секретного ключа как в куки через **CSRF Middleware**, так и в форму через тег шаблона **csrf_token**. Сторонний сайт не будет иметь доступа к **cookies** пользователя, поэтому любое расхождение между двумя ключами приводит к ошибке.

Как всегда, **Django** допускает настройку: вы можете отключить промежуточное ПО **CSRF** и использовать тег шаблона **csrf_protect()** в определенных представлениях. Однако, делайте этот шаг с особой осторожностью.

Шпаргалка **OWASP CSRF Cheat Sheet** дает исчерпывающее представление об этой проблеме. Почти все крупные веб-сайты в какой-то момент времени стали жертвами **CSRF-**атак.

Хорошее эмпирическое правило: когда на вашем сайте есть форма, подумайте, нужно ли включать в нее тег **csrf_token**. В большинстве случаев это необходимо!

Защита от Clickjacking

Clickjacking - это еще одна атака, при которой вредоносный сайт обманывает пользователя, заставляя его нажать на скрытый фрейм. Внутренний фрейм, известный как **iframe**, обычно используется для встраивания одного сайта в другой. Например, если вы хотите разместить на своем сайте карту **Google** или видео с **YouTube**, вы включаете тег **iframe**, который помещает этот сайт в ваш собственный. Это очень удобно.

Но у него есть риск для безопасности, который заключается в том, что фрейм может быть скрыт от пользователя. Например, если пользователь уже вошел в свою учетную запись **Amazon**, а затем посещает вредоносный сайт, который выдает себя за изображение котят. Пользователь переходит на вредоносный сайт, чтобы увидеть больше котят, но на самом деле он нажимает на **iFrame** товара **Amazon**, который неосознанно покупает. Это лишь один из примеров **clickjacking.**

Для защиты от этого **Django** поставляется с промежуточным ПО по умолчанию, которое устанавливает **HTTP-**заголовок **X-Frame- Options**, указывающий, разрешена ли загрузка ресурса во фрейме или **iframe.** При желании вы можете отключить эту защиту или даже установить ее на уровне каждого вида. Однако делать это следует с большой осторожностью и тщательностью.

HTTPS/SSL

Все современные веб-сайты должны использовать **HTTPS**, который обеспечивает зашифрованную связь между клиентом и сервером. Протокол **HTTP (Hypertext Transfer Protocol)** является основой современного Интернета, но по умолчанию он не предусматривает шифрования.

Буква **"s"** в **HTTPS** указывает на его зашифрованный характер сначала благодаря **SSL (Secure Sockets Layer),** а в наши дни его преемнику **TLS (Transport Layer Security).** При включенном **HTTPS,** что мы сделаем в главе о развертывании, злоумышленники не смогут прослушивать входящий и исходящий трафик в поисках таких данных, как учетные данные аутентификации или ключи **API.**

Одним из **4** оставшихся вопросов в нашем контрольном списке развертывания **Django** является то, что **SECURE_SSL_REDIRECT** в настоящее время установлен в **False.** По соображениям безопасности, гораздо лучше заставить это значение быть **True** в производстве. Давайте изменим это сейчас, установив по умолчанию значение **True** и добавив значение для локальной разработки в **docker-compose.yml .**

Code

```
# config/settings.py
SECURE_SSL_REDIRECT = env.bool("DJANGO_SECURE_SSL_REDIRECT",
default=True)
```

Затем добавьте переменную окружения в **docker-compose.yml,** где она имеет значение **False.**

docker-compose.yml

```
# docker-compose.yml
environment:
    - "DJANGO_SECRET_KEY=)*_s#exg*#w+#-xt=vu8b010%%a&p@4edwyj0=(nqq90b9a8*n"
    - "DJANGO_DEBUG=True"
    - "DJANGO_SECURE_SSL_REDIRECT=False" # new
```

Перезапустите **Docker** и снова запустите контрольный список развертывания**.**

Command Line

```
$ docker-compose down
$ docker-compose -f docker-compose-prod.yml up -d --build
$ docker-compose exec web python manage.py check --deploy
```

У нас осталось **3** темы**.**

Строгая транспортная безопасность HTTP (HSTS)

HTTP Strict Transport Security (HSTS) - это политика безопасности**,** которая позволяет нашему серверу обеспечить**,** чтобы веб**-**браузеры взаимодействовали только через **HTTPS** путем добавления заголовка **Strict-Transport-Security.**

В нашем файле **settings.py** есть три неявные конфигурации **HSTS,** которые необходимо обновить для производства**:**

- **SECURE_HSTS_SECONDS = 0**
- **SECURE_HSTS_INCLUDE_SUBDOMAINS = False**
- **SECURE_HSTS_PRELOAD = False**

Параметр **SECURE_HSTS_SECONDS** по умолчанию установлен на **0,** но в целях безопасности чем больше**,** тем лучше**.** В нашем проекте мы установим значение в один месяц**, 2 592 000** секунд**.**

SECURE_HSTS_INCLUDE_SUBDOMAINS заставляет поддомены также исключительно использовать **SSL,** поэтому в производстве мы установим значение **True.**

SECURE_HSTS_PRELOAD имеет эффект только при ненулевом значении для **SECURE_HSTS_- SECONDS,** но поскольку мы только что установили одно значение**,** нам нужно установить его в **True.**

Вот как должен выглядеть обновленный файл настроек**.**

Code

```
# config/settings.py
SECURE_HSTS_SECONDS = env.int("DJANGO_SECURE_HSTS_SECONDS", default=2592000)

SECURE_HSTS_INCLUDE_SUBDOMAINS = env.bool("DJANGO_SECURE_HSTS_INCLUDE_SUBDOMAINS",
        default=True)

SECURE_HSTS_PRELOAD = env.bool("DJANGO_SECURE_HSTS_PRELOAD", default=True)
```

