



# DJANGO

---

*for*

---

# APIs

Build web APIs with Python & Django

WILLIAM S. VINCENT

## Джанг о для API

Создавайте веб-API с помощью Python и Django.

Уильям С. Винсент

Эта книга продается на <http://leanpub.com/djangoforapis>.

Эта версия была опубликована 24 марта 2022 г.



Это [линкаб](#) книга. Leanpub предоставляет авторам и издателям возможность использовать процесс Lean Publishing. **Бережливое издательское дело** — это публикация незавершенной электронной книги с использованием облегченных инструментов и множества интераций для получения отзывов читателей, поворота к творческому процессу, пока у вас не будет нужной книги, и наращивания обработов после того, как вы это сделаете.

© 2018 - 2022 Уильям С. Винсент

Так же Уильям С. Винсент

Джанг о для начинаящих

Джанг о для професионалов

# Содержание

Вступление	1
Почему API	1
Джанго REST Framework	2
Предпосылки	3
Почему эта книга	3
Заключение	4
 Глава 1: Первоначальная настройка	5
Командная строка	5
Команды оболочки	7
Установите Python 3 в Windows	10
Установите Python 3 на Mac	11
Интерактивный режим Python	12
Виртуальные среды	13
Установите Django и Django REST Framework	15
Текстовые редакторы	17
Установить Гит	18
Заключение	20
 Глава 2. Веб-API	21
Всемирная сеть	21
URL-адреса	22
Пакет интернет-протокола	23
Глаголы HTTP	24
Конечные точки	24

## СОДЕРЖАНИЕ

HTTP	25
Коды состояния	27
Без разданс тво	28
ОтдыХ	28
Заключение	29
Г лава 3: Веб-сайт библиотеки	30
Традиционный Джанг о	30
Первое приложение	33
Модели	35
Администратор	36
Просмотры	39
URL-адреса	40
Шаблоны	41
Тесты	43
Гит	45
Заключение	46
Г лава 4: API библиотеки	47
Джанг о REST Framework	47
URL-адреса	49
Просмотры	50
Сериализаторы	51
Документальный для просмотра API	52
Тесты	54
Развертывание	56
Статические файлы	58
Контрольный список развертывания	60
Гитхаб	62
Героку	62
Заключение	64
Г лава 5: Todo API	66

## СОДЕРЖАНИЕ

Односоставничные приложения(SPA)	66
Первоначальная	67
настройка.gitignore	68
Модели	69
Тесты	72
Джанго REST Framework	74
URL-адреса	75
Сериализаторы	76
Просмотры	78
Документальный для просмотра API	79
API-тесты	80
КОРС	82
CSRF	84
Развертывание внутреннего API	84
Заключение	89
 Глава 6: API блога	90
Первоначальная	90
настройка.gitignore	92
Пользовательская модель пользователя	92
Приложение «Сообщения»	97
Почтовая модель	98
Тесты	102
Джанго REST Framework	103
URL-адреса	104
Сериализаторы	106
Просмотры	107
Документальный для просмотра API	108
КОРС	112
Заключение	114
 Глава 7: Разрешения	115
Разрешения на уровне проекта	115

**СОДЕРЖАНИЕ**

Создать новых пользователей	117
Добавить вход и выход од	121
Разрешения на уровне просмотра	123
Пользовательские разрешения	125
Заключение	130
Глава 8: Аутентификация пользователя	131
Базовая аутентификация	131
Аутентификация сеанса	133
Токен Аутентификация	134
Аутентификация по умолчанию	136
Внедрение аутентификации по токену	137
Конечные	140
точки dj-rest-auth	140
Регистрация пользователя	146
Токены	149
Заключение	155
Глава 9. Наборы представлений и маршрутизаторы	156
Пользовательские конечные точки	156
Наборы представлений	161
Маршрутизаторы	162
Разрешения	164
Заключение	166
Глава 10: Схемы и документация	168
Схема	169
Динамическая схема	171
Документация	172
Заключение	175
Глава 11: Развёртывание в производственной среде	177
Переменные среды	177

## СОДЕРЖАНИЕ

ОТЛАДКА И SECRET_KEY	179
РАЗРЕШЕННЫЕ ХОСТИ	181
БАЗЫ ДАННЫХ	182
Статические файлы	183
Требования Pyscopg и	184
Gunicorn.txt	185
Procfile и runtime.txt	186
Контрольный список развертывания	187
Развертывание Героку	187
 Заключение	191
Расширенные темы	191
Следующие шаги	192
Спасибо	192

## Вс тупление

В э той книг е вы узнаете, как с оздавать нес колько веб-API ворас таафей с ложности с ис пользованием Django и Django REST Framework. Django —очень популярная веб-инфраструктура на ос нове Python, которая решает с ложные задачи с оздания веб-сайта аутентификаци я подключение к базе данных , логика, безопасность и т. д. С уществую та же тысячи с торонних пакетов, которые добавляю функц иональность с амому Django, наиболее известным из которых является Django REST Framework, который позволяет разработчикам преобразовать любой сущес твующий проект Django в мощный веб-API.

Django и Django REST Framework ис пользова ся крупнейшими технологиями компаний мира, включая Instagram, Mozilla и Pinterest. Но они также х орошо подходит для начинаний или побочных проектов вых одног одня потому что подх од Django «вклjuченные багарей» маскирует большую часть базовой с ложности, обес печивая быструю безопасную разработку. К конц у э той книг и вы с можете с оздавать готовые к э ксплуатации веб-API с небольшим объемом кода и еще меньшими затратами времени.

время

## Почему API

API (интерфейс прикладног о прог раммирования) —э то с окрашенный с пос об описания того, как два компьютера напрямую взаимодействуют друг с другом. Для веб-API, которые существуют во вс емирной паутине, доминирующ им арх итектурным шаблоном является яREST (передача репрезентативног о состояния), и он будет подробно расмотрен позже в э той книг е.

Е ще в 2005 г оду, когда Django был впервые выпущен, большинс тво веб-сайтов состояло из одной большой монолитной кодовой базы. Серверная часть моделей баз данных , представлений и URL-адресов была объединена с внешней шаблонами для управления презентаци онным слоем каждой веб-страницы.

Но в наши дни для веб-сайтов г ораздо чаще ис пользова ся подход API-first, формально отделяющий серверную часть от внешней. Это позволяет веб-сайту ис пользовать с пециальную интерфейсную библиотеку JavaScript, такую как React или Vue, которые были выпущены в 2013 и 2014 г одах с соответс твенно.

Когда в ближайшие годы текущие интерфейсные фреймворки будут заменены еще более новыми, внутренний API может остаться прежним. Серьезной перезаписи не требуется.

Еще одним важным преимуществом является то, что один API может поддерживать несколько интерфейсов, написанных на разных языках и платформах. Учтите, что JavaScript используется для веб-интерфейсов, в то время как для приложений Android требуется язык программирования Java, а для приложений iOS требуется язык программирования Swift. При традиционном монолитном подходе веб-сайт Django не может поддерживать эти различные внешние интерфейсы. Но с внутренним API все три могут взаимодействовать с одним и тем же базовым сервером, частью базы данных!

Распространенные веб-сайты также могут выигрывать от создания внешнего API, который позволяет сторонним разработчикам создавать свои собственные приложения для iOS или Android. Когда я работал в Quizlet в 2010 году, у нас не было сервисов для разработки собственных приложений для iOS или Android, но у нас был доступный внешний API, который более 30 разработчиков использовали для создания собственных приложений с карточками на основе базы данных Quizlet. Некоторые из этих приложений были загружены более миллиона раз, что побогатило разработчиков и одновременно увеличило охват Quizlet.

Основным недостатком этого подхода, ориентированного на API, является то, что он требует большей настройки, чем традиционное приложение Django. Однако, как мы увидим в этой книге, фантастическая библиотека Django REST Framework избавляет нас от большей части этой сложности.

## ДжангоРEST Framework

Доступны тысячи сторонних приложений, которые расширяют функциональные возможности Django. Вы можете увидеть полный список возможностей поиска на [Django Packages<sup>1</sup>](https://djangopackages.org/), а также кураторский список в [репозитории awesome-django<sup>2</sup>](https://github.com/wsvincent/awesome-django2). Однако среди всех сторонних приложений Django REST Framework, пожалуй, лучшее приложение для Django. Он зрелый, полный функций, настраиваемый, тестируемый и чрезвычайно хорошо документированный. Он также целиком направленно имитирует многие традиционные соглашения Django, что значительно упрощает его изучение. Если вы уже знакомы с Django, то следующим логическим шагом будет изучение Django REST Framework.

<sup>1</sup><https://djangopackages.org/>

<sup>2</sup><https://github.com/wsvincent/awesome-django>

## Предпосылки

Если вы новичок в веб-разработке на Django, я рекомендую начать с моей книги [Django для начинающих](#)<sup>3</sup>.

Первые несколько глав достаточно бесплатно в Интернете и охватывают правильную архитектуру приложения Hello World, приложение Pages и приложение Message Board. Полная версия идет глубже и охватывает веб-сайт блога с формами и учетными записями пользователей, а также готовый к производству сайта газеты, который включает пользовательскую модель пользователя, полный поток аутентификации пользователя, электронные письма, разрешения, развертывание, переменные сюда и многое другое.

Этот фон в традиционном Django важен, поскольку Django REST Framework намеренно имитирует многие соглашения Django. Также рекомендуется, чтобы читатели имели базовые знания о Python. Чтобы по-настоящему освоить Python, нужны годы, но, обладая небольшими знаниями, вы можете сразу погрузиться в него и начать создавать что-то новое.

## Почему эта книга

Я написал эту книгу, потому что для разработчиков, плохо знакомых с Django REST Framework, явно не хватает хороших ресурсов. Предполагается, что все уже знают все об API, HTTP, REST и тому подобном. Мой собственный путь изучения создания веб-API был разочаровывающим... и я уже достаточно хорошо знал Django, чтобы написать о нем книгу! Эта книга — руководство, которое я хотел бы иметь при начале работы с Django REST Framework.

Глава 1 описывает начальную архитектуру установки Python, Django, Git и работу с командной строкой. Глава 2 представляет собой введение в веб-API и протокол HTTP, лежащий в основе всего этого. В главах 3-4 мы смотрим различия между традиционным Django и Django REST Framework, создав веб-сайт библиотеки, преобразовав его в API, добавив тесты и затем развернув его в реальном времени. В главе 5 мы создадим протестируемый развернутый Todo API с списками и подробными конечными точками API. Он также включает совместное использование ресурсов между источниками (CORS).

Глава 6 — это начало с создания готового API для использования в Blog API, в котором используется языком Python. Пользовательская модель и полная функциональность Create-Read-Update-Delete (CRUD). Глава 7 посвящена разрешениям, надлежащим различиям и созданию пользовательского класса разрешений. В главе 8

---

<sup>3</sup><https://djangoforbeginners.com/>

основное внимание уделяется проверке подлинности пользователя и четырьем встроенным методам проверки подлинности. Затем мы добавляем конечные точки для регистрации пользователя, выхода из системы, сброса пароля и подтверждения сброса пароля. В главе 9 расмотрены наборы представлений и маршрутизаторы, встроенные компоненты, которые могут значительно сократить объем кода, необходиимого для стандартных конечных точек API. В главе 10 расмотрены матрицы ячеек и документация, а в главе 11 шагом расматривается производственное развертывание.

Полный исходный код для всех глав можно найти [на Github](#).

## Заключение

Django и Django REST Framework — это мощный и доступный инструмент для создания веб-API. К концу этой книги вы сможете добавлять API-интерфейсы в любые существующие проекты Django или создавать свои собственные с помощью встроенных веб-API с нуля, используя современные передовые методы. Давайте начнем!

---

<sup>4</sup><https://github.com/wsvincent/restapiswithdjango>

## Глава 1: Первоначальная настройка

Если вы уже читали [Django для начинающих 5](#) большая часть этого будет знакома, но есть дополнительные шаги по установке Django REST Framework.

В этой главе рассказывается как правильно настраивать компьютер с Windows или macOS для работы с проектами Django. Мы начнем с обзора командной строки, мощного текстового интерфейса, который разработчики широко используют для установки и настройки программных проектов. Затем мы устанавливаем последнюю версию Python, учимся создавать выделенные виртуальные среды и устанавливаем Django.

В качестве последнего шага мы рассмотрим использование Git для контроля версий и работу с текстовым редактором. К концу этой главы вы создадите свой первый проект Django и Django REST Framework с нуля. В будущем вы сможете создать или изменить любой проект Django всего за несколько нажатий клавиш.

### Командная строка

Командная строка предоставляет собой текстовый интерфейс, в который вводят команды выполнения. Это альтернатива графическому пользовательскому интерфейсу с мышью или пальцами, знакомому большинству пользователей компьютеров. Обычному пользователю компьютера никогда не понадобится использовать командную строку, но разработчикам программного обеспечения это нужно, потому что определенные задачи можно выполнять только с ее помощью. К ним относятся запуск программы, установка программного обеспечения, использование Git для контроля версий или подключение к серверам в облаке. Несколько практиковавшись, большинство разработчиков обнаруживает, что командная строка на самом деле является более быстрым и мощным способом навигации и управления компьютером.

Учитывая минимальный пользовательский интерфейс — густой экран и много курсоров — команда строка пугает новичков. Часто после запуска команды нет обратной связи, и можно с уверенностью сказать о компьютере с помощью одной команды, если вы не будете осторожны: никаких предупреждений не появится. Поэтому командную строку следует использовать с осторожностью. Убедитесь, что вы

несложно копировать и вставлять команды, которые вы найдете в Интернете; полагайтесь только на надежные ресурсы для любой команды, которую вы не полностью понимаете.

На практике для обозначения командной строки используется несолько терминов: интерфейс командной строки (CLI), консоль, терминал, оболочка или приглашение. С точки зрения терминал — это программа, которая открывает новое окно для доступа к командной строке, консоль — это текстовое приложение, оболочка — это программа, которая запускает команды в базовой операционной системе, а подсказка — это место, где находятся команды. набрал и запустил. Поначалу эти термины легкозапутать, но все они, по сути, означают одно и тоже: командная строка — это место, где мы запускаем и выполняем текстовые команды на нашем компьютере.

В Windows встроенный терминал и оболочка называются PowerShell. Чтобы получить к нему доступ, найдите панели задач в нижней части экрана рядом с кнопкой Windows и введите «powershell», чтобы запустить приложение. Откроется новое окно с темно-серым фоном и мигающим курсором после приглашения>. Вот как это выглядит на моем компьютере.

Ракушка

---

PS C:\Пользователи\wsv>

---

Перед приглашением стоит PS, который относится к PowerShell, начальному каталогу С операционной системы Windows, за которым следует каталог Users и текущий пользователь, которым на моих персональных компьютерах является я wsv. Ваше имя пользователя очевидно, будет другим. На этом этапе не беспокойтесь о том, что находится слева от подсказки>: это зависит от каждого компьютера и может быть изменено позднее. Более короткая подсказка> будет использоваться для Windows.

В macOS встроенный терминал называется Терминалом. Его можно открыть через Spotlight: одновременно нажмите клавиши Command и пробел, а затем введите «терминал».

Либо откройте новое окно Finder, перейдите в каталог «Приложения», прокрутите вниз, чтобы открыть каталог «Утилиты», и дважды щелкните приложение под названием «Терминал». Откроется новый экран с белым фоном по умолчанию мигающим курсором после подсказки %. Не беспокойтесь о том, что находится слева от подсказки%. Он зависит от компьютера и может быть изменен позже.

на.

Ракушка

Завершился Macbook-Pro: ~ wsv%

Если в приглашении macOS отображается `$` вместо `%`, это означает, что вы используете Bash в качестве оболочки. Начиная с 2019 года macOS переключилась с Bash на zsh в качестве оболочки по умолчанию, от большинства команд в этом книга будет работать взаимозаменяющими, рекомендуется найти в Интернете, как перейти на zsh через Системные настройки, если ваш компьютер все еще использует Bash.

## Команды оболочки

Существует множество достаточноенных команд оболочки, но большинство разработчиков с нова и с нова полагают языком один и те же команды и по мере необходимости ищут более сложные команды.

Во многих случаях команды для Windows (PowerShell) и macOS похожи. Например, команда `whoami` возвращает имя компьютера/имя пользователя в Windows и только имя пользователя в macOS. Как и для всех команд оболочки, введите саму команду, а затем клавишу возврата. Обратите внимание, что символ `#` представляет собой комментарий и не будет выполняться в командной строке.

Ракушка

---

```
# Окна
> КТО
вс в2021/вс в

# Mac OS
% кто
WSV
```

---

Однако иногда команды оболочки в Windows и macOS совершенно разные. Хорошим примером является команда для вывода приветствия «Hello, World!» с сообщением в консоль. На

в Windows используется команда `Write-Host`, а в macOS — команда `echo`.

Глава 1: Первоначальная настройка

Ракушка

---

# Окна

> Пишите-х ос т "Привет, мир!"

Привет, мир!

# мак OS

% э х о "Привет, мир!"

Привет, мир!

---

Частью задачей в командной строке является навигация по файловой системе компьютера. В Windows и macOS команда pwd (распечатать рабочий каталог) показывает текущее местоположение.

Ракушка

---

# Окна

> pwd

Дорожка

---

C:\Пользователи\WSV

# мак OS

% порошка

/Пользователи/WSV

---

Вы можете сюда занести свой код Django где угодно, но для удобства мы поместим наш код в каталог рабочего стола. Команда cd (сменить каталог), за которой следует предполагаемое местоположение, работает в обеих системах.

Глава 1: Первоначальная настройка

Ракушка

---

### # Окна

```
> cd onedrive\рабочий стол > pwd
```

Дорожка

----

C:\Users\wsv\onedrive\рабочий стол

# мак OS

% cd рабочий

стол % pwd

/Пользователи/wsv/рабочий стол

---

**Совет:** Клавиша табуляц ии автоматичес ки дополняет команду , поэ тому , если вы наберете cd d , а затем нажмете табуляц ию о стальная часть имени буд ет автоматичес ки заполнена . Если сущес твует более двух каталог ов , начинавших с яс буквы d , с нова нажмите клавишу табуляц ии , чтобы просмотреть их .

Чтобы с оздать новый каталог , ис пользуйте команду mkdir , за которой следует имя . Мы с оздали один названный код на рабочем столе , а затем внутри него новый каталог с именем setup .

Ракушка

---

### # Окна

> код mkdir

> код диска

> установка mkdir >

установка компакт-диска

# мак OS

% mkdir код

% cd-код

% установка mkdir %

установка компакт-диска

---

Вы можете убедитс я , что он был с оздан , взг ляну в на рабочий стол или выполнив команду ls .

Полный вывод Windows немног одлиннее , но здес с он сокращен для краткости .

Глава 1: Первоначальная настройка

Ракушка

---

# Окна

> лс

настравить

# macOS

% л.с.

настравить

---

**Совет:** команда очистки очищает Терминал от прошлых команд и вых одных данных , чтобы у вас был чистый лист . Команда табуляц ии автоматически дополняет строку, как мы обсуждали. А клавиши и переключателя между предыдущими командами, чтобы не вводить одно и то же с нова и с нова.

Чтобы выйти, вы можете закрыть терминал с помощью мыши, но я рекомендую использовать вместо этого оконную оболочку. Это работает по умолчанию в Windows, но в macOS необходимо изменить настройки терминала. В верхней части экрана нажмите « Терминал », затем « Настстройки » в раскрытом меню. Нажмите « Профили » в верхнем меню, а затем « Оболочка » из списка ниже.

Существует переключатель для « При выходе из оболочки ». Выберите « Закрыть окно ».

Ракушка

---

# Окна

> вых од

# macOS

% вых од

---

Вроде круто, правда? С практикой командная строка становится гораздо более эффективным способом навигации и управления компьютером, чем использование мыши. Для работы с этой книгой вам не нужно быть экспертом по командной строке: каждый раз я буду давать точные инструкции для запуска. Но если вам интересно, полный список команд оболочки для каждой операционной системы можно найти на [ss64.com](http://ss64.com).

## Установите Python 3 в Windows

В Windows Microsoft размещает выпуск Python 3 для сообщества в Microsoft Store. В строке поиска в нижней части экрана введите «python» и нажмите на лучший результат соответствия.

Глава 1: Первичная настройка

Это автоматически запустит Python 3.10 в Microsoft Store. Нажмите на кнопку «Получить»  
скачать его.

Чтобы убедиться, что Python установлен правильно, откройте новое окно терминала с помощью PowerShell и  
введите `python --version`.

Ракурс

---

```
> python --версияPython
3.10.2
```

---

Результат должен быть не ниже Python 3.10. Затем введите `python`, чтобы открыть интерпретатор Python из  
оболочки командной строки.

Ракурс

---

```
> Python
Python 3.10.2 (tag: v3.10.2:a58ebcc, 17 января 2022 г., 19:00:18)
[MSC v.1929 64 бит (AMD64)] на win32 Введите
"help", "copyright", "credits" или "license" для получения дополнительной информации.
>>>
```

---

## Установите Python 3 на Mac

На Mac лучше всего использовать официальный установщик на веб-сайте Python. В новом окне браузера  
перейдите на страницу загрузки Python<sup>6</sup>, и нажмите кнопку под текстом «Загрузить последнюю версию для Mac  
OS X». На момент написания этой статьи это Python 3.10. Пакет будет находиться в вашем каталоге загрузок.  
Дважды щелкните по нему, чтобы запустить установщик Python, и следуйте инструкциям.

Чтобы подтвердить, что загрузка прошла успешно, откройте новое окно терминала и введите `python3 --версия`

---

<sup>6</sup><https://www.python.org/downloads/>

Глава 1: Первоначальная настройка

Ракурска

---

```
% python3 --версия Python  
3.10.2
```

---

Результат должен быть не менее 3.10. Затем введите `python3`, чтобы открыть интерпретатор Python.

