

# **Django** для начинающих

Научитесь веб-разработки с Django 2.0

William S. Vincent

© 2018 William S. Vincent

# Содержание

<b>Введение</b>	<b>1</b>
Почему Django	1
Почему эта книга	3
Структура книги	3
Макет книги	5
Вывод	7
<b>Глава 1: Первоначальная Настройка</b>	<b>8</b>
Командная строка	8
Установка Python 3 на Mac OS X	11
Установка Python 3 на Windows	13
Установка Python 3 на Linux	14
Установка Виртуальных Среды	15
Джанго	16
Установить Git	20
Текстовый редактор	21
Вывод	21
<b>Глава 2: Hello World приложение</b>	<b>22</b>
Начальная настройка	22
Создание приложения	25
Views и URL конфигурация	27
Привет мир!	30
Git	31
Bitbucket	33
Вывод	37
<b>Глава 3: Страницы приложения</b>	<b>39</b>
Начальная настройка	39
Шаблоны	41
Основа Классов	44
Views URLs	45
Добавление страницы	47

## ОГЛАВЛЕНИЕ

Расширение Шаблонов	49
Тесты	52
Git и Bitbucket	54
Local vs Production	56
Heroku(облачная PaaS-платформа)	57
Дополнительный файл	58
Deploy(развертывание)	61
Вывод	64
<b>Глава 4: Приложение Доска Объявлений</b>	<b>65</b>
Начальная настройка	65
Создание модели базы данных	68
Активация моделей	69
Джанго Админ	70
Views/Templates/URLs	77
Добавление новых записей	81
Тесты	85
Bitbucket	90
Heroku конфигурация	91
Heroku развертывание	93
Выводы	95
<b>Глава 5: Приложение Блог</b>	<b>96</b>
Начальная настройка	96
Модель базы данных	98
Админ	100
URLs	105
Views	107
Templates	108
Static files	111
Отдельные страницы блога	116
Тесты	122
Git	125
Выводы	125
<b>Глава 6: Формы</b>	<b>127</b>
Формы	127
Обновление форм	138
Удаление View	145
Тесты	151
Выводы	155
<b>Глава 7: Учетные Записи Пользователей</b>	<b>156</b>

## ОГЛАВЛЕНИЕ

Вход	156
Обновление Домашней страницы	159
Ссылка "выход"	161
Регистрация	164
Bitbucket	170
Конфигурация Heroku	171
Развертывание на Heroku	173
Выводы	177
<b>Глава 8: Пользовательская Модель</b>	<b>178</b>
Установка	178
Пользовательская User Model	180
Формы	182
Суперпользователь	185
Выводы	187
<b>Глава 9: Аутентификация Пользователя</b>	<b>188</b>
Templates	188
URLs	192
Admin	197
Выводы	203
<b>Глава 10: Bootstrap</b>	<b>204</b>
Приложение Страницы	204
Тесты	208
Bootstrap	211
Форма Регистрации	219
Дальнейшие действия	224
<b>Глава 11: смена и сброс пароля</b>	<b>226</b>
Изменение пароля	226
Настройка изменения пароля	228
Сброс пароля	231
Пользовательские шаблоны	235
Вывод	240
<b>Глава 12: Электронная Почта</b>	<b>241</b>
SendGrid	241
Пользовательская эл.почта	246
Вывод	250
<b>Глава 13: Приложение газета</b>	<b>251</b>
Приложение Статьи	251

## ОГЛАВЛЕНИЕ

URLs и Views	257
Редактирование/Удаление	262
Создание страницы	268
Выводы	276
<b>Глава 14: Права доступа и авторизация</b>	<b>277</b>
Улучшаем CreateView	277
Авторизация	279
Mixins	281
Обновление views	283
Вывод	285
<b>Глава 15: Комментарии</b>	<b>286</b>
Model	286
Admin	288
Template	295
Вывод	299
<b>Вывод</b>	<b>301</b>
Ресурсы Django	302
Питон книги	302
Блоги	303
Обратная связь	303

# Введение

Добро пожаловать в Django для начинающих, проектный подход к обучению веб-разработке с помощью веб-платформы Django. В этой книге вы создадите пять все более сложных веб-приложений, начиная с простого приложения "Привет, мир", переходя к приложению блога с формами и учетными записями пользователей, и, наконец, приложение газеты, используя пользовательскую модель, интеграцию электронной почты, внешние ключи, авторизацию, права доступа и многое другое.

К концу этой книги вы должны чувствовать себя уверенно, создавая свои собственные проекты Django с нуля, используя лучшую современную практику.

Django-это бесплатный веб-фреймворк с открытым исходным кодом, написанный на языке программирования Python и используемый миллионами программистов каждый год. Его популярность обусловлена дружелюбием как к начинающим, так и продвинутым программистам: Django достаточно надежен для использования крупнейшими веб-сайтами в мире – Instagram, Pinterest, Bitbucket, Disqus, но также достаточно гибок, чтобы быть хорошим выбором для стартапов на ранней стадии и прототипирования личных проектов.

Эта книга регулярно обновляется и содержит последние версии Django (2.0) и Python (3.6 x). Он также использует Pipenv, который теперь официально Рекомендуемый менеджер пакетов Python.org для управления пакетами Python и виртуальными средами. На всем протяжении мы будем использовать лучшие современные практики сообщества Django, Python и веб-разработки, особенно тщательное использование тестирования.

## Почему Django

Веб-фреймворк - это набор модульных инструментов, которые абстрагируют большую часть трудностей и повторений, присущих веб-разработке. Например, большинству веб-сайтов

нужен такой же базовый функционал: возможность подключения к базе данных, установка URL маршрутов, отображение контента на странице, корректная защита и так далее. Вместо того, чтобы воссоздавать все это с нуля, программисты на протяжении многих лет создавали веб-фреймворки на всех основных языках программирования: Django и Flask в Python, Rails в Ruby и Express в JavaScript среди многих, многих других.

Django унаследовал подход Python “batteries-included” и включает в себя поддержку из коробки для общих задач в веб-разработке:

- Аутентификация пользователя
- Шаблоны, маршруты и представления
- Интерфейс администратора
- Надежная безопасность
- Поддержка нескольких серверных баз данных
- И многое другое

Такой подход значительно упрощает нашу работу как веб-разработчиков. Мы можем сосредоточиться на этом, что делает наше веб-приложение уникальным, а не изобретать колесо, когда дело доходит до стандартной функциональности веб-приложения. Напротив, несколько популярных фреймворков—в первую очередь Flask на Python и Express на JavaScript—используют подход “микропрограммирования”. Они предоставляют только минимум, необходимый для простой веб-страницы, и оставляют на усмотрение разработчика установку и настройку сторонних пакетов для репликации базовой функциональности веб-сайта. Такой подход обеспечивает большую гибкость для разработчика, но также дает больше возможностей для ошибок. По состоянию на 2018 Год Django находится в активной разработке более 13 лет, что делает его седым ветераном в программных годах. Миллионы программистов уже использовали Django для создания своих сайтов. И это, несомненно, хорошо. Веб-разработка трудна. Не имеет смысла повторять один и тот же код—и ошибки—когда, большое сообщество блестящих разработчиков уже решило эти проблемы для нас.

В то же время, Django остается в стадии активной разработки и имеет годовой график выпуска. Сообщество Django постоянно добавляет новые функции и улучшения безопасности. Если вы создаете сайт с нуля Джанго это отличный выбор.

## Почему эта книга

Я написал эту книгу, не смотря на то, что Django очень хорошо документирован потому что существует серьезная нехватка доступных учебников для начинающих. Когда я впервые узнал Django несколько лет назад, я выбивался из сил пытаюсь выполнять по официальному учебнику. По этому я так хорошо помню, что думал.

С большим опытом сейчас я признаю, что авторы документации Django столкнулись с трудным выбором: они могли бы подчеркнуть простоту использования Django или его глубину, но не то и другое одновременно. Они выбирают последнее, и как профессиональный Разработчик я ценю этот выбор, но как новичок я нашел это как...разочарование!

Моя цель состоит в том, что бы эта книга заполняла пробелы и демонстрировала насколько дружелюбен к новичкам Django.

Вам не нужен прошлый опыт Python или веб-разработки для завершения этой книги. Она специально написан так, что даже новичок может следовать по ней и почувствовать магию написания собственных веб-приложений с нуля. Однако, если вы серьезно относитесь к карьере в веб-разработке, вам в конечном итоге нужно будет потратить время, чтобы изучить Python, HTML и CSS должным образом. В заключение приводится перечень рекомендуемых ресурсов для дальнейшего изучения.

## Структура Книги

Мы начнем с правильного описания того, как настроить локальную среду разработки в **главе 1**. В этой книге мы используем кропотливые инструменты: самую последнюю версию



Django (2.0), Python (3.6) и Pipenv для управления нашими виртуальными средами. Мы также пройдем введение в командную строку и обсудим как работать с современным текстовым редактором.

В **Главе 2** мы создадим наш первый проект, минимальное Hello, World приложение, которое демонстрирует, как настроить новые проекты Django. Так как создание хорошего программного обеспечения важно, мы также сохраним нашу работу в git и загрузим копию в удаленный репозиторий кода на Bitbucket.

В **Главе 3** мы создадим, протестируем и развернем Приложение Pages, которое представляет шаблоны и view на основе классов. Шаблоны это то, как Django позволяет создавать (Не Повторяйтесь - принцип разработки программного обеспечения, нацеленный на снижение повторения информации различного рода) разработку с помощью HTML и CSS, в то время как view на основе классов требуют минимального количества кода для использования и расширения основной функциональности в Django. Они потрясающие, вы это скоро увидите. Мы также добавим наши первые тесты и развернем в Heroku, который имеет свободный уровень и который мы будем использовать на протяжении всей этой книги. Использование таких поставщиков услуг, как Heroku, превращает разработку из болезненного, трудоемкого процесса во что-то, что занимает всего несколько щелчков мыши.

В **Главе 4** мы создадим наш первый проект с базой данных, приложение для доски объявлений. Django предоставляет мощный ORM, который позволяет нам писать сжатый Python для наших таблиц базы данных. Мы рассмотрим встроенное приложение администратора, которое обеспечивает графический способ взаимодействия с нашими данными и может быть даже использовано в качестве системы управления контентом (CMS), аналогичной Wordpress. Также мы напишем тесты для всего нашего кода, сохраним удаленную копию на Bitbucket и развернем в Heroku. Наконец, в **Главах 5-7** мы готовы к нашему финальному проекту: надежному приложению для блога, которое демонстрирует, как выполнять функциональность CRUD (Create-Read-Update-Delete)(**Создание-Чтение-Обновление-Удаление**) в Django. Мы обнаружим, что общие view на основе классов Django означают, что для этого нам нужно написать лишь небольшое количество реального кода! Затем мы добавим формы и интегрируем встроенную систему аутентификации пользователей Django (login, logout, signup).

В **Главе 8** мы начнем сайт газеты и введем понятие пользовательских моделей, Django лучшая практика, которая редко рассматривается в учебных пособиях. Проще говоря все новые

проекты должны использовать пользовательскую модель, и в этой главе вы узнаете, как это сделать. **Глава 9** охватывает аутентификацию пользователей, **Глава 10** добавляет Bootstrap для стилизации, а **главы 11-12** реализуют сброс пароля и изменение по электронной почте. В **главах 13-15** мы добавляем статьи и комментарии к нашему проекту, а также соответствующие разрешения и полномочия. Мы даже узнаем некоторые приемы для настройки администратора, чтобы отобразить наши растущие данные.

В **заключении** дается обзор основных понятий, введенных в книгу, и список рекомендуемых ресурсов для дальнейшего обучения.

Хотя вы можете выбирать главы для чтения, структура книги очень продумана. Каждое **приложение / глава** вводит новую концепцию и усиливает прошлое обучение. Я настоятельно рекомендую прочитать его по порядку, даже если вы хотите пропустить перейдя вперед. Более поздние главы не будут охватывать предыдущий материал в той же глубине, что и предыдущие главы.

К концу этой книги вы будете иметь представление о том, как работает Django, возможность создавать приложения самостоятельно, и основу необходимую, чтобы в полной мере воспользоваться дополнительными ресурсами для изучения промежуточных и передовых методов Django.

## Book layout

В этой книге есть много примеров кода, которые обозначаются следующим образом:

### Code

---

```
# This is Python code
print('Hello, World')
```

---

Для краткости 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 часто будем использовать консоль командной строки (начиная с **главы 1: Начальная настройка** для выполнения команд, которые принимают форму префикса \$ в традиционном стиле Unix.

## Command Line

---

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

---

Результатом этой конкретной команды является следующая строка:

## Command Line

---

```
"hello, world"
```

---

Обычно мы объединяем команду и ее вывод. Команда всегда будет после символа \$, а выходные данные - нет. Например, команда и результат выше будут представлены следующим образом:

## Command Line

---

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

---

Полный исходный код всех примеров можно найти в [официальном репозитории github](#).

## Вывод

Django-отличный выбор для любого разработчика, который хочет создавать современные, надежные веб-приложения с минимальным количеством кода. Он популярен, находится в стадии активной разработки и тщательно протестирован крупнейшими веб-сайтами в мире. В следующей главе мы узнаем, как настроить компьютер для разработки Django.

# Глава 1 : Начальная настройка

В этой главе описывается, как правильно настроить компьютер для работы с Django проектом. Мы начнем с обзора командной строки и используем ее для установки последних версий Django (2.0) и Python (3.6 x). Затем мы обсудим виртуальные среды, git и работу с текстовым редактором.

К концу этой главы вы будете готовы создавать и изменять новые проекты Джанго за несколько кликов.

## Командная строка

Командная строка-это мощное текстовое представление компьютера. Как разработчики мы будем использовать его широко на протяжении всей этой книги, чтобы установить и настроить каждый проект Django.

На Mac командная строка находится в программе Terminal, расположенной в каталоге / Applications / Utilities. Чтобы найти его, откройте новое окно Finder, откройте папку Applications, прокрутите вниз, чтобы открыть папку Utilities, и дважды щелкните приложение под названием Terminal.

В Windows есть встроенная программа командной строки, но это трудно использовать. Я рекомендую вместо этого использовать Babun, бесплатную программу командной строки с открытым исходным кодом. На главной странице сайта [Babun](#) нажмите кнопку "Загрузить сейчас", дважды щелкните загруженный файл, чтобы установить Babun, и по завершении перетащите установщик в корзину. Чтобы использовать Babun в меню Пуск, выберите программы, и нажмите на Babun.

Забегая вперед, когда книга ссылается на "командную строку", это означает открыть новую консоль на вашем компьютере с помощью терминала или Babun.

Хотя есть много возможных команд, которые мы можем использовать, на практике есть шесть наиболее часто используемых в разработке Django.

- `cd` (изменить каталог)
- `cd ..` (перейти в верх)
- `ls` (список файлов в текущем каталоге)
- `pwd` (вывести текущий каталог)
- `mkdir` (создать каталог)
- `touch` (создать новый файл)

Откройте командную строку и попробуйте их. Знак `$` dollar-это наша командная строка: все команды в этой книге предназначены для ввода после `$`.

Например, предположим, что вы находитесь на Mac, давайте перейдем в каталог рабочего стола.

### Command Line

---

```
$ cd ~/Desktop
```

---

Обратите внимание, что наше текущее местоположение `~/Desktop` автоматически добавляется перед командной строкой. Чтобы убедиться, что мы находимся в правильном месте, мы можем использовать `pwd`, который распечатает путь к нашему текущему каталогу.

### Command Line

---

```
~/Desktop $ pwd  
/Users/wsv/desktop
```

---

На моем компьютере Mac это показывает, что я использую пользователя `wsv` и нахожусь на рабочем столе для этой учетной записи.

Давайте создадим новую папку каталога с `mkdir`, `cd` и добавим в нее новый файл `index.html`.

**Command Line**

---

```
~/Desktop $ mkdir new_folder
~/Desktop $ cd new_folder
~/Desktop/new_folder $ touch index.html
```

---

Теперь используйте `ls` для вывода списка всех текущих файлов в нашем каталоге. Вы увидите только что созданный `index.html`.

**Command Line**

---

```
~/Desktop/new_folder $ ls
index.html
```

---

В качестве последнего шага вернитесь в каталог рабочего стола `cd ..` и используйте `pwd` для подтверждения местоположения.

**Command Line**

---

```
~/Desktop/new_folder $ cd ..
~/Desktop $ pwd
/Users/wsv/desktop
```

---

Продвинутые разработчики могут с легкостью использовать клавиатуру и командную строку для навигации по компьютеру ; с практикой этот подход станет намного быстрее, чем с помощью мыши.

В этой книге я дам вам точные инструкции для выполнения, вам не нужно быть экспертом в командной строке и со временем это будет хороший навык для любого профессионального разработчика программного обеспечения. Два хороших бесплатных ресурса для дальнейшего изучения-это [Command Line Crash Course](#) и [CodeCademy's Course on the Command Line](#).

Ниже приведены инструкции для компьютеров Mac, Windows и Linux.

## Установка Python 3 на Mac OS X

Хотя Python 2 установлен по умолчанию на компьютерах Mac, Python3 в нем нет. Вы можете подтвердить это, набрав `python --version` в консоли командной строки и нажав Enter:

### Command Line

---

```
$ python --version
```

```
Python 2.7.13
```

---

Чтобы проверить, установлен ли Python 3, попробуйте выполнить ту же команду, используя `python3` вместо `python`.

### Command Line

---

```
$ python3 --version
```

---

Если ваш компьютер выводит `3.6.x` (любая версия 3.6 или выше), то у вас все в порядке, однако, скорее всего, вы увидите сообщение об ошибке, так как нам нужно установить Python 3 непосредственно.

Наш первый шаг-установить пакет Xcode от Apple, поэтому выполните следующую команду, чтобы установить его:



**Command Line**

---

```
$ xcode-select --install
```

---

Щелкните по всем командам подтверждения (Xcode - это большая программа, поэтому для установки может потребоваться некоторое время, в зависимости от вашего интернет-соединения).

Затем установите менеджер пакетов Homebrew через длинную команду ниже:

**Command Line**

---

```
$ /usr/bin/ruby -e "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/master/install)"
```

---

Чтобы проверить правильность установки Homebrew, выполните следующую команду:

**Command Line**

---

```
$ brew doctor
```

```
Your system is ready to brew.
```

---

И что бы установить последнюю версию Python, выполните следующую команду:

**Command Line**

---

```
$ brew install python3
```

---

Сейчас мы проверим, какая версия была установлена:

### Command Line

---

```
$ python3 --version
```

```
Python 3.6.4
```

---

Чтобы открыть интерактивную оболочку Python 3 (это позволит нам запускать команды Python прямо на нашем компьютере )просто введите **python3** :

### Command Line

---

```
$ python3
```

```
Python 3.6.4 (default, Jan 7 2018, 13:05:00)
```

```
[GCC 4.2.1 Compatible Apple LLVM 9.0.0 (clang-900.0.39.2)] on darwin
```

```
Type "help", "copyright", "credits" or "license" for more information.
```

```
>>>
```

---

Для выхода из интерактивной оболочки Python 3 в любое время нажмите Control-d (клавиши "Control" и "d" одновременно).

Вы все еще можете запускать оболочки Python с Python 2, просто набрав python:

### Command Line

---

```
$ python
```

```
Python 2.7.13 (default, Dec 18 2016, 07:03:39)
```

```
[GCC 4.2.1 Compatible Apple LLVM 8.0.0 (clang-800.0.42.1)] on darwin
```

```
Type "help", "copyright", "credits" or "license" for more information.
```

---

## Установка Python 3 на Windows

Python не включен по умолчанию в Windows, однако мы можем проверить, существует ли какая-либо версия в системе. Откройте консоль командной строки, введя *cmd* в меню Пуск (Поиск Windows)

Или вы можете удерживать клавишу SHIFT и щелкнуть правой кнопкой мыши на рабочем столе, а затем выбрать «Открыть командное окно» здесь или - "Открыть окно PowerShell здесь" в windows 10.

Введите следующую команду и нажмите Enter:

#### Command Line

---

```
python --version
```

```
Python 3.6.4
```

---

Если вы видите такой вывод, Python уже установлен. Но скорее всего, этого не будет! Чтобы загрузить Python 3, перейдите в раздел загрузки официального сайта Python. Загрузите установщик и убедитесь, что вы выбрали опцию Add Python to PATH , которая позволит использовать python прямо из командной строки. В противном случае мы должны были бы ввести полный путь нашей системы и изменить наши переменные среды вручную. После установки Python выполните следующую команду в новой консоли командной строки:

#### Command Line

---

```
python --version
```

```
Python 3.6.4
```

---

Если это сработало, вы закончили!

## Установка Python 3 на Linux

Добавление Python 3 в дистрибутив Linux требует немного больше работы. Здесь рекомендуются последние руководства для Centos и Debian. Если вам нужна Дополнительная помощь в добавлении Python к вашему пути, обратитесь к этому ответу StackOverflow.

## Виртуальная среда

Виртуальные среды являются неотъемлемой частью программирования Python. Они представляют собой изолированный контейнер, содержащий все зависимости программного обеспечения для данного проекта. Это важно, потому что по умолчанию программное обеспечение, такое как Python и Django, установлено в одном каталоге. Это приводит к проблеме, когда требуется работать над несколькими проектами на одном компьютере. Что делать, если проект использует Django 2.0, но ProjectB с прошлого года все еще находится на Django 1.10? Без виртуальных сред это становится очень трудно; с виртуальными средами это вообще не проблема.

Есть много областей разработки программного обеспечения, которые горячо обсуждаются, но использование виртуальных сред для разработки Python не является одним. Для каждого нового проекта Python следует использовать выделенную виртуальную среду.

Исторически разработчики Python использовали `virtualenv` или `pyenv` для настройки виртуальных сред. Но в 2017 известных разработчиков Python Kenneth Reitz выпустил `Pipenv` который теперь официально Рекомендуемый инструмент упаковки питона.

`Pipenv` похож на `npm` и `yarn` из экосистемы узлов: он создает файл канала, содержащий зависимости программного обеспечения и `Pipfile`. Блокировка для обеспечения детерминированных построений. “Детерминизм” означает, что каждый раз, когда вы загружаете программное обеспечение в новой виртуальной среде, у вас будет точно такой же конфигурации. Себастьян Маккензи, создатель `Yarn`, который впервые представил эту концепцию для упаковки JavaScript, имеет краткое сообщение в блоге, объясняющее, что такое детерминизм и почему это имеет значение.

Конечным результатом является то, что мы создадим новую виртуальную среду с `pipenv` для каждого нового проекта Django.

Для установки `Pipenv` мы можем использовать `pip`, который автоматически установился для нас вместе с Python 3.

**Command Line**

---

```
$ pip3 install pipenv
```

---

## Установка Django

Чтобы увидеть Pipenv в действии, давайте создадим новый каталог и установим Django.

Сначала перейдите на рабочий стол, создайте новый каталог django и введите его с cd.

**Command Line**

---

```
$ cd ~/Desktop
```

```
$ mkdir django
```

```
$ cd django
```

---

Теперь используйте Pip env для установки Django.

**Command Line**

---

```
$ pipenv install django
```

---

Если вы посмотрите в наш каталог, есть два новых файла: Pipfile и Pipfile.lock. У нас есть информация, необходимая для новой виртуальной среды, но мы ее еще не активировали.

Давайте сделаем это с помощью оболочки pip env shell.

**Command Line**

---

```
$ pipenv shell
```

---

Если вы находитесь на Mac, вы должны увидеть круглые скобки в командной строке с активированной средой. Он примет формат имени каталога и случайных символов. На моем компьютере я вижу нижеследующее, но вы увидите что-то немного другое: он начнется с django- но закончится случайной серией символов.

Обратите внимание, что из-за открытой ошибки пользователи Windows не будут видеть визуальную обратную связь виртуальной среды в это время. Но если вы сможете запустить `Django-admin startproject` в следующем разделе, то вы узнаете, что в вашей виртуальной среде установлен Django.

### Command Line

---

```
(django-JmZ1NTQw) $
```

---

Это означает, что все работает! Создайте новый проект Django с именем `test` с помощью следующей команды. Не забудьте поставить точку в конце через пробел.

### Command Line

---

```
(django-JmZ1NTQw) $ django-admin startproject test_project .
```

---

Стоит остановиться здесь, чтобы объяснить, почему вы должны добавить точку через пробел к команде. Если вы просто запустите `django-admin startproject test_project`, то по умолчанию Django создаст эту структуру каталогов:

```
└─ test_project
    └─ manage.py
        └─ test_project
            ├── __init__.py
            ├── settings.py
            ├── urls.py
            └─ wsgi.py
```

Посмотрите, как он создает новый каталог `test_project`, а затем в нем `manage.py` файл и каталог `test_project`? Это кажется мне излишним, так как мы уже создали и перешли в папку `django` на нашем рабочем столе. Запустив `django-admin startproject test_project .` с точкой в конце-который говорит, установите в текущем каталоге Результат вместо этого будет:

```
|— manage.py
└─ test_project
    |— __init__.py
    |— settings.py
    |— urls.py
    └─ wsgi.py
```

Вывод заключается в том, что на самом деле **не имеет большого значения**, ставите ли вы точку или нет в конце команды, но я предпочитаю ставить ее, и именно так мы сделаем в этой книге.

Теперь давайте проверим, что все работает, запустив локальный веб-сервер Django.

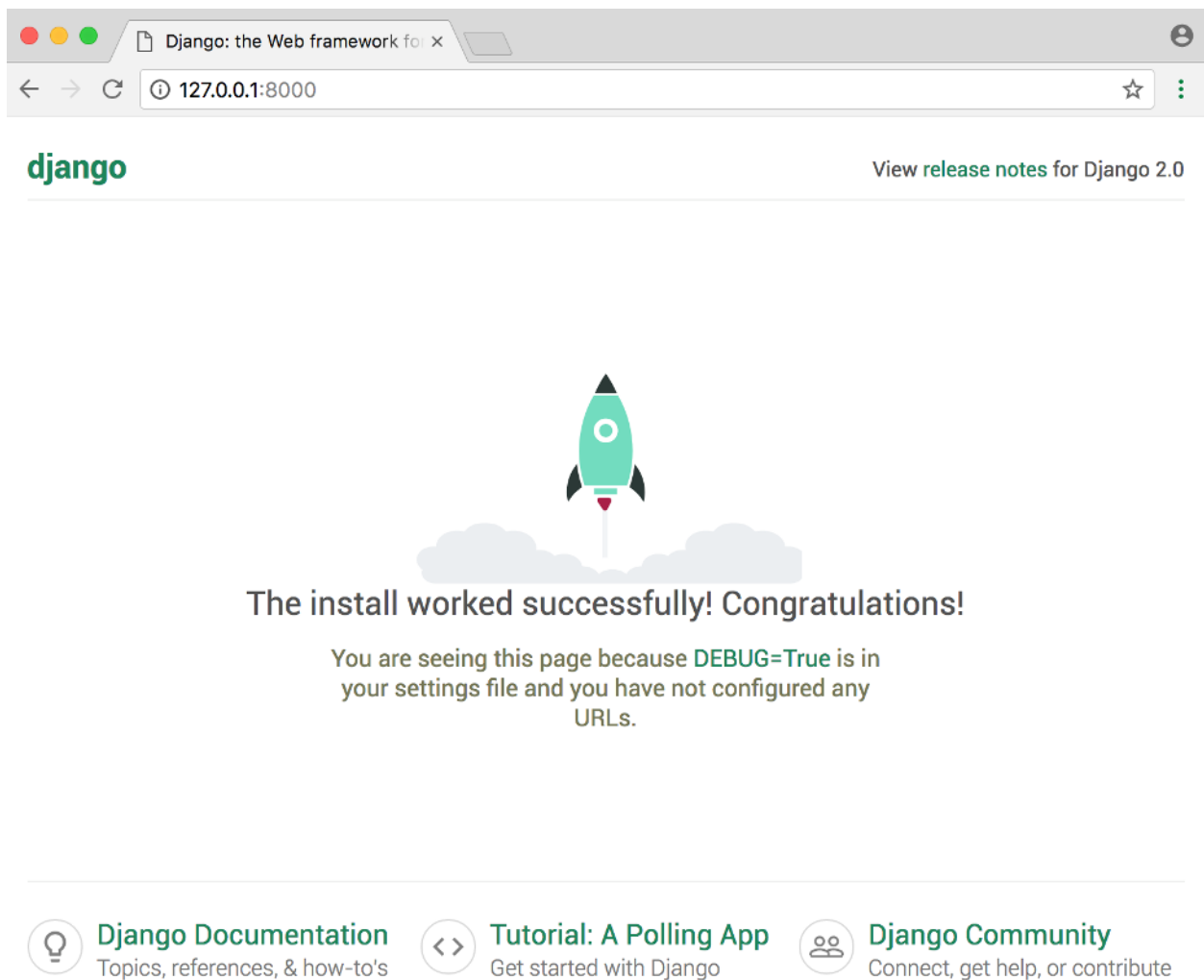
### Command Line

---

```
(django-JmZ1NTQw) $ python manage.py runserver
```

---

Если вы перейдете в <http://127.0.0.1:8000/> вы должны увидеть следующее изображение:



### Страница приветствия Django

Чтобы остановить наш локальный сервер, введите Control-c. Затем выйдите из нашей виртуальной среды с помощью команды `exit`.