Затем обновите **docker-compose.yml** со значениями локальной разработки**.**

docker-compose.yml

```
# docker-compose.yml
environment:
   - "DJANGO_SECRET_KEY=)*_s#exg*#w+#-xt=vu8b010%%a&p@4edwyj0=(nqq90b9a8*n"
   - "DJANGO_DEBUG=True"
   - "DJANGO_SECURE_SSL_REDIRECT=False"
   - "DJANGO_SECURE_HSTS_SECONDS=0" # new
   - "DJANGO_SECURE_HSTS_INCLUDE_SUBDOMAINS=False" # new
   - "DJANGO_SECURE_HSTS_PRELOAD=False" # new
```

Перезапустите **Docker** и снова запустите контрольный список развертывания**.**

Command Line

```
$ docker-compose down
$ docker-compose -f docker-compose-prod.yml up -d --build
$ docker-compose exec web python manage.py check --deploy
```

Осталось всего **2** вопроса**!**

Безопасные файлы cookie

HTTP Cookie используется для хранения информации на компьютере клиента**,** например**,** учетных данных для аутентификации**.** Это необходимо**,** потому что протокол **HTTP** по своей конструкции не имеет состояния**:** нет никакого способа определить**,** аутентифицирован ли пользователь**,** кроме как включить идентификатор в **HTTP-**заголовок**!**

Django использует для этого сессии и **cookies,** как и большинство веб**-**сайтов**.** Но куки можно и нужно принудительно использовать и по **HTTPS** с помощью конфигурации **SESSION_COOKIE_SECURE.** По умолчанию в **Django** установлено значение **False,** поэтому в **production** мы должны изменить его на **True.**

Вторая проблема **-** **CSRF_COOKIE_SECURE,** которая по умолчанию **False,** но в **production** должна быть **True,** чтобы только куки**,** помеченные как "безопасные"**,** отправлялись при **HTTPS-**соединении**.**

Code

```
# config/settings.py
SESSION_COOKIE_SECURE = env.bool("DJANGO_SESSION_COOKIE_SECURE", default=True)
CSRF_COOKIE_SECURE = env.bool("DJANGO_CSRF_COOKIE_SECURE", default=True)
```

Затем обновите файл **docker-compose.yml.**

docker-compose.yml

```
# docker-compose.yml
environment:
      - "DJANGO_SECRET_KEY=)*_s#exg*#w+#-xt=vu8b010%%a&p@4edwyj0=(nqq90b9a8*n"
      - "DJANGO_DEBUG=True"
      - "DJANGO_SECURE_SSL_REDIRECT=False"
      - "DJANGO_SECURE_HSTS_SECONDS=0"
      - "DJANGO_SECURE_HSTS_INCLUDE_SUBDOMAINS=False"
      - "DJANGO_SECURE_HSTS_PRELOAD=False"
      - "DJANGO_SESSION_COOKIE_SECURE=False" # new
      - "DJANGO_CSRF_COOKIE_SECURE=False" # new
```

Перезапустите **Docker** и снова запустите контрольный список развертывания**.**

Command Line

```
$ docker-compose down
$ docker-compose -f docker-compose-prod.yml up -d --build
$ docker-compose exec web python manage.py check --deploy
System check identified no issues (0 silenced).
```

Больше никаких проблем**.** Ух ты**!**

Защита администрации

Пока что может показаться**,** что общий совет по безопасности заключается в том**,** чтобы полагаться на настройки **Django** по умолчанию**,** использовать **HTTPS,** добавлять теги **csrf_token** в формы и устанавливать структуру разрешений**.** Все это верно**.** Но один дополнительный шаг**,** который **Django** не делает от нашего имени **-** это усиление защиты администратора **Django.**

Подумайте о том**,** что каждый сайт **Django** по умолчанию устанавливает администратора на **URL /admin.** Это главный подозреваемый для любого хакера**,** пытающегося получить доступ к сайту **Django.** Поэтому**,** простой шаг **-** просто изменить **URL-**адрес администратора на любой другой**!** Откройте и измените путь **URL.** В данном примере это **anything-but-admin/ .**

Code

```
# config/urls.py
from django.conf import settings
from django.conf.urls.static import static
from django.contrib import admin
from django.urls import path, include

urlpatterns = [
# Django admin
    path('anything-but-admin/', admin.site.urls), # new
# User management
    path('accounts/', include('allauth.urls')),
# Local apps
    path('', include('pages.urls')),
    path('books/', include('books.urls')),
]
if settings.DEBUG:
    import debug_toolbar
    urlpatterns = [
        path('__debug__/', include(debug_toolbar.urls)),
] + urlpatterns
```

Забавный сторонний пакет **django-admin-honeypot** создаст поддельный экран входа в систему администратора и отправит администраторам сайта **IP-**адреса всех**,** кто пытается атаковать ваш сайт по адресу **/admin .** Затем эти **IP-**адреса могут быть добавлены в список заблокированных адресов для сайта**.**

Также с помощью **django-two-factor-auth** можно добавить двухфакторную аутентификацию в админку для еще большего уровня защиты**.**

Git

В этой главе было особенно много изменений кода**,** поэтому убедитесь**,** что все обновления зафиксированы в **Git.**

Command Line

```
$ git status
$ git add .
$ git commit -m 'ch16'
```

Если у вас возникли ошибки**,** проверьте журналы **docker-compose logs** и сравните свой код с официальным исходным кодом на **Github.**

Заключение

Безопасность - одна из главных проблем любого веб-сайта. Используя файл **docker-compose-prod.yml**, мы можем точно протестировать в **Docker** наши производственные настройки перед развертыванием сайта в реальном времени. А используя значения по умолчанию, мы можем упростить переменные окружения в файле, а также гарантировать, что если что-то пойдет не так с переменными окружения, мы по умолчанию будем использовать производственные значения, а не небезопасные локальные! **Django** поставляется со многими встроенными функциями безопасности, и с добавлением контрольного списка развертывания мы можем теперь развернуть наш сайт с высокой степенью уверенности, что он безопасен.