Ракурска

---

```
% python3  
Python 3.10.2 (v3.10.2:a58ebcc701, 13 января 2022 г., 14:50:16)  
[Clang 13.0.0 (clang-1300.0.29.30)] на darwin Введите «помощь»,  
«авторское право», «кредиты» или «лицензия» для получения дополнительной информации.  
>>>
```

---

## Интерактивный режим Python

Если в командной строке ввести `python` в Windows или `python3` в macOS, откроется языковой интерпретатор Python, также известный как интерактивный режим Python. Новое приглашение `>>>` указывает, что вы находитесь внутри сессии Python, а не в командной строке. Если вы попробуете любую из предыдущих команд оболочки, которые мы запускали — `cd`, `ls`, `mkdir` — каждая из них вызовет ошибки. Чем будет работать, так это настоящий код Python. Например, попробуйте как `1 + 1`, так и `print("Hello Python!")`, обязательно нажимая клавишу `Enter` или `Return` после каждого, чтобы запустить их.

Ракурска

---

```
>>> 1 + 1  
2  
>>> print("Привет, Питон!")  
Привет Питон!
```

---

Интерактивный режим Python — отличный способ сэкономить время, если вы хотите попробовать небольшой код Python. Но есть ряд ограничений: вы не можете сюда занести сюда работу в файле, написание более длинных фрагментов кода обременительно. В результате большую часть времени мы потратим на написание Python и Django в файлах с помощью текстового редактора.

Чтобы выйти из Python из командной строки, вы можете ввести либо `exit()` и клавишу `Enter`, либо использовать `Ctrl + z` в Windows или `Ctrl + d` в macOS.

## Виртуальные среды

Установка последней версии Python и Django — правильный подход для любого нового проекта.

Но в реальном мире обычно существует множество проектов, используемых старые версии каждого из них. Рассмотрим следующий пример: проект A использует Django 2.2, а проект B использует Django 4.0? По умолчанию Python и Django устанавливаются глобально на компьютере, что означает, что устанавливать и переустанавливать разные версии каждый раз, когда вы хотите переключаться между проектами, довольно сложно.

К счастью есть простое решение. Виртуальные среды позволяют создавать и управлять отдельными средами для каждого проекта Python на одном компьютере. Есть много областей разработки программного обеспечения, которые являются предметом горячих споров, но использование виртуальных сред для разработки Python не является одной из них. Вы должны использовать выделенную виртуальную среду для каждого нового проекта Python.

Существует несколько способов реализации виртуальных сред, но самый простой — с помощью [venv](#)<sup>7</sup>. Уже установлен как часть стандартной библиотеки Python 3. Чтобы попробовать, перейдите в существующий каталог установки на рабочем столе.

Ракурс

---

```
# Окна  
> cdonedrive\рабочий стол\код\настройка  
  
# Mac OS  
% cd ~/рабочий стол/код/установка
```

---

Чтобы создать виртуальную среду в этом новом каталоге, используйте формат `python -m venv <name_of_env>` в Windows или `python3 -m venv <name_of_env>` в macOS. Развитчик должен выбрать правильное имя среды, но обычно его называют `.venv`.

---

<sup>7</sup><https://docs.python.org/3/library/venv.html>

Ракушка

## # Окна

```
> python -m venv .venv > Set-
ExecutionPolicy -ExecutionPolicy RemoteSigned -Scope CurrentUser
>
```

## # Mac OS

```
% python3 -m venv (.venv)
%
```

Если вы используете команду `ls` для просмотра нашего текущего каталога, он будет казаться пустым. Однако каталог `.venv` существует, просто он «скрыт» из-за точки, что предшествует имени.

Скрытые файлы и каталоги — это способ для разработчиков указать, что их содержимое важно и с ним следует обращаться иначе, чем с обычными файлами. Чтобы просмотреть его, попробуйте `ls -la`, которая показывает все каталоги и файлы, даже скрытые.

Ракушка

```
> ls -la
всегда
drwxr-xr-x 3 wsv staff 96 7 окт 11:10 .
drwxr-xr-x 3 wsv staff 96 7 окт 11:10 ..
drwxr-xr-x 6 wsv staff 192 7 окт 11:10 .venv
```

Вы увидите, что `.venv` есть, и при желании к нему можно получить доступ через компакт-диск. В самом каталоге есть копия интерпретатора Python и несколько сценариев управления, но вам не нужно будет использовать их непосредственно в этой книге.

После создания виртуальной среды должна быть активирована. В Windows необходимо установить политику выполнения, чтобы разрешить выполнение сценариев. Это можно сделать с помощью документации по Python<sup>8</sup> рекомендуем разрешить сценарии только для `CurrentUser`, чтобы мы и сделаем. В macOS нет подобных ограничений для скриптов, поэтому можно напрямую запустить исходный файл `.venv/bin/activate`.

Вот как выглядят полные команды для создания и активации новой виртуальной среды под названием `.venv`:

<sup>8</sup><https://docs.python.org/3/library/venv.html>

Ракушка

## # Окна

```
> python -m venv .venv > Set-
ExecutionPolicy -ExecutionPolicy RemoteSigned -Scope CurrentUser > .venv\Scripts\Activate.ps1
(.venv) >
```

## # Mac OS

```
% python3 -m venv .venv %
источник .venv/bin/activate (.venv)
%
```

Приглашение оболочки теперь имеет префикс имени седы (.venv), который указывает, что виртуальная седа активна. Любые пакеты Python, установленные или обновленные в этом месте, будут ограничиваться активной виртуальной седой.

Чтобы деактивировать и покинуть виртуальную седу, введите deactivate.

Ракушка

## # Окна

```
(.venv) > деактивировать
>
```

## # Mac OS

```
(.venv) % деактивировать
%
```

В приглашении оболочки больше нет префикса имени виртуальной седы, что означает, что сейчас теперь вернулся к нормальной жизни.

## Установите Django и Django REST Framework

Теперь, когда Python установлен и мы знаем, как использовать виртуальные седы, пришло время установить Django и Django REST Framework. В каталоге установки повторно активируйте существующий виртуальный седа.

Ракурса

---

```
# Окна
> .venv\Scripts\Activate.ps1 (.venv) >
```

```
# Mac OS
% источник .venv/bin/activate (.venv)
%
```

---

Django размещается в индексе пакетов Python (PyPI)<sup>9</sup> , центральный репозиторий для большинства пакетов Python.

Мы будем использовать pip, самый популярный установщик пакетов, который входит в состав Python 3. Чтобы установить последнюю версию Django, используйте команду `python -m pip install django ==4.0.0`.

Оператор с равенством гарантирует автоматическую установку последующих обновлений безопасности для Django, таких как 4.0.1, 4.0.2 и т. д. Обратите внимание, что вы можете использовать более краткую форму `pip install <package>`, рекомендуется использовать более длинную и более явную форму `python -m pip install <package>` , чтобы убедиться, что вы используете правильную версию Python. Это может быть проблемой, если на вашем компьютере установлено несколько версий Python.

Ракурса

---

```
(.venv) > python -m pip install django~==4.0.0
```

---

Вы можете увидеть ПРЕДУПРЕЖДЕНИЕ с сообщением об обновлении pip после выполнения этих команд. Всегда полезно использовать последнюю версию pip для обновления и удаления ненадежное ПРЕДУПРЕЖДЕНИЕ с сообщением каждый раз, когда вы используете pip. Вы можете либо скопировать и вставить рекомендуемую команду, либо запустить `python -m pip install --upgrade pip` , чтобы установить последнюю версию pip

Ракурса

---

```
(.venv) > python -m pip install --upgrade pip
```

---

Последняя версия Django REST Framework — 3.12.0. Чтобы установить его или любые будущие обновления 3.12.x, используйте следующую команду:

---

<sup>9</sup><https://pypi.org>

Ракушка

---

```
(.venv) > python -m pip установить djangorestframework==3.13.0
```

---

Команда pip freeze выводит содержимое вашей текущей виртуальной среды.

Ракушка

---

```
(.venv) > замораживание
пипс ов asgiref==3.4.1
Django==4.0.0
djangorestframework==3.12.4
pytz==2021.3 sqlparse==0.4.2
```

---

Наша среда содержит все необходимые пакеты, которые были установлены. Django использует asgiref, pytz и sqlparse, которые автоматически добавляются при установке Django.

Стандартной практикой является явный вывод содержимого виртуальной среды в файл с именем requirements.txt. Это позволяет хранить установленные пакеты, а также позволяет другим разработчикам воспроизводить виртуальную среду на разных компьютерах. Давайте сделаем это сейчас, используя оператор > .

Ракушка

---

```
(.venv) > заморозка пипс а > requirements.txt
```

---

Если вы заглянете в каталог установки, то увидите дополнительный файл с именем requirements.txt. Если вы откроете его с содержимым в текстовом редакторе, вы увидите, что оно совпадает с пятью программами, ранее выводившимися в командную строку.

## Текстовые редакторы

Командная строка — это место, где мы выполняем команды для наших программ, а текстовый редактор — это место, где пишется фактический код. Компьютеру все равно, какой текстовый редактор вы используете — конечным результатом будет простой код, — но хороший текстовый редактор может предложить вам полезные подсказки и выявить опечатки.

Доступно множество современных текстовых редакторов, но очень популярным является Visual Studio Code<sup>10</sup>, который , бес платен, прост в установке и пользуется широкой популярностью с ли вы еще не используете текст редактор, скачайте и установите VSCode с официального сайта.

Не обязательным, но настоятельно рекомендуем дополнительным шагом является яис пользование большой экосистемы расширений, доступных в VSCode. В Windows перейдите в Файл -> Настройки -> Extensions или в macOS Code -> Preferences -> Extensions. Это запускает панель поиска для рынка расширений. Введите «python», который вызовет расширение Microsoft как первый результат. Установите его.

Второе расширение для добавления — Black<sup>11</sup>, который предствляет собой средство форматирования кода Python, к оторое былостроено с стандартом сообщества Python. Чтобы установить Black, откройте окно терминала в VSCode, выбрав «Терминал» -> «Новый терминал» в верхней части страницы. В новом окне терминала, открытом внизу страницы, введите python -m pip install black. Затем откройте настройки VSCode, перейдя в «Файл» -> «Настройки» -> «Настройки» в Windows или «Код» -> «Настройки» -> «Настройки» в macOS. Найдите «поставщик форматирования python» и выберите черный цвет в раскрывающемся списке. Затем найдите «форматировать при сохранении» и включите «Редактор: форматировать при сохранении». Black теперь автоматически форматирует ваш код всякий раз, когда сохраняется файл \*.py .

Чтобы убедиться что это работает, используйте текстовый редактор, чтобы создать новый файл с именем hello.py в каталоге установки, расположенному на вашем рабочем столе, и введите следующие, используя одинарные кавычки:

---

```
привет.py
```

---

```
print("Привет, мир!")
```

---

При сохранении он должен быть автоматически обновлен до использования двойных кавычек, что является предпочтительным параметром Black по умолчанию<sup>12</sup> : print("Привет, мир!"). Это означает, что все работает должным образом.

Установить Гит

Последним шагом является установка Git, системы контроля версий, которая необходима для современной разработки программного обеспечения. С Git вы можете сотрудничать с другими разработчиками, отслеживать всю работу через

---

<sup>10</sup><https://code.visualstudio.com/> <sup>11</sup><https://pypi.org/project/black/> <sup>12</sup>[https://black.readthedocs.io/en/stable/the\\_black\\_code\\_style/current\\_style.html#strings](https://black.readthedocs.io/en/stable/the_black_code_style/current_style.html#strings)

коммитов и вернуться к любой предыдущей версии вашего кода, даже если вы с лучшей стороны удалили что-то важное!

В Windows перейдите на официальный сайт <https://git-scm.com/> и нажмите ссылку «Загрузить», которая должна установить под окно вашего окна компьютера. Сократите файл, а затем откройте папку «Загрузки» и дважды щелкните файл. Это запустит установщик Git для Windows.

Нажмите кнопку «Далее» для большинства ранних шагов по умолчанию так как они в порядке и всегда могут быть обновлены позже по мере необходимости. Однако есть два изменения в разделе «Выбор редактора по умолчанию для Git» выберите VS Code, а не Vim. А в разделе «Настройка имени начальной ветки в новых репозиториях» выберите вариант исользования «main», а не «master» в качестве имени ветки по умолчанию. В противном случае рекомендуемые значения по умолчанию всегда могут быть изменены позже, если нужный.

Чтобы убедиться, что Git установлен в Windows, закройте все текущие окна оболочки, а затем откройте новое, которое загрузит изменения в нашу переменную PATH. Введите git --version, который должен показать это.

установлено.

#### Ракушка

---

##### # Окна

```
> git --version версия
git 2.33.1.windows.1
```

---

В macOS установка Git через Xcode в настоящее время является самым простым вариантом. Чтобы проверить, установлен ли Git на вашем компьютере, введите git --version в новом окне терминала.

#### Ракушка

---

##### # Mac OS

```
% git --version
```

---

Если у вас не установлен Git, всплывающее сообщение спросит, хотите ли вы установить его как часть «инструментов разработчика командной строки». Выберите «Установить», чтобы загрузить Xcode и его пакет инструментов командной строки. Или, если вы по какой-то причине не видите это сообщение, вместо этого введите xcode-select -install, чтобы установить Xcode напрямую.

Имейте в виду, что Xcode — это очень большой пакет, поэтому первоначальная загрузка может занять некоторое время. Xcode в первую очередь предназначена для создания приложений iOS, но также включает в себя множество функций, не общедоступных разработчикам.

мак OS . После завершения загрузки закройте все существующие оболочки терминала, откройте новое окно и введите git --version , чтобы убедиться что установка работает.

Ракушка

---

```
# мак OS
% git --version
версия git 2.30.1 (Apple Git-130)
```

---

После того как Git будет установлен на вашем компьютере, нам нужно выполнить однократную настройку системы , объявив имя и адрес электронной почты, связанные со всеми вашими коммитами Git. Мы также установим имя ветки по умолчанию main . В оболочке командной строки введите следующие две строки. Не забудьте обновить им свое имя и адрес электронной почты.

Ракушка

---

```
> git config --global user.name "Ваше имя" > git config --
global user.email "yourname@email.com" > git config --global
init.defaultBranch main
```

---

Вы всегда можете изменить эти конфигурации позже, если хотите, повторно набрав те же команды с новое имя или адрес электронной почты.

## Заключение

Настойка новой среды разработки программного обеспечения совсем не доставляет удовольствия даже опытным программистам. Но если вы дошли до этого момента, одноразовая боль принесет много dividends в будущем. Теперь мы узнали о командной строке, интерактивном режиме Python и установили последнюю версию Python, Django и Django REST Framework. Мы установили Git и настроили наш текстовый редактор. Далее мы узнаем о веб-API, а затем погружимся в создание собственных с помощью Django.

## Глава 2. Веб-API

Прежде чем мы начнем создавать собственные веб-API с помощью Django, важно рассмотреть, как на самом деле работает Интернет. В конце концов, «веб-API» буквально лежит поверх существующей архитектуры всемирной паутины и опирается на множества технологий, включая HTTP, TCP/IP и другие.

В этой главе мы рассмотрим базовую терминологию веб-API: конечные точки, ресурсы, протоколы HTTP, коды состояния HTTP и REST. Даже если вы уже чувствуете себя комфортно с этими терминами, я рекомендую прочитать эту главу полностью.

### Всемирная сеть

Интернет — это система взаимосвязанных компьютерных сетей, существующая как минимум с 1960-х годов<sup>13</sup>. Однако первое использование Интернета было ограничено небольшим количеством изолированных сетей, в основном правительственных, военных или научных по своему характеру, которые обменивались информацией в электронном виде. К 1980-м годам многие научно-исследовательские институты и университеты использовали Интернет для обмена данными. В Европе крупнейший интернет-узел находился в ЦЕРНе (Европейском ядерном центре) в Женеве, Швейцария, где работает крупнейшая в мире лаборатория физики элементарных частиц. Эти эксперименты генерируют огромные объемы данных, которыми необходимо обмениваться удаленно. с учеными всего мира.

Однако по сравнению с сегодняшним днем общее использование Интернета в 1980-х годах было незначительным. Большинство людей не имели к нему доступа и даже не понимали, почему это важно. Небольшое количество интернет-узлов питалось трафиком, и компьютеры, использующие его, в основном находились в пределах одного итогового небольшого сети.

Все изменилось в 1989 году, когда научный сотрудник ЦЕРНа Тим Бернерс-Ли изобрел HTTP и открыл современную всемирную паутину. Его величайшее изобретение заключалось в том, что существующий гипертекст<sup>14</sup>

---

<sup>13</sup><https://en.wikipedia.org/wiki/Интернет>

<sup>14</sup><https://en.wikipedia.org/wiki/Гипертекст>

## Глава 2. Веб-API

система, где текст, отображаемый на экране компьютера, содержал ссылки (гиперссылки) на другие документы, могут быть перемещены в Интернет.

Его изобретение, [протокол передачи гипертекста \(HTTP\)](#)<sup>15</sup>, был первым стандартным универсальным способом обмена документами через Интернет. Это открыло концепцию веб-страниц: отдельные документы с URL-адресом, с ссылками и ресурсами, такими как изображения, аудио или видео.

Сегодня большинство людей думают об «Интернете», они думают о всемирной паутине, которая в настоящем времени является ядром новым способом общения миллиардов людей и компьютеров в сети.

### URL-адреса

URL-адрес (унифицированный указатель ресурсов) — это адрес ресурсов в Интернете. Например, домашняя страница Google находится по адресу <https://www.google.com>.

Когда вы хотите перейти на главную страницу Google, вы вводите полный URL-адрес в веб-браузере.

Затем ваш браузер отправляет запрос через Интернет и волшебным образом подключается к (мы все прекрасно знаем, что происходит на самом деле) сервером, который отвечает данными, необходимыми для отображения домашней страницы Google в вашем браузере.

Этот шаблон запроса и ответа является ядром новой всего веб-коммуникации. Клиент (как правило, веб-браузер, а также собственное приложение или любое устройство, подключенное к Интернету) запрашивает информацию у сервера, отвечает ответом.

Поскольку веб-связь проходит через HTTP, они более формально известны как HTTP-запросы и HTTP-ответы.

Внутри данного URL-адреса также есть несколько отдельных компонентов. Например, рассмотрим домашнюю страницу Google, расположенную по адресу <https://www.google.com>. Первая часть, https, относится к используемой схеме. Он сообщает веб-браузеру, как получить доступ к ресурсам в этом месте. Для веб-сайта это обычно http или https, но это также может быть ftp для файлов, smtp для электронной почты и так далее. Следующий раздел, www.google.com, является именем хоста или фактическим названием сайта. Каждый URL содержит схему и хосты.

Многие веб-страницы также содержат необязательный путь. Если вы переходите на домашнюю страницу Python

---

<sup>15</sup>[https://en.wikipedia.org/wiki/Hypertext\\_Transfer\\_Protocol](https://en.wikipedia.org/wiki/Hypertext_Transfer_Protocol)

## Глава 2. Веб-API

на <https://www.python.org> и щелкните ссылку на страницу «О программе», после чего вы будете перенаправлены на <https://www.python.org/about/>. Кусок /about/ — это путь.

Таким образом, каждый URL-адрес, такой как <https://python.org/about/>, состоит из трех потенциальных частей:

- схема - https
- имя хоста — www.python.org
- и (необязательно) путь - /about/

## Пакет интернет-протокола

Как только мы узнаем фактический URL-адрес ресурса, весь набор других технологий должен работать должным образом (вместе), чтобы соединить клиента с сервером и загрузить реальную веб-страницу. В широком смысле это называется [набором интернет-протоколов](#)<sup>16</sup>, и есть целые книги, написанные только на эту тему. Однако для наших целей мы можем придерживаться ящиков с новыми.

Когда пользователь вводит <https://www.google.com> в свой веб-браузер и нажимает Enter, происходит следующее. Сначала браузеру нужно найти нужный сервер где-то на просторах интернета.

Он использует службу доменных имен (DNS) для преобразования доменного имени «[google.com](http://google.com)» в IP-адрес<sup>17</sup>, который представляет собой уникальную последовательность чисел, представляющуюся в виде подключенного устройства в Интернете. Доменные имена используются потому, что легче запомнить доменное имя, например «[google.com](http://google.com)», чем IP-адрес, например «172.217.164.68».

После того, как браузер получит IP-адрес для данного домена, ему нужно установить постоянное соединение с нужным сервером. Это происходит через протокол управления передачей (TCP), который обеспечивает надежную последовательную передачу ошибок и доставку байтов между двумя приложениями.

Для установления TCP-соединения между двумя компьютерами происходит трехстороннее «рукопожатие». между клиентом и сервером:

1. Клиент отправляет SYN с просьбой установить соединение 2.

Сервер отвечает SYN-ACK, подтверждая запрос и передавая соединение параметр

---

<sup>16</sup>[https://en.wikipedia.org/wiki/Internet\\_protocol\\_suite](https://en.wikipedia.org/wiki/Internet_protocol_suite)

<sup>17</sup>[https://en.wikipedia.org/wiki/IP\\_address](https://en.wikipedia.org/wiki/IP_address)

3. Клиент отправляет ACK обратно на сервер, подтверждая соединение.

Как только TCP-соединение установлено, два компьютера могут начать обмен данными через HTTP.

## Глаголы HTTP

Каждая веб-страница содержит как адрес (URL-адрес), так и список разрешенных действий, известных как глаголы HTTP. До сих пор мы в основном говорили о создании веб-страницы, но также можно создать, редактировать и удалять содержимое.

Рассмотрим сайт Facebook. После входа в систему вы можете просмотреть свою страницу, создать новую публикацию или отредактировать/удалить существующую. Эти четыре действия Create-Read-Update-Delete известны в программировании как «CRUD» и представляют собой подавляющее большинство действий, предпринимаемых в Интернете.

Протокол HTTP содержит ряд методов запроса, которые можно использовать при запросе информации с сервера. Четыре наиболее распространенных сопоставления функциональности CRUD: POST, GET, PUT, и УДАЛИТЬ.

### Диаграмма

CRUD	Глаголы HTTP
Создать	POST
Читать	GET
Обновление	PUT
Удалить	DELETE

Для создания контента вы используете POST, для чтения контента GET, для обновления PUT и для удаления вы используете УДАЛИТЬ.