### Command Line

```
(django-JmZ1NTQw) $ exit
```

Мы всегда можем снова активировать виртуальную среду с помощью `pipenv shell` в любое время.

Мы получим много практики с виртуальными средами в этой книге, так что не волнуйтесь, если это



немного сбивает вас с толку сейчас. Основной шаблон заключается в установке новых пакетов с `pipenv`, активируйте их с помощью `pipenv shell`, а затем выйти, когда закончите `exit`.

Стоит отметить, что одновременно на вкладке командной строки может быть активна только одна виртуальная среда. В следующих главах мы будем создавать совершенно новую виртуальную среду для каждого нового проекта. Так что либо не забудьте выйти из текущей среды, либо откройте новую вкладку для новых проектов.

## Установка Git

Git является неотъемлемой частью современной разработки программного обеспечения. Это система контроля версий, которая может рассматриваться как чрезвычайно мощная версия отслеживания изменений в Microsoft Word или Google Docs. С помощью git вы можете сотрудничать с другими разработчиками, отслеживать всю вашу работу с помощью фиксаций и возвращаться к любой предыдущей версии вашего кода, даже если вы случайно удалили что-то важное!

На Mac, поскольку Homebrew уже установлен, мы можем просто ввести `brew install git` в командной строке:

### Command Line

---

```
$ brew install git
```

---

В Windows вы должны скачать Git из [Git for Windows](#). Нажмите кнопку «Загрузить» и следуйте инструкциям по установке. Нажмите «Далее» на всех шагах, кроме пятого, “Adjusting your PATH environment.” Вместо этого выберите нижнюю опцию: “Use Git and optional Unix tools from the Windows Command Prompt.”

После установки нам нужно выполнить одноразовую настройку системы, чтобы настроить ее, объявив имя и адрес электронной почты, которые вы хотите связать со всеми вашими обязательствами Git (подробнее об этом в ближайшее время).

В консоли командной строки введите следующие две строки. Не забудьте обновить их имя и адрес электронной почты.

## Command Line

---

```
$ git config --global user.name "Your Name"
$ git config --global user.email "yourname@email.com"
```

---

Вы всегда можете изменить эти настройки, если пожелаете, введя те же команды с новым именем или адресом электронной почты.

## Текстовой редактор

Последний шаг это ваш текстовый редактор. В то время как командная строка, где мы выполняем команды для наших программ по сути текстовый редактор. Компьютеру все равно, какой текстовый редактор вы используете конечный результат просто код, но хороший текстовый редактор может предоставить полезные советы и поймать опечатки для вас.

Опытные разработчики часто предпочитают использовать либо Vim, либо Emacs, оба десятилетних текстовых редактора с лояльными последователями. Однако каждый из них имеет крутую кривую обучения и требует запоминания множества различных комбинаций клавиш. Я не рекомендую их новичкам.

Современные текстовые редакторы сочетают в себе те же мощные функции с привлекательным визуальным интерфейсом. Мой текущий фаворит это Visual Studio Code, который является бесплатным, простым в установке и пользуется широкой популярностью. Если вы еще не используете текстовый редактор, загрузите и установите Visual Studio Code.

## Вывод

УФ! Никто на самом деле не любит настраивать локальную среду разработки, но, к счастью, это одноразовая боль. Теперь мы научились работать с виртуальными средами и установили последнюю версию Python и git. Все готово для вашего первого приложения на Django.

## Глава 2 : Hello World приложение

В этой главе мы будем строить проект Django, который просто говорит: “Привет, мир” на главной странице. Это традиционный способ начать новый язык программирования или фреймворк. Мы также впервые будем работать с git и развертывать наш код в Bitbucket.

### Начальная настройка

Для начала перейдите в новый каталог на компьютере. Например, мы можем создать папку helloworld на рабочем столе с помощью следующих команд.

#### Command Line

---

```
$ cd ~/Desktop  
$ mkdir helloworld  
$ cd helloworld
```

---

Убедитесь, что вы еще не находитесь в существующей виртуальной среде и у вас отображается текст в скобках () перед знаком доллара \$. Чтобы выйти из него, введите exit и нажмите ENTER. Скобки должны исчезнуть, что означает, что виртуальная среда больше не активна. Мы будем использовать pipenv для создания новой виртуальной среды, установить Django и затем активировать его.

**Command Line**

---

```
$ pipenv install django
```

```
$ pipenv shell
```

---

Если Вы находитесь на Mac, вы должны увидеть скобки в начале командной строки в форме (helloworld-XXX) где XXX представляет случайные символы. На моем компьютере я вижу (helloworld-415ivvZC). Я покажу (helloworld) здесь в тексте, но вы увидите что-то немного другое на своем компьютере. Если Вы находитесь в Windows, Вы не увидите визуальную подсказку в это время.

Создайте новый проект Django под названием helloworld\_project включая пробел с точкой в конце команды, так что бы он установился в нашем текущем каталоге.

**Command Line**

---

```
(helloworld) $ django-admin startproject helloworld_project .
```

---

Если сейчас вы используете команду tree вы можете увидеть, что наша структура проекта Django теперь выглядит так. (Примечание: если tree у вас не работает установите его с Homebrew:: brew install tree.)

**Command Line**

---

```
(helloworld) $ tree
```

```
.
├── helloworld_project
│   ├── __init__.py
│   ├── settings.py
│   ├── urls.py
│   └── wsgi.py
└── manage.py
```

## 1 directory, 5 files

---

Файл `settings.py` управляет настройками нашего проекта, `urls.py` сообщает Django какие страницы для сборки выдать в ответ на запрос браузера или `url`, и `wsgi.py` который обозначает интерфейс шлюза веб-сервера и помогает Django обслуживать наши возможные веб-страницы. Последний файл `manage.py` используется для выполнения различных команд Django, таких как запуск локального веб-сервера или создание нового приложения.

Django поставляется со встроенным веб-сервером для локальных целей разработки. Мы можем запустить его с помощью команды `runserver`.

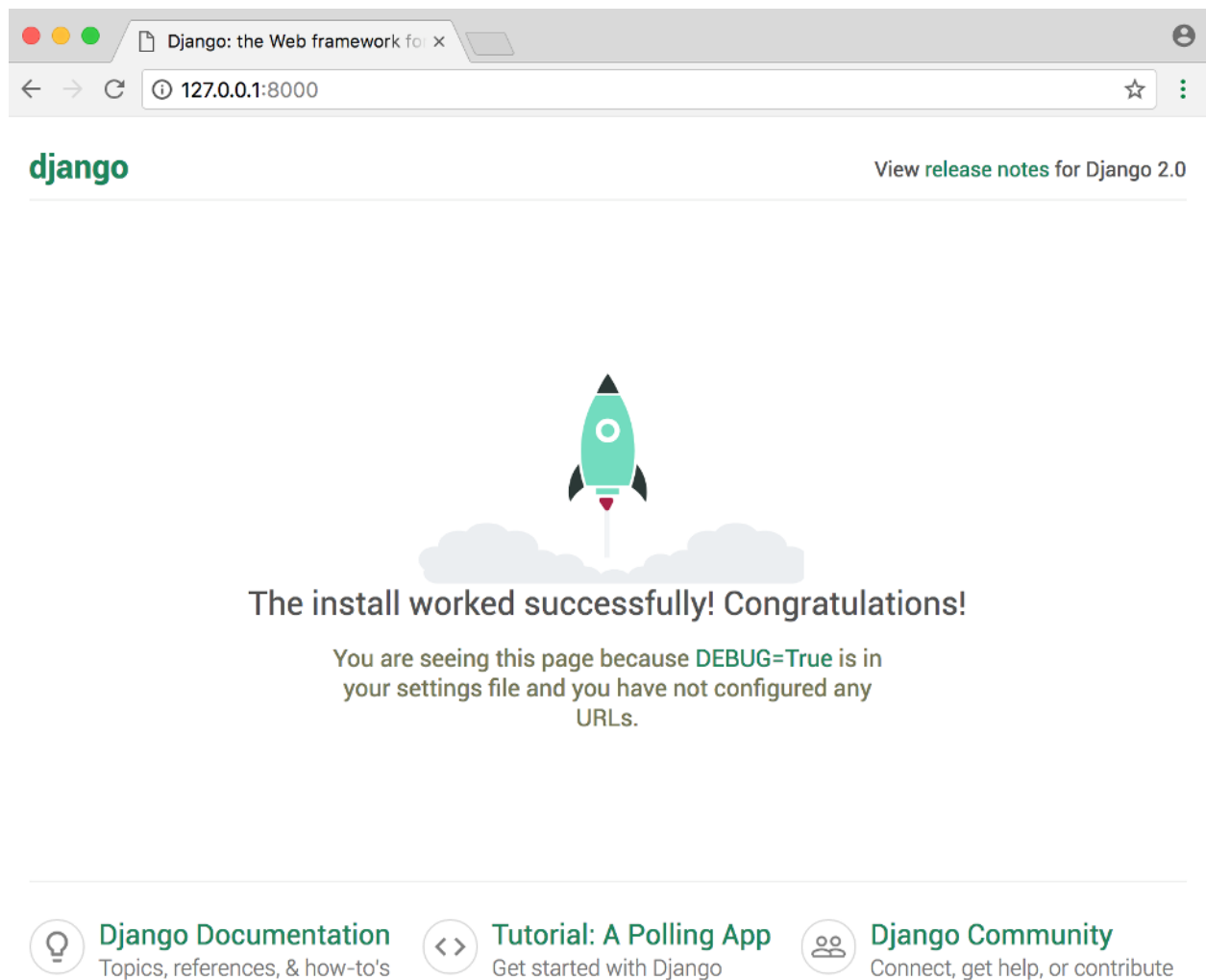
### Command Line

---

```
(helloworld) $ python manage.py runserver
```

---

Если вы посетите <http://127.0.0.1:8000/> вы должны увидеть следующее изображение:



**Django** страница приветствия

## Создание приложения

Django использует концепцию проектов и приложений, чтобы сохранить код чистым и читаемым. Один проект Django содержит одно или несколько приложений, которые все вместе обеспечивают работу всего веб-приложения. Вот почему команда для нового проекта Django-startproject! Например, на реальном сайте электронной коммерции Django может быть одно приложение для проверки подлинности пользователя, другое приложение для платежей и третье приложение для получения сведений о списке элементов. Каждое фокусируется на изолированной части функциональности.

Нам нужно создать наше первое приложение, которое мы назовем `pages`. Из командной строки закройте сервер с помощью `Control+c`. Затем используйте команду `startup`.

### Command Line

---

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

---

Если снова заглянуть в каталог дерева команд вы увидите, что Django создал страницы со следующими файлами:

### Command Line

---

```
|— pages
|   |— __init__.py
|   |— admin.py
|   |— apps.py
|   |— migrations
|   |   └─ __init__.py
|   |— models.py
|   |— tests.py
|   └─ views.py
```

---

Давайте рассмотрим, что делает каждый новый файл приложения `pages`:

- `admin.py` файл конфигурации для встроенного приложения администратора Django
- `apps.py` является конфигурационным файлом для самого приложения
- `migrations/` отслеживает любые изменения в файле `models.py` чтобы синхронизировать нашу базу данных с `models.py`
- `models.py` тут мы определяем наши модели базы данных, которые Django автоматически переводит в таблицы базы данных
- `tests.py` предназначен для тестирования приложений

- `views.py` тут мы обрабатываем логику запроса / ответа для нашего веб-приложения

Несмотря на то, что Наше новое приложение существует в проекте Django, Django не “знает” об этом, пока мы явно не добавим его. В текстовом редакторе откройте `settings.py` файл и прокрутите вниз до `INSTALLED_APPS`, где вы увидите шесть уже встроенных приложений Django. Добавьте наше новое приложение внизу:

### Code

---

```
# helloworld_project/settings.py

INSTALLED_APPS = [

    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    'pages', # new

]
```

---

## Views и URLConfs

В Django, *Views* определяет, **какой** контент выводится на данной странице, *URLConfs* определяет, **где** этот контент будет.

Когда пользователь запрашивает определенную страницу, например домашнюю страницу, *URLConf* использует регулярное выражение для сопоставления этого запроса с соответствующей *Views* функцией, которая затем возвращает правильные данные.

Другими словами, наша функция из *Views* выведет текст «Hello, World», в то время как наш *URL* будет гарантировать, что когда пользователь посещает главную страницу, он будет перенаправляется на правильную функцию *Views*.

Начнем с обновления файла `views.py` в нашем `pages` приложении выглядит это следующим образом:



### Code

---

```
# pages/views.py  
  
from django.http import HttpResponse  
  
def homePageView(request):  
    return HttpResponse('Hello, World!')
```

---

В принципе, мы указываем, что всякий раз, когда вызывается функция `homePageView` вернется текст “Hello, World!” если более конкретно то мы импортировали встроенный `HttpResponse` метод, чтобы мы могли вернуть ответ объекта пользователю. Для этого мы создали функцию с именем `homePageView` которая принимает запрос объекта и возвращает ответ со строкой `Hello, World!`.

Теперь нам нужно настроить наш URL. В приложении `pages` создайте новый `urls.py` файл.

### Command Line

---

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

---

Затем обновите его с помощью следующего кода:

**Code**

---

```
# pages/urls.py

from django.urls import path

from . import views

urlpatterns = [
    path('', views.homePageView, name='home')
]
```

---

В верхней строке мы импортируем `path` из Django, чтобы использовать наш шаблон url, и на следующей строке мы импортируем наш файл `views`. Использованный пробел с точкой `from . import views` означает ссылку на текущий каталог, который является нашим приложением `pages`, содержащим как `views.py`, так и `urls.py`.

Наш шаблон URL состоит из трех частей:

- регулярное выражение Python для пустой строки ""
- указывает функцию `homePageView` из файла `views`
- добавляет не обязательное url имя 'home '

Другими словами, если пользователь запрашивает домашнюю страницу, представленную пустой строкой "", то используется функция файла `views` с именем `homePageView`.

Мы уже почти закончили. Последний шаг это настройка на уровне проекта файла `urls.py`. Помните, что обычно в одном проекте Django есть несколько приложений, поэтому каждому из них нужен свой собственный маршрут.

Обновите файл `helloworld_project/urls.py` следующим образом:

**Code**

---

```
# helloworld_project/urls.py

from django.contrib import admin
from django.urls import path, include

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

---

Мы импортировали `include` во второй строке рядом с `path`, а затем создали новый шаблон url-адреса для нашего приложения `pages`. Теперь, когда пользователь посещает главную страницу `' '`, он сначала будет перенаправлен на приложение `pages`, а затем на `views` домашней страницы.

Это часто путает начинающих, то что нам здесь не нужно импортировать приложение `pages`, но мы ссылаемся на него в нашем шаблоне url как `pages.urls`. Причина, по которой мы делаем это, заключается в том, что метод `django.urls.include()` ожидает, что мы перейдем в модуль или приложение в качестве первого аргумента. Поэтому без использования `include` нам нужно будет импортировать наше приложение `pages`, но поскольку мы используем `include`, мы не должны это делать на уровне проекта!

## Hello, world!

У нас есть весь код, который нам сейчас нужен ! Чтобы подтвердить, что все работает так, как и ожидалось, перезагрузите наш сервер Django:

### Command Line

---

```
(helloworld) $ python manage.py runserver
```

---

Если вы обновите браузер для <http://127.0.0.1:8000/> теперь он отображает текст “Hello, world!”



### Hello world homepage

## Git

В предыдущей главе мы также установили git, который является системой контроля версий. Давайте используем его здесь. Первый шаг - инициализировать (или добавить) git в наш репозиторий.

### Command Line

---

```
(helloworld) $ git init
```

---

Если вы введете `git status` то увидите список изменений с момента последней фиксации(`git commit`). Поскольку это наша первая фиксация, это список всех наших изменений до сих пор.

### Command Line

---

```
(helloworld) $ git status
```

```
On branch master
```

```
No commits yet
```

```
Untracked files:
```

```
(use "git add <file>..." to include in what will be committed)
```

```
db.sqlite3
helloworld_project/
manage.py
pages/
Pipfile
Pipfile.lock
```

ничего не добавлено к фиксации, но присутствуют неотслеживаемые файлы (используйте "git add " для отслеживания)

---

Теперь мы хотим добавить все изменения с помощью команды *add-A*, а затем зафиксировать изменения вместе с сообщением, описывающим, что изменилось.

### Command Line

---

```
(helloworld) $ git add -A
```

```
(helloworld) $ git commit -m 'initial commit'
```

---

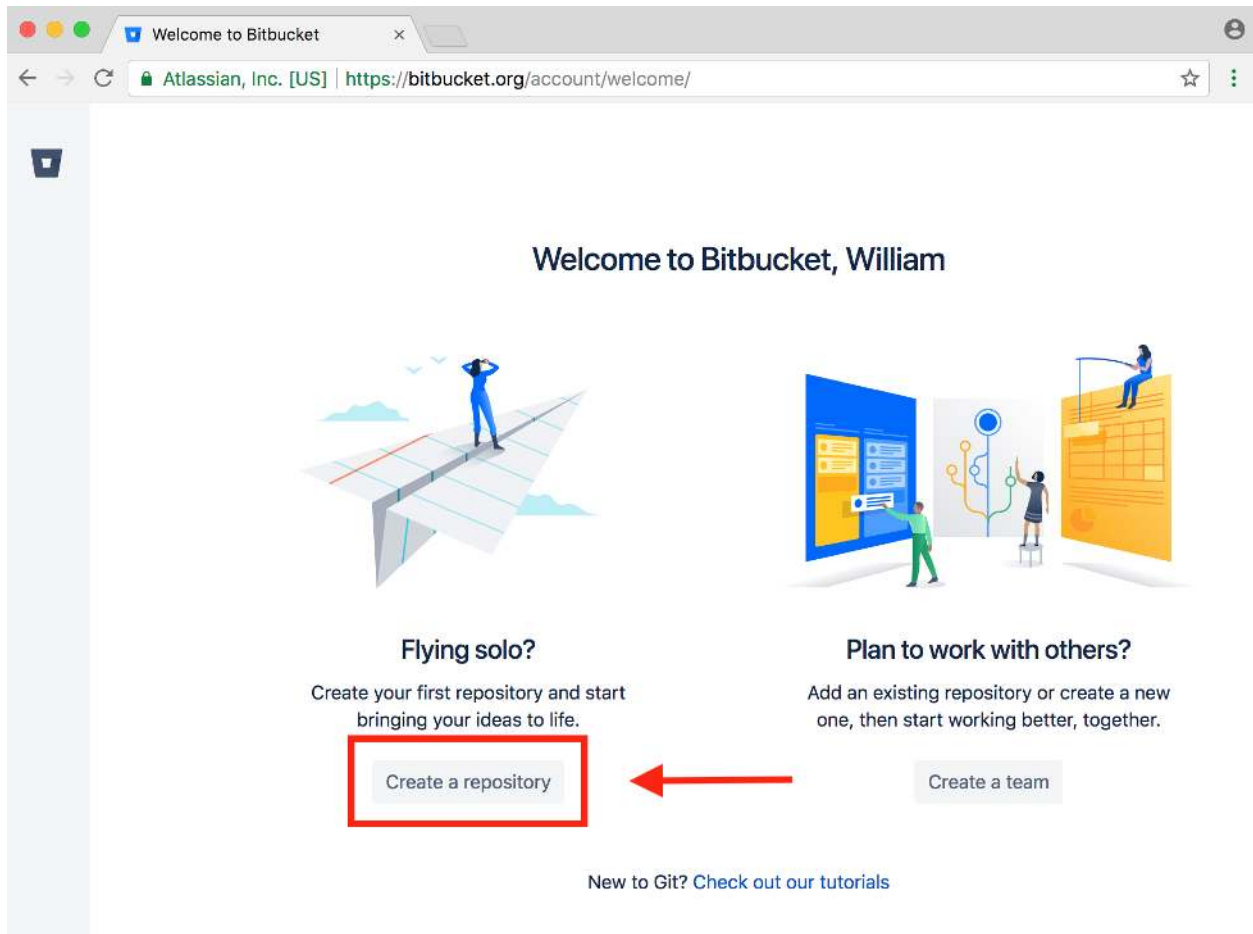
Обратите внимание, что пользователи Windows могут получить сообщение об ошибке **git commit error: pathspec 'commit' did not match any file(s) known to** которая, по-видимому, связана с использованием одинарных кавычек ' ' и отличается от двойных кавычек " ". Если вы видите эту ошибку, используйте двойные кавычки для всех сообщений фиксации в будущем.

## Bitbucket

Хорошей привычкой является создание удаленного хранилища кода для каждого проекта. Таким образом, у вас есть резервная копия на случай, если что-то случится с вашим компьютером, и, что более важно, она позволяет сотрудничать с другими разработчиками программного обеспечения. Два самых популярных варианта [Bitbucket](#) и [Github](#).

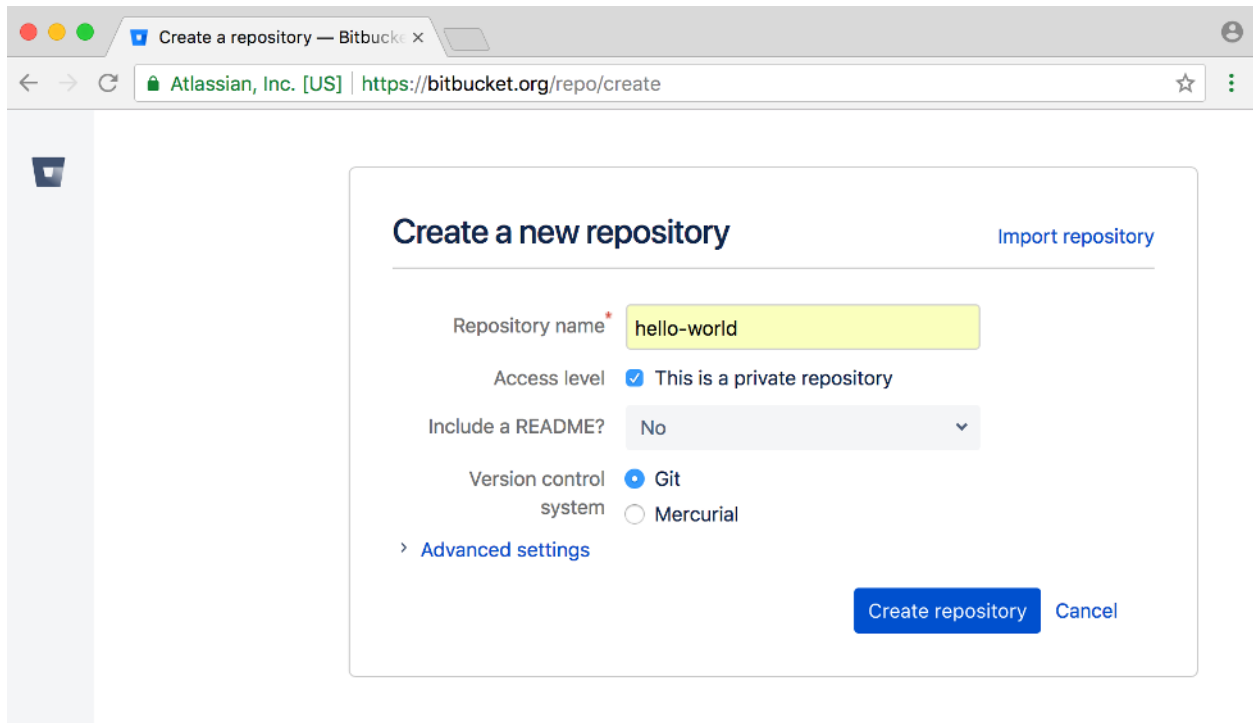
В этой книге мы будем использовать Bitbucket, потому что он позволяет частные репозитории **бесплатно**. Github взимает за это плату. Общедоступные репозитории доступны для всех пользователей интернета, а частные-нет. Когда вы изучаете веб-разработку, лучше всего придерживаться частных репозиториях, чтобы случайно не опубликовать важную информацию, такую как пароли в интернете.

Чтобы начать работу с Bitbucket, зарегистрируйте [бесплатную учетную запись](#). После подтверждения аккаунта по электронной почте вы будете перенаправлены на страницу приветствия. Нажмите на ссылку “создать хранилище”.



Bitbucket страница приветствия

Затем на странице "CreateRepo" введите имя вашего репозитория: "hello-world". Затем нажмите синюю кнопку "Create repository button":



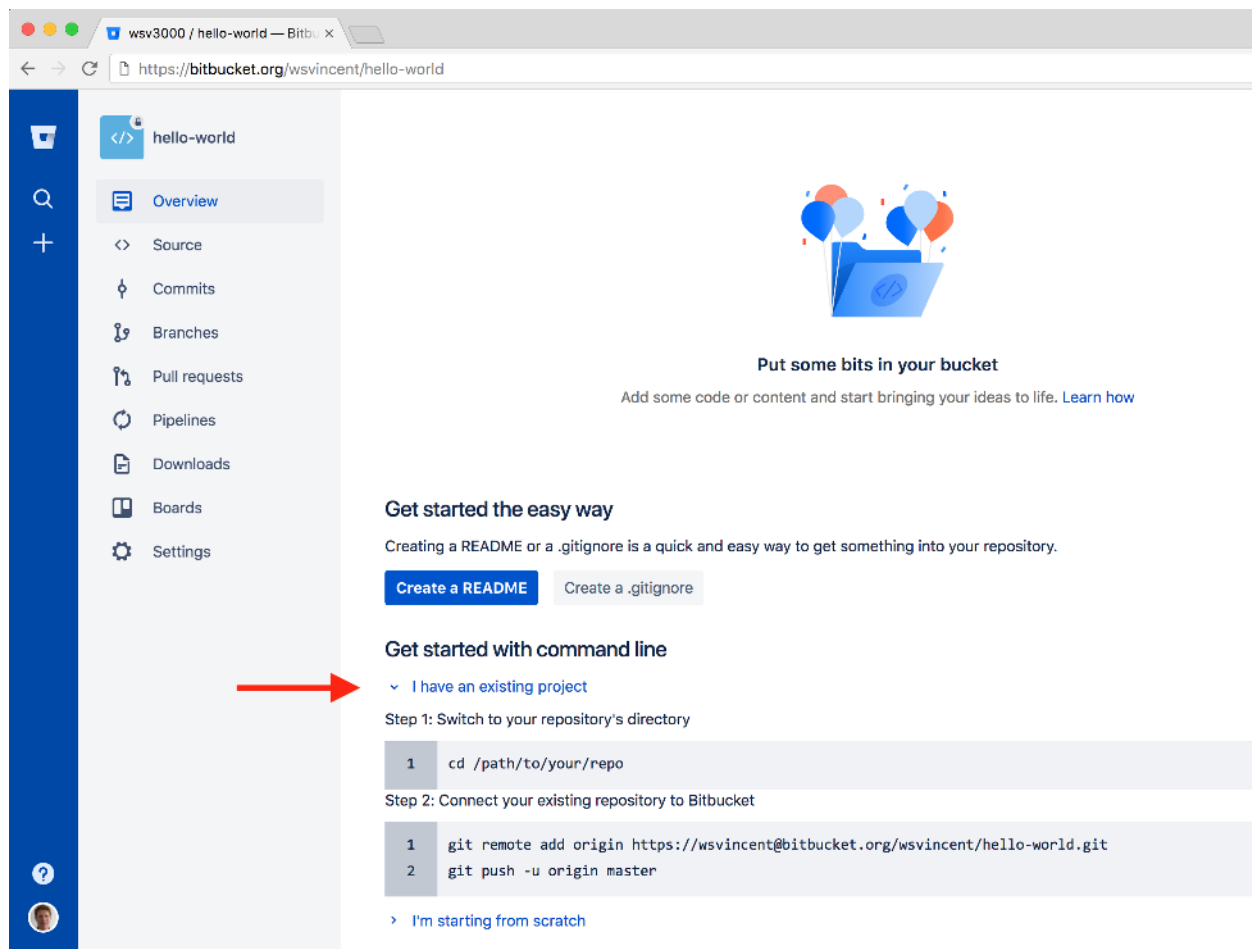
The screenshot shows a web browser window with the address bar displaying "https://bitbucket.org/repo/create". The page title is "Create a repository — Bitbucket". The main content area is titled "Create a new repository" with a link for "Import repository". The form includes the following fields and options:

- Repository name:** A text input field containing "hello-world".
- Access level:** A checkbox labeled "This is a private repository" which is checked.
- Include a README?:** A dropdown menu set to "No".
- Version control system:** Radio buttons for "Git" (selected) and "Mercurial".
- Advanced settings:** A link with a chevron icon.
- Buttons:** "Create repository" and "Cancel".

### Bitbucket create repo

На следующей странице внизу, нажмите на ссылку “I have an existing project” в которой откроется выпадающий список:





### Bitbucket существующий проект

Мы уже в каталоге нашего *repo*, пропустим Шаг 1 (Step 1). В шаге 2(Step 2) мы используем две команды для добавления проекта в Bitbucket. Обратите внимание, что ваша команда будет отличаться от моей, так как у вас другое имя пользователя. Пример общего формата расположен ниже, где <USER> - Ваше имя пользователя Bitbucket.

### Command Line

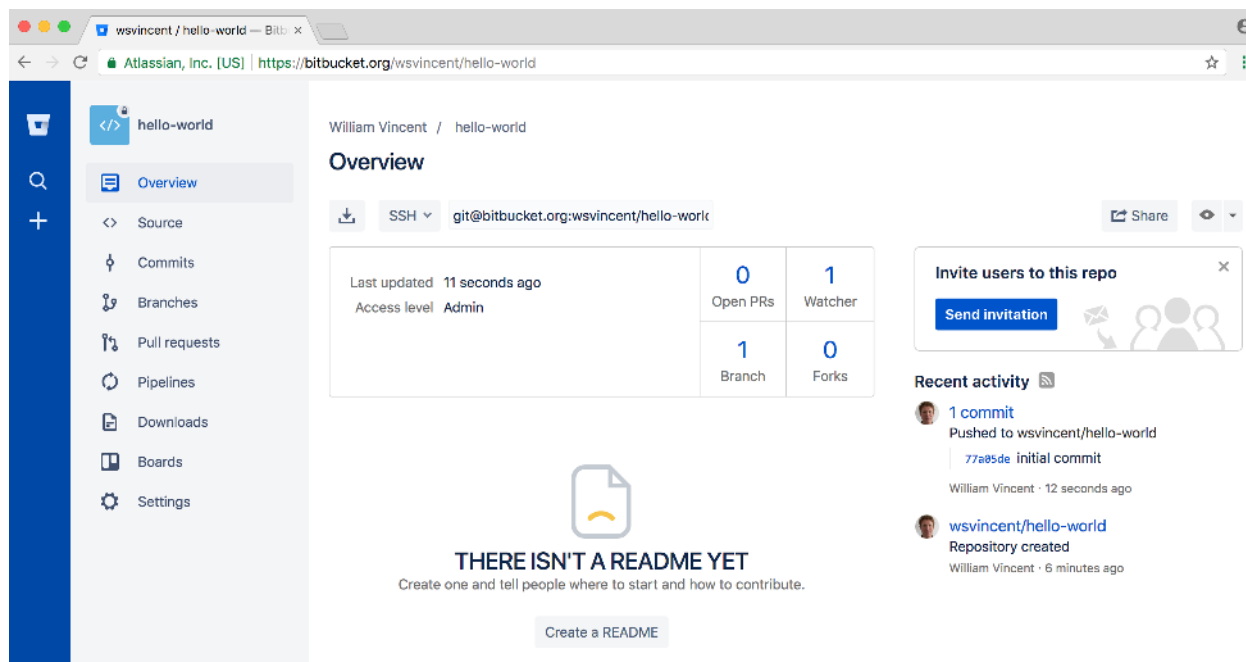
```
(helloworld) $ git remote add origin https://<USER>@bitbucket.org/<USER>/hello-w\
orld.git
```

После выполнения этой команды для настройки git с этим репозиторием Bitbucket мы должны «вставить» наш код в него.

## Command Line