В конечном счете, безопасность - это постоянная борьба, и хотя шаги в этой главе охватывают большинство проблемных областей, поддержание вашего сайта в актуальном состоянии с последней версией **Django** является жизненно важным для постоянной безопасности.

Глава 17: Развертывание

До сих пор мы работали исключительно в локальной среде разработки на своем компьютере. Но теперь пришло время развернуть наш проект, чтобы он был доступен для общественности. По правде говоря, тема развертывания достойна отдельной книги. По сравнению с другими веб-фреймворками **Django** очень свободен и агностичен в этом вопросе. Для большинства хостинговых платформ не существует развертывания одним щелчком мыши, и хотя это требует большей работы разработчиков, это также позволяет, в типичной для **Django** манере, высокую степень кастомизации.

В предыдущей главе мы настроили совершенно отдельный файл **docker-compose-prod.yml** и обновили **config/settings.py**, чтобы сделать его готовым к производству. В этой главе мы рассмотрим, как выбрать хостинг-провайдера, добавить готовый к работе веб-сервер и правильно настроить статические/медиа файлы перед развертыванием нашего сайта книжного магазина!

PaaS против IaaS

Первый вопрос - использовать ли платформу как услугу **(PaaS)** или инфраструктуру как услугу **(IaaS). PaaS -** это вариант хостинга, который выполняет большую часть начальной настройки и масштабирования, необходимых для веб-сайта. Популярные примеры **- Heroku**, **PythonAnywhere**, **Dokku** и многие другие. Хотя **PaaS** стоит больше денег, чем **IaaS**, он экономит невероятное количество времени разработчиков, автоматически обрабатывает обновления безопасности и может быть быстро масштабирован.

IaaS, напротив, обеспечивает полную гибкость, обычно дешевле, но требует больших знаний и усилий для правильной настройки. Известные варианты **IaaS** включают **DigitalOcean**, **Linode**, **Amazon EC2**, **Google Compute Engine** и многие другие.

Так какой же из них использовать? Разработчики **Django**, как правило, относятся к одному из двух лагерей: либо у них уже есть конвейер развертывания, настроенный на выбранную ими **IaaS**, либо они используют **PaaS**. Поскольку первый вариант намного сложнее и сильно варьируется в своей конфигурации, в этой книге мы будем использовать **PaaS**, а именно **Heroku**.

Выбор **Heroku** в некоторой степени произволен, но это зрелая технология, которая поставляется с действительно бесплатным уровнем, достаточным для развертывания нашего проекта **Bookstore**.

WhiteNoise

Для локальной разработки **Django** полагается на приложение **staticfiles**, которое автоматически собирает и обслуживает статические файлы со всего проекта. Это удобно, но довольно неэффективно и, скорее всего, небезопасно.

Для производства необходимо запустить **collectstatic** для компиляции всех статических файлов в один каталог, указанный **STATIC_ROOT**. Затем их можно обслуживать либо на том же сервере, либо на отдельном сервере, либо на выделенном облачном сервисе/**CDN** путем обновления **STATICFILES_STORAGE**.

В нашем проекте мы будем полагаться на обслуживание файлов с нашего сервера с помощью проекта **WhiteNoise**, который отлично работает на **Heroku** и является более быстрым и настраиваемым, чем **Django** по умолчанию.

Первым шагом будет установка **whitenoise** в **Docker** и остановка запущенных контейнеров.

Command Line

```
$ docker-compose exec web pipenv install whitenoise==5.1.0
$ docker-compose down
```

Мы не будем пока пересобирать образ, потому что нам также нужно внести изменения в наши настройки. Поскольку мы используем **Docker**, можно перейти на **WhiteNoise** как локально, так и на производстве. Хотя это можно сделать, передав флаг **--nostatic** в команду **runserver**, на практике это становится утомительным. Лучший подход - это добавить **whitenoise.runserver_nostatic** перед **django.contrib.staticfiles** в конфигурации **INSTALLED_APPS**, что сделает то же самое. Мы также добавим его в наш **MIDDLEWARE** прямо под **SecurityMiddleware** и обновим **STATICFILES_STORAGE**, чтобы теперь использовать **WhiteNoise**.

Code

```
# config/settings.py
INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
     'whitenoise.runserver_nostatic', # new
     'django.contrib.staticfiles',
     'django.contrib.sites',
...
]

MIDDLEWARE = [
    'django.middleware.cache.UpdateCacheMiddleware',
    'django.middleware.security.SecurityMiddleware',
    'whitenoise.middleware.WhiteNoiseMiddleware', # new
...
]

STATICFILES_STORAGE = 'whitenoise.storage.CompressedManifestStaticFilesStorage'
```

После внесения всех изменений мы можем снова запустить наш проект в режиме локальной разработки.

Command Line

```
$ docker-compose up -d --build
```

WhiteNoise имеет дополнительные опции для обслуживания сжатого содержимого и заголовков кэша далекого будущего на содержимом, которое не будет меняться. А пока выполните команду **collectstatic** еще раз.

Command Line

```
$ docker-compose exec web python manage.py collectstatic
```

Появится предупреждение о перезаписи существующих файлов. Это нормально. Введите **"yes"** и нажмите клавишу **"Return"**, чтобы продолжить.

Медиафайлы

WhiteNoise, к сожалению, не очень хорошо работает с загруженными пользователями медиафайлами. Наши обложки книг добавляются через админку **Django**, но методом, аналогичным загружаемым пользователем файлам. В результате, хотя в локальной разработке они будут отображаться так, как нужно, они не будут отображаться в производственной среде. Более подробную информацию смотрите в документации.