## Конечные точки

Традиционный веб-сайт состоит из веб-страниц с HTML, CSS, изображениями, JavaScript и т. д. Для каждой страницы существует специальный URL-адрес, например, example.com/1/. Веб-API также зависит от URL-адресов, и соответствующий URL-адрес может быть example.com/api/1/, но вместе с обслуживанием веб-страниц

## Глава 2. Веб-API

потребляемый людьми, он создает конечные точки API. Конечные точки содержат данные, как правило, в [формате JSON](#)<sup>18</sup>. формат, а также с поиском доступных действий (HTTP-лаголов).

Например, мы могли бы создать следующие конечные точки API для нового веб-сайта с именем mysite.

Диаграмма

---

<code>https://www.mysite.com/api/users https://www.mysite.com/api/users/&lt;id&gt;</code>	# GET возвращает всех пользователей
	# GET возвращает одного пользователя

---

В первой конечной точке, /api/users, простой запрос GET возвращает список всех доступных пользователей. Этот тип конечной точки, которая возвращает несколько records of data, известен как коллекция.

Вторая конечная точка, /api/users/<id>, предоставляет одного пользователя. Запрос GET возвращает информацию о пользователе.

Если бы мы добавили POST к первой конечной точке, мы могли бы создать нового пользователя, а добавление DELETE ко второй конечной точке позволило бы нам удалить одного пользователя.

В ходе чтения этой книги мы разделились на знакомство с конечными точками API, но в конечном итоге создание API включает в себя создание ряда конечных точек: URL-адресов, представляющих данные JSON, и связанные заголовки HTTP.

## HTTP

Мы уже много говорили о HTTP в этой главе, а теперь опишем, что это такое на самом деле. есть и как это работает.

HTTP — это протокол запрос-ответ между двумя компьютерами, имеющими общую TCP-связь. Компьютер, отправляющий запросы, называется клиентом, а отвечающий компьютер называется сервером. Обычно клиент — это веб-браузер, но это также может быть приложение iOS или любое устройство, подключенное к Интернету. Сервер — это причудливое название для любого компьютера, оптимизированного для работы в Интернете. Все, что нам действительно нужно, чтобы превратить обычный ноутбук в сервер, — это специальное программное обеспечение и постоянное подключение к Интернету.

---

<sup>18</sup><https://json.org/>

Каждое HTTP-с сообщение состоит из строк и состояния, заголовков и необязательных данных тела. Например, вот пример HTTP-с сообщения, которое браузер может отправить, чтобы запросить главную страницу Google, расположенную на <https://www.google.com>.

Диаграмма

---

```
GET / HTTP/1.1
Host: google.com
Accept-Language: en-US
```

---

Верхняя строка называется строкой запроса и указывает используемый метод HTTP (GET), путь (/) и конкретную используемую версию HTTP (HTTP/1.1).

Две следующие строки — это заголовки HTTP: Host — это доменное имя, а Accept-Language — используемый язык, в данном случае — американский английский. Существует множество заголовков HTTP<sup>19</sup>, имеется явный наличия.

Сообщение HTTP также имеет необязательный третий раздел, известный как тело, однако мы видим только сообщение тела с ответами HTTP, содержащими данные.

Для простоты предположим, что домашняя страница Google содержит только HTML-код «Hello, World!»

Вот как может выглядеть ответное сообщение HTTP с сервера Google.

Диаграмма

---

```
HTTP/1.1 200 OK
Date: понедельник, 24 января 2022 г., 23:26:07
GMT Server: gws Принятые диапазоны: байты
Content-Length: 13 Content-Type: text/html;
编码 = UTF-8
```

---

Привет, мир!

Верхняя строка — это строка ответа, она указывает, что мы используем HTTP/1.1. Код состояния 200 OK указывает на то, что запрос клиента был успешным (подробнее о кодах состояния чуть позже).

Следующие пять строк — это заголовки HTTP. И, наконец, после разрыва строк находитя наше настолько тело. с содержание «Привет, мир!».

Таким образом, каждое HTTP-с сообщение, будь то запрос или ответ, имеет следующий формат:

---

<sup>19</sup>[https://en.wikipedia.org/wiki/List\\_of\\_HTTP\\_header\\_fields](https://en.wikipedia.org/wiki/List_of_HTTP_header_fields)

## Глава 2. Веб-API

Диаграмма

---

Заголовки с током ответа/  
запроса...

---

(не обязательно) Тело

---

Большинство веб-страниц, содержащие не сколько страниц, для которых требуется не сколько циклов запросов/ответов HTTP. Если бы веб-страница содержала HTML, один файл CSS и изображение, потребовалось бы три отдельных перехода между клиентом и сервером, прежде чем вся веб-страница могла бы быть отображена в браузере.

## Коды состояния

После того, как ваш веб-браузер выполнил HTTP-запрос по URL-адресу, нет никакой гарантии, что все действительны будет работать! Таким образом, существует довольно длинный список кодов состояния HTTP<sup>20</sup>, доступные для сопровождения каждого ответа HTTP.

Вы можете указать общий тип кода состояния на основе следующей системы:

- 2xx Success — действие, запрошенное клиентом, было получено, понято и принято.
- 3xx Redirection — запрошенный URL-адрес был перемещен.
- 4xx Client Error — произошла ошибка, обычно это неправильный запрос URL-адреса клиентом.
- 5xx Server Error — серверу не удалось обработать запрос.

Нет необходимости запоминать все доступные коды состояния. По мере практики вы познакомитесь с наиболее распространенными из них, такими как 200 (OK), 201 (Создано), 301 (Перемещено навсегда), 404 (Не найдено) и 500 (Ошибка сервера).

Важно помнить, что, вообще говоря, любой одинногий HTTP-запрос есть только четыре возможных результата: он сработал (2xx), он был каким-то образом перенаправлен (3xx), клиент сделал ошибку (4xx) или сервер делал ошибку (5xx).

Эти коды состояния автоматически помещаются в строку запроса/ответа вверху каждого HTTP-сообщения.

---

<sup>20</sup>[https://en.wikipedia.org/wiki/List\\_of\\_HTTP\\_status\\_codes](https://en.wikipedia.org/wiki/List_of_HTTP_status_codes)

## Глава 2. Веб-API

### Без ражанс тво

Последнее важное замечание, которое следует отметить в отношении HTTP, заключается в том, что этот протокол без сих ранения состояния. Это означает, что каждая пара запрос / ответ полностью зависит от предыдущей. Нет сих раненой памяти о прошлых взаимодействиях, которая известна как [состояние21](#). в информатике.

Без ражанс тво дает много преимуществ HTTP. Поскольку во всех системах электронной связи с временем происходил потеря сигнала, если бы у нас не было протокола без сих ранений состояния все постоянно ломалось бы, если не прошел бы один цикл запроса/ответа. В результате HTTP известен как оченьустойчивый распределенный протокол.

Недостатком является то, что управление состоянием очень, очень важно в веб-приложениях. Состояние — это то, как веб-сайт запоминает, чтобы вышли в систему, и как сайт электронной коммерции управляет вашей корзиной покупок. Это важно для того, как мы используем современные веб-сайты, но это не поддерживается самим HTTP.

Исторически состояние сих ранялось на сервере, но с современных интерфейсных средах, таких как React, Angular и Vue, оно все больше и больше перемещается на клиент, веб-браузер. мы узнаем больше о состоянии, когда мы расматриваем аутентификацию пользователя но помните, что HTTP не имеет состояния. Это делает это очень хорошо для надежной отправки информации между двумя компьютерами, но плохо для запоминания чего-либо за пределами каждой отдельной пары запрос / ответ.

### ОТДЫХ

[Репрезентативная передача состояния\(REST\)22](#) это архитектура, впервые предложенная в 2000 году Роям Филдингом в его диссертации. Это подход к созданию API поверх Интернета, что означает поверх протокола HTTP.

Были написаны целые книги о том, что делает API действительно RESTful или нет. Но есть три основные черты, на которых мы сосредоточимся здесь для наших целей. Каждый RESTful API:

- не имеет состояния как HTTP
- поддерживает распространенные команды HTTP (GET, POST, PUT, DELETE и т. д.)

---

<sup>21</sup>[https://en.wikipedia.org/wiki/State\\_\(computer\\_science\)](https://en.wikipedia.org/wiki/State_(computer_science)) <sup>22</sup>[https://en.wikipedia.org/wiki/Representational\\_state\\_transfer](https://en.wikipedia.org/wiki/Representational_state_transfer)

- возвращает данные в формате JSON или XML

Любой RESTful API должен, как минимум, следовать этим трем принципам. Стандарт важен, потому что он обеспечивает согласованный способ разработки и ис пользования веб-API.

## Заключение

Несмотря на то, что в основе современной всемирной паутины лежит множество технологий, нам, разработчикам, нужно внедрять все это с нуля. Прекрасное сочетание Django и Django REST Framework правильно справляется с самой большой частью ложности, связанный с веб-API. Однако важно иметь хотя бы общее представление о том, как все части сочетаются друг с другом.

В конечном счете, веб-API — это набор конечных точек, которые предоставляются для определенным частям базовой базы данных. Как разработчики, мы контролируем URL-адреса для каждой конечной точки, какие базовые данные доступны и какие действия возможны с помощью HTTP-команд. Используя заголовки HTTP, мы также можем устанавливать различные уровни аутентификации и разрешений, как мы увидим позже в книге.

## Глава 3: Веб-сайт библиотеки

Django REST Framework работает вместе с веб-платформой Django для создания веб-API. Мы не можем создать веб-API только с помощью Django Rest Framework. Его все еще нужно добавлять в проект после того, как сам Django будет установлен и настроен.

В этой главе мы рассмотрим существенные различия между традиционным Django и Django REST Framework. Самый важный вывод заключается в том, что Django создает веб-сайты, содержащие веб-страницы, а Django REST Framework создает веб-API, которые представляют собой набор конечных точек URL-адресов, содержащих доступные HTTP-методы, которые возвращают JSON.

Чтобы проиллюстрировать эти концепции, мы создадим базовый веб-сайт библиотеки с помощью традиционного Django, а затем расширим его до веб-API с помощью Django REST Framework.

### Традиционный Джанго

Перейдите в существующий каталог кода на рабочем столе и убедитесь, что вы находитесь в текущей виртуальной среде. Вы не должны видеть (.venv) перед приглашением оболочки. Если вы это делаете, используйте команду deaktivировать, чтобы выйти из него. Создайте новый каталог с именем library, создайте новую виртуальную среду, активируйте ее и установите Django.

Ракушка

---

```
# Ok на
> cd onedrive\desktop\code >
библиотека mkdir > библиотека
cd > python -m venv .venv
> .venv\Scripts\Activate.ps1 (.venv)
> python -m pip install django~=4.0.0
```

# Mac OS

```
% cd рабочий стол/рабочий стол/
код % библиотека mkdir %
библиотека cd
```

```
% python3 -m venv .venv %
источник .venv/bin/activate (.venv) %
python3 -m pip install django~=4.0.0
```

---

Традиционный веб-сайт Django состоит из одного проекта с несколькими приложениями, предстающими отдельные функции. Давайте создадим новый проект с помощью команды startproject с именем django\_project. Не забудьте указать период. в конце, который устанавливает код в наш текущий каталог.

Если вы не укажете точку, Django по умолчанию создаст дополнительный каталог.

Ракушка

---

```
(.venv) > django-admin startproject django_project .
```

---

Становитесь на мгновение, чтобы изучить структуру проекта по умолчанию предложенную Django. Вы можете проверить это визуально, если отите, открыв новый каталог с помощью мыши на рабочем столе.

Каталог .venv изначально может быть невидимым, потому что он «скрыт», но, тем не менее, все еще существует.

Код

---

```
django_project
asgi.py23
urls.py
settings.py
manage.py
.venv/
wsgi.py
```

---

Каталог .venv был создан с нашей виртуальной средой, но Django добавил каталог проекта django\_- и файл manage.py. Внутри django\_project пять новых файлов:

- \_\_init\_\_.py указывает, что файлы в папке являются частью пакета Python. Без этого файла мы не можем импортировать файлы из другого каталога, что мы будем делать в Django!
  
- asgi.py позволяет использовать дополнительный [асинхронный интерфейс шлюза сервера](#)<sup>23</sup>, быть запущенным
- settings.py управляет общими настройками нашего проекта Django.

---

<sup>23</sup><https://asgi.readthedocs.io/en/latest/specs/main.html>

- urls.py сообщает Django, какие страницы создавать в ответ на запрос браузера или URL-адреса. urls.py означает [интерфейс шлюза веб-сервера](#)<sup>24</sup>, который помогает Django обслуживать наш возможный интернет страницы.

Файл manage.py не является частью django\_project, но он используется для выполнения различных команд Django, таких как запуск локального веб-сервера или создание нового приложения. Давайте используем его сейчас с помощью миграции, чтобы синхронизировать базу данных с нашими тремя моделями Django по умолчанию. Запустим локальный веб-сервер Django с сервером запуска.

#### Ракурс

---

```
(.venv)> python manage.py migrate (.venv)> python  
manage.py runserver
```

---

Откройте веб-браузер по [адресу](http://127.0.0.1:8000/25) <http://127.0.0.1:8000/25>. чтобы подтвердить, что наш проект успешно установлен и запущен.

---

[24https://en.wikipedia.org/wiki/Web\\_Server\\_Gateway\\_Interface](https://en.wikipedia.org/wiki/Web_Server_Gateway_Interface) 25<http://127.0.0.1:8000/>

The install worked successfully! Congratulations!

You are seeing this page because `DEBUG=True` is in your settings file and you have not configured any URLs.

---

[Django Documentation](#)  
Topics, references, & how-to's

[Tutorial: A Polling App](#)  
Get started with Django

[Django Community](#)  
Connect, get help, or contribute

Первое приложение

Следующим шагом будет добавление нашего первого приложения, которое мы назовем `книгами`. Откройте локальный сервер, нажав `Control+C`, а затем запустите команду `startapp` с именем нашего приложения, чтобы создать его.

Ракушка

(.venv) > `python manage.py startapp книги`

Теперь давайте изучим файлы приложения, которые Django автоматически создал для нас.

Ракушка

---

```
books
__init__.py
admin.py      миграции
__init__.py      models.py
```

просмотры.py

---

Каждое приложение имеет файл `__init__.py`, идентифицирующий его как пакет Python, и есть 6 новых файлов, созданный:

- `admin.py` — это файл конфигурации для встроенного приложения Django Admin.
- `apps.py` — это файл конфигурации для самого приложения.
- `migrations/` — это каталог, в котором хранятся файлы миграции для изменений в базе данных.
- `models.py` — это место, где мы определяем модели нашей базы данных.
- `tests.py` предназначено для наших тестов для конкретных приложений.
- `views.py` — это место, где мы обрабатываем логику запроса от ответа для нашего веб-приложения.

Как правило, разработчик и также создает файл `urls.py` в каждом приложении для маршрутизации. Мы сделаем это в ближайшее время.

Прежде чем двигаться дальше, мы должны добавить новое приложение в конфигурацию `INSTALLED_APPS` в файле `django-project/settings.py`. Сделайте это прямо сейчас в текстовом редакторе.

Код

---

```
# django_project/settings.py
INSTALLED_APPS =
    [ "django.contrib.admin",
      "django.contrib.auth",
      "django.contrib.contenttypes",
      "django.contrib.sessions",
      "django.contrib.messages",
      "django.contrib.staticfiles", # Локальный
      "books.apps.BooksConfig", # новый
  ]
```

---

Для каждой веб-страницы в традиционном Django требуется несолько файлов: views.py, urls.py, template и models.py. Давайте начнем с модели базы данных, чтобы структурировать данные нашей библиотеки.

## Модели

В текстовом редакторе откройте файл books/models.py и обновите его следующим образом:

Код

---

```
# books/models.py из
django.db импортировать модели
```

---

Книга класс `Book` (модели.Модель):

```
title = models.CharField(max_length=250) subtitle =
models.CharField(max_length=250) author =
models.CharField(max_length=100) isbn =
models.CharField(max_length=13)

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

---

Это базовая модель Django, в которой модели импортируются из Django в верхней строке, а новая класс под названием `Book` расширяет его. Есть четыре поля: название, подзаголовок, автор и `isbn`. Мы также включите метод `__str__`, чтобы позже название книги отображалось в читаемом формате в админке. Обратите внимание, что `ISBN` — это уникальный 13-символьный идентификатор, присваиваемый каждой опубликованной книге.

Поскольку мы создали новую модель базы данных, нам нужно создать файл миграции, который будет сопровождать ее.

Указание имени приложения не обязательно, но рекомендуется. Мы могли бы просто ввести `python manage.py makemigrations`, но если бы было несколько приложений с изменениями базы данных, оба были бы добавлены в файл миграции, что усложнило бы отладку в будущем. Страйтесь, чтобы ваши файлы миграции были как можно более конкретными.

Ракушка

---

(.venv) > `python manage.py makemigrations books Миграция для «КНИГ»:`

`КНИГИ/миграции/0001_initial.py`

- Создать книгу модели

---

Затем вторым шагом после создания файла миграции является его перенос, чтобы он применялся к существующей базе данных.

Ракушка

---

(.venv) > `python manage.py migrate Операция для`

`выполнения:`

Применить все миграции: admin, auth, books, contenttypes, session

Выполнение миграции:

Применение books.0001\_initial... OK

---

Все идет нормально. Если что-то из этого окажется вам сорванным новым, я предлагаю делать паузу, чтобы просмотреть Django для начинающих<sup>26</sup>. Для более подробного обзора некоторых традиционных проектов Django.

Администратор

Мы можем начать вводить данные в нашу новую модель через встроенное приложение Django. Чтобы использовать его, нам нужно создать учетную запись суперпользователя и обновить файл `books/admin.py`, чтобы отображалось приложение книг.

Начните с учетной записи суперпользователя. В командной строке выполните следующую команду:

---

<sup>26</sup><https://djangoforbeginners.com/>

Глава 3: Веб-сайт библиотеки

Ракушка

(.venv) &gt; python manage.py создает суперпользователя

Следуйте инструкциям, чтобы ввести имя пользователя, адрес электронной почты и пароль. Обратите внимание, что с изображениями безопасности текст не будет отображаться на экране при вводе пароля.

Теперь обновите файл admin.py нашего приложения для книг .

Код

```
# books/admin.py из
django.contrib import admin

из .models импорт книга

admin.site.register(Забронировать)
```

Это все, что нам нужно! Запустите локальный сервер снова.

Ракушка

(.venv) &gt; сервер запущен python manage.py

Перейдите по адресу <http://127.0.0.1:8000/admin> и войдите в систему. Откроется домашняя страница администратора.

The screenshot shows the Django Admin interface. At the top, there's a header bar with the title "Site administration | Django site". Below it is a navigation bar with links for "127.0.0.1:8000/admin/" and a "Guest" user. The main content area has a dark blue header "Django administration". On the left, there are two main sections: "AUTHENTICATION AND AUTHORIZATION" containing "Groups" and "Users" with "Add" and "Change" buttons, and "BOOKS" containing "Books" with "Add" and "Change" buttons. On the right, there are two panels: "Recent actions" which is currently empty, and "My actions" which also says "None available". At the bottom center, it says "Домашняя страница администратора".

Нажмите на ссылку «+ Добавить» рядом с «Книги».

Django administration

Home > Books > Books > Add book

Add book

Title: Django for Beginners

Subtitle: Build Websites with Python and Django

Author: William S. Vincent

ISBN: 978173546720

Save and add another | Save and continue editing | SAVE

Админ добавить книгу

Я ввел детали трех своих книг: Django для начинающих, Django для API и Django для профессионалов. После нажатия кнопки «Сохранить» нас перенаправляет на страницу «Книги», где перечислены все текущие записи.

The book "Django for Professionals" was added successfully.

Select book to change	
Action:	<input type="button" value="Go"/>
0 of 3 selected	
<input type="checkbox"/> BOOK	
<input type="checkbox"/> Django for Professionals	
<input type="checkbox"/> Django for APIs	
<input type="checkbox"/> Django for Beginners	
3 books	

Список книг администратора

В нашем традиционном проекте Django теперь есть данные, но нам нужен способ представить их в виде веб-страницы. Это означает создание представлений, URL-адресов и файлов шаблонов. Давайте сделаем это сейчас.

### Просмотры

Файл `views.py` управляет тем, как отображается содержимое модели базы данных. Поскольку мы хотим вывести список всех книг, мы можем использовать встроенный общий класс `ListView`<sup>27</sup>. Обновите файл `books/views.py`.

<sup>27</sup><https://docs.djangoproject.com/en/4.0/ref/class-based-views/generic-display/#django.views.generic.list.ListView>.

**Код**


---

```
# books/views.py из
django.views.generic import ListView

из .models импортная книга

клас с BookListView (ListView): модель
    = Книга
    template_name = "book_list.html"
```

---

В верхних строках импортируем ListView и нашу модель Book. Затем мы создаем класс BookListView, указываящий используемую модель и еще не созданный шаблон.

Еще два шага, прежде чем у нас будет рабочая веб-страница с созданным нашим шаблоном и настройте наши URL-адреса. Начнем с URL.

URL-адреса

Нам нужно настроить файл urls.py на уровне проекта, а затем один в приложении «Книги». Когда пользователь просматривает наш сайт, он сначала взаимодействует с файлом django\_project/urls.py, поэтому давайте сначала настроим его. Добавьте импорт включения во второй строке, а затем новый путь для приложения «Книги».

**Код**


---

```
# django_project/urls.py из
django.contrib import admin from
django.urls путь импорта, включите # новый

urlpatterns =
    [ path("admin/", admin.site.urls), path("", include("books.urls")), # новый
]
```

---

Две верхние строки импортируют встроенное приложение администратора, путь для наших маршрутов и включение, которые будут использоваться языком нашего приложения книг. Мы используем пустую строку «» для маршрута приложения «Книги», что означает, что пользователь на главной странице будет перенаправлен непосредственно в приложение «Книги».

Теперь мы можем настроить наш файл `books/urls.py`. Но, упс! Django по какой-то причине не включает файл `urls.py` по умолчанию приложения, поэтому нам нужно создать его самостоятельно. В текстовом редакторе создайте новый файл с именем `books/urls.py` и обновите его следующим образом:

Код

---

```
# books/urls.py из
 пути импорта django.urls
из .views импортировать BookListView
```