```
(helloworld) $ git push -u origin master
```

Сейчас, если вы вернетесь на свою страницу Bitbucket и обновите ее, вы увидите, что код теперь находится в онлайн! Нажмите на вкладку “Source” слева, чтобы увидеть все это.



### Bitbucket обзор

После этого выйдите из виртуальной среды с помощью команды `exit`.

## Command Line

```
(helloworld) $ exit
```

Сейчас в вашей командной строке не должно быть круглых скобок, это указывает на то, что виртуальная среда больше не активна.

## Вывод

Поздравляю! Мы охватили много фундаментальных понятий в этой главе. Мы создали наше первое Приложение Django и узнали о структуре project/app (проект/приложение) в Django.

Мы начали изучать views, urls и внутренний веб-сервер. А так же мы работали с git, чтобы отслеживать наши изменения, и поместили наш код в частный репозиторий Bitbucket.

**Перейдем к главе 3: Простое приложение**, где мы создадим и развернем более сложное приложение Django с использованием шаблонов(**templates**) и представлений(**views**) на основе классов.

# Глава 3: Pages приложение

В этой главе мы создадим, протестируем и развернем Приложение Pages с главной страницей home и страницей about. Мы также узнаем о классовых views и templates Django, которые являются строительными блоками для более сложных веб-приложений, построенных в книге позже .

## Начальная настройка

Как и в **главе 2: Hello World приложение**, наша первоначальная настройка включает в себя следующие шаги:

- создание нового каталога для нашего кода
- установить Django в новой виртуальной среде
- создание нового проекта Django
- создание нового приложения pages
- обновление(настройка) settings.py

В командной строке, убедитесь, что вы не работаете в существующей виртуальной среде.

Проверьте есть ли что-нибудь в скобках перед запросом командной строки. И если есть то просто наберите exit что бы выйти из него.

Мы снова создадим новые страницы каталога для нашего проекта на рабочем столе, но вы можете поместить свой код в любом месте на вашем компьютере. Он просто должен быть в своем собственном каталоге.

В новой консоли командной строки введите следующие команды:

### Command Line

---

```
$ cd ~/Desktop
$ mkdir pages
$ cd pages
$ pipenv install django
$ pipenv shell
(pages) $ django-admin startproject pages_project .
(pages) $ python manage.py startapp pages
```

---

Я использую здесь (pages) для представления виртуальной среды, но в реальности у меня выглядит так (pages-un0YeQ9e). Имя же вашей виртуальной среды будет уникальным и выглядеть будет примерно в таком формате (pages-XXX).

Откройте текстовый редактор и перейдите к файлу settings.py. Добавьте приложение pages в наш проект в разделе INSTALLED\_APPS:

### Code

---

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

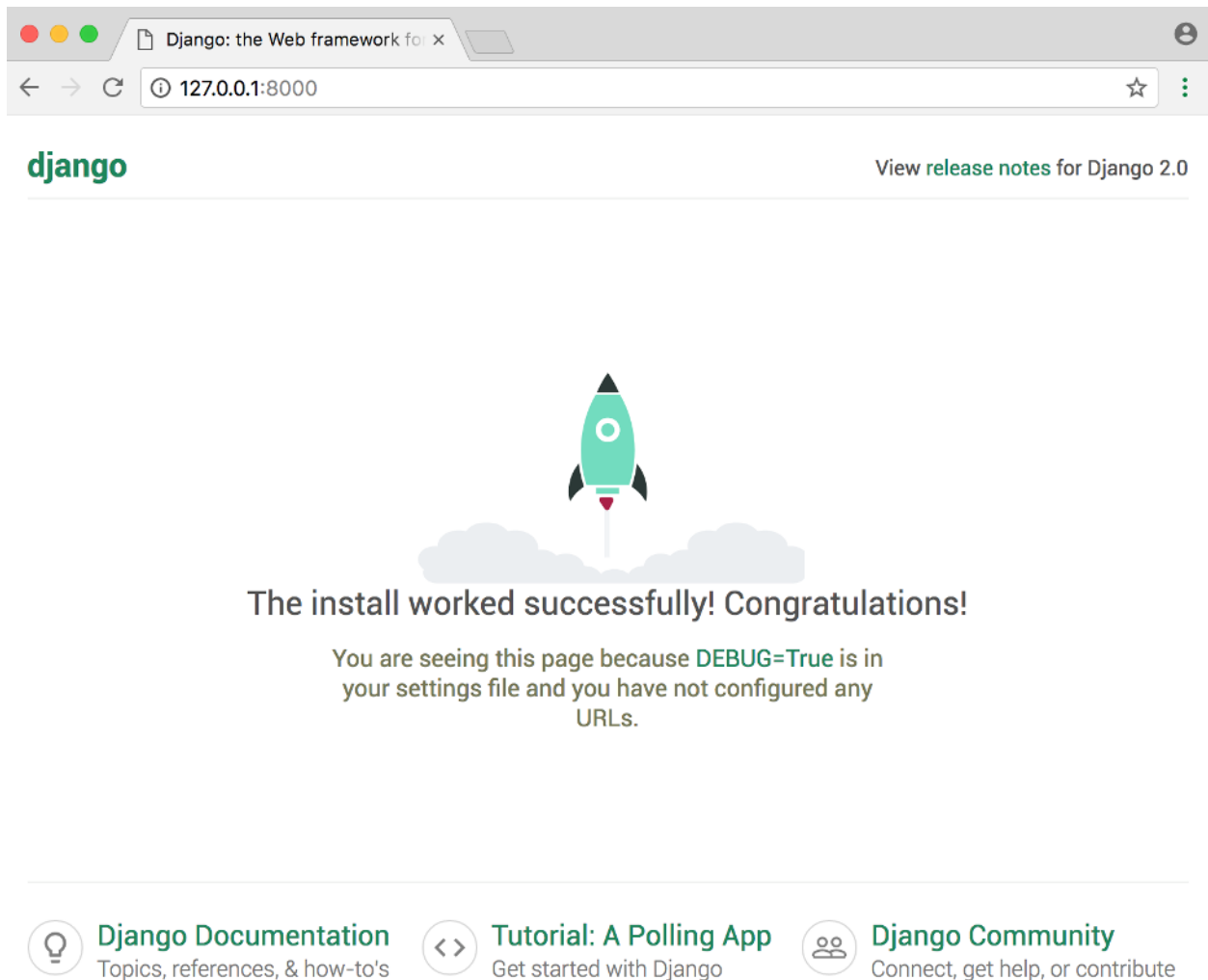
---

Запустите локальный веб-сервер с помощью команды runserver.

## Command Line

```
(pages) $ python manage.py runserver
```

И затем перейдите на <http://127.0.0.1:8000/>.



**Django страница приветствия**

## Templates(Шаблоны)

Каждый веб-фреймворк нуждается в удобном способе создания HTML файлов. В Django, подход заключается в использовании шаблонов(templates), так что бы отдельные HTML файлы могли обслуживаться в view веб-страницы, указанным URL.

Стоит повторить эту модель, так как вы увидите ее снова и снова в Django разработке : Templates, Views и URLs. Порядок, в котором вы их создаете, не имеет большого значения, поскольку все три необходимы и работают в тесном сотрудничестве. URLs управляют начальным маршрутом, точкой входа на страницу, например */about*, views содержат логику, а шаблон имеет HTML. Для веб-страниц, использующих модели базы данных, views делает большую часть работы по определению какие данные доступны в шаблоне.

И так: Templates, Views, URLs. Эта модель будет справедлива для каждой веб-страницы Django которую вы делаете. Однако потребуется некоторая практика, прежде чем вы усвоите это.

Ок, двигаемся дальше. Вопрос о том, где разместить каталог шаблонов(templates), может смутить новичков. По умолчанию Django ищет шаблоны в каждом приложении. Наше pages приложение будет ожидать шаблон *home.html* который расположен в следующей директории:

└─ pages

```
└─ templates
    └─ pages
        └─ home.html
```

Это означает, что нам нужно будет создать новый каталог шаблонов(templates), новый каталог с именем приложения, pages, и наконец, сам наш шаблон *home.html*

Общий вопрос: почему эта повторяющаяся структура? Короткий ответ заключается в том, что загрузчик шаблонов Django хочет быть действительно уверен, что он найдет правильный шаблон, и именно так он запрограммирован на их поиск.

К счастью, есть еще один часто используемый подход построения шаблонов Django-проекта. Вместо этого необходимо создать единый каталог шаблонов на уровне проекта, доступный для всех приложений. Это тот подход, который мы и будем использовать.

Сделав небольшую настройку в файле `settings.py`, мы можем указать Django также искать в папке на уровне проекта для шаблонов.

Сначала остановите наш сервер с помощью Control-c. Затем создайте папку на уровне проекта с именем `templates` и HTML файл с именем `home.html`.

### Command Line

---

```
(pages) $ mkdir templates
(pages) $ touch templates/home.html
```

---

Далее нам нужно обновить `settings.py` и указать Django, чтобы он искал шаблоны на уровне проекта. Для этого изменим строку `'DIRS'` под `TEMPLATES`.

### Code

---

```
# pages_project/settings.py
TEMPLATES = [
    {
        ...
        'DIRS': [os.path.join(BASE_DIR, 'templates')],
        ...
    },
]
```

---

Затем мы можем добавить простой заголовок к нашему `home.html` файлу.



**Code**

---

```
<!-- templates/home.html -->  
<h1>Homepage.</h1>
```

---

Ок, наш шаблон готов! Следующий шаг это настройка url и views.

## Class-Based Views

Ранние версии Django поставлялись только с функциональными views, но вскоре разработчики обнаружили что снова и снова повторяли одни и те же действия. Напишите view со списком всех объектов в модели. Напишите view, в котором отображается только один подробный элемент модели и так далее.

Основанные на функциях view, были представлены, что бы абстрагировать эти закономерности и оптимизировать разработку общими моделями. Однако **не было простого способа расширить или настроить их**. В результате в Django появились универсальные view на основе классов, которые упрощают использование, а также расширяют, и охватывают распространенные варианты использования.

Классы это фундаментальная часть Python, но полное обсуждение их выходит за рамки этой книги. Если Вы нуждаетесь в ведении или напоминании, я предлагаю рассмотреть официальные документы Python, у которых есть превосходное учебное руководство по классам и их использованию.

В нашем view мы будем использовать **встроенный `TemplateView`**. Обновить `pages/` `views.py` файл.

## Code

---

```
# pages/views.py

from django.views.generic import TemplateView


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

---

Обратите внимание, что мы капитализировали наш view, так как теперь это класс Python. Классы, в отличие от функций, всегда должны быть капитализированы. `TemplateView` уже содержит всю логику, необходимую для отображения нашего шаблона, нам просто нужно указать имя шаблона.

## URLs

Последним шагом мы обновим наш `URLConf`s. Вспомним из главы 2, Что нам нужно внести изменения в двух местах. Сначала мы обновляем на уровне проекта `urls.py` файл, чтобы указать на наше `pages` приложение, а затем в `urls.py` файле `pages` приложения мы сопоставим с маршрутами `views`.

Давайте начнем с уровня проекта `urls.py` файл.

**Code**

---

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

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

---

Давайте рассмотрим код на этом этапе. Мы добавили `include` во второй строке, чтобы указать существующий URL-адрес приложения `pages`.  
Далее создайте на уровне приложения `urls.py` файл.

**Command Line**

---

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

---

И добавьте следующий код.

**Code**

---

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

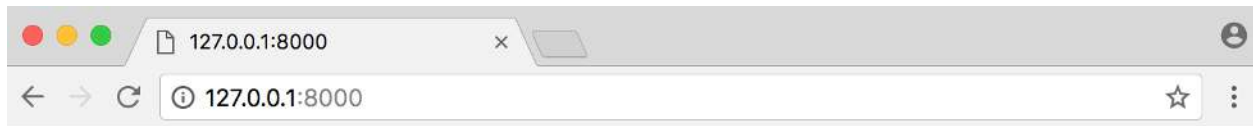
from . import views

urlpatterns = [
    path('', views.HomePageView.as_view(), name='home'),
]
```

---

Этот шаблон почти идентичен тому, что мы сделали в главе 2 с одним существенным отличием. При использовании Views на основе класса мы всегда добавляем `as_view()` в конце имени view.

Мы закончили! Если сейчас вы запустите сервер командой `python manage.py runserver` и перейдете к <http://127.0.0.1:8000/> вы увидите нашу новую домашнюю страницу.



# Номерpage.

Домашняя страница

## Добавить **About** страницу

Процесс добавления страницы очень похож на то, что мы только что сделали. Мы создадим новый файл шаблона, новый view и новый url.

Закройте сервер с помощью Control-c и создайте новый шаблон с именем `about.html`.

### Command Line

---

```
(pages) $ touch templates/about.html
```

---

Затем введите в него короткий HTML заголовок.

**Code**

---

```
<!-- templates/about.html -->
```

```
<h1>About page.</h1>
```

---

Создайте новый view для страницы. (подразумевается создать новый класс или функцию в данном случае класс `AboutPageView`)

**Code**

---

```
# pages/views.py
```

```
from django.views.generic import TemplateView
```

```
class HomePageView(TemplateView):
```

```
    template_name = 'home.html'
```

```
class AboutPageView(TemplateView):
```

```
    template_name = 'about.html'
```

---

А затем подключите его к url-адресу `about/`.

**Code**

---

```
# pages/urls.py

from django.urls import path

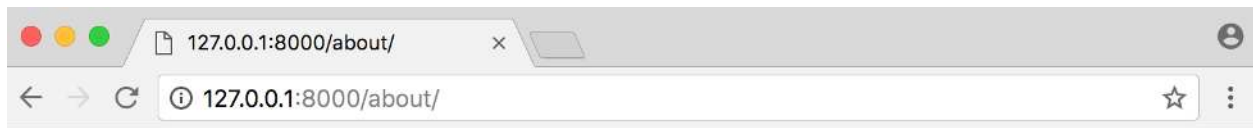
from . import views

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

---

Запустите веб сервер командой *python manage.py runserver*.

Перейдите к <http://127.0.0.1:8000/about> и вы увидите нашу новую страницу “About page”.



## About page.

### About page

## Расширение Шаблонов

Реальная сила шаблонов заключается в их способности расширяться. Если вы сейчас вспомните то у большинства веб сайтов, которые повторяются на каждой странице (заголовок, нижний колонтитул, и т.д.). И было бы неплохо, если бы у нас, как разработчиков, могло быть одно каноническое место для нашего кода заголовка, который бы наследовался всеми другими шаблонами.

И это возможно! Давайте создадим файл `base.html` содержащий заголовок с ссылками на две наши страницы. Сначала `Control-c` а затем введите следующую команду.

### Command Line

---

```
(pages) $ touch templates/base.html
```

---

Django имеет минимальный язык шаблонов для добавления ссылок и базовой логики в наши шаблоны. Вы можете увидеть полный список встроенных шаблонов [здесь в официальной документации](#). Теги шаблонов имеют вот такую форму `{% что-то %}` где “что-то” - это сам тег шаблона. Вы даже можете создать свои собственные теги шаблонов, хотя мы не будем делать этого в этой книге.

Чтобы добавить URL-ссылки в наш проект, мы можем использовать встроенный тег шаблона `url`, который принимает имя шаблона URL в качестве аргумента. Помните, как мы добавили необязательные имена URL в наши URL-маршруты? Вот для чего. Тег `url` автоматически использует эти имена для того чтобы создать соединения для нас.

URL-Адрес нашей домашней страницы называется `home`, поэтому для настройки ссылки на нее мы будем использовать следующее: `{% url 'home' %}`.

### Code

---

```
<!-- templates/base.html -->

<header>

    <a href="{% url 'home' %}">Home</a> | <a href="{% url 'about' %}">About</a>

</header>

{% block content %}

{% endblock %}
```

---

В нижней части мы добавили тег блока `content`. Блоки могут быть перезаписаны дочерними шаблонами с помощью наследования.

Давайте обновим наш `home.html` и `about.html` для расширения `base.html` шаблона. Это означает, что мы можем повторно использовать код из одного шаблона в другой шаблон. Язык шаблонов Django поставляется с методом `extends`, который мы можем использовать для этого.

#### Code

---

```
<!-- templates/home.html -->

{% extends 'base.html' %}


{% block content %}
<h1>Homepage.</h1>
{% endblock %}
```

---

#### Code

---

```
<!-- templates/about.html -->

{% extends 'base.html' %}

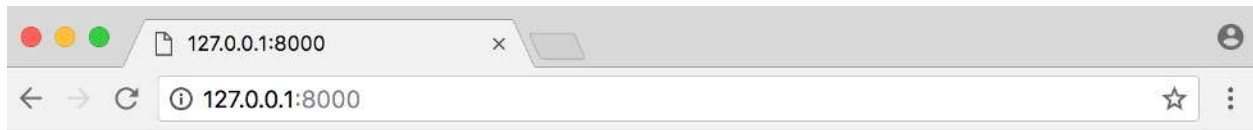

{% block content %}
<h1>About page.</h1>
{% endblock %}
```

---

Запустите сервер командой `python manage.py runserver` и снова откройте наш сайт <http://127.0.0.1:8000/> и <http://127.0.0.1:8000/about> вы увидите, что заголовок волшебным образом включен в обоих местах.

Правда приятно?

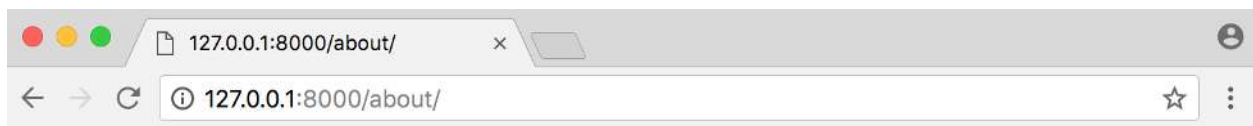




[Home](#) | [About](#)

## Homepage.

Домашняя страница с заголовком



[Home](#) | [About](#)

## About page.

About страница с заголовком

С шаблонами мы можем сделать гораздо больше, на практике мы обычно создаем `base.html` файл, а затем добавляем дополнительные шаблоны поверх него в надежном проекте Django. Мы сделаем это позже в книге.

## Tests

Наконец мы подошли к тестам. Даже в этом базовом приложении важно добавлять тесты и иметь привычку всегда добавлять их в наши проекты Django. По словам Джейкоба Каплан-Мосса, одного из создателей Джанго, “код без тестов нарушен полностью.”

Написание тестов важно, поскольку оно автоматизирует процесс подтверждения того, что код работает должным образом. В таком приложении, как это, мы можем вручную посмотреть и увидеть, что страница `homepage` и страница `about page` существуют и содержат предполагаемое содержание. Но как только Django

проект начинает расти в размерах и в нем могут быть сотни, если не тысячи отдельных веб-страниц и идея вручную посетить каждую страницу становится невозможной. Далее, каждый раз, когда мы вносим изменения в код и добавляем новые функции, обновляем существующие, или удаляем неиспользуемые области сайта мы хотим быть уверены, что случайно не сломали какую-то другую часть сайта. Автоматизированные тесты позволяют нам написать один раз, как мы ожидаем будет работать определенная часть нашего проекта, а затем пусть компьютер делает проверку за нас.

К счастью, Django поставляется с надежными, встроенными инструментами тестирования для написания и запуска тестов.

Если вы посмотрите на наши страницы приложения, Django уже снабжен, `tests.py` файлом, который мы можем использовать. Откройте его и добавьте следующий код:

#### Code

---

```
# pages/tests.py

from django.test import SimpleTestCase

class SimpleTests(SimpleTestCase):

    def test_home_page_status_code(self):
        response = self.client.get('/')
        self.assertEqual(response.status_code, 200)

    def test_about_page_status_code(self):
        response = self.client.get('/about/')
        self.assertEqual(response.status_code, 200)
```

---

Мы используем здесь `SimpleTestCase` поскольку мы не используем базу данных. Если бы мы использовали базу данных, мы бы использовали `TestCase`. Затем мы выполняем проверку состояния кода

для 200 страниц, это стандартный ответ для успешного запроса HTTP. И это причудливый способ сказать, что гарантированно данная веб-страница реально существует, но ничего не говорит о содержании указанной страницы.

Для запуска тестов закройте server Control-с и введите `python manage.py test` в командной строке:

### Command Line

---

```
(pages) $ python manage.py test
Creating test database for alias 'default'...
..
```

```
-----
Ran 2 tests in 0.028s
```

OK

```
Destroying test database for alias 'default'...
```

---

Получилось! В будущем мы сделаем гораздо больше с тестированием, особенно когда начнем работать с базами данных. На данный момент важно видеть, как легко добавлять тесты каждый раз, когда мы добавляем новую функциональность в наш Django проект.

## Git и Bitbucket

Пришло время отслеживать наши изменения с git и выводить в Bitbucket. Начнем с инициализации каталога.

### Command Line

---

```
(pages) $ git init
```

---

Используйте `git status`, чтобы увидеть все наши изменения кода, а затем `git add -A`, чтобы добавить их все. Наконец, мы добавим наше первое сообщение о фиксации.

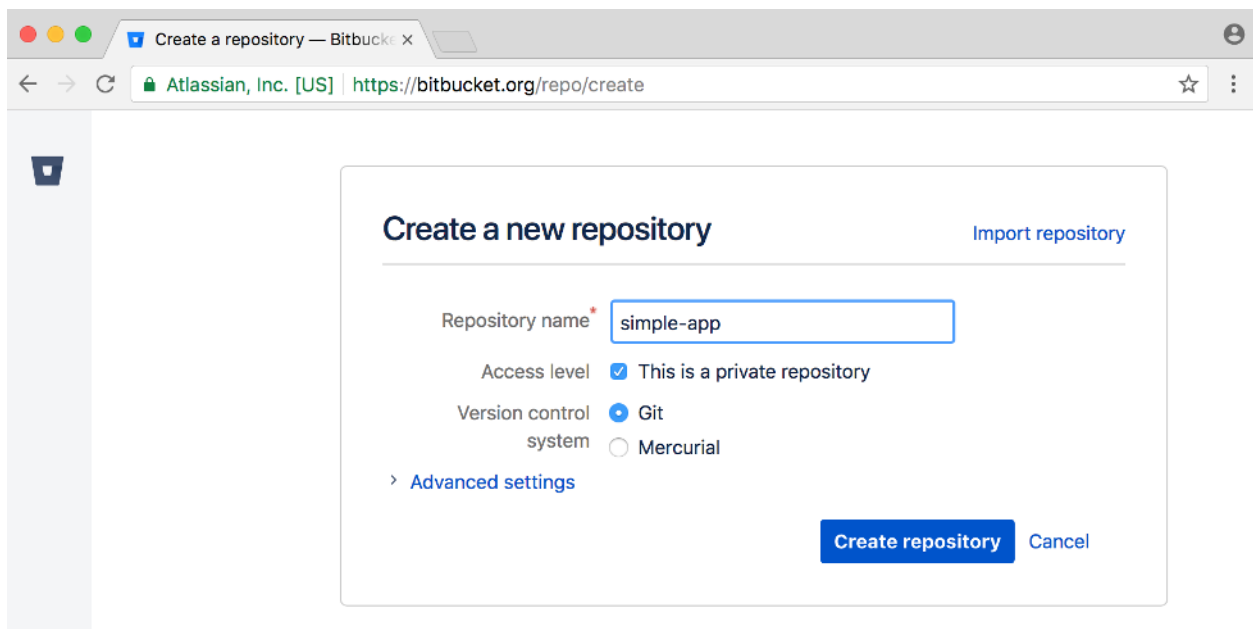
### Command Line

---

```
(pages) $ git status  
(pages) $ git add -A  
(pages) $ git commit -m 'initial commit'
```

---

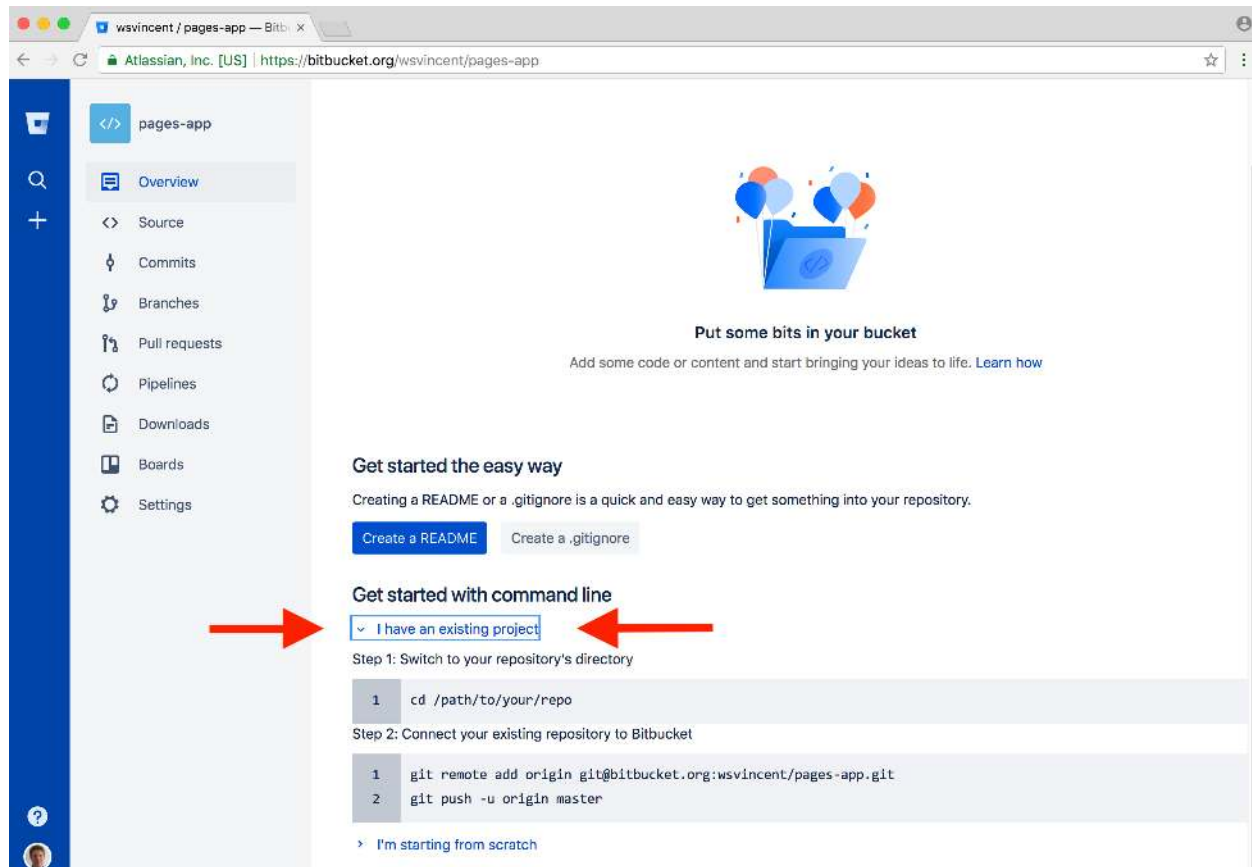
На Bitbucket создайте новый репозиторий, который мы назовем `pages-app`.