Рекомендуемый подход - использовать очень популярный пакет **django-storages** вместе с выделенной **CDN**, такой как **S3**. Однако это требует дополнительной настройки, которая выходит за рамки данной книги.

Gunicorn

Когда мы выполняли команду **startproject** в главе **3**, был создан файл **wsgi.py** с конфигурацией **WSGI (Web Server Gateway Interface)** по умолчанию. Это спецификация того, как веб-приложение (например, наш проект **Bookstore)** взаимодействует с веб-сервером.

В производстве принято менять ее на **Gunicorn** или **uWSGI**. Оба предлагают увеличение производительности, но **Gunicorn** более ориентирован и прост в реализации, поэтому мы выберем его.

Первым шагом будет его установка в нашем проекте и остановка наших контейнеров.

Command Line

```
$ docker-compose exec web pipenv install gunicorn==20.0.4
$ docker-compose down
```

Поскольку мы используем **Docker,** наше локальное окружение может легко имитировать производственное**,** поэтому мы обновим **docker-compose.yml** и **docker-compose-prod.yml,** чтобы использовать **Gunicorn** вместо локального сервера**.**

docker-compose.yml

```
# command: python /code/manage.py runserver 0.0.0.0:8000
command: gunicorn config.wsgi -b 0.0.0.0:8000 # new
```

docker-compose-prod.yml

```
# command: python /code/manage.py runserver 0.0.0.0:8000
command: gunicorn config.wsgi -b 0.0.0.0:8000 # new
```

Теперь снова запустите контейнеры**,** создав новый образ с пакетом **Gunicorn** и нашими обновленными переменными окружения**.**

Heroku

Перейдите на сайт **Heroku** и зарегистрируйте бесплатный аккаунт**.** После подтверждения электронной почты **Heroku** перенаправит вас в раздел сайта с приборной панелью**.**

Далее убедитесь**,** что установили интерфейс командной строки **(CLI) Heroku,** чтобы мы могли выполнять развертывание из командной строки**.** Здесь есть подробные инструкции**.**

Последний шаг **-** войти в систему с учетными данными **Heroku** через командную строку**,** набрав **heroku login .** Используйте электронную почту и пароль для **Heroku,** которые вы только что установили**.**

Command Line

```
$ heroku login
```

Все готово**!** Если у вас возникли вопросы**,** вы можете набрать **heroku help** в командной строке или посетить сайт **Heroku** для получения дополнительной информации**.**

Развертывание с помощью Docker

Теперь перед нами стоит выбор: развертывание традиционным способом на **Heroku** или с помощью контейнеров **Docker**. Последний вариант - это новый подход, который **Heroku** и другие хостинг-провайдеры добавили совсем недавно. Однако так же, как **Docker** захватил локальную разработку, он начинает захватывать и развертывание. И как только вы настроили контейнеры для развертывания, гораздо проще переключаться между потенциальными хостинг-провайдерами, чем если бы вы настраивали их специфическим способом. Поэтому мы будем развертывать с помощью контейнеров **Docker**.

Но даже в этом случае нам снова предстоит сделать выбор, поскольку существует два варианта контейнеров: использование реестра контейнеров для развертывания предварительно созданных образов или добавление файла **heroku.yml**. Мы будем использовать последний подход, поскольку он позволяет использовать дополнительные команды и более точно имитирует традиционный подход **Heroku -** добавление **Procfile** для конфигурации.

heroku.yml

Традиционный не-**Docker Heroku** полагается на пользовательский **Procfile** для настройки сайта для развертывания. Для контейнеров **Heroku** использует аналогичный подход - пользовательский файл, но под названием **heroku.yml** в корневом каталоге. Он похож на **docker-compose.yml**, который используется для создания локальных контейнеров **Docker**.

Давайте теперь создадим наш файл **heroku.yml**.

Command Line

```
$ touch heroku.yml
```

Для настройки доступны четыре раздела верхнего уровня: **setup, build, release** и **run**. Основная функция **setup -** указать, какие дополнения необходимы. Это размещенные решения, которые предоставляет **Heroku**, как правило, за плату. Главное из них - наша база данных, которая будет полагаться на бесплатный уровень **heroku-postgresql**. **Heroku** позаботится о ее создании, обновлениях безопасности, и мы сможем легко увеличить размер и время работы базы данных по мере необходимости.

Раздел **build -** это то, как мы указываем, что **Dockerfile** должен быть, ну, создан. Он опирается на наш текущий **Dockerfile** в корневом каталоге.

Фаза релиза используется для выполнения задач перед развертыванием каждого нового релиза. Например, мы можем убедиться, что **collectstatic** запускается при каждом развертывании автоматически.

Наконец, есть фаза запуска, где мы указываем, какие процессы фактически запускают приложение. Примечательно, что в качестве веб-сервера используется **Gunicorn**.

heroku.yml

```
setup:
    addons:
        - plan: heroku-postgresql
build:
   docker:
        web: Dockerfile
release:
   image: web
   command:
        - python manage.py collectstatic --noinput
run:
   web: gunicorn config.wsgi
```

Не забудьте добавить новые обновления развертывания в **Git** и зафиксировать их**.** В следующем разделе мы перенесем весь наш локальный код на сам **Heroku.**