```
urlpatterns =
    [путь("", BookListView.as_view(), name="home"),
]
```

---

Мы импортируем наш файл представлений, настраиваем `BookListView` на пустую строку "" и добавляем [именованный URL<sup>28</sup>](#), дома, в качестве наилучшей практики.

Теперь, когда пользователь переходит на домашнюю страницу нашего веб-сайта, он сначала нажимает на файл `django_project/urls.py`, а затем перенаправляется на `books/urls.py`, в котором указано использование `BookListView`. В этом файле представления модель `Book` используется вместе с `ListView` для вывода списка всех книг.

## Шаблоны

Последним шагом является создание нашего файла шаблона, который управляет макетом на фактической веб-странице.

Мы уже указали его имя как `book_list.html` в нашем представлении. Есть два варианта его расположения по умолчанию: среди шаблонов Django будет искать шаблоны в нашем приложении книг в следующем месте: `books/templates/books/book_list.html`. Мы также могли бы вместо этого создать отдельный каталог шаблонов на уровне проекта и обновить наш файл `django_project/settings.py`, чтобы он указывал на него.

Какой из них вы в конечном итоге будете использовать в своих проектах, зависит от личных предпочтений. Мы будем использовать структуру по умолчанию здесь.

Начните с создания новой папки шаблонов в приложении «Книги», а в ней — папки «Книги». Это можно сделать из оболочки терминала. Если локальный сервер работает, используйте команду `Control+c`, чтобы остановить его.

---

<sup>28</sup><https://docs.djangoproject.com/en/4.0/topics/http/urls/#название-url-шаблонов>

## Глава 3: Веб-сайт библиотеки

Ракурска

---

```
(.venv) > книг и/шаблоны mkdir (.venv) >
книги и/шаблоны/книги mkdir
```

---

В текстовом редакторе создайте новый файл с именем books/templates/books/book\_list.html. Он будет содержать следующий код:

HTML

---

```
<!-- books/templates/books/book_list.html --> <h1>Все книги</h1>
h1> {% для книг в book_list %} <ul>
```

---

```
<li>Заголовок: {{ book.title }}</li>
<li>Подзаголовок: {{ book.subtitle }}</li> <li>Автор:
{{ book.author }}</li> <li>ISBN: {{ book.isbn }}</li>
</ul> {% endfor %}
```

---

Django предоставляет язык шаблонов<sup>29</sup>, что позволяет использовать базовую логику. Здесь мы используем `for`<sup>30</sup> тег, чтобы просмотреть все доступные книги. Теги шаблона должны быть заключены в открывающие/закрывающие скобки и круглые скобки. Таким образом, формат будет `{% for ... %}`, а затем мы должны закрыть наш цикл позже с `{% endfor %}`.

Мы заинтересованы в объекте, содержащем все доступные книги в нашей модели, либо без предотвращения `ListView`. Имя этого объекта — `<model>_list`, что учитывает, что наша модель называется `book`, означает, что это `book_list`. Поэтому для перебора каждой книги мы пишем `{% for book in book_list %}`. А затем отобразите каждое поле из нашей модели.

Теперь мы можем с новой запустить локальный сервер Django.

---

<sup>29</sup><https://docs.djangoproject.com/en/4.0/ref/templates/language/> <sup>30</sup><https://docs.djangoproject.com/en/4.0/ref/templates/builtins/#std:templatetag-for>

Ракурска

(.venv) > сервер запуска python manage.py

Перейдите на домашнюю страницу по адресу <http://127.0.0.1:8000/>, чтобы увидеть нашу работу.



## All books

- Title: Django for Beginners
- Subtitle: Build Websites with Python and Django
- Author: William S. Vincent
- ISBN: 978173546720
- Title: Django for APIs
- Subtitle: Build web APIs with Python and Django
- Author: William S. Vincent
- ISBN: 978173546722
- Title: Django for Professionals
- Subtitle: Production websites with Python & Django
- Author: William S. Vincent
- ISBN: 978173546723

Веб-страница книги

Если мы добавим дополнительные книги в админку, каждая из них тоже появится здесь.

## Тесты

Тесты являются жизненно важной частью написания программного обеспечения, и мы должны добавить их сейчас, прежде чем перейти к части API этого проекта. Мы хотим быть уверены, что модель Book работает должным образом, а также наше представление, URL-адреса и шаблон. В нашем приложении для книг уже есть пустой файл `books/tests.py`, который мы можем использовать для этого.

## Код

---

```
# books/tests.py из
django.test import TestCase из django.urls
import reverse

из .models импорт книга

класс BookTests (TestCase):
    @classmethod def
    setUpTestData (cls): cls.book =
        Book.objects.create (
            title="Хороший заголовок",
            subtitle="Отличный подзаголовок",
            author="Том Кристи", isbn="1234567890123",
        )

    def test_book_content(self):
        self.assertEqual(self.book.title, "Хорошее название")
        self.assertEqual(self.book.subtitle, "Отличный подзаголовок")
        self.assertEqual(self.book.author, "Том Кристи") self.assertEqual(self.book.isbn,
        "1234567890123")

    def test_book_listview(self): response =
        self.client.get(reverse("home"))
        self.assertEqual(response.status_code, 200)
        self.assertContains(ответ, "отличный подзаголовок")
        self.assertTemplateUsed(ответ, "книги /book_list.html")
```

---

В верхней части файла мы импортируем класс TestCase Django, reverse и используем, чтобы мы могли подтвердить, что мы используем именованный URL-адрес и нашу единственную модель Book.

Затем мы создаем класс BookTests и заполняем setUpTestData фиктивной информацией для книги. Все тесты должны начинаться с имени test\_, чтобы Django запускал их, поэтому мы создаем тест test\_book\_content для проверки содержимого книги, который выполняет assertEquals для каждого поля. Затем мы тестируем наше представление с помощью test\_book\_listview, которое проверяет, что ответ использует именованный URL-адрес «home», возвращающий код состояния HTTP 200, с ожидаемым текстом и использует в книге /book\_list.html.

Убедитесь, что локальный сервер не запущен, а затем используйте команду оболочки python manage.py.

test для выявления тестов.

Ракушка

---

```
(.venv) > python manage.py test Создание тестовой
базы данных для псевдонима "по умолчанию...
Проверка системы не выявила проблем (0 отклонено).
```

..

---

Провел 2 теста за 0,007 с.

х окошко

---

Уничтожение тестовой базы данных для псевдонима "по умолчанию..."

---

Все проходит! Отлично, мы можем двигаться дальше с нашим проектом.

## Гит

Всякий раз, когда мы добавляем новый код, рекомендуется отслеживать наш проект с помощью Git. Убедитесь, чтобы вы установили локальный сервер с помощью Control+C. Затем запустите git init, чтобы инициализировать новый репозиторий, и git status, чтобы проверить его состояние.

Ракушка

---

```
(.venv) > git init (.venv) > git
status На главной ветке
```

Ещё нет коммитов

Но есть лежащие файлы:

```
(используйте "git add <file>...", чтобы указать, что будет зафиксировано)
.venv/
КНИГИ/
db.sqlite3
django_project/
управляемый.py
```

---

ничего не добавлено для фиксации, но приставка не отслеживаемые файлы (используйте «git add» для отслеживания)

---

На данный момент включена виртуальная среда `venv`, что не рекомендуется, поскольку она может содергать секретную информацию, такую как ключи API, которые мы не хотим отслеживать. Чтобы исправить это, создайте в текстовом редакторе новый файл с именем `.gitignore` в каталоге уровня проекта рядом с `manage.py`. Файл `.gitignore` сообщает Git, что следующие должны быть忽略ированы. Добавьте одну строку для `venv`.

---

```
.gitignore
_____
.venv/
```

---

Если вы с нова запустите `git status`, вы увидите, что `venv` больше нет. Он был «прогнорирован» Git. Однако нам нужна запись обо всех пакетах, установленных в виртуальной среде. В настоящий момент рекомендуется запустить команду `pip freeze > requirements.txt` для вывода содержимого нового файла с именем `requirements.txt`.

Ракушка

---

```
(.venv) > заморозка пипса > requirements.txt
```

---

Добавьте добавим вьюшку работу с помощью команды `add -A`, а затем зафиксируем изменения вместе с сообщением (`-m`), описывая, что изменилось.

Ракушка

---

```
(.venv) > git add -A (.venv)
> git commit -m "первоначальная фиксация"
```

---

## Заключение

Эта глава показывает настройку традиционного проекта Django. Мы прошли стандартные этапы создания нового проекта, добавления нового приложения, а затем обновления моделей, представлений, URL-адресов и шаблонов. Файл `admin.py` должен быть обновлен, чтобы мы могли видеть наш новый контент, и мы добавили тесты, чтобы убедиться, что наш код работает, и мы можем добавлять новые функции, не беспокоясь о ошибке.

В следующей главе мы добавим Django REST Framework и посмотрим, как быстро традиционная Django веб-сайт может быть преобразован в веб-API.

## Г лава 4: API библиотеки

Веб-сайт нашей библиотеки в настоящий момент состоит из одной страницы, на которой отображаются все книги в базе данных.

Чтобы преобразовать его в веб-API, мы установим Django REST Framework и создадим новый URL-адрес, который действует как конечная точка API, выводящая все доступные книги. Если вы помните из главы 2, веб-API не выводит традиционную веб-страницу с HTML, CSS и JavaScript. Вместо этого оно просто чистые данные (часто в формате JSON) и определяющие их HTTP-глаголы, указывающие, какие действия пользователя разрешены. В этом случае пользователь API может только читать контент, он не может его каким-либо образом обновлять, хотя мы знаем, как это делать, в следующих главах.

### Джанго REST Framework

Как мы видели в главе 1, добавление Django REST Framework ничем не отличается от установки любого другого стороннего приложения. Обязательно закройте локальный сервер с помощью `Control+c`, если он все еще работает. Затем в командной строке введите следующее.

Ракушка

---

```
(.venv)> python -m pip установить djangorestframework~=3.13.0
```

---

Мы должны официально уведомить Django о новой установке в нашем файле `django_project/settings.py`.

Прокрутите вниз до раздела `INSTALLED_APPS` и добавьте `rest_framework`. мне нравится делать различие между сторонними приложениями и локальными приложениями, поскольку в большинстве проектов количество приложений высокое.

## Код

---

```
# django_project/settings.py
INSTALLED_APPS = [
    "django.contrib.admin",
    "django.contrib.auth",
    "django.contrib.contenttypes",
    "django.contrib.sessions",
    "django.contrib.messages",
    "django.contrib.staticfiles", # 3rd party
    "rest_framework", # новый # Локальный

    "books.apps.BooksConfig",
]
```

---

В конечном итоге наш веб-API предоставит единую конечную точку, в которой перечислены все книги в формате JSON. Для этого нам понадобится новый URL-маршрут, новое представление и новый файл сериализатора (подробнее об этом чуть позже).

Существует несколько способов организации этих файлов. Многие профессиональные разработчики Django просто включают логику API в соответствующее приложение, помешав URL-адрес с префиксом /api/. А пока, чтобы отделить логику API от традиционной логики Django, мы создадим специальное приложение API для нашего проекта.

Давайте сделаем это сейчас, ис пользуя команду `startapp`. Помните, что приложения всегда должны иметь имя во множественном числе, поскольку в противном случае Django автоматически добавит `s` — это имя администратора и другие места расположения.

## Ракушка

---

```
(.venv) > python manage.py startapp API
```

---

Затем добавьте его в `INSTALLED_APPS` в нашем разделе «Локальные».

Код

---

```
# django_project/settings.py
INSTALLED_APPS = [
    "django.contrib.admin",
    "django.contrib.auth",
    "django.contrib.contenttypes",
    "django.contrib.sessions",
    "django.contrib.messages",
    "django.contrib.staticfiles", # 3rd party
    "rest_framework", # Локальный

    "books.apps.BooksConfig",
    "apis.apps.ApisConfig", # новый
]
```

---

Приложение API не будет иметь собственных моделей баз данных, поэтому нет необходимости создавать файл миграций и выполнять миграции для обновления базы данных. На самом деле модели базы данных — это единственная область, которую нам вообще не нужно трогать, поскольку этот новый веб-API предназначен для представления существующих данных, а не для создания новых данных.

URL-адреса

Давайте начнем с наших конфигураций URL. Добавление конечной точки API аналогично настройке традиционного URL-маршрута Django. В файле django\_project/urls.py на уровне проекта включите приложение API и настройте его URL-маршрут, который будет находиться в api/.

## Глава 4: API библиотеки

Код

---

```
# django_project/urls.py из
django.contrib import admin from
django.urls путем импорта, включая

urlpatterns =
    [ path("admin/", admin.site.urls), path("api/",
        include("apis.urls")), # new path("", include("books.urls")),

    ]
```

---

Затем создайте новый файл с именем apis/urls.py в текстовом редакторе. Этот файл будет импортировать будущее представление с именем BookAPIView и установите для него URL-маршрут «», чтобы он отображался в api/. Как всегда, мы также добавим к нему имя book\_list, которое поможет в будущем, когда мы захотим сопоставить янаэ тот конкретный маршрут.

Код

---

```
# apis/urls.py из
пути импорта django.urls

из .views импортировать BookAPIView

urlpatterns =
    [путь("", BookAPIView.as_view(), name="book_list"),
    ]
```

---

Всего.

## Промтры

В традиционных представлениях Django ис пользуясь ядрами тройки того, какие данные отправлять в шаблоны. Представления Django REST Framework аналогичны, за исключением того, что конечным результатом являются сериализованные данные в формате JSON, а не содержимое веб-страницы! Представления Django REST Framework зависят от модели, URL-адреса и нового файла, называемый сериализатором, который мы увидим в следующем разделе.

Существуют общие представления Django REST Framework для различных случаев использования и мы будем использовать [ListAPIView](#)<sup>31</sup>. Здесь, чтобы отобразить все книги.

---

<sup>31</sup><http://www.django-rest-framework.org/api-guide/generic-views/#listapiview>

## Глава 4: API библиотеки

Во избежание путаницы некоторые разработчики называют файл представлений API `apiviews.py` или `api.py`. Лично я работаю в выделенном приложении API, не находя запутанным просто вызывать файл представлений Django REST Framework `views.py`, но мнения по этому поводу различаются.

Обновите файл `apis/views.py`, чтобы он выглядел следующим образом:

Код

---

```
# apis/views.py из
rest_framework import generics

from books.models import Книга
из .serializers import BookSerializer

класс BookAPIView(generics.ListAPIView):
    queryset = Book.objects.all()
    serializer_class = BookSerializer
```

---

В верхних строках мы импортировали `дженерик` и Django REST Framework32. класс представлений, модель Book из нашего приложения `books` и сериализаторы из нашего API - приложения. Мы создадим используемый здесь сериализатор `BookSerializer` в следующем разделе.

Затем мы создаем класс представления с именем `BookAPIView`, который использует `ListAPIView` для создания конечной точки только для чтения для всех экземпляров книг. Доступно множество общих представлений, и мы рассмотрим их подробнее в следующих главах.

С нашей точки зрения требуется явно описать набор запросов, то есть все доступные книги, а затем `serializer_class`, который будет `BookSerializer`.

## Сериализаторы

Мы находимся на этапе! На данный момент мы создали файл `urls.py` и файл `views.py` для нашего API.

Последним, но самым важным действием является создание нашего сериализатора.

`Сериализатор`33 \_ переводит сложные данные, такие как наборы запросов и экземпляры моделей, в формат, который легкодоступен через Интернет, обычно JSON. Также возможно «десериализовать» данные буквально

---

32`https://www.djangoproject.org/api-guide/generic-views/#generic-views` 33`https://www.djangoproject.org/api-guide/serializers/`

## Глава 4: API библиотеки

тот же процесс в обратном порядке, при котором данные JSON с начала проверяются, а затем преобразуются в словарь.

Настоящая часть Django REST Framework заключается в его сериализаторах, которые избавляют нас от большей части ложности. Мы рассматриваем сериализацию JSON более подробно в следующих главах, но сейчас цель состоит в том, чтобы продемонстрировать, как только создать сериализатор с помощью Django REST. Фреймворк.

В текстовом редакторе создайте новый файл с именем `apis/serializers.py` и обновите его следующим образом:

Код

---

```
# apis/serializers.py из
сериализаторов импорта rest_framework

from books.models импорт Книга

класс BookSerializer(сериалайзеры.ModelSerializer):
    Метакласс:
        модель = Книга
        fields = ("название", "подзаголовок", "автор", "isbn")
```

---

В верхних строках мы импортируем класс сериализаторов Django REST Framework и модель Book из нашего приложения `books`. Далее мы расширяем `ModelSerializer`<sup>34</sup> Django REST Framework. в класс `BookSerializer`, который определяет нашу модель базы данных, книгу и поля базы данных, которые мы хотим предоставить: заголовок, подзаголовок, автор и номер телефона.

И это все! Были сделаны. Создав новый маршрут URL, новое представление и класс сериализатора, мы создали конечную точку API для веб-сайта нашей библиотеки, которая будет отображать все существующие книги в формате списка.

Документный для просмотра API

Необработанные данные JSON не особенно удобны для восприятия человеческим глазом. К счастью Django REST Framework предоставляет ясноестроенным API с возможностью запроса, который отображает как контент, так и HTTP-команды, связанные с данной конечной точкой. Чтобы увидеть его в действии, запустите локальный веб-сервер с командой запуска сервера.

---

<sup>34</sup><https://www.djangoproject.com/api-guide/serializers/#modelserializer>

Ракурска

(.venv) &gt; сервер запуска python manage.py

Мы знаем, что конечная точка нашего API находится по адресу <http://127.0.0.1:8000/api/>, поэтому перейдите туда в веб-браузере.

The screenshot shows a browser window titled "Book Api - Django REST frame". The address bar shows the URL [127.0.0.1:8000/api/](http://127.0.0.1:8000/api/). The main content area is titled "Django REST framework" and "Book Api". Below it, there is a "GET /api/" button. To the right of the button are "OPTIONS" and "GET" buttons. The response body shows the following JSON data:

```

HTTP 200 OK
Allow: GET, HEAD, OPTIONS
Content-Type: application/json
Vary: Accept

[
    {
        "title": "Django for Beginners",
        "subtitle": "Build Websites with Python and Django",
        "author": "William S. Vincent",
        "isbn": "978173546720"
    },
    {
        "title": "Django for APIs",
        "subtitle": "Build web APIs with Python and Django",
        "author": "William S. Vincent",
        "isbn": "978173546722"
    },
    {
        "title": "Django for Professionals",
        "subtitle": "Production websites with Python & Django",
        "author": "William S. Vincent",
        "isbn": "978173546723"
    }
]

```

Книжный API

И посмотрите на! Django REST Framework предоставляет эту визуализацию, и это по умолчанию. Он отображает код состояния HTTP для страницы, который равен 200, что означает «OK». Указывает, что Content-Type — это JSON. И отображает информацию о нашей отдельной записи книги в отформатированном виде.

Если вы нажмете кнопку «Получить» в правом верхнем углу и выберите «json» в верхней части раскрывающегося списка, вы увидите, как выглядит необработанная конечная точка API.

## Глава 4: API библиотеки



[Забронировать API JSON](#)

Не очень привлекательно, не так ли? Данные вообще не форматированы, и мы также не видим никакой дополнительной информации о статусе HTTP или допустимых заголовках. Я думаю мы можем согласиться с тем, что версия Django REST Framework более привлекательна.

Профессиональные разработчики обычно используют сторонний инструмент, такой как [Postman](#)<sup>35</sup>, или, если на Mac, [Paw](#)<sup>36</sup> для тестирования и использования API. Но для наших целей в этой книге встроенный API с возможностью просмотра более чем достаточно.

## Тесты

Тестирование в Django основано на встроенным в Python [unittest](#)<sup>37</sup>, модуль и несколько полезных расширений для Django. В частности, Django поставляется с [тестовым клиентом](#)<sup>38</sup>, которые мы можем использовать для имитации запросов GET или POST, проверки цепочки перенаправлений веб-запроса и проверки того, что данный шаблон Django используется и имеет правильные данные контекста шаблона.

Django REST Framework предоставляет [несколько дополнительных вспомогательных классов](#)<sup>39</sup>, которые расширяют существующую среду Django. Одним из них является `APIClient`, расширение стандартного клиента Django, которое мы будем использовать для тестирования извлечения данных API из нашей базы данных.

Поскольку у нас есть тесты в `books/tests.py` для нашей модели Book, мы можем соредакториться на тестировании конечной точки API, в частности на том, что она использует ожидаемый URL, имеет правильный код состояния 200 и содержит правильный контент.

Откройте файл `apis/tests.py` в текстовом редакторе и введите следующий код, который мы обзор ниже.

<sup>35</sup><https://www.postman.com/>

<sup>36</sup><https://paw.cloud/>

<sup>37</sup><https://docs.python.org/3/library/unittest.html#module-unittest>

<sup>38</sup><https://docs.djangoproject.com/en/4.0/topics/testing/tools/#the-test-client>

<sup>39</sup><https://www.djangoproject-rest-framework.org/api-guide/testing/>

Код

---

```
# apis/tests.py из
django.urls import reverse from
rest_framework import status from
rest_framework.test import APITestCase

from books.models импорт Книга

class APITests(APITestCase):
    @classmethod def
    setUpTestData(cls): cls.book =
        Book.objects.create( title="Django для
            API", subtitle="Создание веб-API с
            помощью Python и Django", author="William S. Винсент",
            isbn="9781735467221",

    )

    def test_api_listview(self): response =
        self.client.get(reverse("book_list"))
        self.assertEqual(response.status_code, status.HTTP_200_OK)
        self.assertEqual(Book.objects.count(), 1) self.assertContains (ответ, с одной книгой)
```

---

Вверху мы импортируем reverse из Django и из Django REST Framework как статус, так и APITestCase. Мы также импортируем нашу модель Book, хотя обратите внимание, что, поскольку мы находимся в приложении API, мы должны указать имя приложения book для его импорта.

Мы расширяем APITestCase в новом классе под название APITests, который начинается с настройки данных настройки. Затем мы запускаем четыре равные проверки. Сначала мы проверяем, что именованный URL-адрес «book\_list» используется языком пользователя. Во-вторых, мы подтверждаем, что код состояния HTTP соответствует 200. В-третьих, мы проверяем наличие единственной записи в базе данных. И, наконец, мы подтверждаем, что ответ содержит все данные из созданного нами объекта книги.

Обязательно установите локальный сервер и запустите тест, чтобы убедиться, что он прошел.

## Глава 4: API библиотеки

Ракушка

---