### Bitbucket создание страницы

На следующей странице нажмите на ссылку с низу “I have an existing project”.

Скопируйте две команды для подключения, а затем нажмите repository to Bitbucket.



### Bitbucket Existing Project(существующий проект)

Он должен выглядеть следующим образом, замените wsvincent вашим именем пользователя Bitbucket:

#### Command Line

```
(pages) $ git remote add origin git@bitbucket.org:wsvincent/pages-app.git
(pages) $ git push -u origin master
```

## Local vs Production

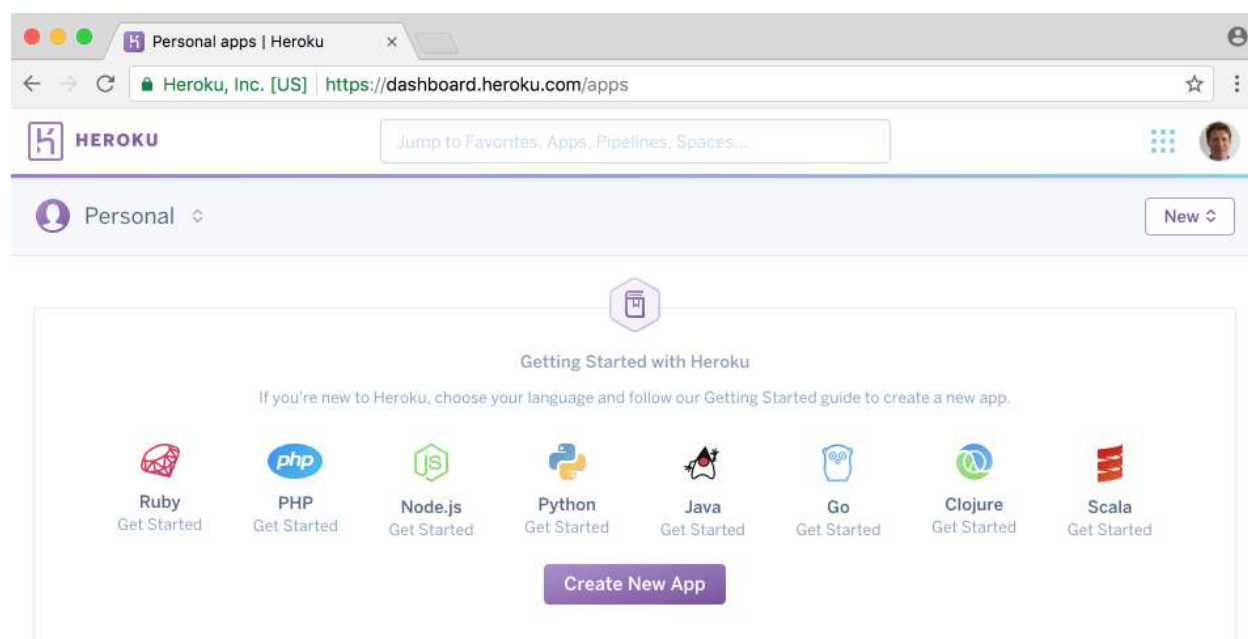
До этого момента мы использовали собственный внутренний веб-сервер Django для работы нашего приложения Pages на нашем локальном компьютере. Но Вы не можете поделиться адресом localhost с кем-то еще. Чтобы сделать наш сайт доступным в интернете, где его может видеть каждый, нам нужно развернуть наш код на внешнем сервере, что бы каждый мог просматривать наш сайт.

Это называется вводом кода в производство (*production*). Локальный код работает только на нашем компьютере; производственный код на внешнем сервере.

Есть много доступных поставщиков серверов, но мы будем использовать Heroku, потому что он бесплатен для небольших проектов, широко используется и имеет относительно простой процесс развертывания.

## Heroku

Вы можете зарегистрировать бесплатный аккаунт на сайте [Heroku](https://heroku.com) и после того, как вы подтвердите свою электронную почту Heroku перенаправит вас в раздел панели инструментов сайта.



### Heroku панель инструментов

Теперь нам необходимо установить Heroku's *Command Line Interface* (CLI), так что мы можем развернуть его из командной строки. Нам нужно установить Heroku глобально, чтобы он был доступен на всем нашем компьютере, поэтому откройте новую вкладку командной строки: Command + t на Mac, Control+ t на Windows. Если бы мы установили Heroku в нашей виртуальной среде, она была бы доступна только там.

В новой вкладке на Mac используйте Homebrew для установки Heroku:

### Command Line

---

```
$ brew install heroku
```

---

Для правильной установки 32-разрядной или 64-разрядной версии Windows см. страницу [Heroku CLI](#).

После завершения установки вы можете закрыть нашу новую вкладку командной строки и вернуться на начальную вкладку с активной виртуальной средой pages.

Введите команду *heroku login* и использованную электронную почту и пароль которые вы указали при регистрации на Heroku.

### Command Line

---

```
(pages) $ heroku login
Enter your Heroku credentials:
Email: will@wsvincent.com
Password: *****
Logged in as will@wsvincent.com
```

---

## Дополнительный файл

Нам необходимо внести следующие четыре изменения в наш проект Pages, чтобы он был готов к развертыванию онлайн на Heroku:

- обновить `Pipfile.lock`
- создать новый файл профиля (`Procfile` file)
- установить `gunicorn` как наш веб-сервер
- внести однострочное изменение в `settings.py` файл

В существующем `Pipfile` укажите версию Python которую мы используем, то есть 3.6.

Добавьте эти две строки в конец файла.

**Code**

---

```
# Pipfile
[requires]
python_version = "3.6"
```

---

Затем запустите `pipenv lock` для создания соответствующего файла `Pipfile.lock`.

**Command Line**

---

```
(pages) $ pipenv lock
```

---

Фактически Heroku ищет в нашем `Pipfile.lock` информацию о нашей виртуальной среде, поэтому мы здесь добавляем настройки языка.

Затем создайте `Procfile` который является специфическим для Heroku.

**Command Line**

---

```
(pages) $ touch Procfile
```

---

Откройте файл `Procfile` в текстовом редакторе и добавьте следующее:

**Code**

---

```
web: gunicorn pages_project.wsgi --log-file -
```

---

Это указывает на то что используется наш существующий файл `pages_project.wsgi` но с [gunicorn](#), который является веб сервером и подходит для производства (production), вместо Django сервера который используется только для локальной разработки.



### Command Line

---

```
(pages) $ pipenv install gunicorn
```

---

Последний шаг это однострочное изменение в settings.py. Прокрутите вниз до раздела ALLOWED\_HOSTS и добавьте '\*', чтобы он выглядело как в примере ниже:

### Code

---

```
# pages_project/settings.py  
ALLOWED_HOSTS = ['*']
```

---

Параметр `ALLOWED_HOSTS` представляет имена хостов/доменов, которые может обслуживать наш Django сайт. Это мера безопасности, предотвращающая атаки на HTTP Host header, которые возможны даже при многих, казалось бы, безопасных конфигурациях веб-сервера. Однако мы использовали подстановочный знак звездочку \*, который означает, что все домены приемлемы, для простоты. На сайте Django уже на уровне производства(production) вы должны явно указать, какие домены разрешены.

Используйте git status, чтобы проверить наши изменения, добавить новые файлы, а затем зафиксировать их:

### Command Line

---

```
(pages) $ git status  
(pages) $ git add -A  
(pages) $ git commit -m "New updates for Heroku deployment"
```

---

И наконец отправим на Bitbucket чтобы у нас была онлайн резервная копия наших изменений кода.

### Command Line

---

```
(pages) $ git push -u origin master
```

---

## Deploy (Развертывание)

Заключительный шаг фактически разворачивает на Heroku. Если вы в прошлом настраивали сервер самостоятельно, то вы будете поражены тем, насколько простой процесс с провайдером сервис платформы таким как Heroku.

Наш процесс будет следующим:

- Создадим новое приложение на Heroku и отправляем на него наш код.
- Добавим удаленный git “hook” для Heroku
- Настроим приложение, чтобы оно игнорировало статические файлы, которые мы рассмотрим в последующих главах.
- Запустим сервер Heroku, чтобы приложение заработало.
- Посетим приложение на Heroku с предоставленным URL

Мы можем сделать первый шаг, создав новое приложение Heroku, из командной строки `heroku create`. Heroku создаст случайное имя для нашего приложения, в моем случае это `cryptic-oasis-40349`. у вас имя будет другим.

### Command Line

---

```
(pages) $ heroku create
Creating app... done,    cryptic-oasis40349
https://cryptic-oasis-40349.herokuapp.com/ | https://git.heroku.com/cryptic-oasis-40349.git
```

---

Теперь нам нужно добавить “hook” для Heroku в git. Это означает, что git сохранит обе наши настройки для отправки кода на Bitbucket и Heroku. Мое Приложение Heroku называется `cryptic-oasis-40349` поэтому моя команда выглядит следующим образом.

**Command Line**

---

```
(pages) $ heroku git:remote -a cryptic-oasis-40349
```

---

Вы должны заменить `cryptic-oasis-40349` на имя приложения которое предоставил Heroku.

На данный момент нам нужно сделать только одну настройку конфигурации Heroku, которая должна указать Heroku игнорировать статические файлы, такие как CSS и JavaScript, которые Django по умолчанию пытается оптимизировать для нас. Мы рассмотрим это в последующих главах, так что пока просто выполните следующую команду.

**Command Line**

---

```
(pages) $ heroku config:set DISABLE_COLLECTSTATIC=1
```

---

Теперь мы можем отправить наш код на Heroku. Потому что ранее мы установили наш “hook” и он уйдет на Heroku.

**Command Line**

---

```
(pages) $ git push heroku master
```

---

Если мы просто наберем `git push origin master`, то код будет перенесен в Bitbucket, а не в Heroku. Добавив `heroku` к команде пошлет код на Heroku. Это немного сбивает с толку первое время.

Наконец, нам необходимо сделать ваше приложение Heroku рабочим. Поскольку веб-сайты растут в трафике, им нужны дополнительные услуги Heroku, но для нашего основного примера мы можем использовать самый низкий уровень, `web=1`, который также является бесплатным!

Введите следующую команду.

### Command Line

---

```
(pages) $ heroku ps:scale web=1
```

---

Мы закончили! Последний шаг, это подтвердить что наше приложение работает в интернете. Если вы используете команду `heroku open` веб-браузер откроет новую вкладку с URL вашего приложения:

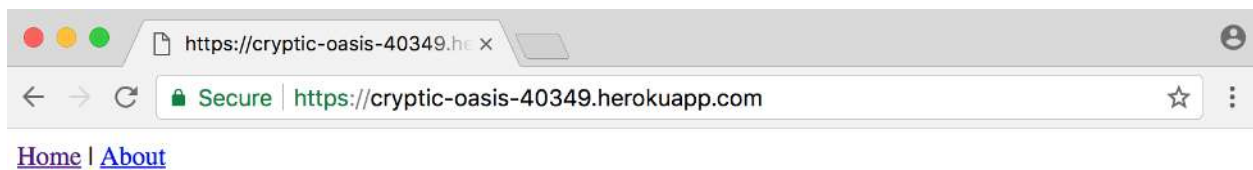
### Command Line

---

```
(pages) $ heroku open
```

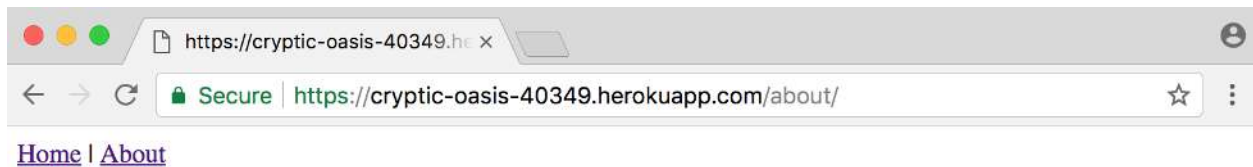
---

У меня это <https://cryptic-oasis-40349.herokuapp.com/about/>. Вы сможете увидеть страницу homepage и страницу about page .



## Homepage.

### Страница Homepage на Heroku



## About page.

### Страница About page на Heroku

## Вывод

Поздравляем с созданием и развертыванием второго проекта Django! На этот раз мы использовали шаблоны, классические `views`, более полно исследовали `URLConfs`, добавили базовые тесты и использовали Heroku. Далее мы перейдем к нашему первому проекту с базой данных и посмотрим, где действительно Django блистает.

## Глава 4: Message Board app (Приложение доска объявлений)

В этой главе мы будем использовать базу данных в первый раз, чтобы построить базовое приложение доска объявлений, где пользователи могут отправлять и читать короткие сообщения. Мы рассмотрим мощный встроенный интерфейс администратора Django, который обеспечивает визуальный способ внесения изменений в наши данные. И после добавления тестов мы будем отправлять наш код в Bitbucket и развернем приложение на Heroku.

Django предоставляет встроенную поддержку для нескольких типов баз данных. Имея всего несколько строк в нашем файле `settings.py`, он может поддерживать PostgreSQL, MySQL, Oracle или SQLite. Но самым простым на сегодняшний день-является SQLite, потому что он работает с одним файлом и не требует сложной установки. В отличие от других, опции которых требуют, чтобы процессы работали в фоновом режиме и могут быть довольно сложными для настройки. Django использует SQLite по умолчанию и это идеальный выбор для небольших проектов.

### Начальная настройка

Так как мы уже создали несколько проектов Django в этой книге, мы можем быстро выполнить наши команды, чтобы начать новый проект. Нам необходимо выполнить следующее:

- создать новый каталог для нашего кода на рабочем столе под названием `mb`
- установить Django в новой виртуальной среде
- создать новый проект с именем `mb_project`
- создать новое приложение `posts`
- обновить `settings.py`

В новой консоли командной строки введите следующие команды. Обратите внимание, что я использую здесь (mb) для представления имени виртуальной среды, хотя на самом деле (mb-XXX), где XXX случайные символы.

### Command Line

---

```
$ cd ~/Desktop
$ mkdir mb
$ cd mb
$ pipenv install django
$ pipenv shell
(mb) $ django-admin startproject mb_project .
(mb) $ python manage.py startapp posts
```

---

Укажите Django о новом приложении posts, добавив его в нижнюю часть раздела INSTALLED\_APPS вашего settings.py файла открыв его в текстовом редакторе.

### Code

---

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

---

Затем выполните команду migrate, чтобы создать исходную базу данных на основе настроек Django по умолчанию.

**Command Line**

---

```
(mb) $ python manage.py migrate
```

---

Если вы посмотрите в наш каталог с помощью команды `ls`, вы увидите, что у нас появился файл `db.sqlite3` представляющий нашу базу данных SQLite.

**Command Line**

---

```
(mb) $ ls  
db.sqlite3 mb_project manage.py
```

---

**Отступление:** Технически файл `db.sqlite3` создается при первом запуске `migrate` или `runserver`. Использование `runserver` формирует базу данных, используя настройки Django по умолчанию, однако `migrate` будет синхронизировать базу данных с текущим состоянием любых моделей баз данных, находящихся в проекте и внесенных в `INSTALLED_APPS`. Другими словами, чтобы убедиться, что база данных отражает текущее состояние проекта, необходимо использовать `migrate` (а также `makemigrations`) при каждом обновлении модели. Более подробно об этом чуть позже.

Чтобы убедиться, что все работает правильно, запустите наш локальный сервер.

**Command Line**

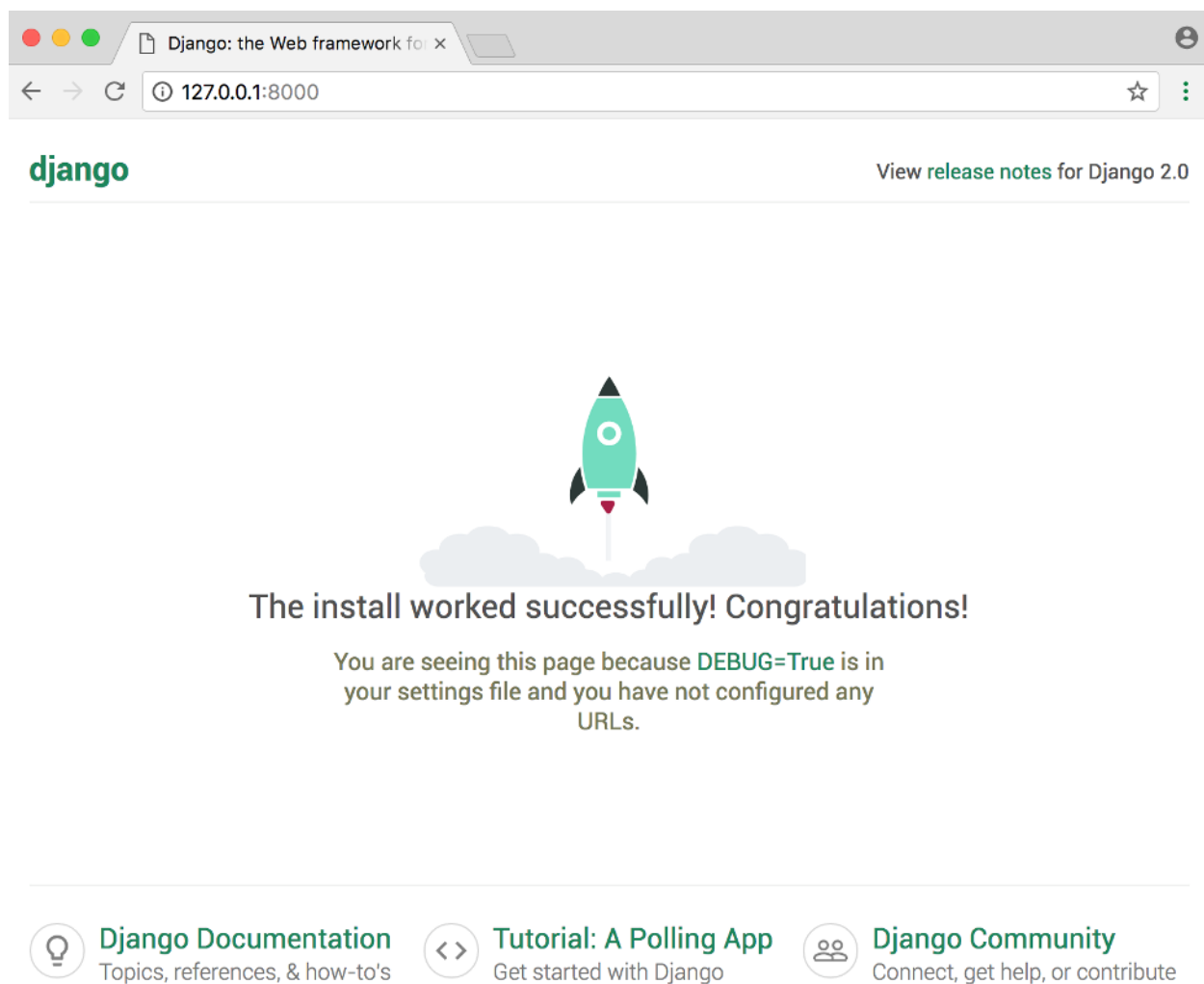
---

```
(mb) $ python manage.py runserver
```

---

И перейдите к <http://127.0.0.1:8000/> что бы увидеть хорошо знакомую страницу, Django установлен правильно.





Django welcome page

## Создание модели базы данных

Наша первая задача создать модель базы данных, где мы можем хранить и отображать сообщения от наших пользователей. Для нас Django превратит эту модель в таблицу базы данных. В реальных проектах Django часто бывает так, что будет много сложных взаимосвязанных моделей баз данных, но в нашем простом приложении для доски объявлений нам нужна только одна.

Я не буду рассматривать конструкцию базы данных в этой книге, но я написал краткое руководство, которое вы можете [найти здесь](#), если это новое для вас.

Откройте файл `posts/models.py` и посмотрите на код по умолчанию, который предоставляет Django:

**Code**

---

```
# posts/models.py

from django.db import models

# Create your models here
```

---

Django импортирует модуль *models*, чтобы помочь нам построить новые модели баз данных, которые будут “моделировать” характеристики данных в нашей базе данных. Мы хотим создать модель для хранения текстового содержимого сообщений в доске объявлений, которую мы можем выполнить следующим образом:

**Code**

---

```
# posts/models.py

from django.db import models

class Post(models.Model):
    text = models.TextField()
```

---

Обратите внимание, что мы создали новую модель базы данных, которая имеет текстовое поле данных. Мы также определили тип контента, который он будет содержать, `TextField()`. Django предоставляет множество полей модели, поддерживающих общие типы контента, такие как символы, даты, целые числа, электронные письма и так далее.

## Активация моделей

Теперь, когда наша новая модель создана, нам нужно ее активировать. В будущем, когда мы создаем или изменяем существующую модель необходимо обновлять Django в два этапа.

1. Сначала мы создаем файл миграции с помощью команды `makemigrations`, которая генерирует команды SQL для предустановленных приложений в нашем параметре `INSTALLED_APPS`. Файлы миграции не выполняют эти команды в файле базы данных, а являются ссылкой на все новые изменения в наших моделях. Этот подход означает, что у нас есть запись изменений в наших моделях.
2. Во-вторых, мы создаем фактическую базу данных с помощью `migrate`, которая выполняет инструкции в нашем файле `migrations`.

### Command Line

---

```
(mb) $ python manage.py makemigrations posts
```

```
(mb) $ python manage.py migrate posts
```

---

Обратите внимание, что вам не нужно включать имя после `makemigrations` или `migrate`. Если вы просто запустите команды, то они будут применены ко всем доступным изменениям. Это хорошая привычка, чтобы быть точным. Если бы у нас было два отдельных приложения в нашем проекте, и мы обновили модели в обоих, а затем запустили `makemigrations`, это создало бы файл миграции, содержащий данные об обоих изменениях. Это усложняет отладку в будущем. Вам потребуется, чтобы каждый файл миграции был как можно меньше и возможно отдельным. Таким образом, если вам нужно просмотреть прошлые миграции, есть только одно изменение на миграцию, а не одно, которое применяется к нескольким приложениям.

## Django Admin

Django предоставляет нам мощный интерфейс администратора для взаимодействия с нашей базой данных. Это поистине потрясающий функционал, который предлагают немногие веб-фреймворки. Он имеет свои маршруты [как в Django проекте newspaper](#) (газета - в этой книге ниже). Разработчики хотели CMS (систему управления контентом), чтобы журналисты могли писать и редактировать свои истории без необходимости затрагивать "код." Постепенно встроенное приложение администратора превратилось в фантастический, из коробки инструмент для управления всеми аспектами проекта Django.

Чтобы использовать Django админ, нам сначала нужно создать суперпользователя, который может войти в систему. В консоли командной строки введите `python manage.py createsuperuser` и заполните ответы на запросы для имени пользователя, электронной почты и пароля:

### Command Line

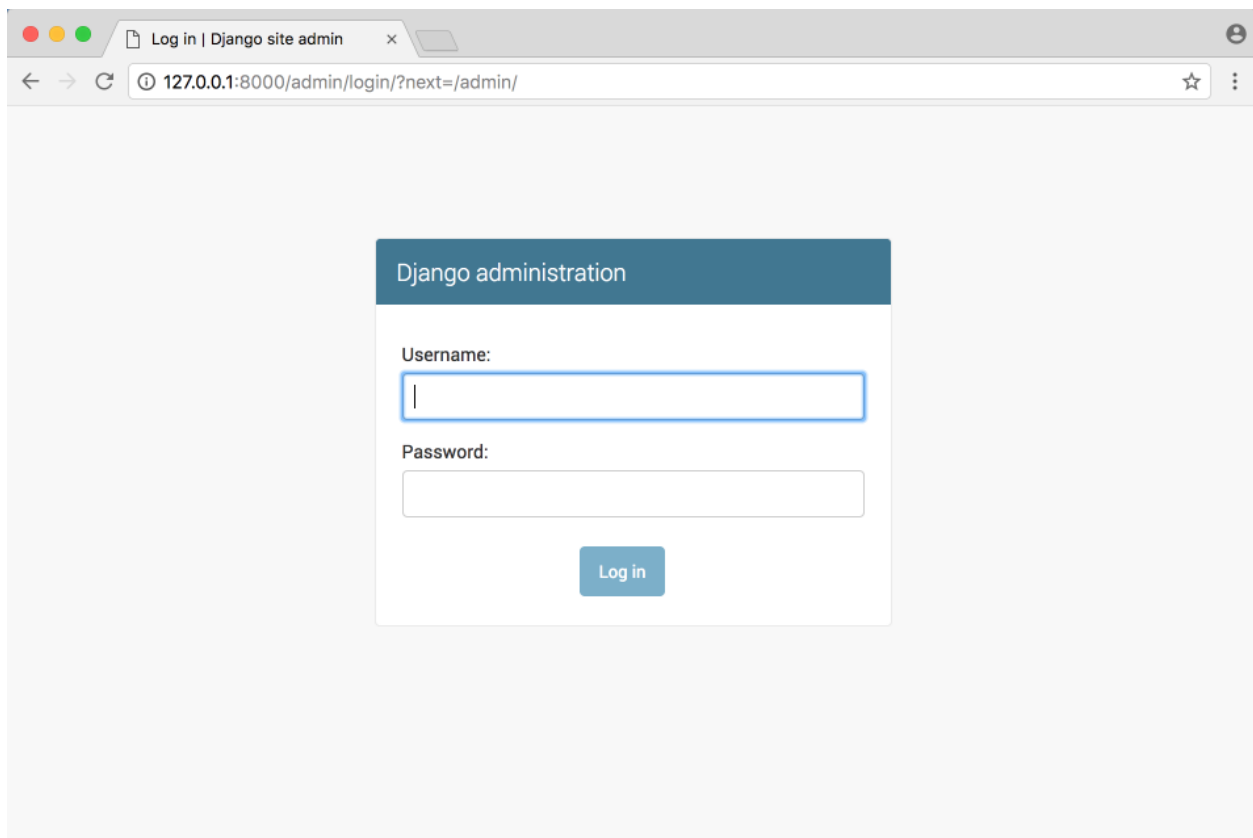
---

```
(mb) $ python manage.py createsuperuser
Username (leave blank to use 'wsv'): wsv
Email:
Password:
Password (again):
Superuser created successfully.
```

---

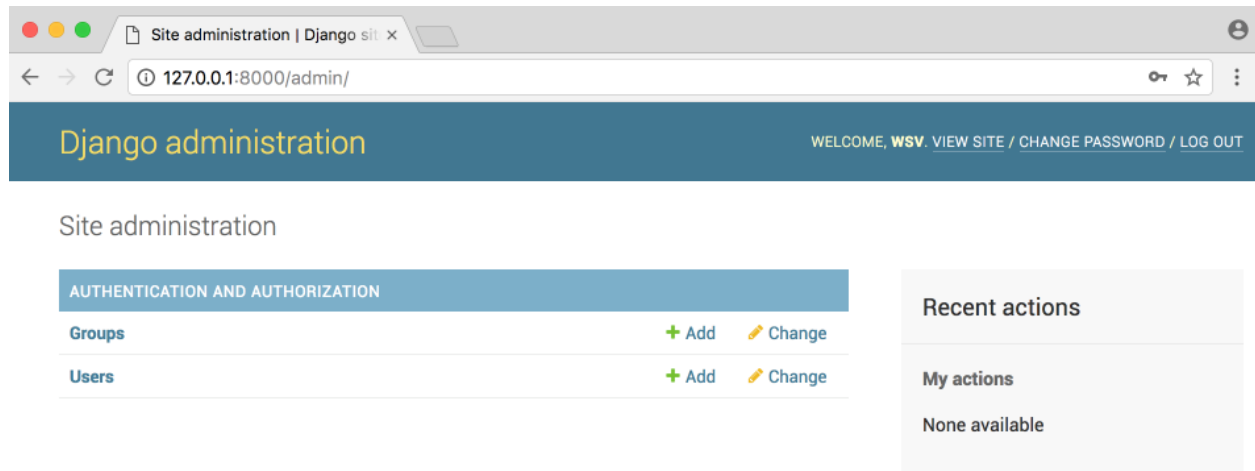
**Примечание:** При вводе пароля он не отображается в консоли командной строки по соображениям безопасности.

Перезапустите сервер Django с помощью `python manage.py runserver` и в браузере перейдите к <http://127.0.0.1:8000/admin/>. Вы должны увидеть экран входа администратора:



### Admin login page

Войдите в систему, введя имя пользователя и пароль, которые вы только что создали. Далее вы увидите домашнюю страницу администратора Django:



### Admin homepage

Но где наше приложение `posts`? Оно не отображается на главной странице администратора!