Command Line

```
$ git status
$ git add .
$ git commit -m 'ch17'
```

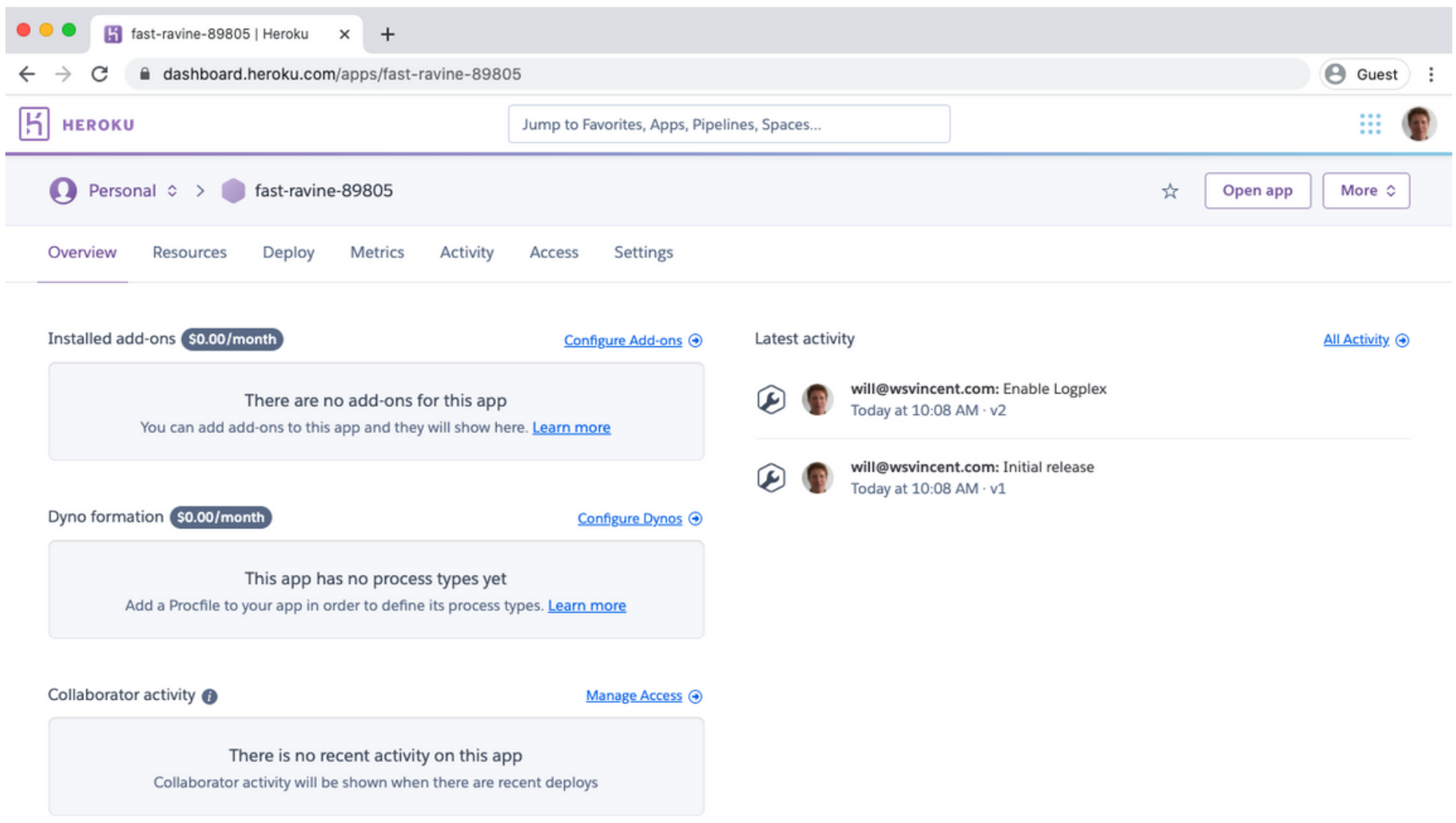
Развертывание на Heroku

Теперь создайте новое приложение на **Heroku** для нашего проекта **Bookstore.** Если вы наберете **heroku create**, **Heroku** присвоит случайное имя**.** Поскольку имена в **Heroku** глобальные**,** маловероятно**,** что будут доступны такие распространенные имена**,** как **"blog"** или **"webapp".** Имя всегда можно изменить позже в **Heroku** на доступное глобальное пространство имен**.**