```
(.venv) > python manage.py test Создание тестовой
базы данных для псевдонима "по умолчанию...".
Проверка с исключением не выявила проблем (0 отключено).
...
Провел 3 теста за 0,009 с.
```

х оправ

---

```
Уничтожение тестовой базы данных для псевдонима "по умолчанию..."
```

---

Обратите внимание, что в этих одних данных описываются прохождение трех тестов, потому что у нас было два в books/tests.py и один здесь. На больших веб-сайтах с сомнениями или даже тысячами тестов производительность может стать проблемой, и иногда вы можете проверить только тест в данном приложении, прежде чем запустить полный набор тестов веб-сайта. Для этого запрос добавьте название приложения, которое хотите проверить, в конец

теста `python manage.py`.

Ракушка

---

```
(.venv) > python manage.py test apis Создание тестовой
базы данных для псевдонима «по умолчанию»...
Проверка с исключением не выявила проблем (0 отключено).
```

---

Выполнить 1 тест за 0,005 с

х оправ

---

```
Уничтожение тестовой базы данных для псевдонима "по умолчанию..."
```

---

## Развертывание

Развертывание веб-API почти идентично развертыванию традиционного веб-сайта. В этой книге мы будем использовать Heroku, так как он предоставляет бесплатный уровень и является широко используемой платформой как услугой, которая уделяет большую часть с функциями, связанными с развертыванием.

Если вы впервые используете Heroku, вы можете зарегистрироваться бесплатно учетную запись на их [веб-сайте](https://www.heroku.com/)<sup>40</sup>. После заполнения регистрационной формы дождитесь письма с подтверждением для подтверждения вашей учетной записи. Будет

---

<sup>40</sup><https://www.heroku.com/>

приведет вас на страницу настройки пароля и после настройки вы будете перенаправлены в раздел панели инструментов сайта Heroku. Теперь Heroku также требует регистрацию в многофакторной аутентификации (MFA), которую можно выполнить с помощью SalesForce или такого инструмента, как Google Authenticator.

Мы будем использовать интерфейс командной строки Heroku (CLI), чтобы мы могли выполнять развертывание из командной строки. В настоящие времена работаем в виртуальной среде для нашего проекта библиотеки, но мы хотим, чтобы Heroku был доступен по всему миру, то есть везде на нашей машине. Самый простой способ сделать это — открыть новую вкладку командной строки — Control+T в Windows, Command+T в Mac, которая не работает в виртуальной среде. Все, что здесь установлено, будет глобальным.

В Windows см. [страницу командной строки Heroku 41](#), для правильной установки 32-битной или 64-битной версии. На Mac менеджер пакетов [Homebrew42](#) использует языковые установки. Если это еще не сделано на вашем компьютере, установите Homebrew, скопировав и вставив длинную команду на веб-сайте Homebrew в командную строку и нажав «Return». Это будет выглядеть примерно так:

Ракушка

---

```
% /bin/bash -c "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install.sh)"
```

---

Затем установите CLI Heroku, скопировав и вставив следующее в командную строку и нажав

Возвращаться

Ракушка

---

```
% варить кран героку/варить && варить установить героку
```

---

Если вы работаете на новом компьютере Apple с чипом M1, вы можете получить ошибку с чем-то вроде Bad CPU type в исполняемом файле. Установка [Розетты 243](#) решит проблему.

После завершения установки вы можете закрыть новую вкладку командной строки и вернуться к исходному окну вкладки с активной виртуальной средой страницы.

Чтобы убедиться, что установка работает правильно, запустите `heroku --version`. Должен быть вывод с установленной текущей версией Heroku CLI.

---

<sup>41</sup><https://devcenter.heroku.com/articles/heroku-cli#download-and-install>

<sup>42</sup><https://brew.sh/> <sup>43</sup><https://support.apple.com/en-us/HT211861>

## Глава 4: API библиотеки

Ракушка

---

```
(.venv) > heroku --версияheroku/
7.59.2 darwin-x64 node-v12.21.0
```

---

Если вы видите здесь сообщение об ошибке в VSCode для Windows о том, что «термин 'heroku' не распознан...», скорее всего, это проблема с разрешениями. Попробуйте напрямую открыть приложение PowerShell и выполнить heroku --version. Он должен работать правильно. К сожалению, время от времени в терминальной оболочке VSCode возникают некоторые тонкие проблемы.

Если вы получите «Предупреждение» о том, что ваша версия Heroku устарела, попробуйте запустить обновление Heroku. Для установки последней версии.

Как только вы увидите установленную версию Heroku, введите команду heroku login и используйте адрес электронной почты и пароль для Heroku, которые вы только что установили.

Ракушка

---

```
(.venv) > вх од на heroku
Ведите свои учетные данные Heroku:
Электронная почта: will@wsvincent.com
Пароль: ****
Вы вошли как will@wsvincent.com
```

---

Возможно, вам потребуется подтвердить свои учетные данные на веб-сайте Heroku, но как только оболочка терминала подтвердит ваш вход в систему, вы будете готовы продолжить.

## Статические файлы

[Статические файлы](#)<sup>44</sup> несколько ложны для правильного развертывания в проектах Django, но с орошаемостью заключается в том, что процесс для API Django, по сути, такой же. Несмотря на то, что на данный момент у нас нет собственных, есть статические файлы, включенные в Django admin и API для просмотра Django REST Framework, по тому для них правильного развертывания мы должны настроить все статические файлы.

Сначала нам нужно создать выделенный статический каталог.

---

<sup>44</sup><https://docs.djangoproject.com/en/4.0/howto/static-files/>

## Глава 4: API библиотеки

Ракушка

`(.venv) > mkdir статический`

Git не будет отслеживать пустые каталоги, поэтому важно добавить файл .keep, чтобы статический каталог был включен в систему управления версиями. Сделайте это прямо сейчас в текстовом редакторе.

Затем мы установим [WhiteNoise](#)<sup>45</sup>. package, так как Django не поддерживает об служивание статических файлов в производстве.

Ракушка

`(.venv) > python -m pip install whitenoise == 6.0.0`

WhiteNoise необходимо добавить в django\_project/settings.py в следующих местах:

- белый шум над django.contrib.staticfiles в INSTALLED\_APPS
- WhiteNoiseMiddleware выше CommonMiddleware
- Конфигурация STATICFILES\_STORAGE, указывающая на WhiteNoise.

Код

```
# django_project/settings.py
INSTALLED_APPS = [
    ...
    "whitenoise.runserver_nostatic", # новый
    "django.contrib.staticfiles",
]

[

    "django.middleware.security.SecurityMiddleware",
    "django.contrib.sessions.middleware.SessionMiddleware",
    "whitenoise.middleware.WhiteNoiseMiddleware", # new
    ...
]

STATIC_URL = "/статический/"
STATICFILES_DIRS = [BASE_DIR / "static"] # новый
STATIC_ROOT = BASE_DIR / "staticfiles" # новый
STATICFILES_STORAGE =
    "whitenoise.storage.CompressedManifestStaticFilesStorage" # новый
```

---

<sup>45</sup><http://whitenoise.evans.io/en/stable/>

Последним шагом является запуск команды `collectstatic` в первый раз, чтобы скомпилировать все каталоги и файлы статических файлов в один автономный модуль, пригодный для развертывания.

Ракушка

---

```
(.venv) > python manage.py collectstatic
```

---

Всегда готово. Теперь, когда наши статические файлы настроены правильно, нам не нужно много думать о них в будущем!

### Контрольный список развертывания

Для базового развертывания есть пять пунктов в контрольном списке развертывания:

- установить [Gunicorn<sup>46</sup>](#) как производственный веб-сервер
- создать файл `requirements.txt`
- создать файл `runtime.txt`
- обновить конфигурацию `ALLOWED_HOSTS`
- создать `Procfile` для Heroku

Встроенный веб-сервер Django подходит для локального тестирования, но либо Gunicorn, либо [uWSGI<sup>47</sup>](#) следует использовать в производстве. Поскольку Gunicorn проще в использовании, это будет наш выбор. Установите его через Pip.

Ракушка

---

```
(.venv) > python -m pip install gunicorn~=20.1.0
```

---

В предыдущей главе мы создали файл `requirements.txt`, но с тех пор мы установили Django.

REST Framework и Gunicorn в нашей виртуальной среде. Ни то, ни другое не отражено в текущем файле. Достаточно просто снова запустить команду `> для обновления`.

Это

---

<sup>46</sup><https://gunicorn.org/>

<sup>47</sup><https://uwsgi-docs.readthedocs.io/en/latest/>

Ракушка

---

(.venv)> python -m pip замораживание> требования.txt

---

Третий шаг — с создать файл runtime.txt в корневом каталоге рядом с файлом requirements.txt, в котором указывается какую версия Python запускать на Heroku. Если не указано явно, это то в [настоящее время установлено 48](#) к среде выполнения Python-3.9.10, но со временем меняется.

Поскольку мы используем Python 3.10, мы должны создать специальный файл runtime.txt<sup>49</sup>. файл, чтобы использовать его. В текстовом редакторе создайте этот новый файл runtime.txt на уровне проекта, что означает, что он находится в том же каталоге, что и файл manage.py. На момент написания этой статьи последняя версия — 3.10.2. Убедитесь, что все в нижнем регистре!

время выполнения.txt

---

python-3.10.2

---

Четвертый шаг — обновить ALLOWED\_HOSTS. По умолчанию он принимает все хосты, но мы хотим ограничить доступ к сайту и API. Мы хотим иметь возможность использовать локальный хост, либо 127.0.0.1. локально, и мы также знаем, что любой сайт Heroku заканчивается на .herokuapp.com. Добавьте все три хоста в наш конфигурационный ALLOWED\_HOSTS.

Код

---

```
# django_project/settings.py
ALLOWED_HOSTS = [".herokuapp.com", "localhost", "127.0.0.1"]
```

---

И последний шаг в вашем текстовом редакторе — создать новый Procfile в корневом каталоге проекта рядом с файлом manage.py. Это файл с помощью которого мы будем запускать наш веб-сайт на Heroku, который содержит инструкции по запуску нашего веб-сайта. Мы говорим ему использовать Gunicorn в качестве веб-сервера, использовать конфигурацию WSGI в django\_project.wsgi, а также выводить файлы журнала, которые являются необязательными, но полезными дополнительными настройками.

---

<sup>48</sup><https://devcenter.heroku.com/articles/python-support#specifying-a-python-version><sup>49</sup><https://devcenter.heroku.com/articles/python-runtimes>

## Профайл

---

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

---

Все готово. Добавьте и зафиксируйте наши новые изменения в Git.

## Ракушка

---

```
(.venv) > git status (.venv)
> git add -A (.venv) > git
commit -m "Новые обновления для развертывания Heroku"
```

---

## Гитхаб

Также рекомендуется хранить коду хостинг-провайдера, такого как GitHub, GitLab или BitBucket.

GitHub очень популярен и предоставляет щедрый бесплатный уровень, поэтому мы будем использовать его в этой книге. Ты можешь создать бесплатную учетную запись на сайте.

После настройки [создайте новый репозиторий](#) называется библиотекой и обязательно выберите переключатель «Частная».

Затем нажмите на кнопку «Создать репозиторий». На следующей странице прокрутите вниз до места, где написано «...или нажмите существующий репозиторий из командной строки». Скопируйте и вставьте две команды в свой терминал.

Это должно выглядеть так, как показано ниже, хотя вместе с wsvincent в качестве имени пользователя это будет ваш GitHub.

имя пользователя

## Ракушка

---

```
(.venv) > git удаленное добавление источника https://github.com/wsvincent/library.git (.venv) >
git push -u origin main
```

---

## Героку

Последний шаг — создать новый проект на Heroku и把他 в него наш код. Вы уже должны войти в Heroku через командную строку из предыдущей главы.

---

<sup>50</sup><https://github.com/новый>

Вы можете либо запустить `heroku create`, и Heroku с случайным образом присвоит имя вашему проекту, либо вы можете указать собственное имя, но оно должно быть уникальным для вас в Heroku! Так что чем больше, тем лучше. Я звоню с веббиблиотеку `wsvincent`. Префикс вашего имени пользователя GitHub — это то, что оно будет сказывать об этом, чтобы вы можете указать имя своего проекта Heroku, хотя вы всегда можете изменить его позже.

на тоже.

#### Ракурс

---

```
(.venv) > heroku create wsvincent-library Сздание
библиотеки wsvincent... сделано https://wsvincent-
library.herokuapp.com/ | https://git.heroku.com/wsvincent-library.git
```

---

Затем мы отправим код в сам Heroku и добавим веб-процесс, чтобы динамометр заработал.

#### Ракурс

---

```
(.venv) > git push heroku main (.venv) >
heroku ps:scale web=1
```

---

URL-адрес вашего нового приложения будет в выводе командной строки, или вы можете запустить `heroku open`, чтобы найти это.

Вот домашняя страница моей библиотеки.



## All books

- Title: Django for Beginners
- Subtitle: Build Websites with Python and Django
- Author: William S. Vincent
- ISBN: 978173546720
  
- Title: Django for APIs
- Subtitle: Build web APIs with Python and Django
- Author: William S. Vincent
- ISBN: 978173546722
  
- Title: Django for Professionals
- Subtitle: Production websites with Python & Django
- Author: William S. Vincent
- ISBN: 978173546723

[Домашняя страница библиотеки](#)

А также конечная точка API в `/api/`.

```

HTTP 200 OK
Allow: GET, HEAD, OPTIONS
Content-Type: application/json
Vary: Accept

[
    {
        "title": "Django for Beginners",
        "subtitle": "Build Websites with Python and Django",
        "author": "William S. Vincent",
        "isbn": "978173546720"
    },
    {
        "title": "Django for APIs",
        "subtitle": "Build web APIs with Python and Django",
        "author": "William S. Vincent",
        "isbn": "978173546722"
    },
    {
        "title": "Django for Professionals",
        "subtitle": "Production websites with Python & Django",
        "author": "William S. Vincent",
        "isbn": "978173546723"
    }
]

```

## API библиотеки

Развертывание — с ложной темой, и мы намеренно использовали здесь не сколько сокращений. Но цель состоит в том, чтобы пройти по очень простому веб-сайту Django и API, чтобы показать, как его можно создать с нуля.

## Заключение

В этой главе мы рассмотрели много материала, так что не беспокойтесь, если что-то сейчас покажется вам немногозапутанным. Мы добавили Django REST Framework на наш существующий веб-сайт библиотеки и создали конечную точку API для наших книг. Затем мы добавили тесты и развернули наш проект на Heroku.

Однако веб-API мог уметь делать гораздо больше, чем просто отображать информацию из вашей базы данных. В следующей главе мы создадим и развернем наш собственный серверный интерфейс Todo API, который может быть легко использован любым пользователем.

Г лава 4: API библиотеки

внешний интерфейс.

# Глава 5: Todo API

В этой главе мы создадим и развернем первую часть Todo API, которая содержит как конечную точку API списка для всех задач, так и выделенные конечные точки для каждой отдельной задачи. Мы также узнаем о совместном использовании ресурсов между источниками (CORS), которое является необходиимой функцией безопасности, когда развернутому внутреннему интерфейсу необходимо взаимодействовать с внешним интерфейсом. Мы уже создали наш первый API и расмотрели, как работают HTTP и REST в абсолютном виде, но, скорее всего, вы еще не совсем понимаете, как все это сочетается друг с другом. К концу этих двух глав вы это сделаете.

## Односоставнические приложения(SPA)

SPA требуется для мобильных приложений, работающих на iOS или Android, и является доминирующим шаблоном для веб-приложений, которые хотят использовать преимущества интерфейсных технологий JavaScript, таких как React, Vue, Angular, и другие.

Использование SPA-подхода имеет множество преимуществ. Развработчики могут сосредоточиться на языке с собственной областью знаний, как правило, либо на интерфейсе, либо на сервере, но редко на обоих. Это позволяет использовать инструменты тестирования и сборки, подходящие для поставленной задачи, поскольку сборка, тестирование и развертывание проекта Django сильно отличаются от того же самого для проекта JavaScript, такого как React. Апринципиальное разделение устраняет риск сцепления; внешние изменения не могут нарушить задний конец.

Для больших команд SPA имеет большой смысл, так как уже есть встроенное разделение задач.

Даже в небольших командах с теми же принципами подхода SPA относительно невелика. Основной риск разделения с первоначальной частью и клиентской частью заключается в том, что для этого потребуется языка программирования предметной области в обеих областях.

В то время как Django относительно зрелый на данный момент, интерфейсная часть системы явно не такова. Индивидуальный разработчик должен тщательно подумать о том, стоит ли добавленная сложность выделенного интерфейса JavaScript того, чтобы добавить JavaScript в существующие шаблоны Django с помощью временных инструментов, таких как [htmx<sup>51</sup>](https://htmx.org/).

---

<sup>51</sup><https://htmx.org/>

## Начальная настройка

Первым шагом для любого Django API всегда является установка Django, а затем добавление Django REST Framework поверх него. В командной строке перейдите в каталог кода на рабочем столе и создайте обе папки todo.

Ракушка

---

```
# Окна
> cd onedrive\рабочий стол\код
> mkdir todo && cd todo

# мак OS
% cd рабочий стол/рабочий стол/
код % mkdir todo && cd todo
```

---

Затем выполните стандартные шаги по созданию новой виртуальной среды, ее активации и установке Django.

Ракушка

---

```
# Окна
> python -m venv .venv
> .venv\Scripts\Activate.ps1 (.venv) >
python -m pip install django~=4.0.0

# мак OS
% python3 -m venv .venv %
ис точник .venv/bin/activate (.venv)
% python3 -m pip install django~=4.0.0
```

---

Теперь, когда Django установлен, мы должны начать с создания традиционного проекта Django с именем django\_project, добавления в него приложения todos и переноса исходной базы данных.

## Ракушка

```
(.venv) > django-admin startproject django_project . (.venv) > python  
manage.py startapp todos (.venv) > python manage.py migrate
```

В Django нам все равно нужно добавлять новые приложения в наш параметр `INSTALLED_APPS`, поэтому сделайте это сейчас. Откройте `django_project/settings.py` в текстовом редакторе и добавьте `todos` в конец установленного

пакета.

## Код

```
# django_project/settings.py  
INSTALLED_APPS =  
    [ "django.contrib.admin",  
      "django.contrib.auth",  
      "django.contrib.contenttypes",  
      "django.contrib.sessions",  
      "django.contrib.messages",  
      "django.contrib.staticfiles",  
      # местный  
      "todos.apps.TodosConfig", # новый  
    ]
```

Если вы сейчас запустите `python manage.py runserver` в командной строке и перейдете в браузере по адресу `http://127.0.0.1:8000/`, вы увидите, что наш проект успешно установлен. Мы готовы продолжить.

## .gitignore

Поскольку мы будем использовать Git для управления версиями, важно заранее создать файл `.gitignore`, чтобы указать, что не следует отслеживать. Это включает в себя нашу новую виртуальную среду `.venv`. Чтобы решить эту проблему, создайте в текстовом редакторе новый файл с именем `.gitignore` и добавьте одну строку для `.venv`.

---

```
.gitignore
```

---



---

```
.venv/
```

---

Затем давайте инициализируем новый репозиторий Git для нашего проекта и запустим `git status`, чтобы убедиться что файл `.venv` не отображается. Мы также можем добавить в нашу работу по настройке через `git add -A` и написать наш первый коммит с сообщением.

Ракушка

---

```
(.venv) > с тагом git (.venv)
> git add -A (.venv) > git
commit -m "начальный коммит"
```

---

## Модели

Далее нужно определить модель базы данных `Todo` в приложении `todos`. Мы с ох раним базовые вещи и будем иметь только два поля `title` и `body`.

Код

---

```
# todos/models.py из
django.db импортировать модели
```

---

```
Класс Todo(models.Model):
    title = models.CharField(max_length=200)
    body = models.TextField()

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

---

Мы импортируем модели вверху, а затем подключаем их, чтобы создать нашу собственную модель `Todo`. Также добавлен метод `__str__` для представления удобочитаемого имени для каждого экземпляра будущей модели.

Поскольку мы обновили нашу модель, пришло время для двух шагов от Django: создания нового файла миграции и последующей синхронизации базы данных с изменениями каждого раза. В командной строке введите `Control+c`, чтобыстановить наш локальный сервер. Затем запустите команду `makemigrations`.

Ракурска

---

```
(.venv) > python manage.py makemigrations todos Миграц ии  
для «todos»:
```

```
todos/migrations/0001_initial.py —Создать  
модель Todo
```

---

И затем команда миграции.

Ракурска

---

```
(.venv) > python manage.py migrate Операци ии  
для выполнения
```

```
Применить все миграции: admin, auth, contenttypes, session, todos  
Выполнение миграции:
```

```
применение todos.0001_initial... OK
```

---

Не обязательно добавлять конкретное приложение, для которого мы хотим создать файл миграции — вместе с тем оно может быть введено просто `python manage.py makemigrations` — однако это хорошая практика. Файлы миграции — это фантастический способ отладки приложений, и вы должны стремиться создавать файл миграции для каждого небольшого изменения. Если бы мы обновили модели в двух разных приложениях, а затем запустили `python manage.py makemigrations`, результатирующий файл миграции содержал бы данные для обоих приложений.

Это только усугубляет отладку. Страйтесь, чтобы ваши миграции были как можно меньше.

Теперь мы можем использовать встроенное приложение администрирования Django для взаимодействия с нашей базой данных. Если бы мы сразу вошли в админку, наше приложение Todos не появилось бы. Нам нужно явно добавить его через файл `todos/admin.py`. Покажем это делаем, мы можем создать класс `TodoAdmin`, который использует `list_display`, чтобы обозначить модели, заголовок и тело, были видны.

## Код

---

```
# todos/admin.py из
django.contrib import admin

из .models импортировать Todo

класс с TodoAdmin(admin.ModelAdmin):
    list_display = ("заголовок", "тело",
    )

admin.site.register(Todo, TodoAdmin)
```

---

Вот и все! Теперь мы можем создать учетную запись суперпользователя для входа в админку.