Нам необходимо явно указать Django, что отображать в админке. К счастью, мы можем легко изменить это, открыв файл `posts/admin.py` и отредактировать его следующим образом:

## Code

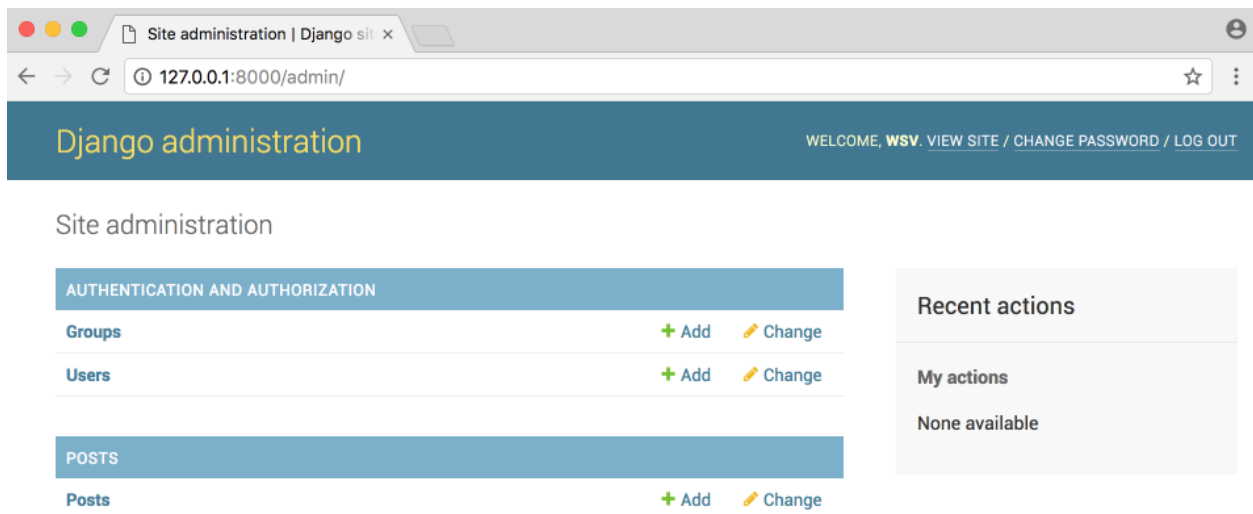
```
# posts/admin.py

from django.contrib import admin

from .models import Post

admin.site.register(Post)
```

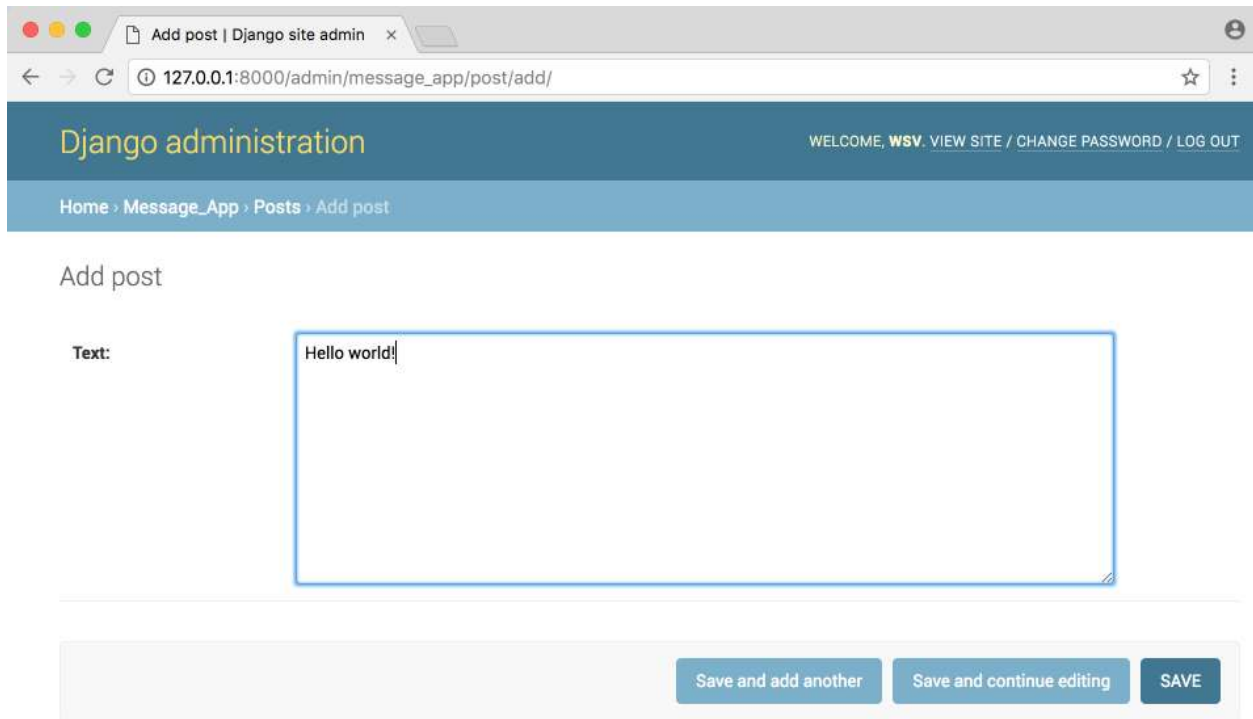
Django теперь знает, что он должен отображать наше posts приложение и его модель базы данных Post на странице администратора. Если вы обновите свой браузер, вы увидите, что теперь оно появилось:



### Admin homepage updated

Теперь давайте создадим наш первый пост в доске объявлений для нашей базы данных.

Нажмите на кнопку + Add напротив Posts. Введите свой текст в текстовое поле.

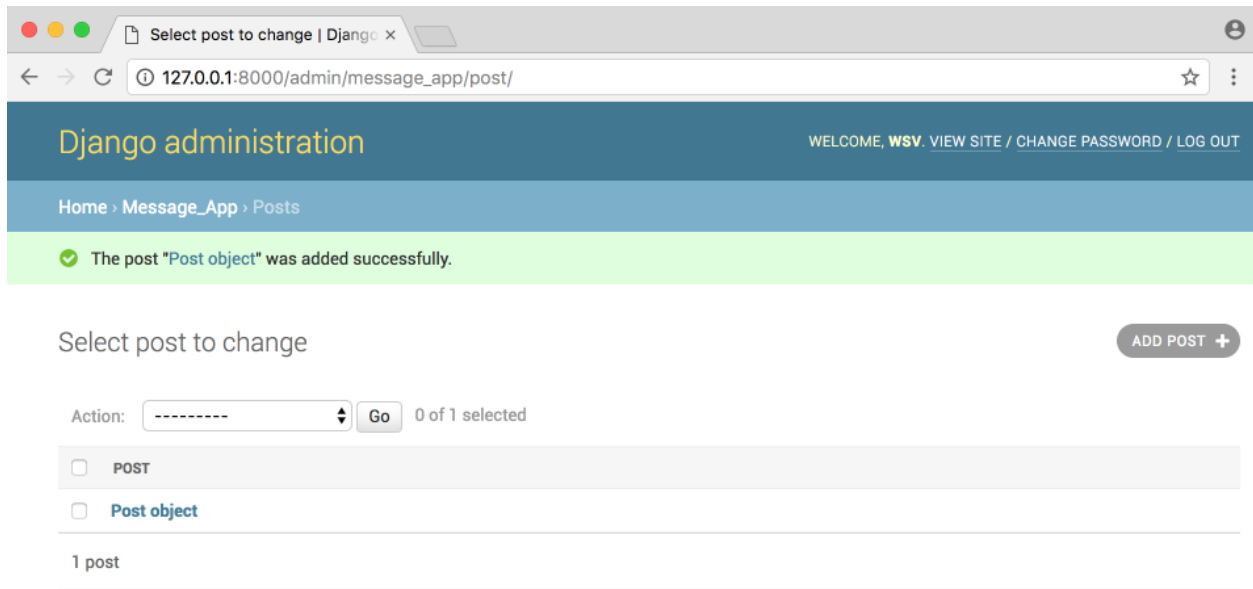


The screenshot shows a web browser window with the title "Add post | Django site admin". The address bar displays the URL "127.0.0.1:8000/admin/message\_app/post/add/". The page header is a dark blue bar with "Django administration" on the left and "WELCOME, wsv. VIEW SITE / CHANGE PASSWORD / LOG OUT" on the right. Below the header is a light blue breadcrumb trail: "Home > Message\_App > Posts > Add post". The main content area is titled "Add post" and features a "Text:" label next to a large text input field. The input field contains the text "Hello world!". At the bottom of the form, there are three buttons: "Save and add another", "Save and continue editing", and "SAVE".

### Admin new entry

Затем нажмите кнопку "Save", которая перенаправит вас на главную страницу Post. Однако, если вы посмотрите внимательно, то есть проблема: Наша новая запись называется "Post object", что не очень полезно.





### Admin new entry

Давайте изменим эту ситуацию. В файле `posts/models.py` добавьте новую функцию `__str__` как в примере:

#### Code

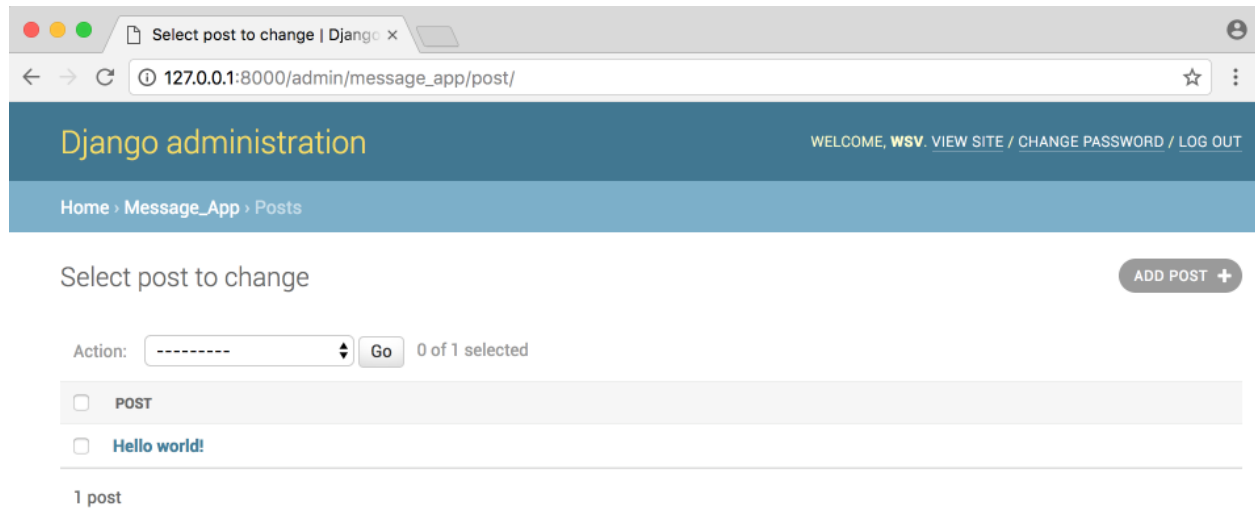
```
# posts/models.py

from django.db import models

class Post(models.Model):
    text = models.TextField()

    def __str__(self):
        """Строковое отображение модели """
        return self.text[:50]
```

Если вы обновите страницу администратора в браузере, вы увидите, что она изменена на гораздо более наглядное и полезное отображение нашей записи базы данных.



### Admin new entry

Гораздо лучше! Рекомендуется добавлять методы `str()` ко всем моделям, чтобы улучшить их читаемость.

## Views/Templates/URLs

Для того, чтобы отобразить содержимое нашей базы данных на нашей домашней странице, мы должны подключить наши `views`, `templates` и `URLConfs`. Сейчас этот образец должен начать казаться знакомым.

Начнем с `view`. Ранее в книге мы использовали встроенный универсальный [Template-View](#) для отображения файла шаблона на нашей домашней странице. Теперь мы хотим перечислить содержимое нашей модели базы данных. К счастью, это общая задача в веб-разработке и Django поставляется с универсальным основным-классом `ListView`.

В `posts/views.py` файле введите код Python ниже:

### Code

---

```
# posts/views.py

from django.views.generic import ListView

from .models import Post


class HomePageView(ListView):
    model = Post
    template_name = 'home.html'
```

---

В первой строке мы импортируем `ListView`, а во второй строке нам нужно явно определить, какую модель мы используем. В view мы создаем подкласс `ListView`, указываем название модели и указываем ссылку на шаблон. Внутри `ListView` возвращает объект с именем `object_list`, который мы хотим отобразить в нашем шаблоне (template).

Наш view готов, что означает, что нам все еще нужно настроить наши URL-адреса и сделать наш шаблон(template). Начнем с шаблона. Создадим каталог(папку) на уровне проекта с именем `templates` и файл шаблона `home.html`.

### Command Line

---

```
(mb) $ mkdir templates
(mb) $ touch templates/home.html
```

---

Затем обновите поле `DIRS` в `settings.py` файле, чтобы Django знал, что нужно искать в этой папке шаблонов.

**Code**

---

```
# settings.py

TEMPLATES = [

    {

        ...

        'DIRS': [os.path.join(BASE_DIR, 'templates')],

        ...

    },

]
```

---

В нашем файле шаблона `home.html` мы можем использовать [Django Templating Language's](#)(Язык шаблонов Django) для цикла, чтобы перечислить все объекты в `object_list`. Помните, что `object_list`-это то, что `ListView` возвращает нам.

**Code**

---

```
<!-- templates/home.html -->

<h1>Message board homepage</h1>

<ul>

    {% for post in object_list %}

        <li>{{ post }}</li>

    {% endfor %}

</ul>
```

---

Последний шаг заключается в том, чтобы настроить наш `URLConfs`. Давайте начнем с файла `urls.py` на уровне проекта где мы просто подключаем `include` нашего `posts` и добавляем `include` во вторую строку.

**Code**

---

```
# mb_project/urls.py

from django.contrib import admin
from django.urls import path, include

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

---

Затем создайте файл `urls.py` на уровне приложения.

**Command Line**

---

```
(mb) $ touch posts/urls.py
```

---

И обновите его таким образом:

**Code**

---

```
# posts/urls.py

from django.urls import path

from . import views

urlpatterns = [
    path('', views.HomePageView.as_view(), name='home'),
]
```

---

Перезапустите сервер с `python manage.py runserver` и перейдите на наш сайт <http://127.0.0.1:8000/> в котором теперь перечислены наши сообщения на доске объявлений.



## Message board homepage

- Hello world!

### Homepage with posts

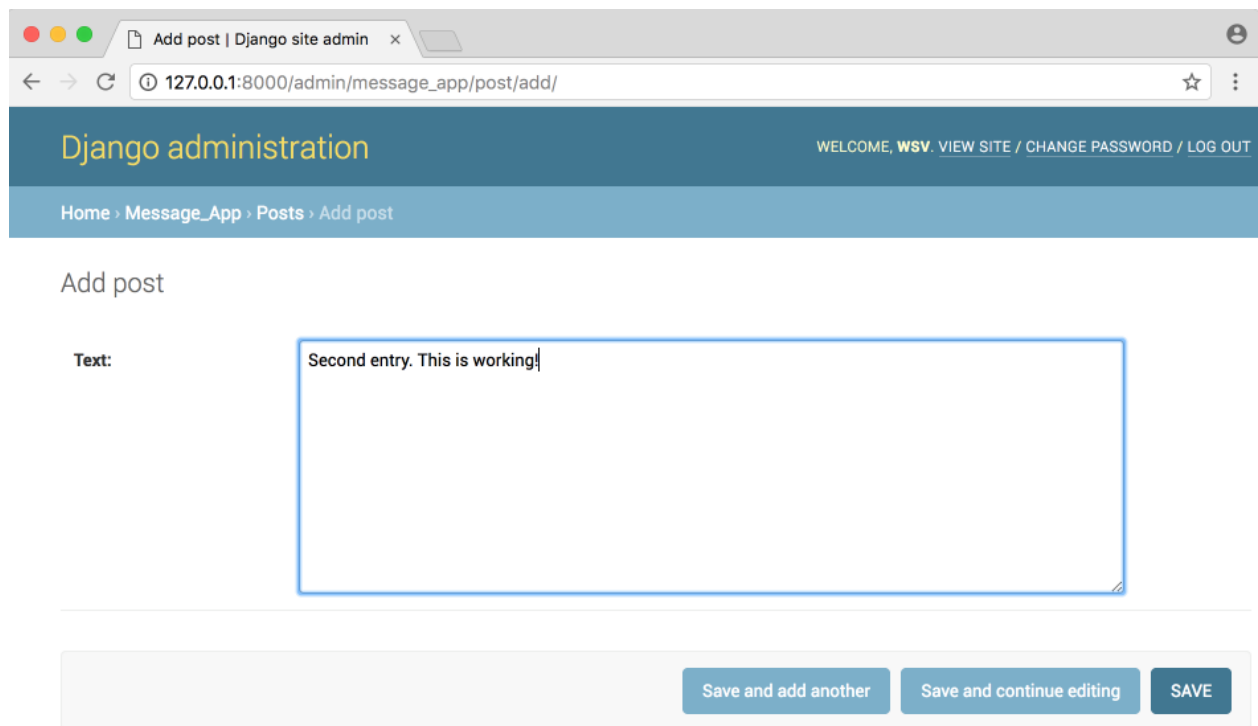
В основном мы закончили, но давайте создадим еще несколько сообщений на доске объявлений в администраторе Django, чтобы убедиться, что они будут отображаться правильно на главной странице.

## Добавление новых записей

Чтобы добавить новые сообщения в нашу доску объявлений, вернитесь в Admin:

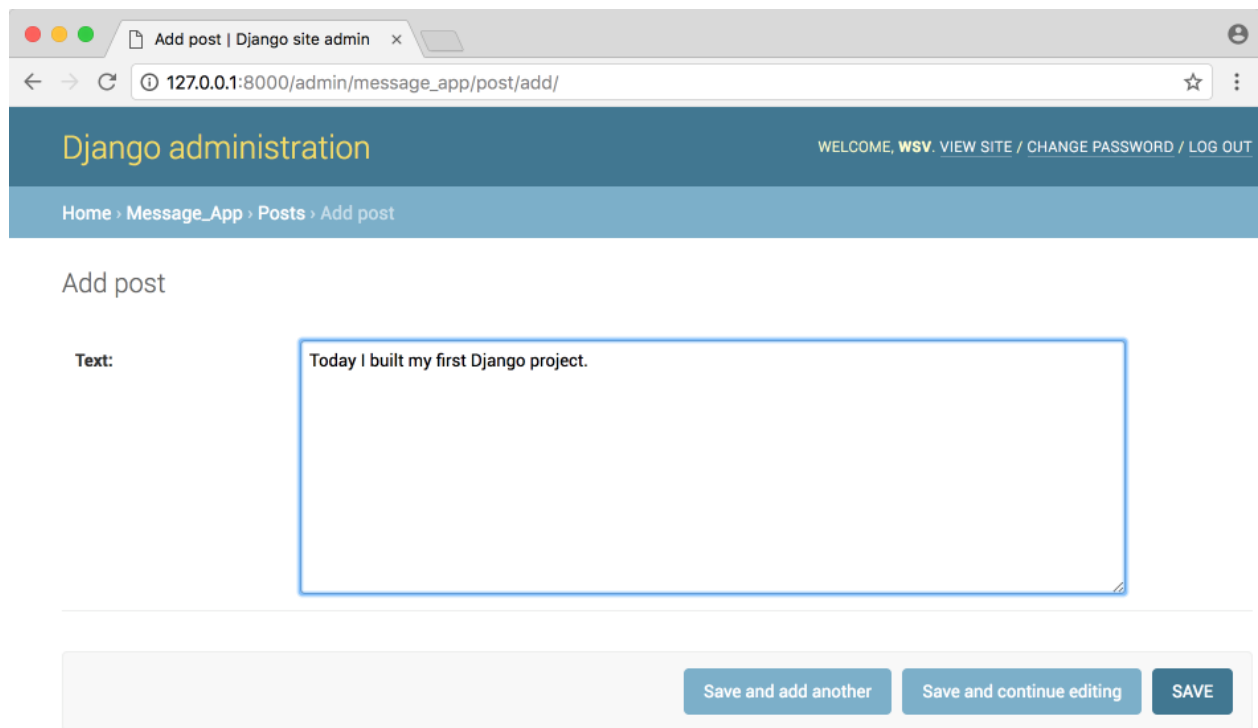
<http://127.0.0.1:8000/admin/>

И создайте еще два поста. Вот как выглядит мой:



The screenshot shows a web browser window with the address bar displaying `127.0.0.1:8000/admin/message_app/post/add/`. The page title is "Add post | Django site admin". The Django administration header is visible, showing "Django administration" and a welcome message for "WSV" with links for "VIEW SITE", "CHANGE PASSWORD", and "LOG OUT". The breadcrumb trail is "Home > Message\_App > Posts > Add post". The main heading is "Add post". Below it, there is a "Text:" label and a large text area containing the text "Second entry. This is working!". At the bottom of the form, there are three buttons: "Save and add another", "Save and continue editing", and "SAVE".

**Admin entry**



The screenshot shows a web browser window with the title "Add post | Django site admin". The address bar displays "127.0.0.1:8000/admin/message\_app/post/add/". The page header is "Django administration" with a welcome message "WELCOME, wsv." and links for "VIEW SITE", "CHANGE PASSWORD", and "LOG OUT". The breadcrumb trail is "Home > Message\_App > Posts > Add post". The main heading is "Add post". Below it, there is a "Text:" label and a large text area containing the text "Today I built my first Django project." At the bottom, there are three buttons: "Save and add another", "Save and continue editing", and "SAVE".

Add post

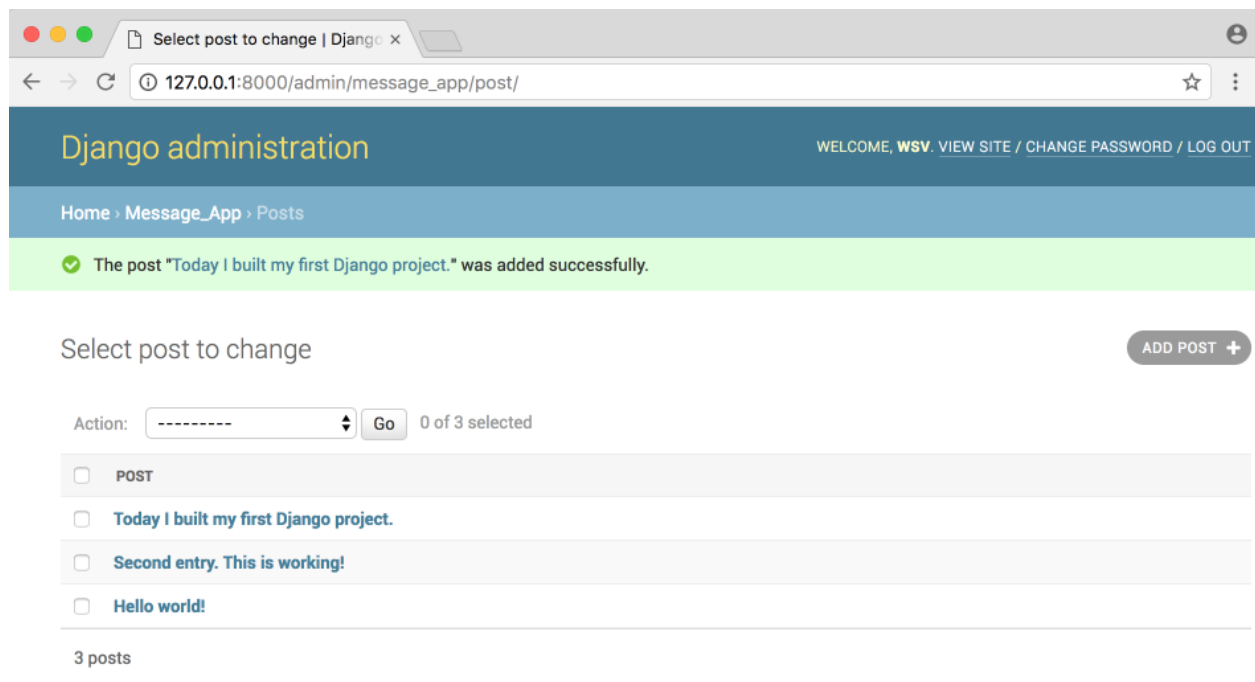
Text:

Today I built my first Django project.

Save and add another Save and continue editing SAVE

### Admin entry





### Updated admin entries section

Если вы вернетесь на домашнюю страницу, вы увидите, что она автоматически отображает наши отформатированные сообщения. Ура!



- Hello world!
- Second entry. This is working!
- Today I built my first Django project.

### Homepage with three entries

Все работает так, что самое время инициализировать наш каталог, добавить новый код и

и включить наш первый `git commit`.

### Command Line

---

```
(mb) $ git init
(mb) $ git add -A
(mb) $ git commit -m 'initial commit'
```

---

## Tests

Раньше мы тестировали только статические страницы, поэтому мы использовали `SimpleTestCase` (простой тест). Но теперь, когда наша домашняя страница работает с базой данных, нам нужно использовать `TestCase`, которая позволит нам создать «тестовую» базу данных, которую мы можем проверить. Другими словами, нам не нужно запускать тесты в нашей реальной базе данных, но вместо этого можно создать отдельную тестовую базу данных, заполнить ее образцами данных, а затем протестировать ее.

Начнем с добавления образца записи в текстовое поле базы данных, а затем проверим, правильно ли она хранится в базе данных. Важно, чтобы все наши методы тестирования начинались с `test_`, чтобы Django знал что тестировать! Код будет выглядеть следующим образом:

### Code

---

```
# posts/tests.py

from django.test import TestCase
from .models import Post


class PostModelTest(TestCase):


    def setUp(self):
        Post.objects.create(text='just a test')
```

```
def test_text_content(self):  
    post=Post.objects.get(id=1)  
    expected_object_name = f'{post.text}'  
    self.assertEqual(expected_object_name, 'just a test')
```

---

В верхней части мы импортируем модуль `TestCase`, который позволяет нам создать образец базы данных, а затем импортировать нашу `Post` модель. Мы создаем новый класс `PostModelTest` и добавляем метод `setUp` для создания новой базы данных, которая имеет только одну запись: сообщение с текстовым полем, содержащим строку `'just a test'`.

Затем мы запускаем наш первый тест, `test_text_content`, чтобы проверить, что поле базы данных на самом деле содержит только тест. Мы создаем переменную `post`, которая представляет первый идентификатор (`id`) в нашей модели `Post`. Запомните, что Django автоматически устанавливает этот `id` для нас. Если бы мы создали еще одну запись, у нее был бы идентификатор 2, следующий был бы 3 и так далее.

Следующая строка использует  `f-strings` которая является очень классным дополнением к Python 3.6. Она позволяет помещать переменные непосредственно в строки, если переменные окружены фигурными скобками `{}`. Здесь мы устанавливаем `expected_object_name` как строку значения в `post.text`, который должен быть просто тест.

В последней строке мы используем `assertEqual`, чтобы проверить, что наша недавно созданная запись действительно соответствует тому, что мы вводим наверху. Продолжайте и запустите тест в командной строке `python manage.py test`.

**Command Line**

---

```
(mb) $ python manage.py test
Creating test database for alias 'default'...
System check identified no issues (0 silenced).
```

```
.
```

```
-----
Ran 1 test in 0.001s
```

```
OK
```

```
Destroying test database for alias 'default'...
```

---

Прошло!

Не волнуйтесь, если предыдущее объяснение было похоже на информационную перегрузку. Это естественно, когда вы впервые начинаете писать тесты, но вскоре вы обнаружите, что большинство тестов, которые вы пишете, на самом деле довольно повторяющиеся.

Время для нашего второго теста. Первый тест был для модели, но теперь мы хотим проверить нашу единственную страницу: домашнюю страницу (homepage). В частности, мы хотим проверить, что она существует (throws an HTTP 200 response), использует view home и использует home.html шаблон(template).

Нам нужно добавить еще один импорт наверху для reverse и совершенно новый класс HomePageViewTest для нашего теста.

**Code**

---

```
from django.test import TestCase
from django.urls import reverse
from .models import Post

class PostModelTest(TestCase):

    def setUp(self):
        Post.objects.create(text='just a test')

    def test_text_content(self):
        post=Post.objects.get(id=1)
        expected_object_name = f'{post.text}'
        self.assertEqual(expected_object_name, 'just a test')

class HomePageViewTest(TestCase):

    def setUp(self):
        Post.objects.create(text='this is another test')

    def test_view_url_exists_at_proper_location(self):
        resp = self.client.get('/')
        self.assertEqual(resp.status_code, 200)

    def test_view_url_by_name(self):
        resp = self.client.get(reverse('home'))
        self.assertEqual(resp.status_code, 200)
```

```
def test_view_uses_correct_template(self):  
    resp = self.client.get(reverse('home'))  
    self.assertEqual(resp.status_code, 200)  
    self.assertTemplateUsed(resp, 'home.html')
```

---

Если вы запустите наши тесты снова, вы должны увидеть, что они проходят.

### Command Line

---