Command Line

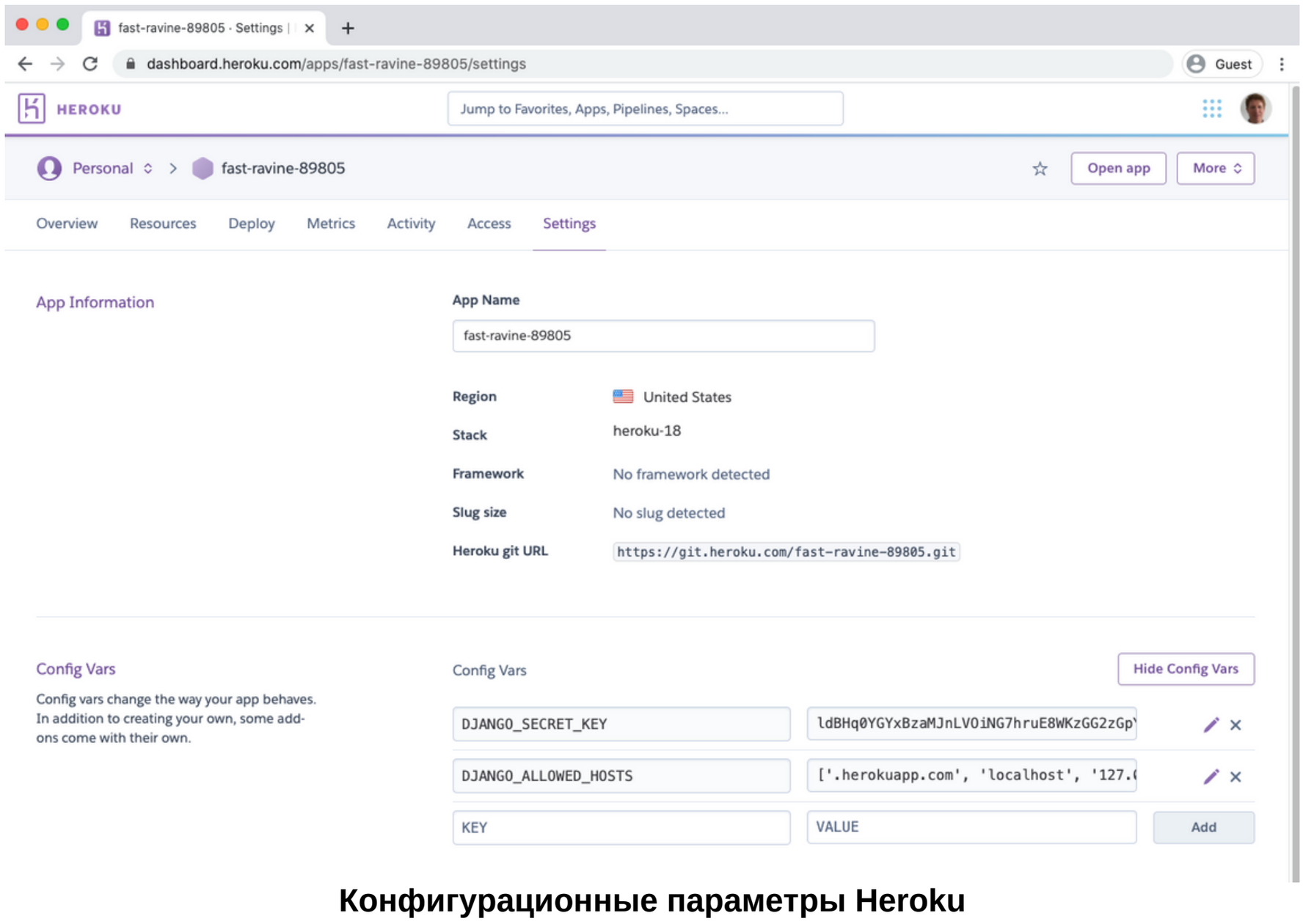
```
$ heroku create
Creating app... done, â¬¢ fast-ravine-89805
https://fast-ravine-89805.herokuapp.com/ |
https://git.heroku.com/fast-ravine-89805.git
```

В данном случае **Heroku** присвоил моему приложению имя **fast-ravine-89805 .** Если вы обновите приборную панель **Heroku** на сайте**,** вы увидите только что созданное приложение**.** Нажмите на новое приложение**,** чтобы открыть страницу "Обзор"**.**



Страница обзора Heroku

Следующим шагом будет добавление переменных производственного окружения**.** Нажмите на опцию "Настройки" в верхней части страницы**,** а затем нажмите на "Раскрыть конфигурационные переменные"**.** Поскольку мы так свободно используем значения по умолчанию**,** необходимо установить только два значения**: DJANGO_SECRET_KEY** и **DJANGO_ ALLOWED_HOSTS .** И поскольку мы только что узнали конкретное доменное имя нашего производственного сайта **- fast-ravine-89805.herokuapp.com/** в моем случае **-** мы можем добавить его в **ALLOWED_HOSTS** для максимальной безопасности**.**



Конфигурационные параметры Heroku

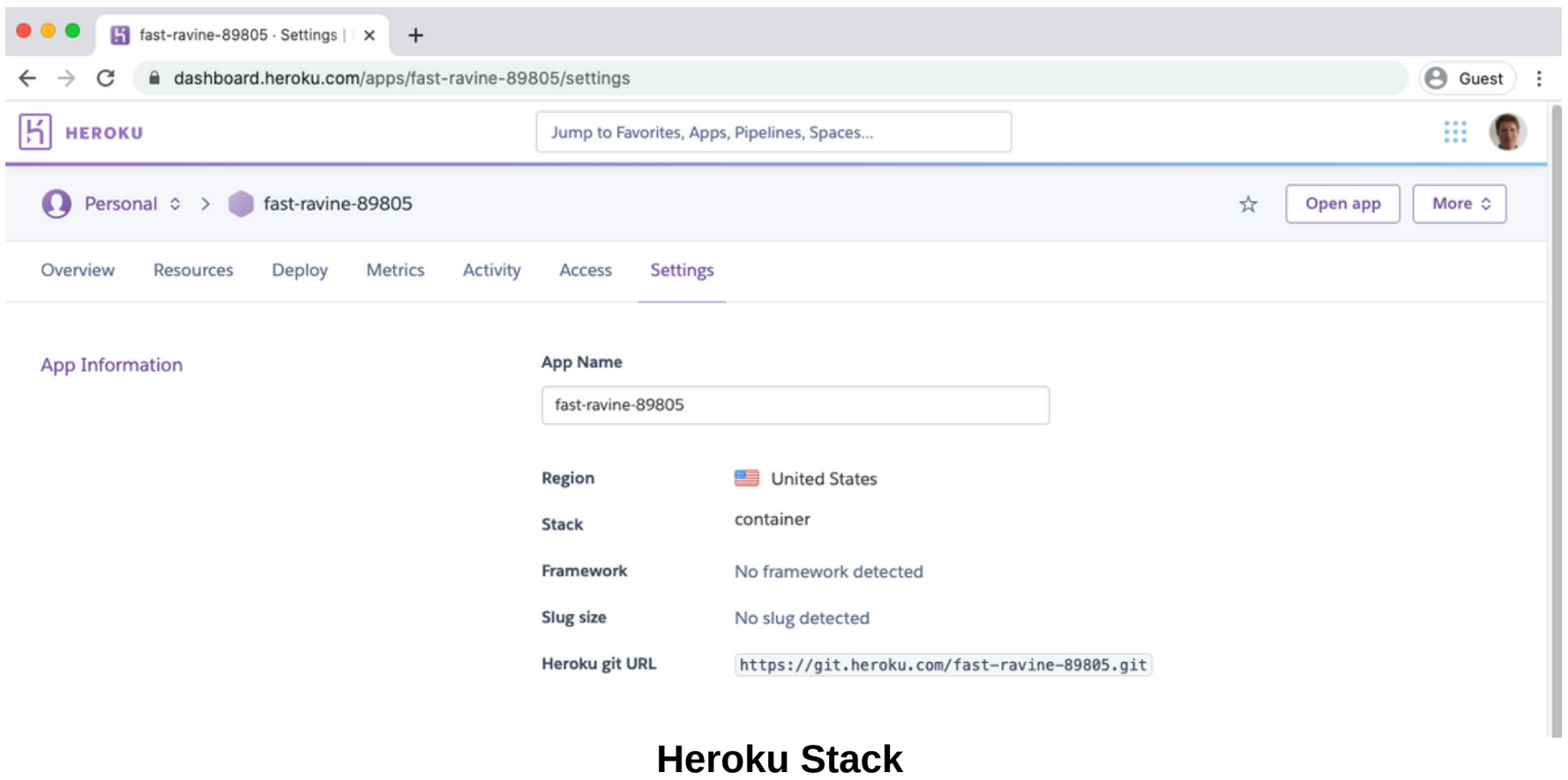
Также можно добавить переменные конфигурации из командной строки в **Heroku.** Оба подхода работают**.**