## Работа

---

```
(.venv) > python manage.py создает суперпользователя
```

---

Снова запустите локальный сервер с помощью `python manage.py runserver` и перейдите в раздел администратора по адресу `http://127.0.0.1:8000/admin/`. Войдите в систему и нажмите «+ Добавить» рядом с `Todos`. Создайте 3 новых элемента списка задач, обязательно добавив заголовок и текст для обоих. Вот как выглядит мой:

The screenshot shows the Django administration interface for a 'Todos' model. On the left, there's a sidebar with 'AUTHENTICATION AND AUTHORIZATION' sections for 'Groups' and 'Users'. The main content area is titled 'Select todo to change' and displays three items:

- Learn HTTP**: BODY: It's important.
- Second item**: BODY: Learn Python.
- 1st todo**: BODY: Learn Django properly.

A button 'ADD TODO +' is located at the top right of the main content area. The URL in the browser bar is 127.0.0.1:8000/admin/todos/todo/.

## Тесты

Код без тестов неполный, поэтому мы должны добавить их сейчас для нашей модели Todo. Мы будем использовать `TestCase`52 Django, для создания тестовой базы данных и использования `setUpTestData` для создания тестовых данных для нашего класса `TodoModelTest`. Мы хотим подтвердить, что заголовок и текст выглядят так, как ожидалось, а также метод `__str__` в модели.

Откройте файл `todos/tests.py` и заполните его следующим образом:

52 <https://docs.djangoproject.com/en/4.0/topics/testing/tools/# testcase>

Код

---

```
# todos/tests.py из
django.test импортировать TestCase

из .models импортировать Todo

class TodoModelTest(TestCase): @classmethod
    def setUpTestData(cls): cls.todo =
        Todo.objects.create( title="First Todo",
            body="Текст здесь"
        )

    определение test_model_content (я):
        self.assertEqual(self.todo.title, "Первое задание")
        self.assertEqual(self.todo.body, "Текст здесь") self.assertEqual(str(self.todo),
            "Первое задание")
```

---

Убедитесь, что локальный сервер не запущен, набрав `Control+c` в командной строке, а затем запустите тест с помощью команды `python manage.py`.

Ракушка

---

```
(.venv) > python manage.py test Создание
тестовой базы данных для псевдонима "по умолчанию...
Проверка системы не выявила проблем (0 отключено).
..
=====
Выполнить 1 тест за 0,002 с
```

---

```
Хорошо
Уничтожение тестовой базы данных для псевдонима "по умолчанию..."
```

---

На данный момент мы закончили с традиционной частью Django нашего Todo API! Поскольку мы не утруждаем себя с созданием веб-страниц для этого проекта, все, что нам нужно, это модель и Django REST. Фреймворк позаботится об этом сам.

## Джанг о REST Framework

Чтобы добавить Django REST Framework, установите локальный сервер, нажав **Control+C**, а затем установите его с помощью Pip.

Ракушка

---

```
(.venv)> python -m pip установить djangorestframework~=3.13.0
```

---

Затем добавьте `rest_framework` в наш параметр `INSTALLED_APPS`, как и любое другое стороннее приложение. Мы также хотим начать настройку конкретных параметров Django REST Framework, которые все существуют в конфигурации с именем `REST_FRAMEWORK`, которую можно добавить в конец файла.

Для начала давайте явно установим разрешения [AllowAny](#)<sup>53</sup> который разрешает неограниченный доступ независимо от того, был ли запрос аутентифицирован или нет. В рабочей среде разрешения API контролируются в учебных целях мы будем использовать `AllowAny`.

Код

---

```
# django_project/settings.py
INSTALLED_APPS = [
    "django.contrib.admin",
    "django.contrib.auth",
    "django.contrib.contenttypes",
    "django.contrib.sessions",
    "django.contrib.messages",
    "django.contrib.staticfiles", # 3rd party
    "rest_framework", # новый # Локальный

    "todos.apps.TodosConfig",
]

REST_FRAMEWORK =
{ "DEFAULT_PERMISSION_CLASSES":
    [ "rest_framework.permissions.AllowAny",
    ],
}
```

---

<sup>53</sup><http://www.django-rest-framework.org/api-guide/permissions/#allowany>

## Глава 5: Todo API

Django REST Framework имеет длинный список неявно заданных настроек по умолчанию. Полный список можно посмотреть [здесь](#)<sup>54</sup>. `AllowAny` является одним из них, что означает, что когда мы установляем его явно, как мы делали выше, этот эффект точно такой же, как если бы у нас не было конфигурации `DEFAULT_PERMISSION_CLASSES`.

Установлен

Изучение настроек по умолчанию требует времени. С некоторыми из них мы познакомимся позже в книге. Главный вывод, который следует помнить, заключается в том, что неявные настройки по умолчанию предназначены для того, чтобы разработчики могли начать работу в локальной среде разработки. Однако, как и в традиционном Django, настройки Django REST Framework по умолчанию подходит для производства. Перед развертыванием мы обычно вносим в них ряд изменений в коде проекта.

Итак, Django REST Framework установлен. Что дальше? В отличие от проекта «Библиотека» в предыдущих главах, где мы сдавали веб-страницы, и API, здесь мы просто сдаваем API. Поэтому нам не нужно сдавать какие-либо файлы шаблонов или традиционные представления Django. Также, возможно, нет необходимости сдавать отдельное приложение API, поскольку этот проект изначально ориентирован на API. В то время как Django устанавливается с множеством ограничений по структуре проекта, разработчик сам решает, как организовать свои приложения. Это обычный путь для новичков, но при создании нескольких проектов с разными структурами приложений становится ясно, что приложения — это просто организационный инструмент для разработчика. Пока приложение добавлено в `INSTALLED_APPS` и использует правильную структуру импорта, его можно использовать практически в любой конфигурации.

Чтобы преобразовать нашу существующую модель базы данных в веб-API, нам потребуется обновить URL-адреса, добавить представления Django Rest Framework и создать сериализатор. Давайте начнем!

### URL-адреса

Мне нравится начинать с URL-адресов, поскольку они являются яточкой ввода для наших конечных точек API. Начните с файла уровня проекта Django, расположенного по адресу `django_project/urls.py`. Мы импортируем `include` во вторую строку и добавим маршрут для нашего приложения `todos` по пути `api/`. Рекомендуется иметь все конечные точки API на едином пути, таком как `api/`, если вы решите добавить традиционные веб-страницы Django позже.

---

<sup>54</sup><http://www.django-rest-framework.org/api-guide/settings/>

Код

---

```
# django_project/urls.py из
django.contrib import admin from
django.urls путь импорта, включите # новый

urlpatterns =
    [ path("admin/", admin.site.urls), path("api/",
        include("todos.urls")), # новый
]
```

---

Затем сядьте файл todos/urls.py на уровне приложения с помощью текстового редактора и добавьте следующий код:

Код

---

```
# todos/urls.py из
пути импорта django.urls

из .views импортировать ListTodo, DetailTodo

urlpatterns =
    [ path("<int:pk>/", DetailTodo.as_view(), name="todo_detail"), path("",
        ListTodo.as_view(), name="todo_list"),
]
```

---

Обратите внимание, что здесь мы создаем на два представления ListTodo и DetailTodo, которые нам еще предстоит создать. Но теперь маршрутизация завершена. Внутри той строке "", другими словами, в api/, будет список всех задач, и каждая отдельная задача будет доступна по своему первичному ключу, pk, который является значением, которое Django устанавливает автоматически в каждой таблице базы данных. Первая запись — 1, вторая — 2 и так далее. Поэтому наша первая задача в конечном итоге будет расположена в конечной точке API api/1, вторая — в API/2/ и так далее.

## Сериализаторы

Давайте рассмотрим, где мы находимся до сих пор. Мы начали с традиционного проекта Django, добавили сценарийальное приложение, настроили нашу модель базы данных и добавили исходные данные. Затем мы установили Django REST Framework и создали приложение API, для которого мы только что настроили наши URL-адреса. Есть два

## Глава 5: Todo API

оставшиеся шаги: сериализатор и представления. Давайте начнем с сериализатора, который преобразует данные нашей модели в JSON, который будет выводиться по нужным нам URL-адресам. Создайте новый файл с именем `todos/serializers.py` и обновите его следующим кодом.

Код

---

```
# todos/serializers.py из
с сериализаторов импорта rest_framework
из .models импортировать Todo

класс TodoSerializer (сериализаторы.ModelSerializer):
    Метакласс:
        модель = Todo
        fields = ("id",
                  "название",
                  "тело",
        )
```

---

Вверху мы импортировали сериализаторы из Django REST Framework вместе с нашим Todo.

модель базы данных. Затем мы расширили ModelSerializer до нового класса TodoSerializer.

Формат здесь очень похож на то, как мы создаем классы моделей или формы в самом Django. Мы указываем, какую модель использовать, и конкретные поля, которые мы хотим предоставить. Помните, что идентификатор (похожий на pk) создается я Django автоматически, поэтому нам не нужно определять его в нашей модели задач, но мы будем отображать его в нашем отдельном представлении для каждой задачи. Итак, в Django REST Framework волшебным образом преобразует наши данные в JSON, предоставляя поля для идентификатора, заголовка и тела из нашей модели Todo.

В чем разница между id и pk? Оба они относятся к полю автоматически добавляемому ORM в модели Django. [id55](#) — это то, что троенное функция с стандартной библиотекой Python, а [pk56](#) — с самим Django. Общие представления на основе классов, такие как DetailView в Django, ожидают передачи параметра с именем pk, в то время как в полях модели часто просто ссылаются на id.

Последнее, что нам нужно сделать, это настроить файл `views.py`, который будет сопровождать наш сериализатор и URL-адреса.

[55](https://docs.python.org/3.10/library/functions.html#id)<https://docs.python.org/3.10/library/functions.html#id>

[56](https://docs.djangoproject.com/en/4.0/ref/models/instances/#the-pk-property)<https://docs.djangoproject.com/en/4.0/ref/models/instances/#the-pk-property>

## Глава 5: Todo API

### Просмотры

Здесь мы будем использовать два общих представления DRF: [ListAPIView57](#) для отображения всех задач и [RetrieveAPIView58](#) для отображения одного из элементов модели.

Обновите файл todos/views.py, чтобы он выглядел следующим образом:

#### Код

---

```
# todos/views.py из
rest_framework import generics

из .models импортировать Todo
из .serializers импортировать TodoSerializer

класс ListTodo (generics.ListAPIView): набор
    запросов = Todo.objects.all() serializer_class
        = TodoSerializer
```

```
класс DetailTodo (generics.RetrieveAPIView): набор
    запросов = Todo.objects.all() serializer_class =
        TodoSerializer
```

---

Вверху мы импортируем общие представления Django REST Framework, нашу модель Todo и только что созданный TodoSerializer.

Напомним из файла todos/urls.py, чтоу нас есть два маршрута

и, следовательно, два разных вида ляда. Новое представление под названием ListTodo подклассы ListAPIView, в то время как

Подклассы DetailTodo RetrieveAPIView.

Проницательные читатели заметят, что здесь есть некоторая избыточность. Последний, мы повторяем набор запросов и класс с сериализатором для каждого представления даже несмотря на то, что расширенное общее представление различные. Позже в книге мы узнаем о наборах представлений и маршрутизаторах, которые решают эту проблему. И позволяют нам создавать те же представления API и URL-адреса с гораздо меньшим количеством кода.

Но пока мы закончили! Наш API готов к использованию

---

[57](http://www.django-rest-framework.org/api-guide/generic-views/#listapiview)[58](http://www.django-rest-framework.org/api-guide/generic-views/#retrieveapiview)<http://www.djangoproject.com/en/1.11/topics/class-based-views/generic-display/>

## Г лава 5: Todo API

Доступный для просмотра API

Теперь давайте воспользуемся доступным для просмотра API Django REST Framework для взаимодействия с нашими данными. Убедитесь, что локальный сервер работает, и перейдите по адресу <http://127.0.0.1:8000/api/>, чтобы увидеть конечную точку нашего рабочего списка представлений API.

Django REST framework

List Todo

OPTIONS    GET

```
GET /api/
HTTP 200 OK
Allow: GET, HEAD, OPTIONS
Content-Type: application/json
Vary: Accept

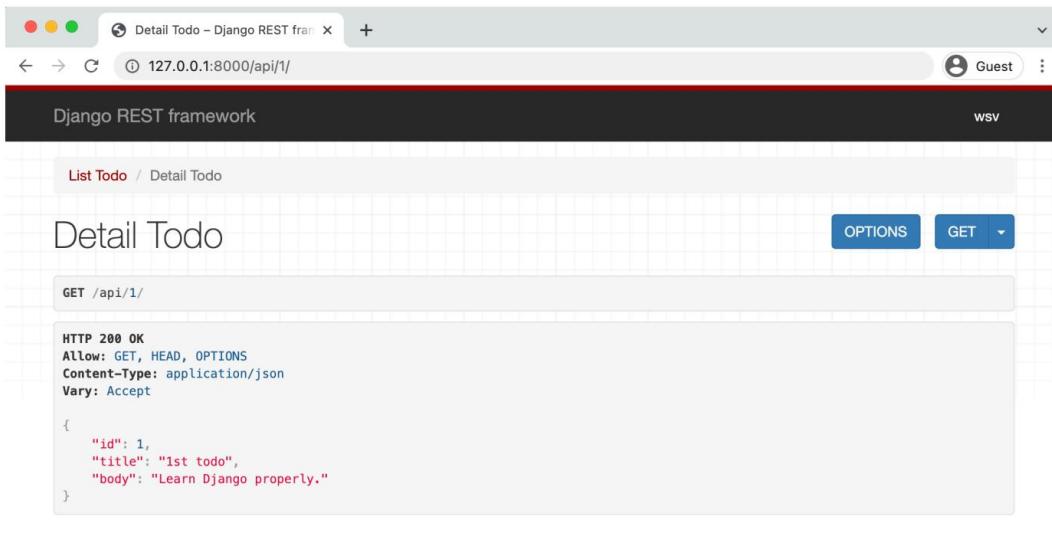
[
    {
        "id": 1,
        "title": "1st todo",
        "body": "Learn Django properly."
    },
    {
        "id": 2,
        "title": "Second item",
        "body": "Learn Python."
    },
    {
        "id": 3,
        "title": "Learn HTTP",
        "body": "It's important."
    }
]
```

Список API

На этой странице показаны три задачи, которые мы создали ранее в модели базы данных. Конечная точка API относится к URL-адресу, используемому для выполнения запроса. Если в конечной точке есть несколько элементов, она называется коллекцией, а один элемент называется ресурсом. Термины «конечная точка» и «ресурс» часто используются разработчиками взаимодействия, но означают разные вещи.

Мы также сделали представление `DetailTodo` для каждой отдельной модели, которое должно быть видно по адресу:

<http://127.0.0.1:8000/api/1/>.



Сведения об API

Вы также можете перейти к конечным точкам для

- <http://127.0.0.1:8000/api/2>
- <http://127.0.0.1:8000/api/3>

## API-тесты

Как мы видели в предыдущей главе, Django REST Framework содержит несколько помощниковых классов для тестирования наших конечных точек API. Мы хотим убедиться, что при использовании правильных URL-адресов, возвращается код состояния 200 и содержится правильный контент. На этот раз есть две страницы для тестирования нашей страницы с списком всех задач и отдельные задачи из списка с выделенной конечной точкой.

Откройте файл todos/tests.py в текстовом редакторе. Чтобы протестировать API, нам нужно импортировать три новых элемента вверху: reverse из Django, TestCase из Django REST Framework и APITestCase из Django REST Framework. Затем мы добавляем два теста — test\_api\_listview и test\_api\_detailview — чтобы проверить, что страницы с списком и подробностями используют правильный именованный URL-адрес, возвращают 200 кодов состояния с содержанием только один объект и содержат все ожидаемые данные. Единственная ложность здесь в том, что для детального просмотра мы должны передать pk объекта.

Код

---

```
# todos/tests.py из
django.test import TestCase из django.urls
import reverse # new from rest_framework import
status # new from rest_framework.test import
APITestCase # new

из .models импортировать Todo

class TodoModelTest(TestCase):
    @classmethod def setUpTestData(cls):
        cls.todo = Todo.objects.create( title="First
            Todo", body=" Текст здесь"

    )

    определение test_model_content ():
        self.assertEqual(self.todo.title, "Первое задание")
        self.assertEqual(self.todo.body, "Текст здесь") self.assertEqual(str(self.todo),
        "Первое задание")

    def test_api_listview(self): # новый
        ответ = self.client.get(reverse("todo_list"))
        self.assertEqual(response.status_code, status.HTTP_200_OK)
        self.assertEqual(Todo.objects.count(), 1) self.assertContains(ответ, self.todo )

    def test_api_detailview(self): # новый
        response =
            self.client.get( reverse("todo_detail", kwargs={"pk": self.todo.id}),
            format="json"

        ) self.assertEqual(response.status_code, status.HTTP_200_OK)
        self.assertEqual(Todo.objects.count(), 1) self.assertContains(ответ, "Первая
        задача")
```

---

Запустите тесты с помощью команды `python manage.py`.

Ракушка

---

```
(.venv) > python manage.py test Создание тестовой
базы данных для пакета "помолчаний...".
Проверка систе... не выявила проблем (0 отключено).
..
-----
```

Провел 3 теста за 0,007 с.

Хорошо

---

Уничтожение тестовой базы данных для пакета "помолчаний..."

---

Мы почти закончили, но есть два дополнительных сюжета, так как наш бэкенд будет связываться с внешним интерфейсом на другом порту. Это вызывает множество опасений по поводу безопасности, которые мы сейчас разберем.

## Корс

Совместное использование ресурсов между источниками (CORS)<sup>59</sup> относится к тому факту, что всякий раз, когда клиент взаимодействует с API, размещенным в другом домене ([mysite.com](http://mysite.com) или [yoursite.com](http://yoursite.com)) или порту (`localhost:3000` или `localhost:8000`), возникают потенциальные проблемы с безопасностью

В частности, CORS требует, чтобы веб-сервер включал определенные заголовки HTTP, которые позволяют клиенту определять, следует ли ему разрешать междоменные запросы. Поскольку мы используем архитектуру SPA, интерфейс во время разработки будет находиться на другом локальном порту, а после развертывания — на совершенно другом домене!

Самый простой способ справиться с этой проблемой — это тот, который рекомендуется Django REST Framework<sup>60</sup> — использовать промежуточное программное обеспечение, которое будет автоматически включать соответствующие заголовки HTTP на основе наших настроек. Сторонний пакет [django-cors-headers<sup>61</sup>](https://github.com/adamchainz/django-cors-headers) является явным выбором по умолчанию и поддерживается Django и может быть легко добавлено в наш существующий проект.

Обязательно установите локальный сервер с помощью `Control+C`, а затем установите `django-cors-headers` с помощью `Pip`.

---

<sup>59</sup><https://developer.mozilla.org/en-US/docs/Web/HTTP/CORS> <sup>60</sup><http://www.djangoproject.com/topics/ajax-csrf-cors/> <sup>61</sup><https://github.com/adamchainz/django-cors-headers>

Ракурска

---

```
(.venv) > python -m pip install django-cors-headers~=3.10.0
```

---

Затем обновите наш файл django\_project/settings.py в трех местах :

- добавить заголовки в INSTALLED\_APPS
- добавить CorsMiddleware выше CommonMiddleware в MIDDLEWARE.
- создать конфигурацию CORS\_ALLOWED\_ORIGINS внизу файла

Код

---

```
# django_project/settings.py
INSTALLED_APPS =
    [
        "django.contrib.admin",
        "django.contrib.auth",
        "django.contrib.contenttypes",
        "django.contrib.sessions",
        "django.contrib.messages",
        "django.contrib.staticfiles", # с внешней стороны
        "rest_framework",
        "corsheaders", # новый
    ]

    # местный
    "todos.apps.TodosConfig",
]

# ПРИМЕЧАНИЕ: ТРОГАТЬ НЕЛЬЗЯ С ПРЕДУПРЕЖДЕНИЕМ!
CORS_ALLOWED_ORIGINS = (
    "http://локальный:3000", "http://
    локальный:8000",
)
```

---

## Глава 5: Todo API

Очень важно, чтобы corsheaders.middleware.CorsMiddleware отображался в нужном месте, поскольку промежуточное ПО Django загружается с верху вниз. Также обратите внимание, что мы внесли в белый список два домена: localhost:3000 и localhost:8000. Первый — порт по умолчанию для React (если используется внешний интерфейс), второй — порт Django по умолчанию

### CSRF

Как CORS является проблемой при работе с архитектурой SPA, так и формы. Django поставляется с надежной [защитой от CSRF](#). Это должно быть добавлено к формам в любом шаблоне Django, но с выделенной настройкой внешнего интерфейса React эта защита по своей сути недоступна. К счастью, мы можем разрешить определенные междоменные запросы из нашего внешнего интерфейса, установив [CSRF\\_TRUSTED\\_ORIGINS](#)<sup>62</sup>.

В нижней части файла `settings.py`, рядом с `CORS_ORIGIN_WHITELIST`, добавьте эту дополнительную строку для локального порта React по умолчанию 3000:

Код

---

```
# django_project/settings.py
CSRF_TRUSTED_ORIGINS = ["localhost:3000"]
```

---

И это все! Теперь наша первая часть завершена и способна взаимодействовать с любым внешним интерфейсом, используя нашим порт 3000. Если наш внешний интерфейс требует другого порта, который можно легко обновить в нашем коде.

### Развертывание внутреннего API

Мы снова развернем серверную часть Django API с помощью Heroku. Если вы помните наш контрольный список развертывания API библиотеки из главы 4, он включал следующие:

- настроить статические файлы и установить WhiteNoise
- установить Gunicorn в качестве рабочего веб-сервера
- создавать файлы requirements.txt, runtime.txt и Procfile.

<sup>62</sup><https://docs.djangoproject.com/en/4.0/ref/csrf/>

<sup>63</sup><https://docs.djangoproject.com/en/4.0/ref/settings/#csrf-trusted-origins>

- обновить конфигурацию ALLOWED\_HOSTS

Теперь мы можем пройти по каждому из них быстрее. Для статических файлов создайте новый статический каталог из оболочки терминала.

Ракушка

---

(.venv) > mkdir статический

---

С помощью текстового редактора создайте файл .keep в статическом каталоге, чтобы Git подхватил его.