```
(mb) $ python manage.py test  
Creating test database for alias 'default'...  
System check identified no issues (0 silenced).  
.
```

-----  
Ran 4 tests in 0.036s

OK

```
Destroying test database for alias 'default'...
```

---

Почему там написано четыре теста? Запомните, что наш setUp метод на самом деле не тест, он просто позволяет нам запускать последующие тесты. Наши четыре реальных теста это test\_test\_content, test\_view\_url\_exists\_at\_proper\_location, test\_view\_url\_by\_name, и test\_view\_uses\_correct\_template.

Любая функция, имеющая в начале слово test\* в существующем файле tests.py будет запущена при выполнении команды python manage.py test.

Мы закончили добавление кода для нашего тестирования, пришло время внести изменения в git.

## Command Line

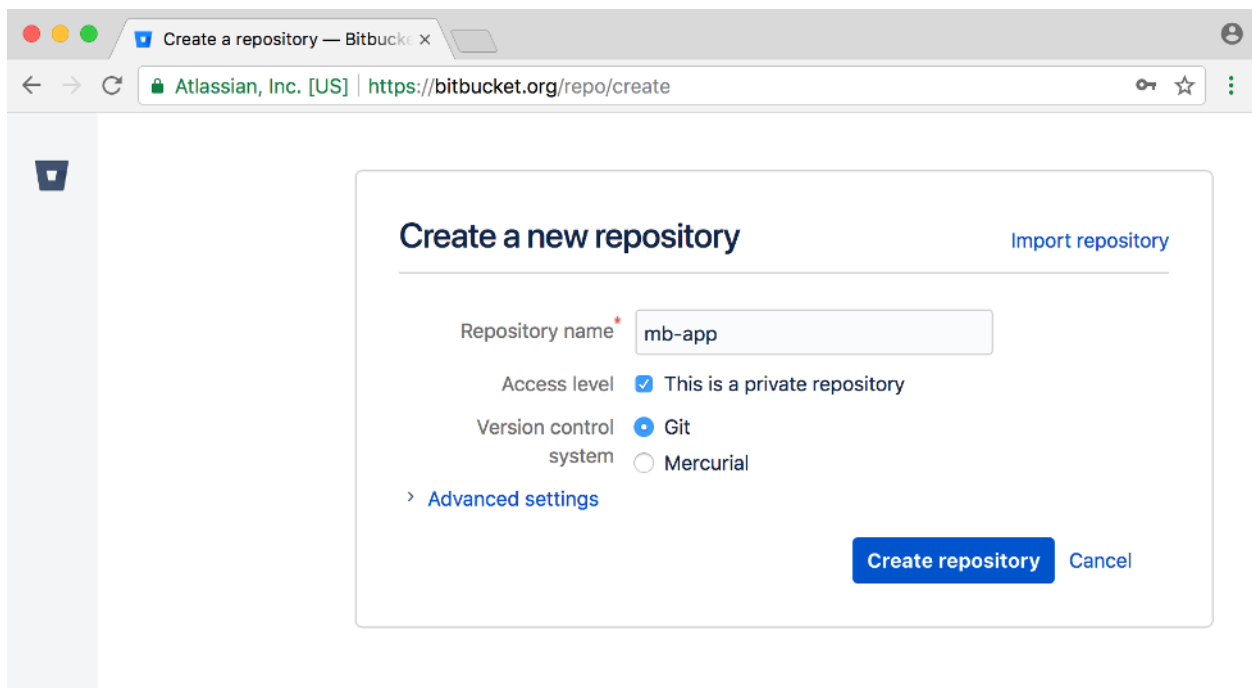
```
(mb) $ git add -A
```

```
(mb) $ git commit -m 'added tests'
```

## Bitbucket

Нам также нужно сохранить наш код на Bitbucket. Это хорошая привычка, чтобы сохранить код в случае, если что-нибудь случится с вашим локальным компьютером, и это также позволяет вам делиться и сотрудничать с другими разработчиками.

У Вас уже должна быть учетная запись Bitbucket из главы 3, поэтому создайте новый репозиторий, который мы назовем mb-app.



### Bitbucket create app

На следующей странице нажмите на нижнюю ссылку “I have an existing project”.

Скопируйте две команды для подключения, а затем нажмите repository to Bitbucket.

Это должно выглядеть так, замените `wsvincent` (мое имя пользователя) на ваш логин в системе `bitbucket` :

### Command Line

---

```
(mb) $ git remote add origin git@bitbucket.org:wsvincent/mb-app.git
(mb) $ git push -u origin master
```

---

## Heroku configuration

Вы также должны уже иметь настройки учетной записи Heroku установленные из главы 3. Нам необходимо внести следующие изменения в наш проект доски объявлений, чтобы развернуть его в интернете:

- обновить `Pipfile.lock`
- новый `Procfile`
- установить `gunicorn`
- обновить `settings.py`

В вашем `Pipfile` укажите версию Python мы используем 3.6. Добавьте эти две строки в конец файла.

### Code

---

```
# Pipfile
[requires]
python_version = "3.6"
```

---

Запустите `pipenv lock` для создания соответствующего `Pipfile.lock`.



**Command Line**

---

```
(mb) $ pipenv lock
```

---

Затем создайте Procfile, который указывает Heroku, как запустить удаленный сервер, где наш код будет работать.

**Command Line**

---

```
(mb) $ touch Procfile
```

---

Теперь мы укажем Heroku использовать gunicorn в качестве нашего рабочего сервера и посмотреть в наш mb\_project.wsgi файл для получения дальнейших инструкций.

**Command Line**

---

```
web: gunicorn mb_project.wsgi --log-file -
```

---

Далее установим [gunicorn](#) который мы будем использовать для production но при этом используя внутренний сервер Django для локальной разработки.

**Command Line**

---

```
(mb) $ pipenv install gunicorn
```

---

Наконец, обновите ALLOWED\_HOSTS в нашем файле settings.py

**file. Code**

---

```
# mb_project/settings.py  
ALLOWED_HOSTS = ['*']
```

---

Все мы закончили! Добавьте и зафиксируйте наши новые изменения в git, а затем отправьте их в Bitbucket.

**Command Line**

---

```
(mb) $ git status
(mb) $ git add -A
(mb) $ git commit -m 'New updates for Heroku deployment'
(mb) $ git push -u origin master
```

---

## Heroku развертывание

Убедитесь, что вы вошли в свою учетную запись Heroku.

**Command Line**

---

```
(mb) $ heroku login
```

---

Затем запустите команду `heroku create` и Heroku случайным образом сгенерирует для вас имя приложения. Вы сможете настроить это позже, если будет необходимо.

**Command Line**

---

```
(mb) $ heroku create
Creating app... done, agile-inlet-25811
https://agile-inlet-25811.herokuapp.com/ | https://git.heroku.com/agile-inlet-25811.git
```

---

Настройте git на использование имени вашего нового приложения, когда вы отправляете код на Heroku. Мое сгенерированное имя Heroku это `agile-inlet-25811` поэтому команда выглядит следующим образом.

**Command Line**

---

```
(mb) $ heroku git:remote -a agile-inlet-25811
```

---

Укажите Heroku игнорировать статические файлы, которые мы подробно рассмотрим при развертывании нашего приложения блога в книге позже.

**Command Line**

---

```
(mb) $ heroku config:set DISABLE_COLLECTSTATIC=1
```

---

Отправьте код в Heroku и добавьте бесплатное масштабирование, чтобы он действительно работал в интернете, иначе код просто сидит там.

**Command Line**

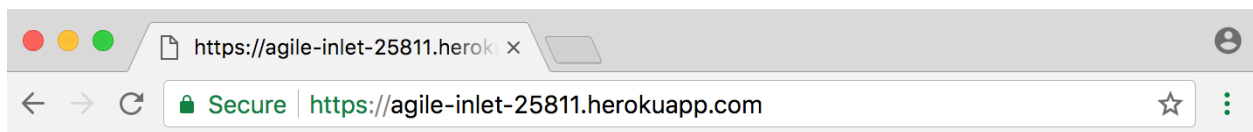
---

```
(mb) $ git push heroku master  
(mb) $ heroku ps:scale web=1
```

---

Если вы откроете новый проект heroku open, он автоматически запустит новое окно браузера с URL вашего приложения. Мой находится по адресу:

<https://agile-inlet-25811.herokuapp.com/>.



## Message board homepage

- Hello world!
- Second entry. This is working!
- Today I built my first Django project.

**Live site**

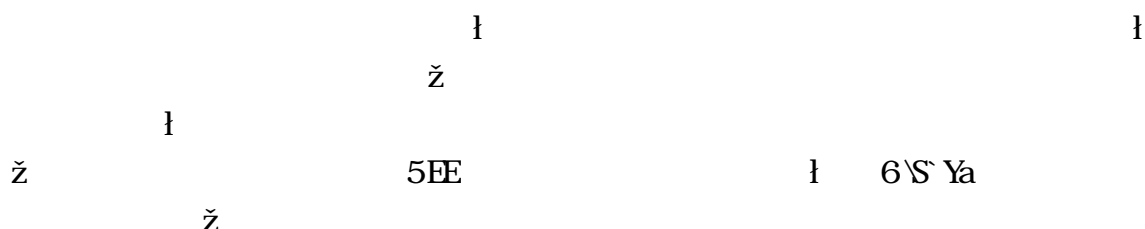
## **Вывод**

На данный момент мы создали, протестировали и развернули наше первое приложение на основе базы данных. Хотя это намеренно довольно просто, теперь мы знаем, как создать модель базы данных, обновить ее с помощью панели администратора, а затем отобразить содержимое на веб-странице. Но чего-то не хватает, не так ли?

В реальном мире пользователям нужны формы для взаимодействия с нашим сайтом. Ведь не у всех должен быть доступ к админ панели. В следующей главе мы создадим приложение блога, которое использует формы, чтобы пользователи могли создавать, редактировать и удалять записи. Мы также добавим стили CSS.

## 5: Blog app/

fi



Как описано в предыдущих главах, наши шаги для создания нового проекта Django будут выглядеть следующим образом:

- Создать новую папку для нашего проекта на рабочем столе с именем `blog`
- Установить Django в новой виртуальной среде
- Создать новый проект Django с именем `blog_project`
- Создать новое приложение `blog`
- Выполнить миграцию для установки базы данных
- Обновить `settings.py`

Выполните следующие команды в новой консоли командной строки. Обратите внимание, что фактическое имя виртуальной среды будет `(blog-XXX)`, где `XXX` представляет случайные символы. Я использую здесь `(blog)`, чтобы было проще, так как мое имя будет отличаться от вашего.

И не забудьте указать пробел с точкой "." в конце команды для создания нашего нового `blog_project`. (прм. переводчика. Создает проект Django в существующей директории без создания новой папки, не обязательно)

### Command Line

---

```
$ cd ~/Desktop
$ mkdir blog
$ cd blog
$ pipenv install django
$ pipenv shell
(blog) $ django-admin startproject blog_project .
(blog) $ python manage.py startapp blog
(blog) $ python manage.py migrate
(blog) $ python manage.py runserver
```

---

Чтобы убедиться, что Django знает о нашем новом приложении, откройте текстовый редактор и добавьте новое приложение в `INSTALLED_APPS` в нашем `settings.py` файле:

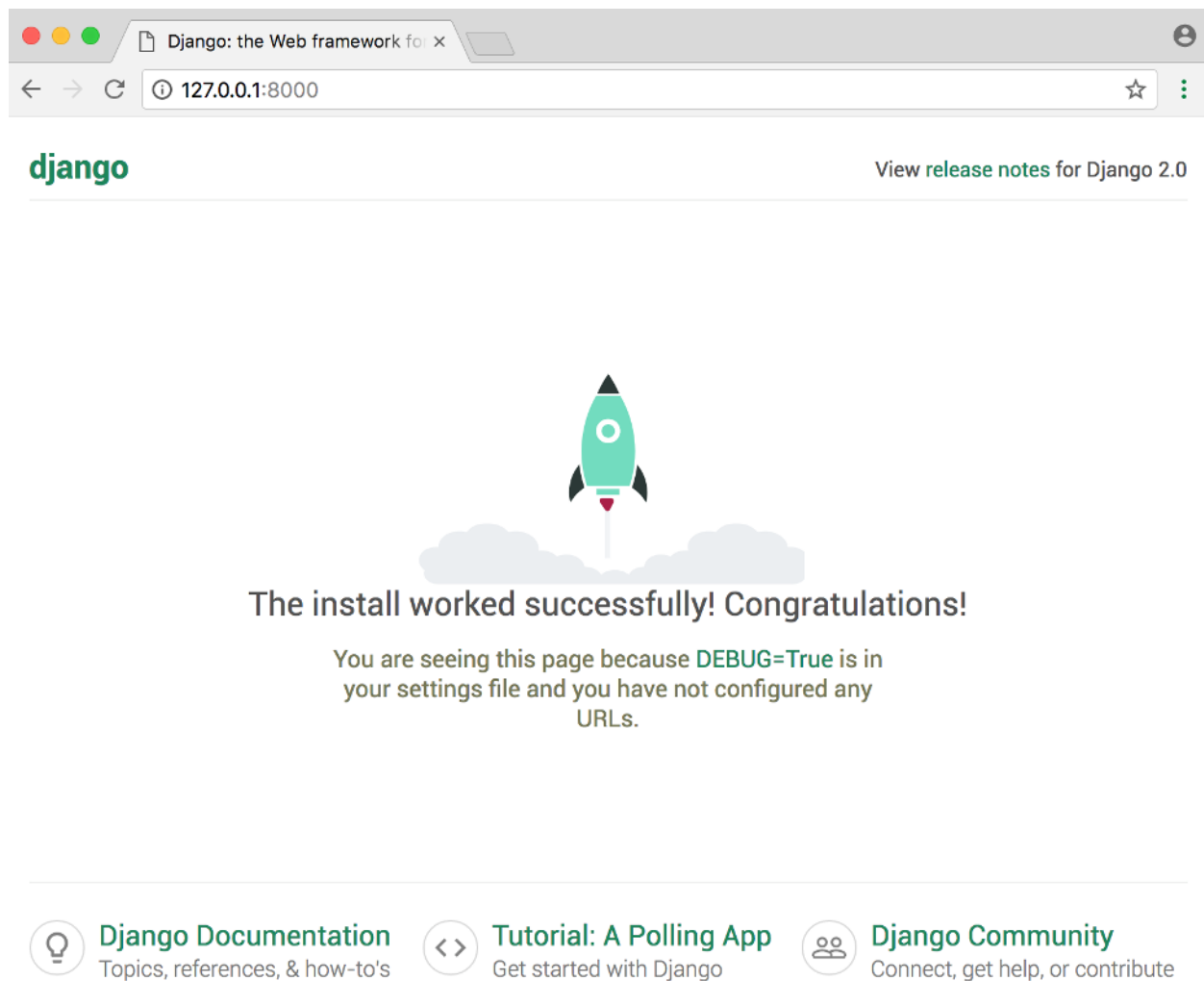
### Code

---

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

---

Если вы перейдете по <http://127.0.0.1:8000/> должны увидеть следующую страницу.



### Django welcome page

Хорошо, первичная настройка закончена! Далее мы создадим модель базы данных для записей блога.

## Database Models(модели базы данных)

Каковы характеристики типичного приложения для блога? В нашем случае давайте придерживаться простоты и предположим, что у каждого сообщения есть название, автор и тело. Мы можем превратить это в модель базы данных, откроем файл `blog/models.py` и введем показанный ниже код :

**Code**

---

```
# blog/models.py

from django.db import models

class Post(models.Model):
    title = models.CharField(max_length=200)
    author = models.ForeignKey(
        'auth.User',
        on_delete=models.CASCADE,
    )
    body = models.TextField()

    def __str__(self):
        return self.title
```

---

В верхней части мы импортируем класс `models` а затем создаем подкласс `models.Model` с именем `Post`. Используя функционал подкласса, мы автоматически имеем доступ ко всему в [django.db.models.Models](#) и можем добавлять дополнительные поля и методы по своему желанию.

Для `title` мы ограничиваем длину до 200 символов, а для `body` мы используем текстовое поле, которое будет автоматически расширяться по мере необходимости, чтобы соответствовать тексту пользователя. В Django доступно много типов полей; вы можете увидеть [полный список здесь](#).

Для поля `author` мы используем [ForeignKey](#) это предоставляет связь *многие-к-одному*. Это означает, что данный пользователь может быть автором многих различных постов в блоге, но не наоборот. Ссылка на встроенную модель `User` которой Django обеспечивает аутентификацию. Для всех связей многие-к-одному таких как `ForeignKey` нам также необходимо указать параметр `on_delete`.



Теперь, когда наша новая модель базы данных создана, нам нужно создать для нее новую запись миграции и перенести изменения в нашу базу данных. Это двухэтапный процесс и может быть выполнен следующими командами:

#### Command Line

---

```
(blog) $ python manage.py makemigrations blog
(blog) $ python manage.py migrate blog
```

---

Наша база настроена! Что дальше?

## Admin

Нам нужен способ доступа к нашим данным. Войдем в Django админ! Но сначала создадим учетную запись суперпользователя, введя приведенную ниже команду и после отвечая на запросы введем адрес электронной почты и пароль. Обратите внимание, что при вводе пароля он не будет отображаться на экране по соображениям безопасности.

#### Command Line

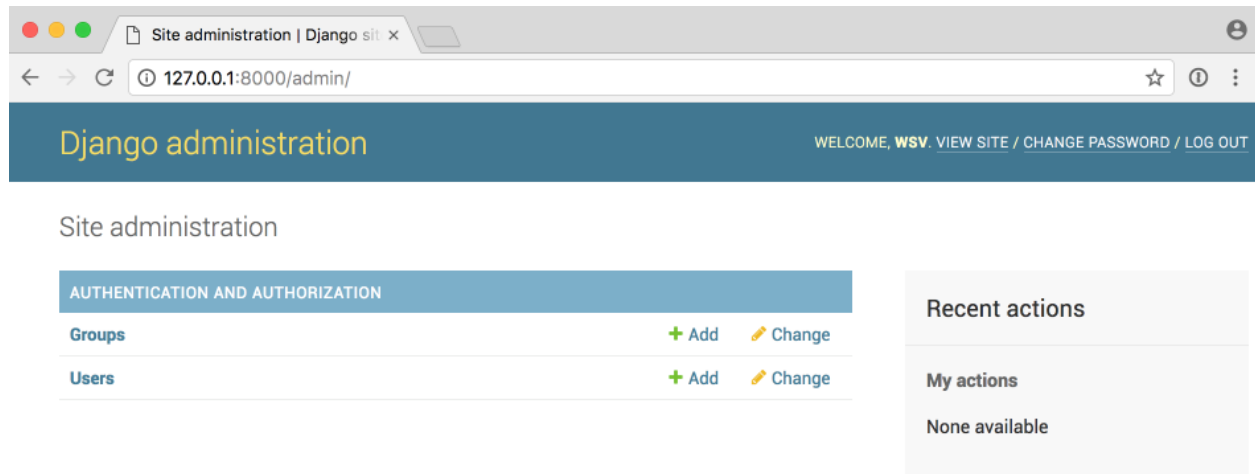
---

```
(blog) $ python manage.py createsuperuser
Username (leave blank to use 'wsv'): wsv
Email:
Password:
Password (again):
Superuser created successfully.
```

---

Теперь снова запустите сервер Django с помощью команды `python manage.py runserver` и откройте Django админ <http://127.0.0.1:8000/admin/>. Войдите с новой учетной записью суперпользователя.

Упс! А где наша новая Post модель?



### Admin homepage

Мы забыли обновить `blog/admin.py` так что давайте сейчас сделаем это.

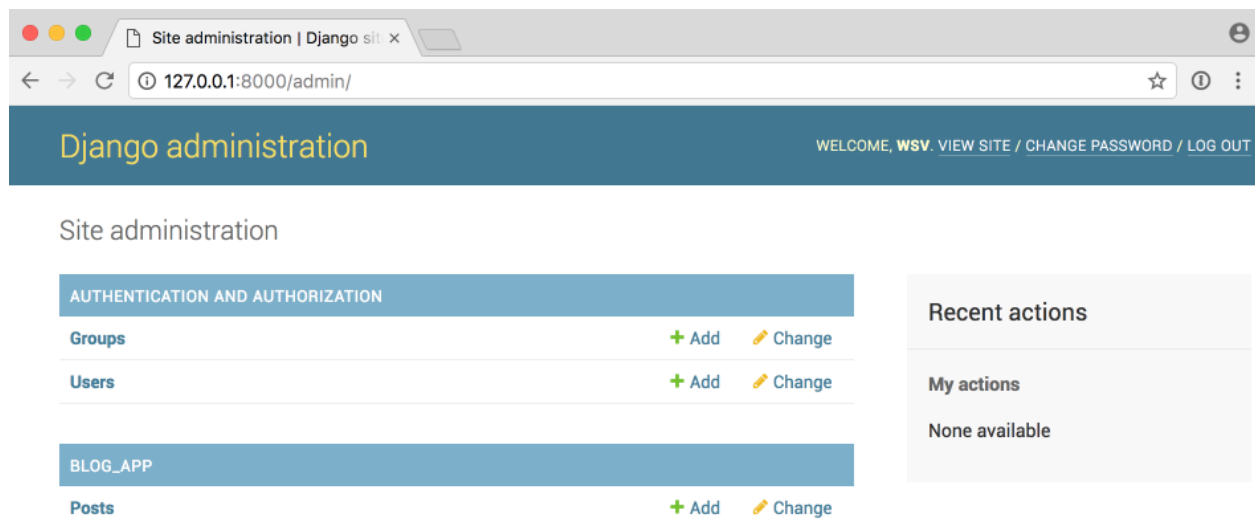
#### Code

```
# blog/admin.py

from django.contrib import admin
from .models import Post

admin.site.register(Post)
```

Если вы обновите страницу, вы увидите это обновление.



### Admin homepage

Давайте добавим два сообщения в блог, чтобы у нас были примеры данных для работы. Нажмите кнопку + Add рядом с Posts чтобы создать новую запись. Не забудьте добавить “author” к каждому сообщению, так как по умолчанию все поля модели обязательны для заполнения. Если вы попытаетесь ввести сообщение без автора вы увидите ошибку. Если мы захотим изменить это, мы можем добавить **field options**(параметры поля) в нашу модель, чтобы сделать это поле необязательным или заполнить его значением по умолчанию

The screenshot shows a web browser window with the address bar displaying `127.0.0.1:8000/admin/blog_app/post/add/`. The page title is "Add post | Django site admin". The Django administration header is visible, showing "Django administration" and a welcome message for user "wsv" with links for "VIEW SITE", "CHANGE PASSWORD", and "LOG OUT". The breadcrumb trail is "Home > Blog\_App > Posts > Add post".

The main content area is titled "Add post" and contains the following form fields:

- Author:** A dropdown menu showing "wsv" with edit and add icons.
- Title:** A text input field containing "Hello world!".
- Text:** A large text area containing "My first blog post. Woohoo!".

At the bottom of the form, there are three buttons: "Save and add another", "Save and continue editing", and "SAVE".

**Первый пост**

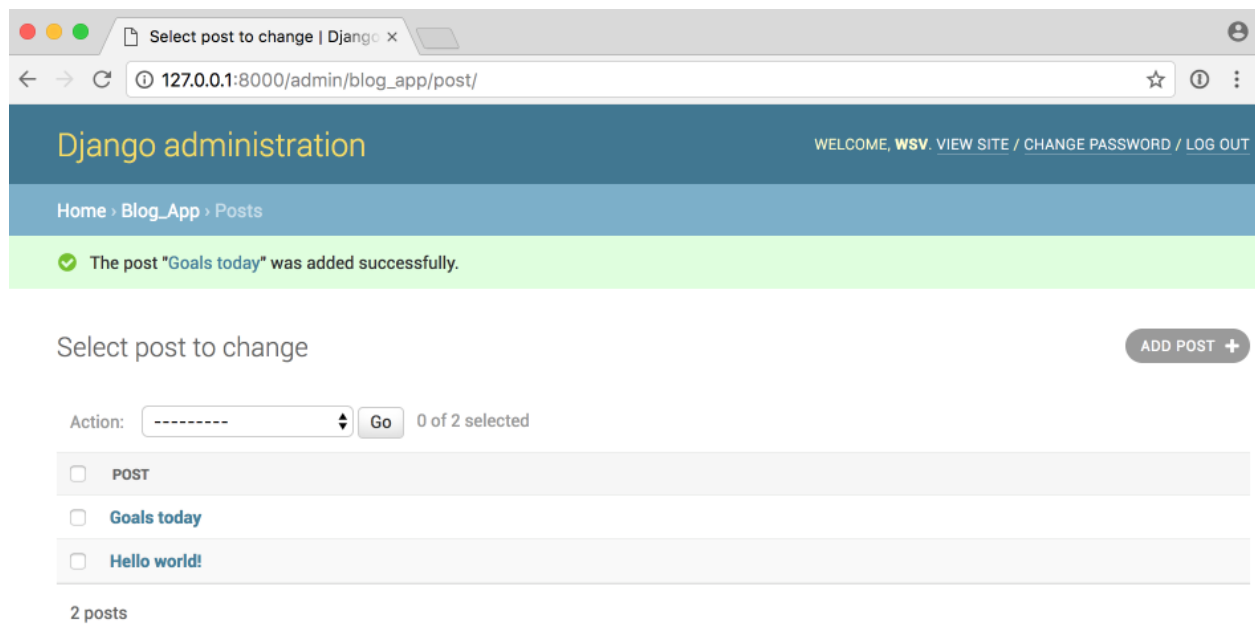
The screenshot shows a web browser window with the address bar displaying `127.0.0.1:8000/admin/blog_app/post/add/`. The page title is "Django administration" and the user is logged in as "wsv". The breadcrumb trail is "Home > Blog\_App > Posts > Add post".

The "Add post" form contains the following fields:

- Author:** A dropdown menu showing "wsv" with a pencil icon and a plus sign to add a new user.
- Title:** A text input field containing "Goals today".
- Text:** A large text area containing "Learn Django and build a blog application."

At the bottom of the form, there are three buttons: "Save and add another", "Save and continue editing", and "SAVE".

### Второй пост



### Админка с двумя постами

Теперь, когда наша модель базы данных завершена, нам нужно создать необходимые view, URL-адреса и шаблоны, чтобы мы могли отображать информацию в нашем веб-приложении.

## URLs

Мы хотим, чтобы сообщения в блоге отображались на главной странице, поэтому, как и в предыдущих главах, мы сначала настроим URLConfs уровня проекта, а затем URLConfs уровня приложения, что бы достичь этого. Обратите внимание на то, что “уровень проекта” означает в той же родительской папке где папка `blog_project` и папка приложения `blog`.

В командной строке остановите существующий сервер с помощью `Control-c` и создайте новый `urls.py` файл в вашем `blog`:

### Command Line

---

```
(blog) $ touch blog/urls.py
```

---

Теперь обновите его с помощью кода ниже.

### Code

---

```
# blog/urls.py
from django.urls import path

from . import views

urlpatterns = [
    path('', views.BlogListView.as_view(), name='home'),
]
```

---

Мы импортируем `views` (которые скоро создадим) в верхней части. Пустая строка `''` говорит Python что соответствует всем значениям, далее мы делаем URL именованным `home`, к которому мы можем обратиться в наших `views` позже. Хотя это не обязательно но добавление имени к URL это хорошая практика которую необходимо принять так как это помогает сохранять организованность когда количество ваших URL растет.

Также мы должны обновить файл `urls.py` на уровне проекта, чтобы он мог пересылать все запросы непосредственно на `blog` приложение.

**Code**

---

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

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

---

Мы добавили `include` во второй строке и url образец используя пустую строку регулярного выражения `' '` указывающее, что URL запросы должны быть перенаправлены на URL-адреса блога(`blog`) для дальнейших инструкций.

## Views

Мы собираемся использовать `views` на основе классов, но если вы хотите увидеть функциональный способ создания приложения для блога, я настоятельно рекомендую Django Girls Tutorial. Это отличный учебник.

В нашем файле `views` добавьте код ниже, чтобы отобразить содержимое нашей модели `Post` с помощью `ListView`.



**Code**


---

```
# blog/views.py

from django.views.generic import ListView

from . models import Post


class BlogListView(ListView):
    model = Post
    template_name = 'home.html'
```

---

**Templates/**

```
GD>5a`X hWe l
, /fWb'SfWz узнали из ы &
l наш ž TSeVZf_ ^
Za_ VZf_ 1 ž l
l base.html.
```

Начнем с создания каталога шаблонов(templates) на уровне проекта с двумя файлами шаблонов.

### Command Line

---

```
(blog) $ mkdir templates
(blog) $ touch templates/base.html
(blog) $ touch templates/home.html
```

---

Затем обновите settings.py чтобы Django знал, где искать наши шаблоны.

### Code

---

```
# blog_project/settings.py
TEMPLATES = [
    {
        ...
        'DIRS': [os.path.join(BASE_DIR, 'templates')],
        ...
    },
]
```

---

Теперь обновите шаблон base.html следующим образом.

### Code

---

```
<!-- templates/base.html -->
<html>
  <head>
    <title>Django blog</title>
  </head>
  <body>
    <header>
      <h1><a href="/">Django blog</a></h1>
    </header>
```

```

    <div class="container">
        {% block content %}
        {% endblock content %}
    </div>
</body>
</html>

```

---

Обратите внимание на то, что код между `{% block content %}` и `{% endblock content %}` может быть заполнен другими шаблонами. Говоря об этом, вот код для `home.html`.

### Code

```

<!-- templates/home.html -->
{% extends 'base.html' %}

{% block content %}
    {% for post in object_list %}
        <div class="post-entry">
            <h2><a href="">{{ post.title }}</a></h2>
            <p>{{ post.body }}</p>
        </div>
    {% endfor %}
{% endblock content %}

```

---

В верхней части отметим, что этот шаблон расширяет(`extends`) `base.html` и обертывает нужный вам код с помощью контент-блоков. Мы используем язык шаблонов [Django Templating Language](#), чтобы создать простой цикл `for` для каждой записи блога. Обратите внимание, что `object_list` происходит из `ListView` и содержит все объекты в нашем `view`.

Если вы снова запустите сервер Django: `python manage.py runserver`.

И обновите <http://127.0.0.1:8000/> мы видим, что это работает.



## Django blog

### Hello world!

My first blog post. Woohoo!

### Goals today

Learn Django and build a blog application.

Главная страница блога с двумя постами

Но выглядит это ужасно. Давайте это исправим!

## Static files(статичные файлы)

Нам нужно добавить CSS, который называется статическим файлом, потому что, в отличие от нашего динамического контента базы данных, он не изменяется. К счастью добавлять, статические файлы, такие как CSS, JavaScript и изображения в наш проект Django очень просто.

В готовом к производству Django проекте вы, как правило, храните его в сети доставки контента (CDN) для лучшей производительности, но для наших целей хранение файлов локально это нормально.

Сначала закройте наш локальный сервер с *Control-c*. Затем создайте папку на уровне проекта с именем static.

### Command Line

---

```
(blog) $ mkdir static
```

---

Как и в нашей папке с шаблонами, нам нужно обновить settings.py, чтобы сообщить Django, где искать эти статические файлы. Мы можем обновить settings.py с изменением строки для STATICFILES\_DIRS. Добавьте его в нижнюю часть файла под заголовком STATIC\_URL.

### Code

---

```
# blog_project/settings.py
STATICFILES_DIRS = [os.path.join(BASE_DIR, 'static')]
```

---

Теперь создайте папку css в static и добавьте в нее новый файл base.css.

### Command Line

---

```
(blog) $ mkdir static/css
(blog) $ touch static/css/base.css
```

---

Что мы должны поместить в ваш файл? Как насчет изменения title на красный?

### Code

---

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

---

Последний шаг. Нам нужно добавить статические файлы в ваши шаблоны, добавив {% load staticfiles %} в начало base.html. Потому что другие наши шаблоны наследуются от base.html мы должны добавить это только один раз. Добавьте новую строку в нижней части кода <head>< / head>, которая явно ссылается на наш новый файл base.css.

**Code**

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

<html>

  <head>

    <title>Django blog</title>

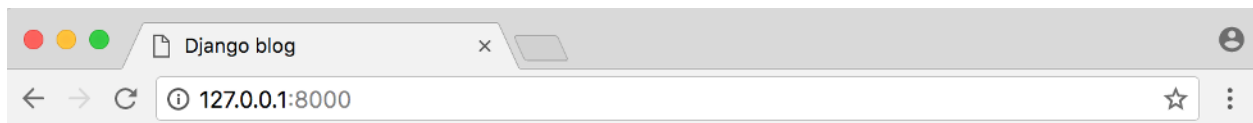
    <link rel="stylesheet" href="{% static 'css/base.css' %}">

  </head>

  ...
```

УФ! Это было немного больно, но это одноразовая боль. Теперь мы можем добавить статические файлы в нашу папку static, и они автоматически появятся во всех наших шаблонах.