Теперь настройте стек на использование наших контейнеров **Docker,** а не **buildpack** по умолчанию от **Heroku.** Включите имя вашего приложения в конце команды после **heroku stack:set container -a .**

Command Line

```
$ heroku stack:set container -a fast-ravine-89805
Setting stack to container... done
```

Чтобы убедиться, что это изменение было выполнено правильно, обновите веб-страницу панели **Heroku** и обратите внимание, что в разделе **" Info"** для **"Stack"** теперь есть **"container".** Это то, что нам нужно**.**



Heroku Stack

Перед размещением нашего кода на **Heroku** укажите базу данных **PostgreSQL,** которую мы хотим разместить на хостинге**.** В нашем случае хорошо подходит бесплатный уровень **hobby-dev;** его всегда можно обновить в будущем**.**

Command Line

```
$ heroku addons:create heroku-postgresql:hobby-dev -a fast-ravine-89805
Creating heroku-postgresql:hobby-dev on â¬¢ fast-ravine-89805... free
Database has been created and is available
 ! This database is empty. If upgrading, you can transfer
 ! data from another database with pg:copy
Created postgresql-curved-34718 as DATABASE_URL
Use heroku addons:docs heroku-postgresql to view documentation
```

Вы заметили, что переменная **DATABASE_URL** была создана автоматически**.** Поэтому нам не пришлось устанавливать ее в качестве переменной окружения**.** Мы готовы**!** Создайте **Heroku remote,** то есть версию нашего кода, которая будет жить на сервере, размещенном на **Heroku.** Не забудьте указать **-a** и название вашего приложения**.** Затем "протолкните" код в **Heroku,** что приведет к сборке нашего образа **Docker** и запуску контейнеров**.**

Command Line

```
$ heroku git:remote -a fast-ravine-89805
set git remote heroku to https://git.heroku.com/fast-ravine-89805.git
$ git push heroku master
```

Первоначальный толчок может занять некоторое время. Вы можете видеть активный прогресс, перейдя на вкладку " **Activity**" на приборной панели **Heroku.**

Теперь наш проект **Bookstore** должен быть доступен онлайн. Помните, что, хотя код зеркально отражает наш собственный локальный код, на производственном сайте есть своя собственная база данных, в которой нет никакой информации. Для запуска команд на нем добавьте **heroku run** к стандартным командам. Например, мы должны перенести нашу начальную базу данных, а затем создать учетную запись суперпользователя.

Command Line

```
$ heroku run python manage.py migrate
$ heroku run python manage.py createsuperuser
```

Есть два способа открыть только что развернутое приложение. Из командной строки вы можете набрать **heroku open -a** и название вашего приложения. Или вы можете нажать на кнопку "Открыть приложение" в правом верхнем углу панели **Heroku.**

Command Line

```
$ heroku open -a fast-ravine-89805
```


Перенаправления Heroku

Но... ах! Что это? Ошибка перенаправления. Добро пожаловать в радость развертывания, где подобные проблемы будут возникать постоянно.

SECURE_PROXY_SSL_HEADER

Некоторое исследование показало, что проблема связана с нашим параметром **SECURE_SSL_REDIRECT**. **Heroku** использует прокси, поэтому мы должны найти соответствующий такой заголовок и обновить **SECURE_PROXY_SSL_HEADER** соответствующим образом.

По умолчанию он установлен на **None**, но поскольку мы доверяем **Heroku**, мы можем обновить его **('HTTP_X_FORWARDED_PROTO', 'https').** Эта настройка не повредит нам для локальной разработки, поэтому мы добавим ее непосредственно в файл **config/settings.py** следующим образом:

Code

```
# config/settings.py
SECURE_PROXY_SSL_HEADER = ('HTTP_X_FORWARDED_PROTO', 'https') # new
```

Зафиксируйте эти изменения в **Git** и перенесите обновленный код на **Heroku.**

Command Line

```
$ git status
$ git commit -m 'secure_proxy_ssl_header and allowed_hosts update'
$ git push heroku master
```

После завершения сборки обновите веб-страницу вашего сайта. Вот он!

Живой сайт Heroku

Журналы **Heroku**

В какой-то момент неизбежно возникнут ошибки при развертывании. Когда это произойдет, выполните команду **heroku logs --tail**, чтобы увидеть журналы ошибок и информации и отладить происходящее.