Затем установите whitenoise для обработки статических файлов в рабочей среде.

Ракушка

---

(.venv) > python -m pip install whitenoise == 5.3.0

---

WhiteNoise необходимо добавить в django\_project/settings.py в следующих местах:

- белый шум над django.contrib.staticfiles в INSTALLED\_APPS
- WhiteNoiseMiddleware выше CommonMiddleware
- Конфигурация STATICFILES\_STORAGE, указывающая на WhiteNoise.

Код

---

```
# django_project/settings.py
INSTALLED_APPS = [
    ...
    "whitenoise.runserver_nostatic", # новый
    "django.contrib.staticfiles",
]
[

    "django.middleware.security.SecurityMiddleware",
    "django.contrib.sessions.middleware.SessionMiddleware",
    "whitenoise.middleware.WhiteNoiseMiddleware", # new
    "corsheaders.middleware.CorsMiddleware",
    ...
]

STATIC_URL = "/статический/"
```

```
STATICFILES_DIRS = [BASE_DIR / "static"] # новый
STATIC_ROOT = BASE_DIR / "staticfiles" # новый
STATICFILES_STORAGE =
    "whitenoise.storage.CompressedManifestStaticFilesStorage" # новый
```

---

Наконец, запустите команду `collectstatic`, чтобы все статические каталоги и файлы были скомпилированы в одном месте для целей развертывания.

Ракушка

---

```
(.venv) > python manage.py collectstatic
```

---

Gunicorn будет использовать явно в качестве производственного веб-сервера и может быть установлено напрямую

Ракушка

---

```
(.venv) > python -m pip install gunicorn~=20.1.0
```

---

В текстовом редакторе создайте файл `runtime.txt` в корневом каталоге проекта рядом с `manage.py`.

В нем будет одна строка, указывающая версию Python для запуска на Heroku.

время выполненияtxt

---

```
питон-3.10.2
```

---

Теперь создайте пустой файл `Procfile` в том же корневом каталоге проекта. Он должен содержать следующую одну строку команд:

Procfile

---

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

---

Мы можем автоматически сгенерировать файл `requirements.txt` с необходимым нашей виртуальной среды в одной команде:

## Г лава 5: Todo API

Ракушка

```
(.venv)> python -m pip замораживание> требованийtxt
```

Последний шаг — обновить конфигурацию ALLOWED\_HOSTS в django\_project/settings.py.

Доступ должен быть ограничен локальными местами, 127.0.0.1 и .herokuapp.com.

Код

```
# django_project/settings.py  
ALLOWED_HOSTS = [".herokuapp.com", "localhost", "127.0.0.1"]
```

Обязательно добавьте и зафиксируйте новые изменения в Git.

Ракушка

```
(.venv)> git status (.venv)  
> git add -A (.venv)> git  
commit -m "Новые обновления для развертывания Heroku"
```

Затем войдите в интерфейс командной строки Heroku, введя команду heroku login, которая потребует от вас подтверждения учетные данные на самом сайте Heroku.

Ракушка

```
(.venv)> heroku login heroku:  
Нажмите любую клавишу, чтобы открыть браузер для ввода или q, чтобы выйти:  
Открытие браузера для...  
Вход в систему...  
Готово Вы вошли как will@wsvincent.com
```

После входа в систему нам нужно создать новый проект Heroku. Поскольку имена Heroku уникальны, вам нужно будет придумать свой собственный вариант. Я позвонил своему wsvincent-todo.

Глава 5: Todo API

Ракушка

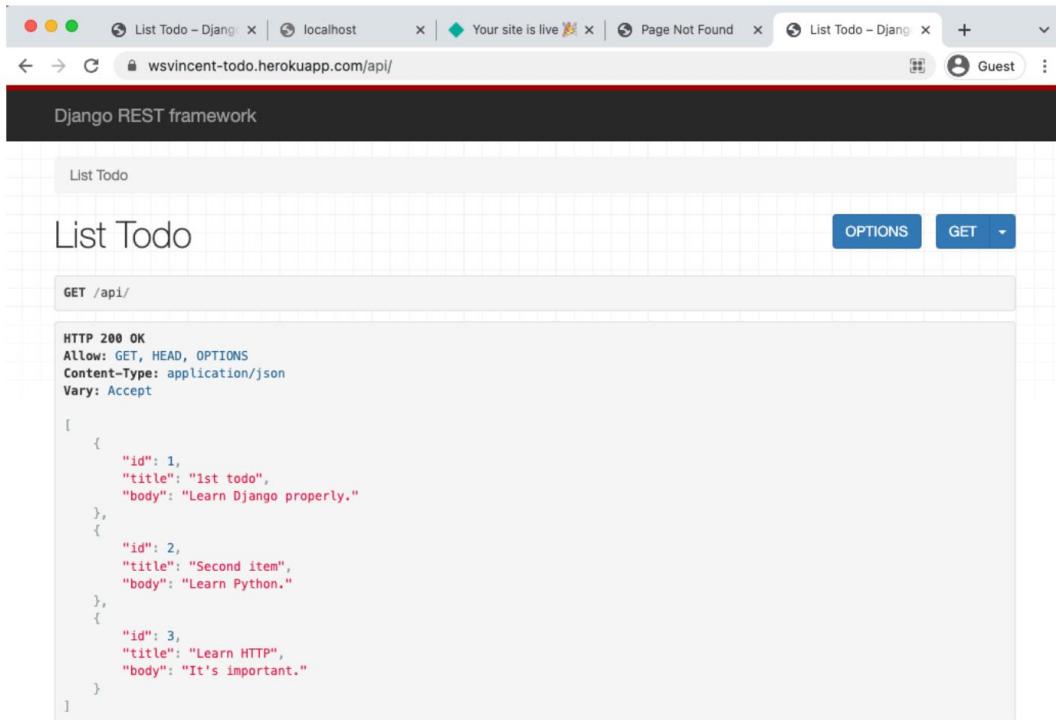
```
(.venv) > heroku create wsvincent-todo Создание
wsvincent-todo... сделано https://wsvincent-
todo.herokuapp.com/ | https://git.heroku.com/wsvincent-todo.git
```

Отправьте код в Heroku и добавьте веб-процессы, чтобы дупо работал.

Ракушка

```
(.venv) > git push heroku main (.venv) >
heroku ps:scale web=1
```

URL-адрес вашего нового приложения будет в выводе командной строки, или вы можете запустить `heroku open`, чтобы найти его. Обязательно перейдите к конечной точке `/api/`, чтобы увидеть список всех элементов Todo. Вот моя конечная точка Todo API со списком всех элементов:



Конечная точка списка API Todo

Отдельные конечные точки API для каждого элемента Todo также будут доступны в /api/1/, /api/2/ и т. д. Развернутый Todo API теперь можно использовать. Как только развернутые URL-адреса внешнего кода становятся известны, их можно добавить в разделы CORS и CSRF по мере необходимости.

## Заключение

Используя минимальный объем кода, Django REST Framework позволил нам создать Django API с нуля. В отличие от нашего примера из предыдущей главы, мы не создавали никаких веб-страниц для этого проекта, поскольку нашей целью было просто создать API. Однако в любой момент в будущем мы легко могли бы! Потребуется просто добавить новое представление, URL-адрес и шаблон, чтобы показать наш существующий модель базы данных.

Важным моментом в этом примере является то, что мы добавили заголовки CORS и явно установили доступ к нашему API только для доменов localhost:3000 и localhost:8000. Когда вы впервые начинаете создавать API, легкозагружаться в правильной настройке заголовков CORS.

Мы можем и продолжать больше настроек позже, но в конце концов создание Django API с водится как с созданием модели, написанием некоторых URL-маршрутов, а затем добавлением немного волшебства, предоставляемого сериализаторами и представлениями Django REST Framework.

# Глава 6: API блога

Основной проект в этой книге — Blog API, использующий полный набор функций Django REST Framework. У него будут пользователи, разрешения и полная функциональность CRUD (создание-чтение-обновление-удаление). Мы также рассмотрим наборы представлений, маршрутизаторы и документацию.

В этой главе мы создадим базовый раздел API. Как и в случае с нашей библиотекой Todo API, мы начинаем с традиционного Django, а затем добавляем Django REST Framework. Основные отличия заключаются в том, что мы будем использовать настраиваемую модель и поддерживать операции CRUD с самим с самого начала, что, как мы увидим, Django REST Framework делает довольно легко.

## Начальная настройка

Наша установка также, как и раньше. Перейдите в каталог кода и в нем создайте один для этого проекта под названием blogapi. Затем установите Django в новую виртуальную среду и создайте новый проект Django с именем django\_project.

Ракурс

---

# Ok на

```
> cd onedrive\desktop\code > mkdir
blogapi > cd blogapi > python -m
venv .venv > .venv\Scripts\Activate.ps1
(.venv) > python -m pip install
django~=4.0.0 (.venv) > django-admin
startproject django_project .
```

# Mac OS

```
% cd desktop/desktop/code %
mkdir blogapi % cd blogapi %
python3 -m venv .venv %
source .venv/bin/activate (.venv)
% python3 -m pip install django~=4.0.0
(.venv) % django-admin startproject django_project .
```

---

Запустите команду `python manage.py runserver`, и она должна вызвать приветствие Django.  
с страницы по адресу `http://127.0.0.1:8000/`.

The screenshot shows a web browser window with the address bar displaying `127.0.0.1:8000`. A message at the top says "The install worked successfully". Below it is a cartoon rocket ship icon launching from clouds. The text "The install worked successfully! Congratulations!" is displayed. A note below states: "You are seeing this page because `DEBUG=True` is in your settings file and you have not configured any URLs." At the bottom, there are links for "Django Documentation", "Tutorial: A Polling App", and "Django Community".

Оболочка терминала, вероятно, отображает сообщение с жалобой на то, что у вас есть 18 непримененных миграций.

Мы намеренно пока не запускаем миграции потому что будем использовать пользовательскую модель пользователя и хотим подождать, пока она не будет настроена, прежде чем запустить нашу первую команду миграции.

## .gitignore

Использование Git рано и часто его применяется для проверки ошибок в проектах. Это позволяет разработчикам отслеживать изменения в проекте с течением времени и выявлять любые ошибки, которые могут возникнуть. Давайте инициализируем новый репозиторий Git и проверьте его состояние.

Ракушка

---

```
(.venv) > git init (.venv)
> стартуем git
```

---

Должен появиться файл .venv, который нам не нужен в системе управления версиями, поэтому используйте текстовый редактор для создания файла .gitignore в каталоге проекта рядом с файлом manage.py. Добавьте одну строку для .venv, чтобы Git игнорировал ее.

.gitignore

---

```
.venv/
```

---

Затем снова запустите git status, чтобы подтвердить, что .venv больше не отображается, добавьте нашу текущую работу и создайте первый коммит Git.

Ракушка

---

```
(.venv) > стартуем git (.venv)
> git add -A (.venv) > git
commit -m "начальный коммит"
```

---

Пользовательская модель пользователя

Добавление пользовательской модели пользователя является необязательным, но рекомендуемым следующим шагом. Даже если вы не планируете его использовать, предприняв несколько шагов, вы получите открытой возможность использовать его в будущем в своем проекте.

Для этого начните новое приложение под названием «Учетные записи».

## Глава 6: API блога

Ракурска

(.venv) &gt; учетные записи и запуск python manage.py

Затем добавьте его в нашу конфигурацию INSTALLED\_APPS, чтобы Django знал, что он существует.

Код

```
# django_project/settings.py
INSTALLED_APPS =
    [ "django.contrib.admin",
      "django.contrib.auth",
      "django.contrib.contenttypes",
      "django.contrib.sessions",
      "django.contrib.messages", "
      django.contrib.staticfiles", # Локальный

      "accounts.apps.AccountsConfig", # новый
    ]
```

В файле account/models.py определите пользовательскую модель пользователя с именем CustomUser, расширяя AbstractUser<sup>64</sup>. и добавление одного поля имя на данный момент. Мы также добавим метод \_\_str\_\_ для возврата адреса электронной почты пользователя в админке и в других местах.

Код

```
# account/models.py из
django.contrib.auth.models import AbstractUser из django.db import
models
```

**класс** CustomUser (AbstractUser):

имя = модели.CharField (нуль = Истина, пусто = Истина, max\_length = 100)

Последний шаг — обновить AUTH\_USER\_MODEL<sup>65</sup>. конфигурация в settings.py, для которой неявно установлено значение auth.User, а не для account.CustomUser. Это можно добавить внизу файла.

---

<sup>64</sup><https://docs.djangoproject.com/en/4.0/topics/auth/customizing/#django.contrib.auth.models.AbstractUser> <sup>65</sup><https://docs.djangoproject.com/en/4.0/ref/settings/#auth-user-model>

## Код

---

```
# django_project/settings.py
AUTH_USER_MODEL = "accounts.CustomUser" # новый
```

---

Теперь мы можем запустить makemigrations для изменений нашей модели, выполнить миграции и создать нового пользователя в базе данных и создать суперпользователя для создания учетной записи и управления им. Не забудьте указать адрес электронной почты для каждого пользователя.

## Ракурс

---

```
(.venv) > python manage.py makemigrations (.venv) >
python manage.py migrate (.venv) > python manage.py
createsuperuser
```

---

Затем запустите внутренний веб-сервер Django с помощью команды runserver:

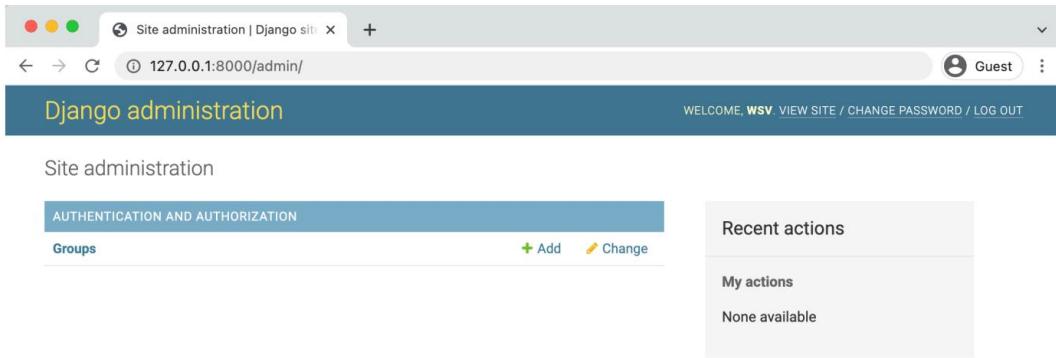
## Ракурс

---

```
(.venv) > сервер запускает python manage.py
```

---

Если мы направимся к административному порталу по адресу <http://127.0.0.1:8000/admin/> и войдем в систему, будет казаться, что что-то не входит, не так ли?



Открывается только раздел «Группы». У нас нет пользователей, как обычно с моделью пользователя по умолчанию. Чего не входит, так это двух вещей: нам нужно настроить account/admin.py для отображения нашей новой пользовательской модели пользователя и создать новый файл с именем account/forms.py, в котором CustomUser будет использоваться при создании или изменении пользователей. Начнем с account/forms.py.

## Глава 6: API блога

Код

---

```
# account/forms.py из
django.contrib.auth.forms import UserCreationForm, UserChangeForm

из .models импортируем CustomUser

класс CustomUserCreationForm(UserCreationForm): класс Meta
    (UserCreationForm):
        модель = CustomUser
        fields = UserCreationForm.Meta.fields + ("имя")

класс CustomUserChangeForm(UserChangeForm):
    Мета класса:
        модель = CustomUser
        поля = UserChangeForm.Meta.fields
```

---

Вверху импортируем `UserCreationForm`<sup>66</sup> и `UserChangeForm`<sup>67</sup> которые используются для создания или обновления пользователя.

Мы также импортируем нашу модель `CustomUser`, чтобы ее можно было интегрировать в новые классы `CustomUserCreationForm` и `CustomUserChangeForm`.

После этого последним шагом в настройке пользователя является обновление `account/admin.py` для правильного отображения нового пользователя.

Код

---

```
# account/admin.py из
django.contrib import admin
django.contrib.auth.admin import UserAdmin

из .forms импортируем CustomUserCreationForm, CustomUserChangeForm из .models
импортируем CustomUser
```

```
класс CustomUserAdmin(UserAdmin):
    add_form = CustomUserCreationForm
    form = CustomUserChangeForm
    модель = CustomUser
```

```
list_display = [
```

---

<sup>66</sup><https://docs.djangoproject.com/en/4.0/topics/auth/default/#django.contrib.auth.forms.UserCreationForm> <sup>67</sup><https://docs.djangoproject.com/en/4.0/topics/auth/default/#django.contrib.auth.forms.UserChangeForm>

Глава 6: API блога

```
"электронная
 почта", "имя
 пользователя",
 "имя", "is_staff",

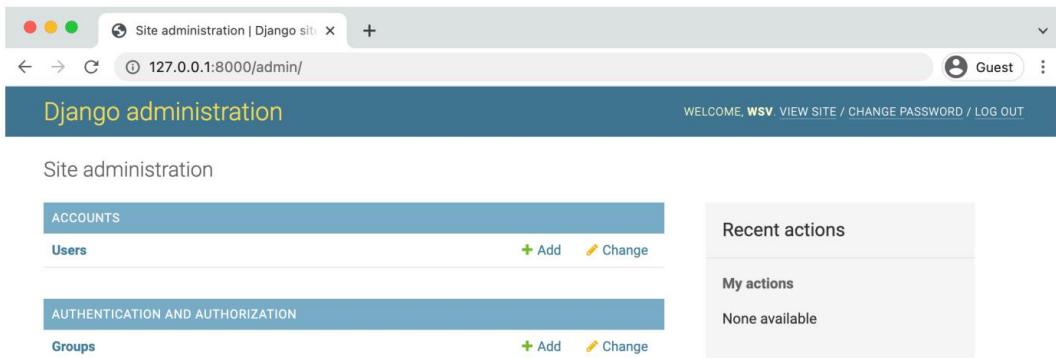
] fieldsets = UserAdmin.fieldsets + ((Нет, {"поля": ("имя",)}),) add_fieldsets =
UserAdmin.add_fieldsets + ((Нет, {"поля": ("имя",)}), )
```

---

```
admin.site.register(CustomUser, CustomUserAdmin)
```

---

И мы закончили. Если вы перезагрузите страницу администратора, она теперь отображает пользователей.



The screenshot shows the Django admin interface at the URL `127.0.0.1:8000/admin/`. The top navigation bar includes links for Site administration, Django site, and a guest user. The main header says "Django administration". On the right, there are "Recent actions" and "My actions" sections, both currently empty. The left sidebar has two main categories: "ACCOUNTS" (with "Users" listed) and "AUTHENTICATION AND AUTHORIZATION" (with "Groups" listed). Each category has "Add" and "Change" buttons. The "Users" section shows a table with columns for "username", "first name", "last name", "email", "is\_staff", and "is\_superuser". A single row is visible, representing the superuser account created earlier.

Если вы нажмете «Пользователи», вы увидите, что наш суперпользователь тоже находится там.

Select user to change

Action:  0 of 1 selected

<input type="checkbox"/>	EMAIL ADDRESS	USERNAME	NAME	STAFF STATUS
<input type="checkbox"/>	will@wsvincent.com	wsv	-	<input checked="" type="checkbox"/>

1 user

FILTER

By staff status

- All
- Yes
- No

By superuser status

- All
- Yes
- No

By active

- All
- Yes
- No

Приложение «Сообщения»

Пришло время создать специальное приложение для нашего блога. Именование все еще должно быть отвязано и добавить новое приложение под названием blog и связанные с ним модель под названием Blog, это редко делается поскольку несколько областей в Django добавляются к именам приложений и моделей, а «блог и» просто не выглядят очень хороший. Поэтому причине приложение блог лучше называть чем-то вроде сообщений, а связанную модель базы данных — просто публикаций. Этого, что мы будем делать здесь.

Введите Control+c, чтобы установить локальный сервер, а затем используйте команду управления startapp для создания нового приложения с сообщений.

Редактор

---

```
(.venv) > python manage.py startapp посты
```

---

Затем немедленно обновите INSTALLED\_APPS в файле django\_project/settings.py, пока мы не забыли.

Код

---

```
# django_project/settings.py
INSTALLED_APPS =
    [ "django.contrib.admin",
      "django.contrib.auth",
      "django.contrib.contenttypes",
      "django.contrib.sessions",
      "django.contrib.messages",
      "django.contrib.staticfiles", # Локальный

      "accounts.apps.AccountsConfig",
      "posts.apps.PostsConfig", # новый
    ]
```

---

## Почтовая модель

Наша модель базы данных с сообщений блога будет иметь пять полей: автор, заголовок, тело, created\_at и updated\_at. Мы также импортируем настройки Django, чтобы мы могли ссылаться на AUTH\_USER\_MODEL в нашем поле автора. И мы добавим метод \_\_str\_\_ в качестве общей рекомендации.