Запустите сервер снова с `python manage.py runserver` и посмотрите на нашем обновленном сайте <http://127.0.0.1:8000/>.



## Django blog

### Hello world!

My first blog post. Woohoo!

### Goals today

Learn Django and build a blog application.

#### Домашняя страница блога с красным заголовком

Мы можем сделать немного лучше. Как насчет того, чтобы добавить пользовательский шрифт и еще немного CSS? Поскольку эта книга не является учебником по CSS, просто вставьте следующее между тегами `<head>< / head>`, чтобы добавить **Source Sans Pro**, бесплатный шрифт от Google.

**Code**

---

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

<html>
<head>
    <title>Django blog</title>
    <link href="https://fonts.googleapis.com/css?family=Source+Sans+Pro:400" rel="\
stylesheet">
    <link rel="stylesheet" href="{% static 'css/base.css' %}">
</head>

...
```

---

Затем обновите наш css файл, скопировав и вставив следующий код:

**Code**

---

```
/* static/css/base.css */

body {
    font-family: 'Source Sans Pro', sans-serif;
    font-size: 18px;
}

header {
    border-bottom: 1px solid #999;
    margin-bottom: 2rem;
    display: flex;
}

header h1 a {
    color: red;
}
```

```
    text-decoration: none;
}

.nav-left {
    margin-right: auto;
}

.nav-right {
    display: flex;
    padding-top: 2rem;
}

.post-entry {
    margin-bottom: 2rem;
}

.post-entry h2 {
    margin: 0.5rem 0;
}

.post-entry h2 a,
.post-entry h2 a:visited {
    color: blue;
    text-decoration: none;
}

.post-entry p {
    margin: 0;
```

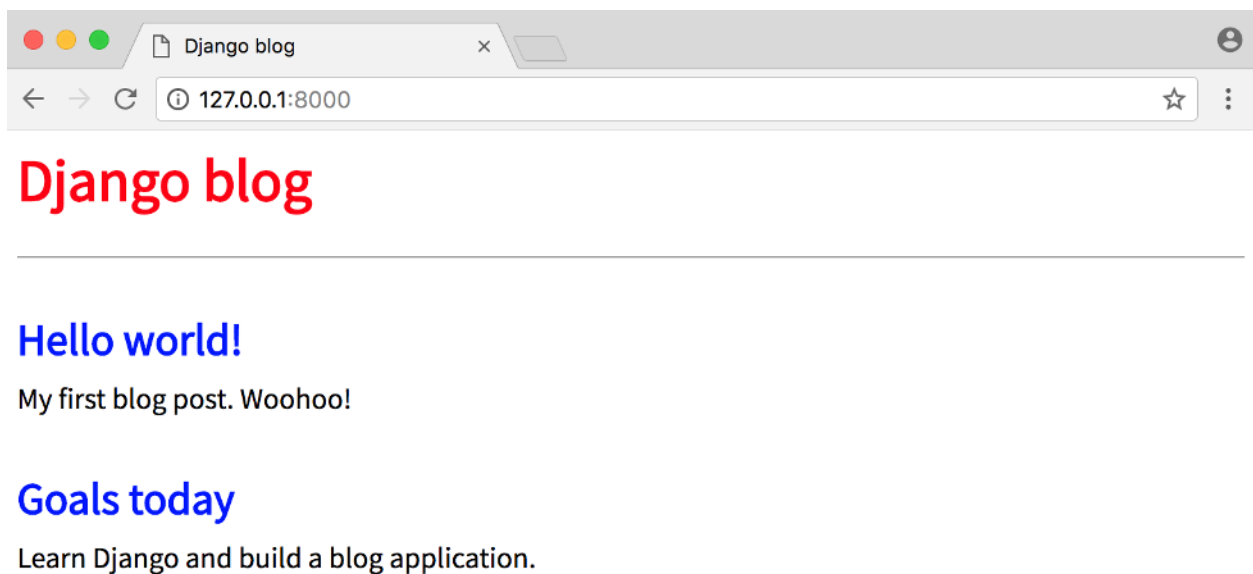


```
    font-weight: 400;
}

.post-entry h2 a:hover {
    color: red;
}
```

---

Обновить домашнюю страницу <http://127.0.0.1:8000/> и вы должны увидеть следующее.



Домашняя страница блога с CSS

## Отдельные страницы блога

Теперь мы можем добавить функциональность для отдельных страниц блога. Как мы можем это сделать? Мы должны создать новое представление, URL и шаблон. Я надеюсь, что вы заметили шаблон в разработке с Django сейчас!

Начните с представления. Для упрощения можно использовать общее представление `DetailView` на основе классов. В верхней части файла добавьте `DetailView` в список импорта, а затем создайте наше новое представление под названием `BlogDetailView`.

view called `BlogDetailView`.

### Code

---

```
# blog/views.py

from django.views.generic import ListView, DetailView

from . models import Post


class BlogListView(ListView):
    model = Post
    template_name = 'home.html'


class BlogDetailView(DetailView):
    model = Post
    template_name = 'post_detail.html'
```

---

В этом новом представлении мы определяем модель, которую мы используем, `Post` и шаблон, с которым мы хотим ее связать, `post_detail.html`. По умолчанию `DetailView` предоставит контекстный объект, который мы можем использовать в нашем шаблоне, называемый либо объектом, либо строчным именем нашей модели `post`. Кроме того, `DetailView` ожидает либо первичный ключ, либо `slug`, переданный ему в качестве идентификатора. Подробнее об этом вкратце.

Теперь выйдите из локального сервера `Control-c` и создайте наш новый шаблон для деталей поста следующим образом:

## Command Line

---

```
(blog) $ touch templates/post_detail.html
```

---

Затем введите следующий код:

## Code

---

```
<!-- templates/post_detail.html -->
{% extends 'base.html' %}

{% block content %}
    <div class="post-entry">
        <h2>{{ post.title }}</h2>
        <p>{{ post.body }}</p>
    </div>

{% endblock content %}
```

---

В верхней части мы указываем, что этот шаблон наследуется от `base.html`. Затем отобразите `title` и `body` из нашего объекта контекста, который `DetailView` делает доступным как `post`. Лично я нашел именование объектов контекста в общих представлениях чрезвычайно запутанным при первом изучении Django. Поскольку наш объект контекста из подробного представления является либо вашим именем модели `post`, либо объектом, мы также можем обновить наш шаблон следующим образом, и он будет работать точно так же.

**Code**

---

```
<!-- templates/post_detail.html -->

{% extends 'base.html' %}


{% block content %}

    <div class="post-entry">
        <h2>{{ object.title }}</h2>
        <p>{{ object.body }}</p>
    </div>

{% endblock content %}
```

---

Если вы обнаружите, что использование `post` или `object` сбивает с толку, мы также можем явно задать имя объекта контекста в нашем представлении. Поэтому, если бы мы хотели назвать его `anything_you_want`, а затем использовать его в шаблоне, код выглядел бы следующим образом, и он работал бы так же.

**Code**

---

```
# blog/views.py

...

class BlogDetailView(DetailView):
    model = Post
    template_name = 'post_detail.html'
    context_object_name = 'anything_you_want'
```

---

**Code**

---

```
<!-- templates/post_detail.html -->

{% extends 'base.html' %}


{% block content %}

    <div class="post-entry">
        <h2>{{ anything_you_want.title }}</h2>
        <p>{{ anything_you_want.body }}</p>
    </div>

{% endblock content %}
```

---

"Волшебное" именование объекта контекста-это цена, которую вы платите за простоту и удобство использования общих представлений. Они великолепны, если вы знаете, что они делают, но их трудно настроить, если вы хотите другого поведения. Хорошо, что будет дальше? Как насчет добавления нового URLConf для нашего представления, который мы можем сделать следующим образом.

**Code**

---

```
# blog/urls.py

from django.urls import path

from . import views

urlpatterns = [
    path('', views.BlogListView.as_view(), name='home'),
    path('post/<int:pk>/', views.BlogDetailView.as_view(), name='post_detail'),
]
```

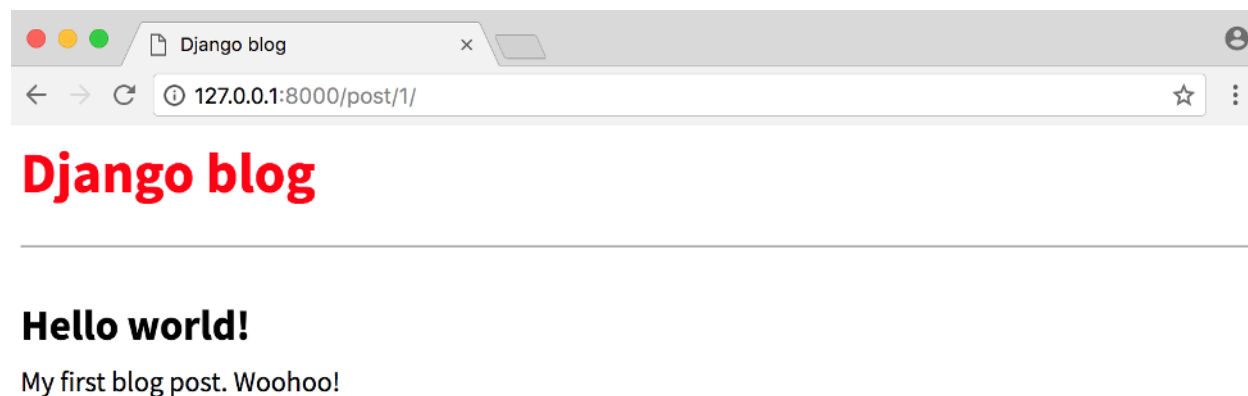
---

Все записи в блоге будут начинаться с `post/`. Далее идет первичный ключ для нашей записи `post`, которая будет представлена в виде целого числа `<int: pk>`. Вероятно вы спросите что это за первичный ключ? Django автоматически добавляет автоматически увеличивающийся первичный ключ к нашим моделям баз данных. Таким образом, как только мы объявили название поля, `author` и `body` в нашей модели `Post`, Django под капотом также добавил еще одно поле под названием `id`, который является нашим первичным ключом. Мы можем получить к нему доступ как `id` или `pk`.

Этот `pk` для нашего первого поста "Привет, Мир" равен 1. Для второго поста это 2. и так далее. Поэтому, когда мы переходим на страницу отдельной записи для нашего первого сообщения, мы можем ожидать, что его `url` шаблон будет `post/1`.

Примечание: понимание того, как первичные ключи работают с `DetailView` является очень распространенной путаницей для начинающих. Стоит перечитать предыдущие два абзаца несколько раз, если не понимаете. С практикой это станет второй натурой.

Если теперь запустить сервер с `python manage.py runserver` и перейти непосредственно к <http://127.0.0.1:8000/post/1/> вы увидите специальную страницу для нашего первого поста в блоге.



### Blog post one detail

Ура! Вы также можете перейти к <http://127.0.0.1:8000/post/2> посмотреть вторую запись. Чтобы сделать нашу жизнь проще, мы должны обновить ссылку на главной странице, чтобы мы могли напрямую получить доступ к отдельным сообщениям в блоге оттуда. В настоящее время в `home.html` наша ссылка пуста: `<a href="">`. Обновите его до `<a href="{% url 'post_detail' post.pk %}">`.

### Code

---

```
<!-- templates/home.html -->

{% extends 'base.html' %}


{% block content %}
    {% for post in object_list %}
        <div class="post-entry">
            <h2><a href="{% url 'post_detail' post.pk %}">{{ post.title }}</a></h2>
            <p>{{ post.body }}</p>
        </div>
    {% endfor %}
{% endblock content %}
```

---

Мы начинаем с того, что говорим нашему шаблону Django, что мы хотим ссылаться на URLConf, используя код `{%url ... %}`. Какой URL? Тот, который называется `post_detail`, который мы назвали `BlogDetailView` в нашей url-конфигурации всего минуту назад. Если мы посмотрим на `post_detail` в нашем URLConf, мы увидим, что он ожидает, что будет передан аргумент `pk`, представляющий первичный ключ для записи блога. К счастью, Django уже создал и включил это поле `pk` в наш объект `post`. Мы передаем его в URLConf, добавляя его в шаблон как `post.pk`.

Чтобы убедиться, что все работает, обновите главную страницу <http://127.0.0.1:8000/> и нажмите на название каждого поста в блоге, чтобы подтвердить работу новых ссылок.

## Тесты

Сейчас нам нужно протестировать нашу модель и представления. Мы хотим убедиться, что модель `Post` работает должным образом, включая ее представление. И мы хотим протестировать как `ListView`, так и `DetailView`.

Вот как выглядят примеры тестов в файле `blog/tests.py`.

#### Code

---

```
# blog/tests.py

from django.contrib.auth import get_user_model
from django.test import Client, TestCase
from django.urls import reverse


from .models import Post


class BlogTests(TestCase):

    def setUp(self):
        self.user = get_user_model().objects.create_user(
            username='testuser',
            email='test@email.com',
            password='secret'
        )

        self.post = Post.objects.create(
            title='A good title',
            body='Nice body content',
            author=self.user,
        )

    def test_string_representation(self):
        post = Post(title='A sample title')
        self.assertEqual(str(post), post.title)
```



```
def test_post_content(self):
    self.assertEqual(f'{self.post.title}', 'A good title')
    self.assertEqual(f'{self.post.author}', 'testuser')
    self.assertEqual(f'{self.post.body}', 'Nice body content')

def test_post_list_view(self):
    response = self.client.get(reverse('home'))
    self.assertEqual(response.status_code, 200)
    self.assertContains(response, 'Nice body content')
    self.assertTemplateUsed(response, 'home.html')

def test_post_detail_view(self):
    response = self.client.get('/post/1/')
    no_response = self.client.get('/post/100000/')
    self.assertEqual(response.status_code, 200)
    self.assertEqual(no_response.status_code, 404)
    self.assertContains(response, 'A good title')
    self.assertTemplateUsed(response, 'post_detail.html')
```

---

В этих тестах много нового, поэтому мы будем проходить их медленно. Вверху импортируем `get_user_model` для ссылки на активного пользователя. Мы импортируем `TestCase`, который мы видели раньше, а также `Client` (), который является новым и используется в качестве фиктивного веб-браузера для имитации GET и POST запросов на URL. Другими словами, всякий раз, когда вы тестируете представления, вы должны использовать `Client()`.

В нашем методе `setUp` мы добавляем образец поста в блоге для тестирования, а затем подтверждаем, что его строковое представление и содержимое верны. Затем мы используем `test_post_list_view`, чтобы подтвердить, что наш сайт возвращает код статуса HTTP 200, содержит наш текст, и использует правильный шаблон `home.html`.

Наконец `test_post_detail_view` проверяет, что наша страница работает должным образом и что неправильная страница возвращает 404. Всегда хорошо проверить, что что-то существует, и что что-то неправильное не существует в ваших тестах.

Продолжайте и выполните эти тесты сейчас. Они все должны быть пройдены

### Command Line

---

```
(testy) $ python manage.py test
```

---

## Git

Сейчас самое время для нашего первого `git commit`. Сначала инициализируйте наш каталог.

---

```
(testy) $ git init
```

---

Затем просмотрите все содержимое, которое мы добавили, проверив состояние. Добавьте все новые файлы. И сделайте наш первый `commit`.

### Command Line

---

```
(testy) $ git status
(testy) $ git add -A
(testy) $ git commit -m 'initial commit'
```

---

## Conclusion

Мы создаем простое приложение блог с нуля! С помощью администратора Django мы можем создавать, редактировать или удалять контент. И мы использовали `DetailView` впервые создав детальное отдельное представление каждого поста блога.

В следующем разделе **Глава 6: Приложение блога с формами** мы добавим формы, чтобы нам вообще не пришлось использовать администратора Django для этих изменений.

## Глава 6: Формы

В этой главе мы продолжим работу над нашим приложением блога из главы 5, добавив формы, чтобы пользователь мог создавать, редактировать или удалять любую из своих записей блога.

### Формы

Формы очень распространены и очень сложны для правильной реализации. Каждый раз, когда вы принимаете пользовательский ввод, возникают проблемы безопасности (XSS Атаки), требуется правильная обработка ошибок, а также вопросы пользовательского интерфейса о том, как предупредить пользователя о проблемах с формой. Не говоря уже о необходимости перенаправления при успехе.

К счастью для нас, встроенные формы Django абстрагируют большую часть сложности и предоставляют богатый набор инструментов для обработки распространенных случаев использования, работающих с формами.

Чтобы начать, обновите наш базовый шаблон, чтобы отобразить ссылку на страницу для ввода новых сообщений в блоге. Он примет форму `<a href= "{% url 'post_new'%} " ></a>`, где `post_new`-это имя нашего URL.

Обновленный файл должен выглядеть следующим образом:

**Code**

---

```
<!-- templates/base.html -->
{% load staticfiles %}

<html>
  <head>
    <title>Django blog</title>
    <link href="https://fonts.googleapis.com/css?family=Source+Sans+Pro:400" rel\
="stylesheet">
    <link rel="stylesheet" href="{% static 'css/base.css' %}">
  </head>
  <body>
    <div class="container">
      <header>
        <div class="nav-left">
          <h1><a href="/">Django blog</a></h1>
        </div>
        <div class="nav-right">
          <a href="{% url 'post_new' %}">+ New Blog Post</a>
        </div>
      </header>
      {% block content %}
      {% endblock content %}
    </div>
  </body>
</html>
```

---

Давайте добавим новый URLConf для post\_new:

**Code**

---

```
# blog/urls.py

from django.urls import path

from . import views

urlpatterns = [
    path('', views.BlogListView.as_view(), name='home'),
    path('post/<int:pk>/', views.BlogDetailView.as_view(), name='post_detail'),
    path('post/new/', views.BlogCreateView.as_view(), name='post_new'),
]
```

---

Ваш url будет начинаться с post / new/, представление называется Blog Create View, и url будет называться post\_new. Просто, правда?

Теперь давайте создадим наше представление, импортировав новый универсальный класс под названием Create View, а затем создадим его подкласс для создания нового представления с именем BlogCreateView.

**Code**

---

```
# blog/views.py

from django.views.generic import ListView, DetailView
from django.views.generic.edit import CreateView
from . models import Post

class BlogListView(ListView):
    model = Post
    template_name = 'home.html'
```

```
class BlogDetailView(DetailView):  
    model = Post  
    template_name = 'post_detail.html'
```

```
class BlogCreateView(CreateView):  
    model = Post  
    template_name = 'post_new.html'  
    fields = '__all__'
```

---

В BlogCreateView мы указываем нашу модель базы данных Post, имя нашего шаблона post\_new.html и все поля с '\_\_all\_\_', так как у нас только два: title и author. Последний шаг - создать наш шаблон, который мы назовем post\_new.html.

### Command Line

---

```
(blog) $ touch templates/post_new.html
```

---

А затем добавьте следующий код:

**Code**

---

```
<!-- templates/post_new.html -->

{% extends 'base.html' %}


{% block content %}

    <h1>New post</h1>

    <form action="" method="post">{% csrf_token %}

        {{ form.as_p }}

        <input type="submit" value="Save" />

    </form>

{% endblock %}
```

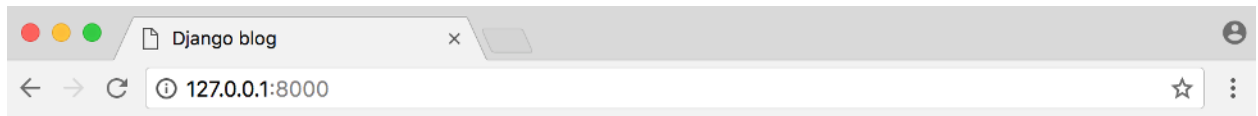
---

Давайте разберемся с тем, что мы сделали:

- В верхней строке мы наследуем от нашего базового шаблона.
- Используются HTML-теги `<form>` с методом POST, так как мы отправляем данные. Если бы мы получали данные из формы, например, в окне поиска, мы бы использовали GET.
- Добавлен `{% csrf_token %}`, который предоставляется Django для защиты нашей формы от атак межсайтового скриптинга. **Вы должны использовать его для всех ваших форм в Django.**
- Затем для вывода данных формы используется `{{ form.as_p }}`, который отображает его в тегах параграфа `<p>`.
- Наконец, указываем тип ввода `submit` и присваиваем ему значение "Save".

Для просмотра нашей работы запустите сервер с `python manage.py runserver` и перейдите на домашнюю страницу по адресу <http://127.0.0.1:8000/>.





# Django blog

[+ New Blog Post](#)

## Hello world!

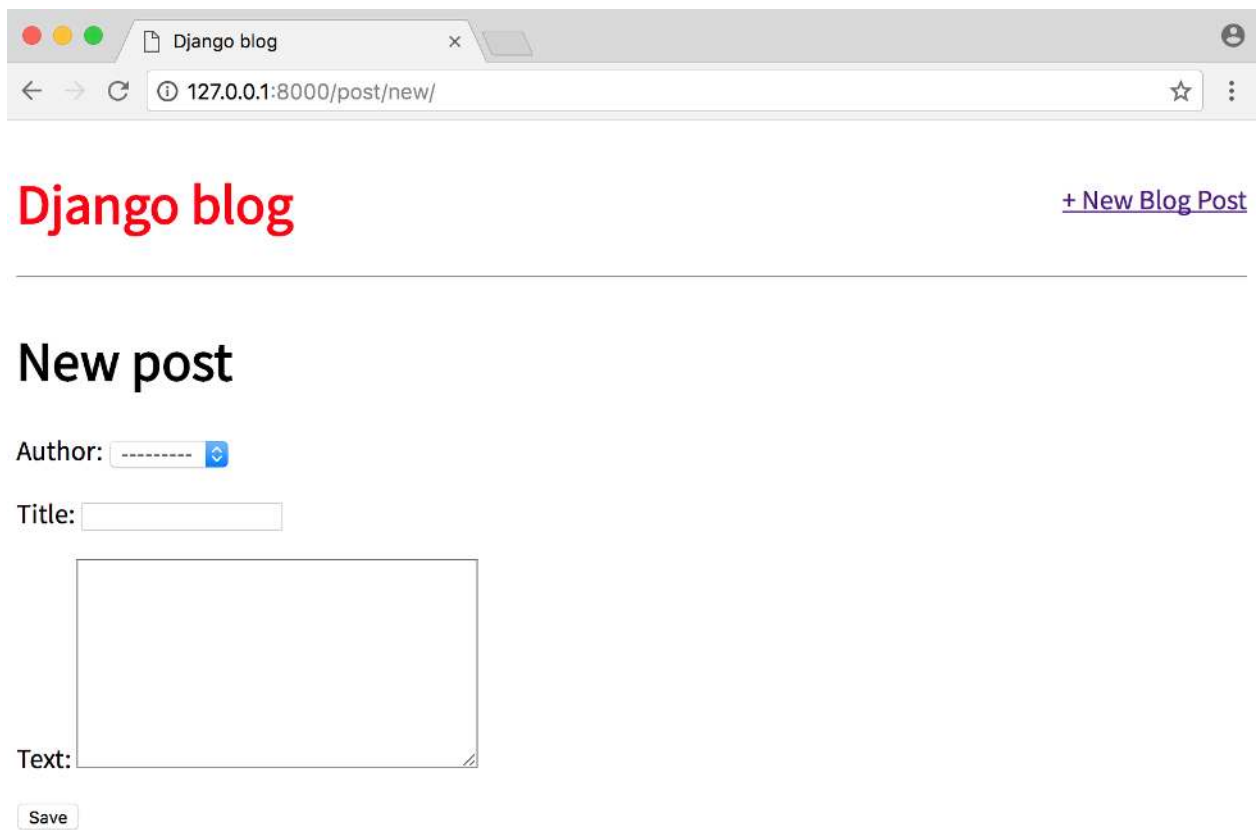
My first blog post. Woohoo!

## Goals today

Learn Django and build a blog application.

### Homepage with New button

Нажмите на нашу ссылку "+ New Blog Post", которое перенаправит вас на: <http://127.0.0.1:8000/post/new/>.




The screenshot shows a web browser window with the title 'Django blog'. The address bar displays '127.0.0.1:8000/post/new/'. The page content includes the 'Django blog' header in red, a '+ New Blog Post' link, and a 'New post' section. The form has three fields: 'Author' with a dropdown menu, 'Title' with a text input, and 'Text' with a large text area. A 'Save' button is located at the bottom of the form.

Django blog [+ New Blog Post](#)

---

## New post

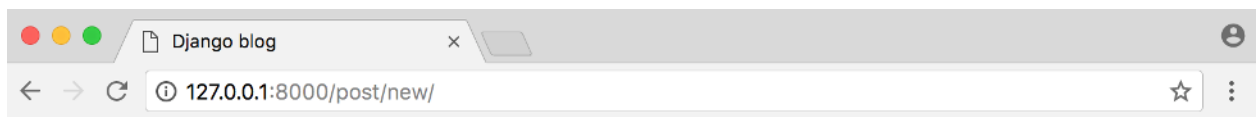
Author:  

Title:

Text:

### Blog new page

Попробуйте создать новую запись в блоге и отправить ее.



# Django blog

[+ New Blog Post](#)

## New post

Author:  

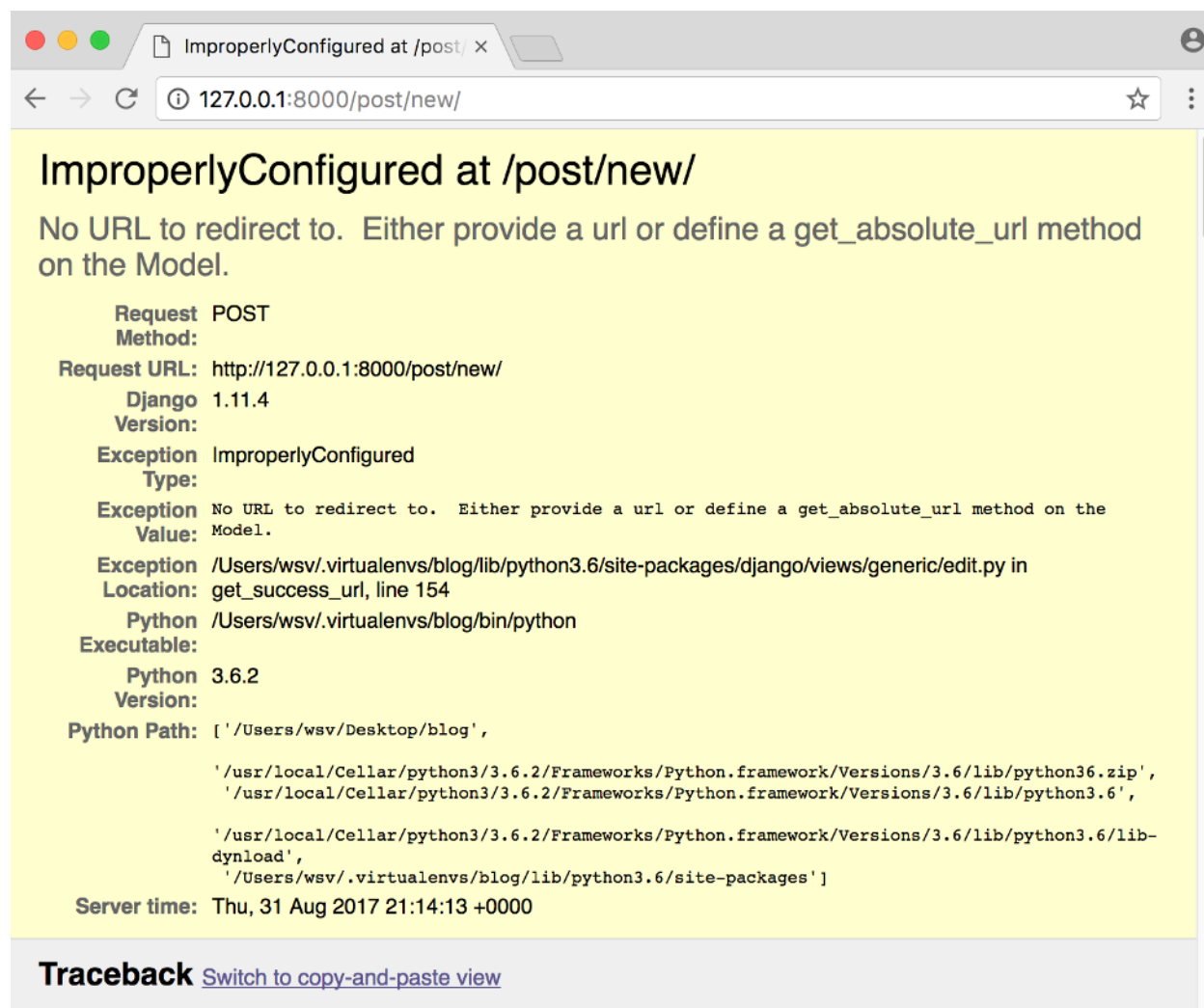
Title:

I wonder if this will work?

Text:

**Blog new page**

Упс! что случилось?



### Blog new page

Сообщение об ошибке Django является весьма полезным. Он жалуется, что мы не указали, куда отправить пользователя после успешной отправки формы. Давайте отправим пользователя на страницу сведений после успешной отправки; таким образом они смогут увидеть свою завершенную публикацию.

Мы можем следовать предложению Django и добавить `get_absolute_url` в нашу модель. Это лучшая практика, которую вы всегда должны делать. Он устанавливает канонический URL-адрес для объекта, поэтому, даже если структура ваших URL-адресов изменится в будущем, ссылка на конкретный объект будет одинаковой. Короче говоря, вы должны добавить `get_absolute_url()` и `__str__()` метод к каждой модели, которую вы пишете.

Откройте models.py файл. Добавьте импорт во второй строке для reverse и новый метод get\_absolute\_url.

### Command Line

---