Надеюсь, процесс развертывания прошел гладко. Но на практике, даже при использовании такой надежной платформы как **Heroku**, высока вероятность возникновения проблем. Если вы видите страницу с ошибкой, введите **heroku logs --tail** , который отображает журналы информации и ошибок, чтобы диагностировать проблему.

Дополнения для Heroku

Heroku поставляется с большим списком дополнительных сервисов**,** которые за определенную плату можно быстро добавить к любому сайту**.** Например**,** чтобы включить кэширование с помощью **Memcache,** следует рассмотреть вариант **Memcachier.**

Ежедневное резервное копирование является дополнительной**,** но необходимой функцией любой производственной базы данных**.**

А если вы используете пользовательский домен для своего сайта**,** обеспечение **SSL** является жизненно важным для любого сайта**.** Чтобы включить эту функцию**,** вам потребуется платный уровень на **Heroku.**

Заключение

В этой главе было много кода**,** поэтому если у вас возникли ошибки**,** пожалуйста**,** проверьте официальный исходный код на **Github.**

Даже при всех преимуществах современной платформы как сервиса**,** такой как **Heroku,** развертывание остается сложной и часто разочаровывающей задачей для многих разработчиков**.** Лично я хочу**,** чтобы мои веб-приложения "просто работали"**.** Но многим инженерам нравится работать над производительностью**,** безопасностью и масштабированием**.** В конце концов**,** гораздо легче измерить улучшения в этой сфере**:** уменьшилось ли время загрузки страницы**?** Увеличилось ли время работы сайта**?** Обновлена ли система безопасности**?** Работа над этими проблемами часто приносит гораздо больше удовольствия**,** чем обсуждение того**,** какую новую функцию добавить на сайт**.**

Заключение

Создание "профессионального" веб-сайта - задача не из легких, даже с учетом всей той помощи, которую оказывает входящий в состав батарей веб-фреймворк, такой как **Django**. **Docker** дает большое преимущество в стандартизации локальных и производственных сред независимо от локальной машины - и особенно в контексте команды. Однако **Docker** - это сложный зверь сам по себе. Хотя мы разумно использовали его в этой книге, он может сделать гораздо больше в зависимости от потребностей проекта.

Сам **Django** дружелюбен к небольшим проектам, поскольку его настройки по умолчанию ориентированы на быструю локальную разработку, но эти настройки должны систематически обновляться для производства, от обновления базы данных до **PostgreSQL**, использования пользовательской модели, переменных окружения, настройки потока регистрации пользователей, статических активов, электронной почты... и так далее.

Хорошей новостью является то, что шаги, необходимые для подхода на уровне производства, довольно схожи. Поэтому первая половина этой книги намеренно не зависит от конечного проекта, который создается: вы обнаружите, что эти шаги являются стандартными практически для любого нового проекта **Django**. Вторая половина книги была посвящена созданию реального сайта книжного магазина с использованием современных лучших практик, добавлению отзывов, загрузке изображений, установке прав доступа, добавлению поиска, рассмотрению производительности и мер безопасности, и, наконец, развертыванию на **Heroku** с помощью контейнеров.

При всем содержании этой книги мы действительно только поцарапали поверхность того, что может **Django**. Такова природа современной веб-разработки: постоянная итерация.

Django - великолепный партнер в создании профессионального веб-сайта, потому что многие необходимые соображения уже продуманы и учтены. Но необходимы знания, чтобы знать, как включить эти производственные переключатели, чтобы использовать все преимущества настройки, которую позволяет **Django**. В конечном счете, это и есть цель данной книги: познакомить вас, читателя, со всем спектром того, что требует **Django** и профессиональные веб-сайты.

По мере того, как вы будете узнавать больше о веб-разработке и **Django**, я бы призвал вас к осторожности, когда дело доходит до преждевременной оптимизации. Всегда заманчиво добавить в свой проект функции и оптимизации, которые, как вы думаете, понадобятся вам позже. Короткий список включает в себя добавление **CDN** для статических и мультимедийных файлов, разумный анализ запросов к базе данных, добавление индексов в модели и т.д.

Правда заключается в том, что в любом конкретном веб-проекте всегда будет больше работы, чем позволяет время. В этой книге рассмотрены основы, на которые стоит потратить время заранее. Дополнительные шаги, связанные с безопасностью, производительностью и возможностями, будут появляться у вас в режиме реального времени. Постарайтесь не поддаваться желанию усложнять систему до тех пор, пока это не станет абсолютно необходимым.

Обучающие ресурсы

По мере освоения **Django** и веб-разработки в целом, вы обнаружите, что официальная документация **Django** и исходный код становятся все более ценными. Я обращаюсь к ним почти ежедневно. Существует также официальный форум **Django** - отличный ресурс, хотя и недостаточно используемый для решения вопросов, связанных с **Django**.

Для продолжения вашего путешествия по **Django** хорошим источником дополнительных учебников и курсов является сайт **LearnDjango.com**, который я поддерживаю. Также есть еженедельный подкаст **Django Chat**, со-ведущим которого является **Django Fellow** Карлтон Гибсон, и **Django News**, еженедельная рассылка, наполненная последними новостями, статьями и учебниками по **Django**.

Обратная связь

В заключение я хотел бы услышать ваши мысли о книге. Она постоянно находится в процессе работы, и подробные отзывы читателей помогают мне продолжать ее совершенствовать. Я отвечаю на каждое письмо, и со мной можно связаться по адресу **will@learndjango.com**.

Если вы купили эту книгу на **Amazon**, пожалуйста, подумайте о том, чтобы оставить честный отзыв. Такие отзывы оказывают огромное влияние на продажи книг.

Спасибо, что прочитали книгу, и удачи вам в вашем путешествии с **Django!**