```
# blog/models.py

from django.db import models
from django.urls import reverse


class Post(models.Model):
    title = models.CharField(max_length=200)
    author = models.ForeignKey(
        'auth.User',
        on_delete=models.CASCADE,
    )
    body = models.TextField()

    def __str__(self):
        return self.title

    def get_absolute_url(self):
        return reverse('post_detail', args=[str(self.id)])
```

---

Reverse - очень удобная утилита Django, позволяющая ссылаться на объект по имени шаблона URL, в данном случае post\_detail. Если вы вспомните наш шаблон URL, то он выглядит следующим образом:

**Code**

```
path('post/<int:pk>/', views BlogDetailView.as_view(), name='post_detail'),
```

Это означает, что для того, чтобы этот маршрут работал, мы должны также передать аргумент с ключом `pk` или первичным ключом объекта. Запутанно что `pk` и `id` взаимозаменяемы в Django, хотя документация Django рекомендуют использовать `self.id` с параметром `get_absolute_url`. Поэтому мы сообщаем Django, что конечным местоположением записи `Post` является ее представление `post_detail`, которое представляет собой `posts / <int: pk> /`, поэтому маршрут для первой созданной нами записи будет находиться в `posts / 1`.


Попробуйте создать новую запись в блоге еще раз по адресу <http://127.0.0.1:8000/post/new/>, и в случае успеха вы будете перенаправлены на страницу подробного просмотра, где размещена запись.



## Django blog

[+ New Blog Post](#)

### New post

Author:  Title: 

Text:

Новая страница блога с вводом

Вы также заметите, что наш предыдущий пост в блоге тоже есть. Он был успешно отправлен в базу данных, но Django не знал, как перенаправить нас после этого.



## Django blog

[+ New Blog Post](#)

### Hello world!

My first blog post. Woohoo!

### Goals today

Learn Django and build a blog application.

### 3rd post

I wonder if this will work?

### Is this form working?

Yes it is!

#### Blog homepage with four posts

Хотя мы можем обратиться к администратору Django, чтобы удалить нежелательные сообщения, но лучше добавить формы, чтобы пользователь мог обновлять и удалять существующие сообщения непосредственно с сайта.

## Форма обновления

Процесс создания формы обновления, чтобы пользователи могли редактировать сообщения в блоге, должен быть знакомым. Мы снова будем использовать встроенное общее представление на основе классов Django UpdateView и создадим необходимый шаблон, URL и представление.

Для начала добавим новую ссылку на `post_detail.html`, чтобы возможность редактирования сообщения блога появилась на отдельной странице блога.

### Code

---

```
<!-- templates/post_detail.html -->

{% extends 'base.html' %}

{% block content %}

    <div class="post-entry">
        <h2>{{ object.title }}</h2>
        <p>{{ object.body }}</p>
    </div>

    <a href="{% url 'post_edit' post.pk %}">+ Edit Blog Post</a>

{% endblock content %}
```

---

Мы добавили ссылку, используя `<a href> ... </a>` и тег движка шаблонов Django `{% url ...%}`. В нем мы указали целевое имя нашего URL, которое будет называться `post_edit`, а также передали необходимый параметр, который является первичным ключом поста `post.pk`.

Затем мы создаем шаблон для нашей страницы редактирования с именем `post_edit.html`.

### Command Line

---

```
(blog) $ touch templates/post_edit.html
```

---

И добавьте следующий код:



**Code**

---

```
<!-- templates/post_edit.html -->

{% extends 'base.html' %}


{% block content %}

    <h1>Edit post</h1>

    <form action="" method="post">{% csrf_token %}

        {{ form.as_p }}

        <input type="submit" value="Update" />

    </form>

{% endblock %}
```

---

Мы снова используем теги HTML `<form></form>`, `csrf_token` от Django для безопасности, `form.as_p`, чтобы отобразить наши поля формы с тегами абзаца, и, наконец, дать ему значение “Update” на кнопке отправки.

Теперь к нашему представлению нужно импортировать `UpdateView` на второй строке с верху, а затем создать его подкласс в нашем новом представлении `BlogUpdateView`.

**Code**

---

```
# blog/views.py

from django.views.generic import ListView, DetailView
from django.views.generic.edit import CreateView, UpdateView
from . models import Post


class BlogListView(ListView):

    model = Post

    template_name = 'home.html'
```

```
class BlogDetailView(DetailView):  
    model = Post  
    template_name = 'post_detail.html'
```

```
class BlogCreateView(CreateView):  
    model = Post  
    template_name = 'post_new.html'  
    fields = '__all__'
```

```
class BlogUpdateView(UpdateView):  
    model = Post  
    fields = ['title', 'body']  
    template_name = 'post_edit.html'
```

---

Обратите внимание, что в представлении BlogUpdateView мы явно перечисляем поля, которые мы хотим использовать ['title', 'body'], а не "\_\_all\_\_". Это потому, что мы предполагаем, что автор сообщения не меняется; мы хотим, чтобы только заголовок и текст были редактируемыми.

Заключительный этап-это обновление вашего файла urls.py как указано:

**Code**

```
# blog/urls.py

from django.urls import path

from . import views

urlpatterns = [
    path('', views.BlogListView.as_view(), name='home'),
    path('post/<int:pk>/', views.BlogDetailView.as_view(), name='post_detail'),
    path('post/new/', views.BlogCreateView.as_view(), name='post_new'),
    path('post/<int:pk>/edit/',
        views.BlogUpdateView.as_view(), name='post_edit'),
]
```

В верхней части мы добавляем наше представление `BlogUpdateView` в список импортированных представлений, затем создали новый шаблон url для `/post/pk / edit` и дали ему имя `post_edit`.

Теперь, если вы нажмете на запись в блоге, вы увидите нашу новую кнопку редактирования.



## Django blog

[+ New Blog Post](#)

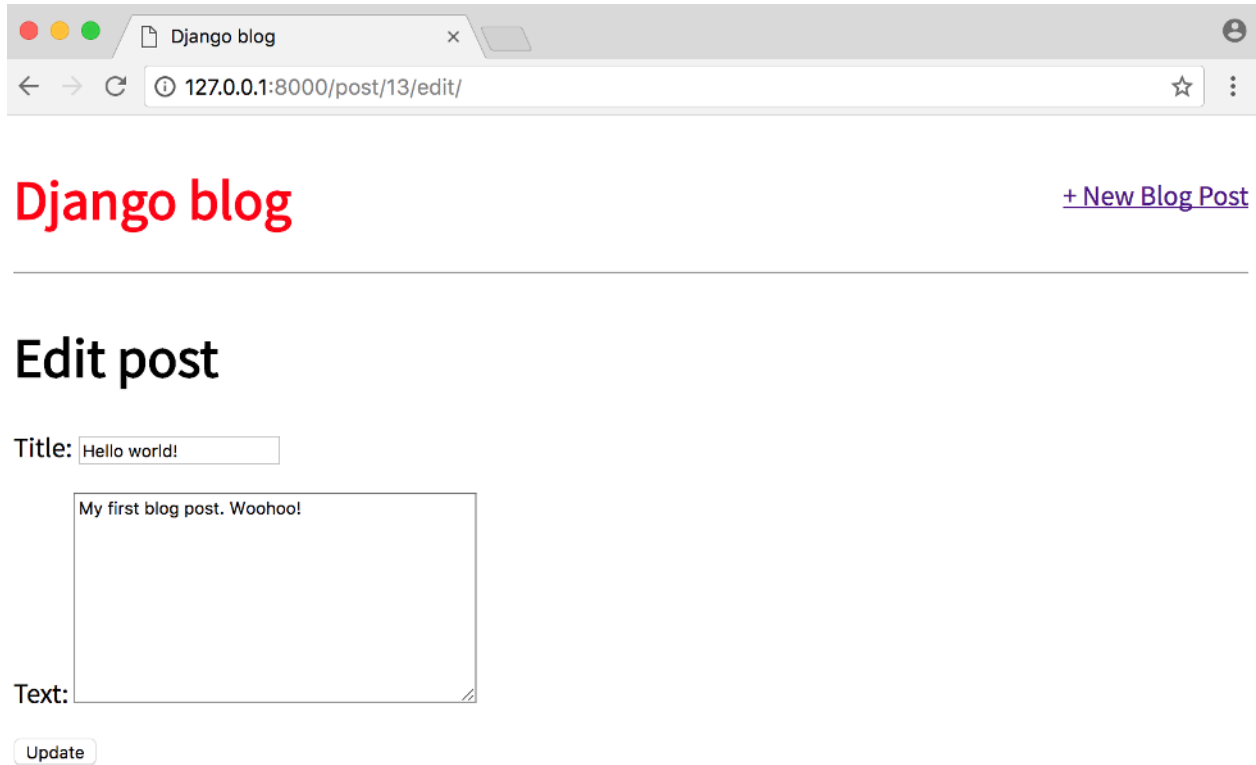
### Hello world!

My first blog post. Woohoo!

[+ Edit Blog Post](#)

Страница блога с кнопкой редактировать

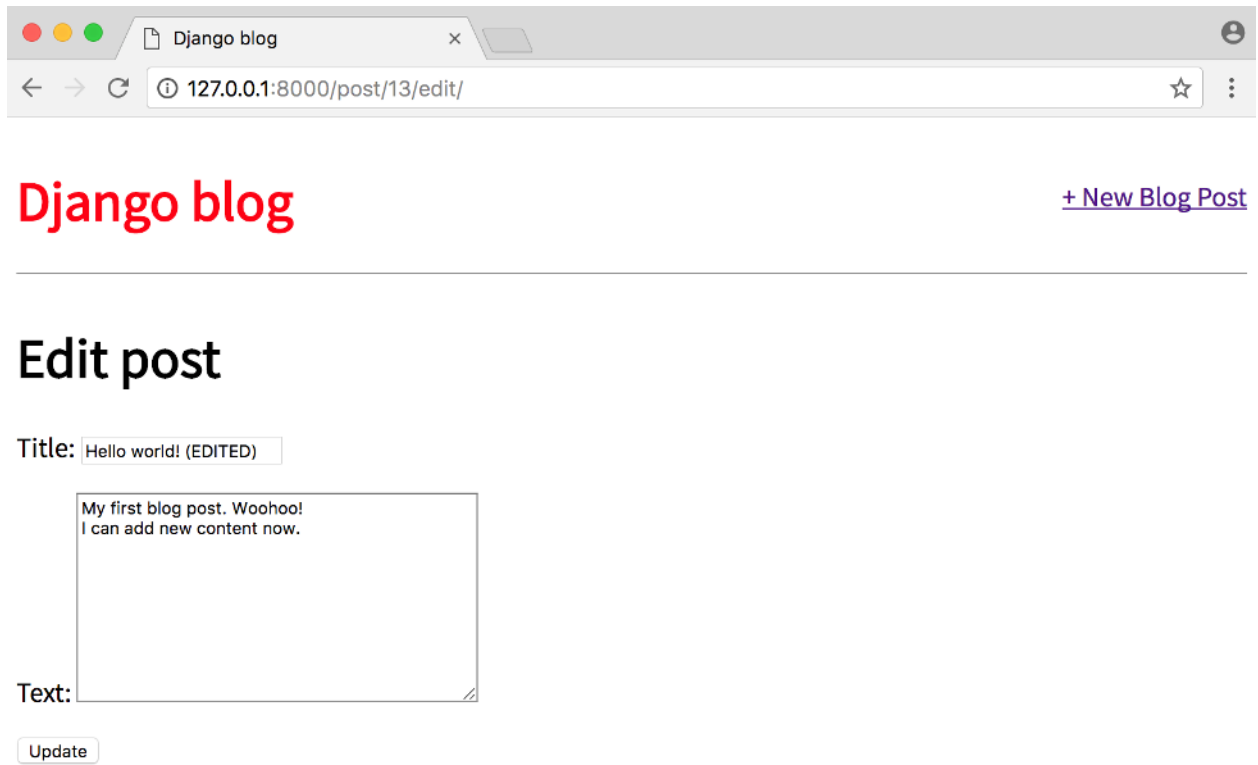
Если вы нажмете на "+ Edit Blog Pos", вы будете перенаправлены на `http://127.0.0.1:8000/post/1/edit/` if это ваш первый пост в блоге.



The screenshot shows a web browser window with the title "Django blog". The address bar displays the URL `127.0.0.1:8000/post/13/edit/`. The page content includes the "Django blog" header in red, a link "+ New Blog Post" in purple, and a section titled "Edit post". Below the title, there is a text input field containing "Hello world!". Below that is a large text area containing "My first blog post. Woohoo!". To the left of the text area is the label "Text:". At the bottom left is an "Update" button.

#### Страница редактирования

Обратите внимание, что форма предварительно заполняется существующими данными нашей базы данных для публикации. Давайте внесем изменения...



Django blog [+ New Blog Post](#)

---

## Edit post

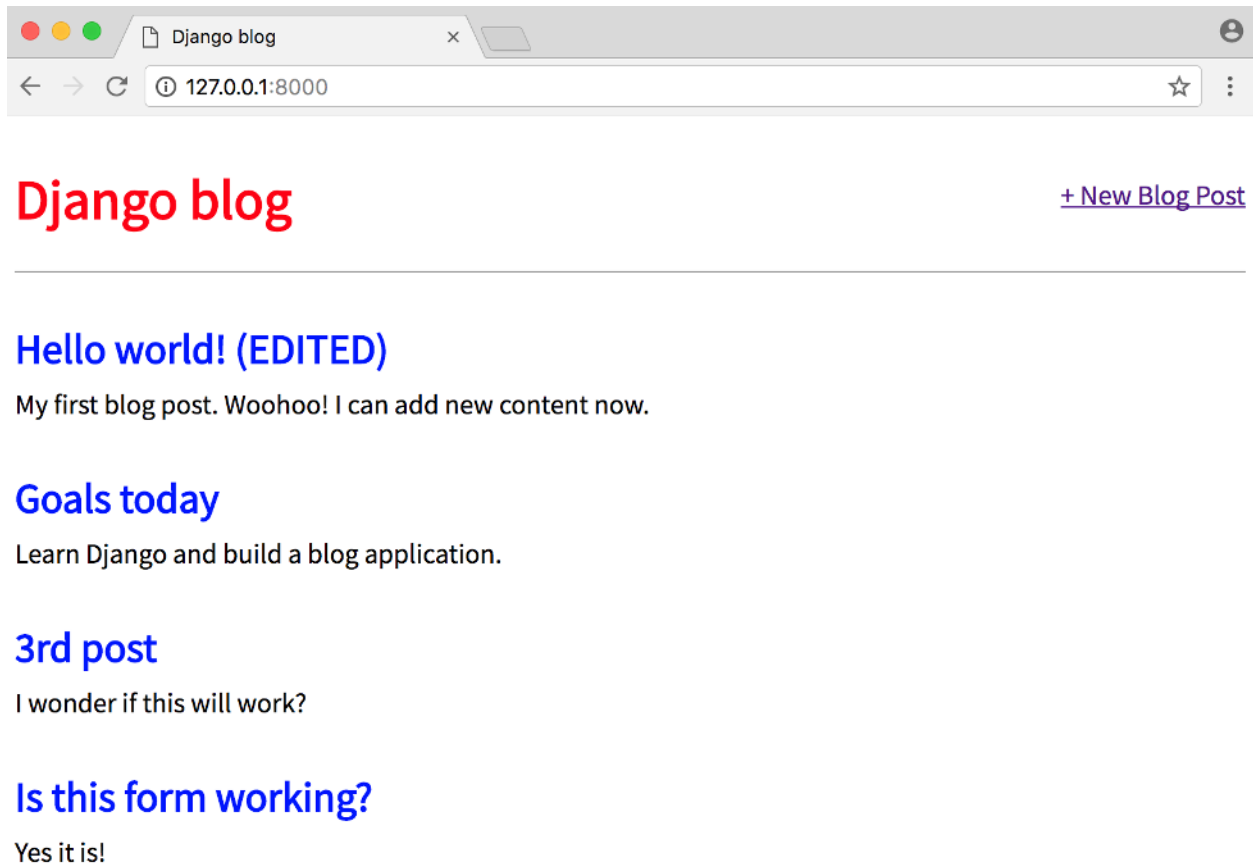
Title:

Text: 

My first blog post. Woohoo!  
I can add new content now.

### Blog edit page

И после нажатия кнопки "Update" мы перенаправляемся в подробный вид поста, где вы можете увидеть изменения. Это из-за нашей установки `get_absolute_url`. Перейдите на главную страницу, и вы увидите изменения рядом со всеми остальными записями.



Домашняя страница блога с отредактированным сообщением

## Удаление View

Процесс создания формы для удаления записей блога очень похож на процесс обновления записи. Мы будем использовать еще одно общее представление на основе классов, `DeleteView`, для создания представления, url-адреса и шаблона функциональности. Давайте начнем с добавления ссылки для удаления записей блога на Вашей странице блога, `post_detail.html`.

**Code**

---

```
<!-- templates/post_detail.html -->

{% extends 'base.html' %}


{% block content %}

    <div class="post-entry">
        <h2>{{ object.title }}</h2>
        <p>{{ object.body }}</p>
    </div>


    <p><a href="{% url 'post_edit' post.pk %}">+ Edit Blog Post</a></p>
    <p><a href="{% url 'post_delete' post.pk %}">+ Delete Blog Post</a></p>

{% endblock content %}
```

---

Затем создайте новый файл для нашего шаблона удаления страницы. Сначала выйдите из локального сервера Control-c, а затем введите следующую команду:

**Command Line**

---

```
(blog) $ touch templates/post_delete.html
```

---

И заполните его этим кодом:

**Code**

---

```
<!-- templates/post_delete.html -->

{% extends 'base.html' %}

{% block content %}
    <h1>Delete post</h1>
    <form action="" method="post">{% csrf_token %}
        <p>Are you sure you want to delete "{{ post.title }}"?</p>
        <input type="submit" value="Confirm" />
    </form>
{% endblock %}
```

---

Обратите внимание, что мы используем `post.title` для отображения заголовка нашего блога. Мы также могли бы просто использовать `object.title`, поскольку он также предоставляется `DetailView`.

Теперь обновите наш файл `views.py`, импортировав `DeleteView` и `reverse_lazy` вверху, затем создайте новое представление, которое подкласс `DeleteView`.

**Code**

---

```
# blog/views.py

from django.views.generic import ListView, DetailView
from django.views.generic.edit import CreateView, UpdateView, DeleteView
from django.urls import reverse_lazy

from . models import Post

class BlogListView(ListView):
    model = Post
```



```
template_name = 'home.html'
```

```
class BlogDetailView(DetailView):  
    model = Post  
    template_name = 'post_detail.html'
```

```
class BlogCreateView(CreateView):  
    model = Post  
    template_name = 'post_new.html'  
    fields = '__all__'
```

```
class BlogUpdateView(UpdateView):  
    model = Post  
    fields = ['title', 'body']  
    template_name = 'post_edit.html'
```

```
class BlogDeleteView>DeleteView):  
    model = Post  
    template_name = 'post_delete.html'  
    success_url = reverse_lazy('home')
```

---

Мы используем `reverse_lazy` вместо просто `reverse`, чтобы он не выполнял перенаправление URL, пока наше представление не завершило удаление сообщения в блоге.

Наконец, добавьте URL, импортировав наш `BlogDeleteView` и добавив новый шаблон:

**Code**

---

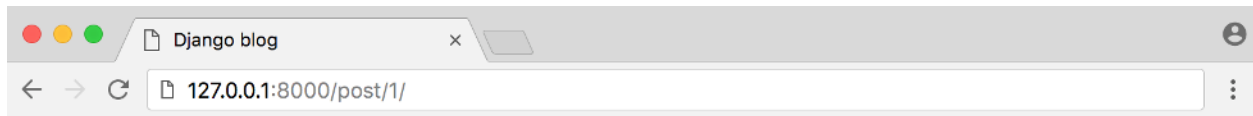
```
# blog/urls.py
from django.urls import path

from . import views

urlpatterns = [
    path('', views.BlogListView.as_view(), name='home'),
    path('post/<int:pk>/', views.BlogDetailView.as_view(), name='post_detail'),
    path('post/new/', views.BlogCreateView.as_view(), name='post_new'),
    path('post/<int:pk>/edit/',
        views.BlogUpdateView.as_view(), name='post_edit'),
    path('post/<int:pk>/delete/',
        views.BlogDeleteView.as_view(), name='post_delete'),
]
```

---

Если вы снова запустите сервер `python manage.py runserver` и обновите страницу отдельного сообщения, вы увидите ссылку «Удалить сообщение в блоге».



## Django blog

[+ New Blog Post](#)

### Hello world! (EDITED)

My first blog post. Woohoo! I can add new content now.

[+ Edit Blog Post](#)[+ Delete Blog Post](#)

#### Blog delete post

При нажатии на ссылку мы попадаем на страницу удаления записи блога, где отображается название записи блога.



## Django blog

[+ New Blog Post](#)

### Delete post

Are you sure you want to delete "Hello world! (EDITED)"?

#### страница удалить запись

Если вы нажмете на кнопку “Confirm”, то будете перенаправлены на домашнюю страницу, где пост был удален!



## Django blog

[+ New Blog Post](#)

### Goals today

Learn Django and build a blog application.

### 3rd post

I wonder if this will work?

### Is this form working?

Yes it is!

Домашняя страница с удаленным сообщением

Так это работает!

## Tests

Время для тестов, чтобы убедиться, что все работает сейчас–и в будущем–как и ожидалось. Мы добавили метод `get_absolute_url` в нашу модель и новые представления для создания, обновления и редактирования записей. Это означает, что нам нужно четыре новых теста:

- `def test_get_absolute_url`
- `def test_post_create_view`
- `def test_post_update_view`
- `def test_post_delete_view`

Обновите существующий `tests.py` файл следующим образом.

**Code**

---

```
# blog/tests.py

from django.contrib.auth import get_user_model
from django.test import Client, TestCase
from django.urls import reverse


from .models import Post


class BlogTests(TestCase):

    def setUp(self):
        self.user = get_user_model().objects.create_user(
            username='testuser',
            email='test@email.com',
            password='secret'
        )

        self.post = Post.objects.create(
            title='A good title',
            body='Nice body content',
            author=self.user,
        )

    def test_string_representation(self):
        post = Post(title='A sample title')
        self.assertEqual(str(post), post.title)
```

```
def test_get_absolute_url(self):
    self.assertEqual(self.post.get_absolute_url(), '/post/1/')

def test_post_content(self):
    self.assertEqual(f'{self.post.title}', 'A good title')
    self.assertEqual(f'{self.post.author}', 'testuser')
    self.assertEqual(f'{self.post.body}', 'Nice body content')

def test_post_list_view(self):
    response = self.client.get(reverse('home'))
    self.assertEqual(response.status_code, 200)
    self.assertContains(response, 'Nice body content')
    self.assertTemplateUsed(response, 'home.html')

def test_post_detail_view(self):
    response = self.client.get('/post/1/')
    no_response = self.client.get('/post/100000/')
    self.assertEqual(response.status_code, 200)
    self.assertEqual(no_response.status_code, 404)
    self.assertContains(response, 'A good title')
    self.assertTemplateUsed(response, 'post_detail.html')

def test_post_create_view(self):
    response = self.client.post(reverse('post_new'), {
        'title': 'New title',
        'body': 'New text',
        'author': self.user,
    })
```

```
self.assertEqual(response.status_code, 200)

self.assertContains(response, 'New title')

self.assertContains(response, 'New text')


def test_post_update_view(self):
    response = self.client.post(reverse('post_edit', args='1'), {
        'title': 'Updated title',
        'body': 'Updated text',
    })
    self.assertEqual(response.status_code, 302)


def test_post_delete_view(self):
    response = self.client.get(
        reverse('post_delete', args='1'))
    self.assertEqual(response.status_code, 200)
```

---

Мы ожидаем, что URL нашего теста будет в `post / 1 /`, поскольку там только один пост, а `1` - его первичный ключ, который Django автоматически добавляет для нас. Чтобы протестировать создание представления, мы создаем новый ответ и затем проверяем, что ответ проходит (код состояния 200) и содержит наш новый заголовок и основной текст. Для просмотра обновлений мы обращаемся к первому сообщению, в котором в качестве единственного аргумента указан `pk`, равный `1`, и подтверждаем, что оно приводит к перенаправлению 302. Наконец, мы проверяем наше представление удаления, подтверждая, что если мы удаляем сообщение, код состояния 200 успех.

Всегда можно добавить больше тестов, но это, по крайней мере, охватывает все наши новые функциональные возможности.

## Заключение

В небольшом количестве кода мы создали приложение blog которое позволяет создавать, читать, обновлять и удалять сообщения в блоге. Эта основная функциональность известна под аббревиатурой CRUD: Create-Read-Update-Delete. Хотя есть несколько способов достичь этой же функциональности-мы могли бы использовать представления на основе функций или написать наши собственные представления на основе классов-мы продемонстрировали, как мало кода требуется в Django, чтобы это произошло.

В следующей главе мы добавим учетные записи пользователей, а также функции входа, выхода и регистрации.