## Глава 6: API блога

Код

---

```
# posts/models.py из
django.conf импортировать настройки из
django.db импортировать модели
```

Сообщение класса (models.Model):

```
title = models.CharField(max_length=50) body =
models.TextField() author =
models.ForeignKey(settings.AUTH_USER_MODEL, on_delete=models.CASCADE) created_at =
models.DateTimeField(auto_now_add=True) updated_at = models.DateTimeField(auto_now = Верно)

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

---

Это выглядит достаточно просто. Теперь обновите нашу базу данных, сначала создав новый файл миграции с помощью команды makemigrations posts, а затем запустив migrate для их применения базы данных с изменениями нашей модели.

Ракушка

---

```
(.venv) > с сообщениями makemigrations python manage.py (.venv) >
python manage.py migrate
```

---

Хорошо! Мы хотим просматривать наши данные в административном приложении Django, поэтому будем обновить posts/admin.py. следующее.

Код

---

```
# posts/admin.py из
django.contrib import admin

из поста импорта .models

admin.site.register(Post)
```

---

Снова запустите локальный веб-сервер с помощью python manage.py runserver и посетите администратора, чтобы увидеть наша работа в действии.

The screenshot shows the Django administration interface at the URL `127.0.0.1:8000/admin/`. The top navigation bar includes the title "Site administration | Django site", a search bar, and links for "WELCOME, wsv. VIEW SITE / CHANGE PASSWORD / LOG OUT". The top right corner shows a "Guest" user icon.

The main content area is titled "Django administration" and displays three main sections:

- ACCOUNTS**: Contains a "Users" list with a "+ Add" button and a "Change" link.
- AUTHENTICATION AND AUTHORIZATION**: Contains a "Groups" list with a "+ Add" button and a "Change" link.
- POSTS**: Contains a "Posts" list with a "+ Add" button and a "Change" link.

To the right of the main content, there is a sidebar with two sections:

- Recent actions**: A list showing no recent actions.
- My actions**: A list showing "None available".

С сообщениями администратора

Вот и наше приложение «Сообщения»! Нажмите кнопку «+ Добавить» рядом с «Сообщениями» и создайте новый пост в блоге. Рядом с «Автор» будет выпадающее меню вашей учетной записи пользователя (меня называетя wsv). Убедитесь, что выбран автор, добавьте заголовок, добавьте новое содержимое, а затем нажмите кнопку «Сохранить».

Глава 6: API блога

Django administration

WELCOME, **WSV**. VIEW SITE / CHANGE PASSWORD / LOG OUT

Home › Posts › Posts › Add post

Add post

Title: 1st Blog Post

Body: Hello, World!

Author: wsv

Save and add another Save and continue editing SAVE

Админис трат измабавиль запись в блог

Вы будете перенаправлены на страницу с сообщений, на которой отображаются все существующие сообщения блога.

Select post to change | Django

127.0.0.1:8000/admin/posts/post/

Django administration

WELCOME, **WSV**. VIEW SITE / CHANGE PASSWORD / LOG OUT

Home › Posts › Posts

✓ The post "1st Blog Post" was added successfully.

Select post to change ADD POST +

Action: ----- Go 0 of 1 selected

POST

1st Blog Post

1 post

Посты администратора блога

## Тесты

Мы написали новый код, так что пришло время тестов. Они добавляются в существующий файл `posts/tests.py`, созданный с помощью команды `startapp`.

В верхней части файла импортируйте `get_user_model()`<sup>68</sup>, для ссылки на нашего пользователя вместе с `TestCase` и моделью `Post`. Затем создайте класс `BlogTests` с установленными данными и единственным тестом, `test_post_model`, который проверяет поля в модели `Post` вместе с методом `__str__`.

### Код

---

```
# posts/tests.py из
django.contrib.auth import get_user_model из django.test
import TestCase

из поста импорта .models

class BlogTests (TestCase):
    @classmethod def
    setUpTestData (cls):
        cls.user = get_user_model().objects.create_user( username="testuser",
                                                       email="test@email.com", password="secret",

    )

    cls.post = Post.objects.create( author=cls.user,
                                   title="Хорошее название",
                                   body="Хорошее содержание тела",
    )

    def test_post_model (я):
        self.assertEqual(self.post.author.username, "testuser")
        self.assertEqual(self.post.title, "Хорошее название")
        self.assertEqual(self.post.body, "Хорошее содержание тела") self.
        assertEquals(str(self.post), "Хорошее название")
```

---

Чтобы убедиться, что наши тесты работают, закройте локальный сервер с помощью `Control+C` и запустите наши тесты.

<sup>68</sup>[https://docs.djangoproject.com/en/4.0/topics/auth/customizing/#django.contrib.auth.get\\_user\\_model](https://docs.djangoproject.com/en/4.0/topics/auth/customizing/#django.contrib.auth.get_user_model)

Ракушка

```
(.venv) > python manage.py test Создание
тестовой базы данных для псевдонима "по умолчанию...".
Проверка системы не выявила проблем (0 отключено).
```

Выполнить 1 тест за 0,105 с

X окошко

```
Уничтожение тестовой базы данных для псевдонима "по умолчанию..."
```

Мы закончили с обычной частью Django нашего API. Все, что нам действительно нужно, это модель и некоторые данные в нашей базе данных. Теперь пришло время добавить Django REST Framework, чтобы позаботиться о преобразовании данных нашей модели в API.

## Джанг о REST Framework

Как мы видели ранее, Django REST Framework берет на себя тяжелую работу по преобразованию наших моделей баз данных в RESTful API. В этом процессе есть три основных шага:

- файл urls.py для URL-маршрутов.
- файл serializers.py для преобразования данных в формат JSON.
- файл views.py для применения логики к каждой конечной точке API.

В командной строке используйте pip для установки Django REST Framework.

Ракушка

```
(.venv) > python -m pip установить djangorestframework~=3.13.0
```

Затем добавьте его в раздел INSTALLED\_APPS нашего файла django\_project/settings.py. Также рекомендуется явно установить наши разрешения. По умолчанию Django REST Framework настроен на AllowAny, чтобы облегчить нам локальную разработку, однако это далеко не безопасно. Мы обновим этот параметр разрешения в следующей главе.

Код

---

```
# django_project/settings.py
INSTALLED_APPS = [
    "django.contrib.admin",
    "django.contrib.auth",
    "django.contrib.contenttypes",
    "django.contrib.sessions",
    "django.contrib.messages",
    "django.contrib.staticfiles", # 3rd
    с сторонние приложения "rest_framework",
    # новый # Local

    "accounts.apps.AccountsConfig",
    "posts.apps.PostsConfig",
]

REST_FRAMEWORK = { # новый
    "DEFAULT_PERMISSION_CLASSES":
        [ "rest_framework.permissions.AllowAny",
    ],
}
```

---

Теперь нам нужно создать наши URL-адреса, представляющие и сериализаторы.

URL-адреса

Начнем с маршрутов URL для фактического отображения конечных точек. Обновите файл urls.py на уровне проекта, включив импорт во второй строке и новый маршрут api/v1/ для наших сообщений.

приложение.

## Глава 6: API блога

Код

---

```
# django_project/urls.py из
django.contrib import admin from
django.urls путь импорта, включите # новый

urlpatterns =
    [ path("admin/", admin.site.urls), path("api/
v1/", include("posts.urls")), # новый
]
```

---

Хорошей практикой является всегда версировать ваши API, поскольку при внесении больших изменений может возникнуть некоторая задержка, прежде чем различные потребители API также смогут обновляться. Таким образом, вы можете поддерживать API версии 1 в течение определенного периода времени, а также загружать новую обновленную версию API 2 и не ломать другие приложения, которые полагаются на серверную часть вашего API.

Обратите внимание, что, поскольку наше единственное приложение на данный момент — это сообщения, мы можем включить его прямо здесь. Если бы у нас было несколько приложений в проекте, было бы разумнее создать специальное приложение API, а затем включить в него все остальные маршруты URL-адресов API. Но для базовых проектов, подобных этому, я предлагаю избегать приложения API, которое используется только для маршрутизации. Мы всегда можем добавить один позже, если это необходимо.

Затем сядьте на новый файл posts/urls.py и добавьте следующий код:

Код

---

```
# posts/urls.py из
пути импорта django.urls

из .views импортировать PostList, PostDetail

urlpatterns =
    [ path("<int:pk>/", PostDetail.as_view(), name="post_detail"), path("", PostList.as_view(), name="post_list"),
]
```

---

В верхней части файла мы импортировали два представления — PostList и PostDetail — которые мы напишем в следующем разделе, но они соответствуют списку всех сообщений блога впустую с трактацией «», что означает в api/v1/. Отдельные сообщения с подробностями будут иметь первичный ключ pk, поэтому первое сообщение в блоге будет иметь адрес api/v1/1/, второе — api/v1/2/ и так далее. Пока это все стандартные вещи Django.

## Сериализаторы

Теперь о наших сериализаторах. Создайте новый файл `posts/serializers.py` в текстовом редакторе.

Сериализатор не только преобразует данные в JSON, но и может указать, какие поля включать или исключать. В нашем случае мы включим поле `id`, которое Django автоматически добавляет в модели базы данных, но мы исключим поле `updated_at`, не включив его в наши поля.

Возможность легкого включения/исключения полей в нашем API — замечательная функция. Чаще всего базовая модель базы данных будет иметь гораздо больше полей, чем нужно показать.

Мощный класс сериализатора Django REST Framework упрощает управление этим.

Код

---

```
# posts/serializers.py из
сериалайзаторов импорта rest_framework

из поста импорта .models
```

класс с PostSerializer (сериалайзеры.ModelSerializer):

Метакласс:

```
fields = ("id",
          "автор",
          "название",
          "тело",
          "создано_в",
```

```
) модель = Пост
```

---

В верхней части файла мы импортировали класс сериализаторов Django REST Framework и наши собственные модели. Затем мы создали `PostSerializer` и добавили класс `Meta`, в котором мы указали, какие поля следует включать, и явно задали используемую модель `Post`. Есть много способов настроить сериализатор, но для обычных случаев использования таких как `Post`, это все, что нам нужно.

## Глава 6: API блога

## Просмотры

Последним шагом является создание наших представлений. В Django REST Framework есть несколько полезных общих представлений. Мы уже использовали [ListAPIView69](#) в API-интерфейсах библиотеки Todos для создания коллекции конечных точек, доступной только для чтения, по умолчанию с классом модели. В Todos API мы использовали [RetrieveAPIView70](#), для единственный конечной точки только для чтения, как оправданного аналогичного подробному представлению традиционном Django.

Для нашего API блога мы хотим перечислить все доступные сообщения в блоге как конечную точку для чтения и записи, что означает использование [ListCreateAPIView71](#), который наследует от [ListAPIView](#), который мы использовали ранее, но допускает запись и, следовательно, запросы POST. Мы также хотим делать отдельные сообщения в блоге доступными для чтения, обновления или удаления. И, конечно же, именно для этой цели существует встроенное универсальное представление Django REST Framework: [RetrieveUpdateDestroyAPIView72](#). Этого, чтобы мы будем использовать здесь.

Обновите файл `posts/views.py` следующим образом.

## Код

---

```
# posts/views.py из
rest_framework import generics

из .models import Post
из .serializers import PostSerializer

class PostList(generics.ListCreateAPIView): queryset =
    Post.objects.all() serializer_class = PostSerializer

    class PostDetail(generics.RetrieveUpdateDestroyAPIView):
        queryset = Post.objects.all()
        serializer_class = PostSerializer
```

---

[69](http://www.djangoproject.org/api-guide/generic-views/#listapiview)[70](http://www.djangoproject.org/api-guide/generic-views/#retrieveapiview)[71](http://www.djangoproject.org/api-guide/generic-views/#listcreateapiview)[72](http://www.djangoproject.org/api-guide/generic-views/#retrieveupdatedestroyapiview)

В верхней части файла мы импортируем дженерики из Django REST Framework, а также наши модели и файлы сериализаторов.

Затем мы создаем два представления PostList и использует общий ListCreateAPIView.

в то время как PostDetail использует RetrieveUpdateDestroyAPIView.

Удивительно, что все, что нам нужно сделать, это обновить наше общее представление, чтобы радикально изменить поведение данной конечной точки API. В этом преимуществе использования полнофункционального фреймворка, такого как Django REST Framework: весь этот функционал доступен, протестирован и просто работает. Как разработчики нам не нужно изобретать велосипед здесь.

Фу. Теперь наш API завершен, и нам действительно не придется писать много кода самостоятельно. Мы внесли дополнительные улучшения в наш API в следующих главах, но стоит отметить, что он уже выполняет основные функции с помощью CRUD, которые нам нужны. Пришло время протестировать работу API-интерфейса Django Rest Framework с возможностью просмотра.

Документ для просмотра API

Запустите локальный сервер для взаимодействия с нашим API.

Ракушка

---

(.venv) > сервер запуска python manage.py

---

Затем перейдите по адресу <http://127.0.0.1:8000/api/v1/>, чтобы увидеть конечную точку с списком сообщений.

Глава 6: API блога

The screenshot shows a browser window with the title "Post List – Django REST framework". The URL in the address bar is "127.0.0.1:8000/api/v1/". The page header says "Django REST framework" and "wsf". Below the header, there's a "Post List" section. On the right, there are "OPTIONS" and "GET" buttons. A "Raw data" tab is selected at the bottom right. The main content area shows the response for a GET request to "/api/v1/". It includes the HTTP status code "HTTP 200 OK", headers ("Allow: GET, POST, HEAD, OPTIONS", "Content-Type: application/json", "Vary: Accept"), and a JSON response body:

```
[{"id": 1, "author": 1, "title": "1st Blog Post", "body": "Hello, World!", "created_at": "2022-02-07T17:34:00.892277Z"}]
```

Below this, there's a form with fields for "Author" (set to "wsf"), "Title", and "Body", and a "POST" button.

Список с обобщенными API

На странице отображается список наших постов в блоге — только один на данный момент — в формате JSON. Обратите внимание, что разрешены методы GET и POST. Идентификатор равен 1, что означает, что это первая запись в блоге, и автор также равен 1, поскольку мы использовали учетную запись суперпользователя, которая была создана первой. Возможно, было бы более идеальным отображать имя пользователя или, возможно, требовать отображения полного имени.

Сериалайзаторы очень мощные и могут быть настроены для вывода почти всего, что мы хотим, с любыми ограничениями. Многие из них [перечислены в документах](#)<sup>73</sup>, которые также отмечают, что «REST Framework не пытается автоматически оптимизировать наборы запросов, передаваемые сериализатором, с точки зрения

<sup>73</sup><https://www.django-rest-framework.org/api-guide/relations/#serializer-relations>

## Глава 6: API блога

`select_related` и `prefetch_related`, так как это было бы слишком много волшебства. На больших сайтах сериализаторы обычно не обходимо настраивать по соображениям производительности.

В отличие от наших предыдущих API для блога, у нас есть конечная точка для каждого модели, отображающая один пост. Давайте подтвердим, что он также существует, перейдя по адресу <http://127.0.0.1:8000/api/v1/1/> в Интернете браузер.

The screenshot shows a browser window displaying the Django REST framework's `Post Detail` view. The URL in the address bar is `127.0.0.1:8000/api/v1/1/`. The page title is "Post Detail – Django REST framework". The main content area shows a "Post Detail" heading and a "DELETE" button. Below this, there is a "Raw data" section containing the following JSON response:

```
HTTP 200 OK
Allow: GET, PUT, PATCH, DELETE, HEAD, OPTIONS
Content-Type: application/json
Vary: Accept

{
    "id": 1,
    "author": 1,
    "title": "1st Blog Post",
    "body": "Hello, World!",
    "created_at": "2022-02-07T17:34:00.892277Z"
}
```

Below the raw data, there is an "HTML form" section with fields for "Author" (set to "wsv"), "Title" ("1st Blog Post"), and "Body" ("Hello, World!"). A "PUT" button is located at the bottom right of the form.

## Детали поста API

Вы можете видеть в заголовке, что GET, PUT, PATCH и DELETE поддерживаются, но не POST. И на самом деле вы можете использовать HTML-форму ниже, чтобы внести изменения или даже использовать красную кнопку «УДАЛИТЬ», чтобы удалить этот экземпляр.

Давайте попробуем. Обновите наш заголовок с дополнительным текстом (отредактированным) в конце. Затем нажмите

нажмите на кнопку «ПОСТАВИТЬ».

The screenshot shows a browser window titled "Post Detail - Django REST framework" at the URL "127.0.0.1:8000/api/v1/1/". The page displays the details of a blog post with ID 1, including its author (wsv), title ("1st Blog Post (edited)"), body ("Hello, World!"), and creation timestamp ("2022-02-07T17:34:00.892277Z"). Below the details, there are four buttons: "DELETE" (red), "OPTIONS" (blue), "GET" (blue), and a dropdown menu. A "PUT /api/v1/1/" button is also present. The response to the PUT request is shown in a modal box:

```

HTTP 200 OK
Allow: GET, PUT, PATCH, DELETE, HEAD, OPTIONS
Content-Type: application/json
Vary: Accept

{
    "id": 1,
    "author": 1,
    "title": "1st Blog Post (edited)",
    "body": "Hello, World!",
    "created_at": "2022-02-07T17:34:00.892277Z"
}

```

Below the modal, there are two tabs: "Raw data" (selected) and "HTML form". The "HTML form" tab shows a form with fields for "Author" (wsv), "Title" (1st Blog Post (edited)), and "Body" (Hello, World!). A "PUT" button is located to the right of the form. At the bottom of the page, a note reads: "Деталь публикации API отредактирована".

Вернитесь к представлению «Список с сообщениями», щелкнув ссылку на него в верхней части страницы или перейдя непосредственно по адресу <http://127.0.0.1:8000/api/v1/>, и вы также увидите там обновленный текст.

The screenshot shows the Django REST framework's browsable API interface. At the top, there's a browser header with tabs for 'Post List – Django REST framework' and '+'. Below it, the URL is 127.0.0.1:8000/api/v1/. The main title is 'Django REST framework' with a user icon and 'Guest' next to it. A sidebar on the right says 'wsv'.

The main content area has a title 'Post List' with 'OPTIONS' and 'GET' buttons. Below that, a 'GET /api/v1/' button is shown. The response is displayed in a box:

```

HTTP 200 OK
Allow: GET, POST, HEAD, OPTIONS
Content-Type: application/json
Vary: Accept

[
    {
        "id": 1,
        "author": 1,
        "title": "1st Blog Post (edited)",
        "body": "Hello, World!",
        "created_at": "2022-02-07T17:34:00.892277Z"
    }
]

```

At the bottom, there are 'Raw data' and 'HTML form' tabs. The 'HTML form' tab is active, showing fields for 'Author' (set to 'wsv'), 'Title', and 'Body'. A 'POST' button is at the bottom right. A note at the bottom center says 'Список с сообщениями API отредактирован'.

## КОРС

Поскольку вполне вероятно, что наш API будет использовать я в другом домене, мы должны настроить CORS и указать, какие домены будут иметь доступ. Как мы видели в более раннем API Todo, процесс прост.

Во-первых, мы установим локальный веб-сервер с помощью `Control+c` и установим сторонний пакет [django-cors-headers](https://github.com/adamchainz/django-cors-headers)<sup>74</sup>.

<sup>74</sup><https://github.com/adamchainz/django-cors-headers>

Ракурс

---

```
(.venv) > python -m pip install django-cors-headers~=3.10.0
```

---

Затем мы добавляем corsheaders в INSTALLED\_APPS, добавляем CorsMiddleware в настройку MIDDLEWARE и создаем список CORS\_ALLOWED\_ORIGINS. Обновите файл settings.py следующим образом, чтобы справляться с тремя:

Код

---

```
# django_project/settings.py
INSTALLED_APPS =
    [
        "django.contrib.admin",
        "django.contrib.auth",
        "django.contrib.contenttypes",
        "django.contrib.sessions",
        "django.contrib.messages",
        "django.contrib.staticfiles", # с внешней стороны
        "rest_framework", "corsheaders", #
        "новый" # локальный

        "accounts.apps.AccountsConfig",
        "posts.apps.PostsConfig",
    ]

# ПОНЕМАНИЕ: ПРИ РАБОТЕ СЕРЕВЕРАМ – [
    "django.middleware.security.SecurityMiddleware",
    "django.contrib.sessions.middleware.SessionMiddleware",
    "corsheaders.middleware.CorsMiddleware", # new
    "django.middleware.common.CommonMiddleware",
    "django.middleware.csrf.CsrfViewMiddleware",
    "django.contrib.auth.middleware.AuthenticationMiddleware",
    "django.contrib.messages.middleware.MessageMiddleware",
    "django.middleware.clickjacking.XFrameOptionsMiddleware",
]

# новый
CORS_ORIGIN_WHITELIST = (
    "http://локальный:3000", "http://
    локальный:8000",
)
```

---

В случае, если наш API используется с формами, рекомендуется разрешить определенные междоменные запросы.

из нашего интерфейса, установив `CSRF_TRUSTED_ORIGINS75` также. Мы можем сделать это в `settings.py` сразу после раздела `CORS_ORIGIN_WHITELIST`. На данный момент мы устанавливаем локальный порт 3000, который используется React, хотя мы можем легко изменить порт в будущем в зависимости от нашего внешнего интерфейса.

потребности.

#### Код

---

```
# django_project/settings.py
CSRF_TRUSTED_ORIGINS = ["http://localhost:3000"] # новый
```

---

В качестве последнего шага передайте нашу работу с блогом в Git.

#### Ракушка

---

```
(.venv) > git status (.venv)
> git add -A (.venv) > git
commit -m "Нас тройка API блог а"
```

---

## Заключение

И это все! Мы намеренно повторили несколько шагов из наших предыдущих примеров, поэтому шаблон создания нового проекта Django, а затем его API должен казаться я более знакомым. Модели представляют собой чистый традиционный Django, но в основном URL-адреса, представления и сериализаторы поступают из DRF.

Мы добавили детальную оконечную точку в наш API и начали изучать возможности сериализаторов.

На данный момент Blog API полностью функционален для локального использования, однако есть большая проблема: любой может обновить или удалить существующую запись в блоге! Другими словами, у нас нет никаких разрешений. В следующий главе мы узнаем, как применять разрешения для защиты API.

---

<sup>75</sup><https://docs.djangoproject.com/en/4.0/ref/settings/#csrf-trusted-origins>

## Г лава 7: Р азрешения

Безопасность — важная часть любого веб-сайта, но в случае с веб-API она важна вдвое. В настоящем время наш Blog API предоставляет полный доступ любому. Нет никаких ограничений; любой пользователь может с делать что-нибудь через чрезвычайно опасное. Например, анонимный пользователь может создавать, читать, обновлять или удалять любые сообщения в блоге. Даже если одни не создали! Ясно, что мы это хотим.

Django REST Framework предоставляет с несколькими готовыми настройками разрешений, которые мы можем использовать для защиты нашего API. Их можно применять на уровне проекта, на уровне представления или в любой отдельной модели. уровни.

В этой главе мы рассмотрим все три и закончим настраиваемым разрешением, чтобы только автор сообщения в блоге имел возможность обновлять или удалять его.

### Разрешения на уровне проекта

Django REST Framework имеет [множество конфигураций](#)<sup>76</sup>, которые находятся внутри одного параметра Django с именем REST\_FRAMEWORK. Мы уже сделали один из них, `AllowAny`<sup>77</sup>, явным образом в файл `django_project/settings.py`.

---

<sup>76</sup><https://www.djangoproject.com/en/2.0/ref/settings/> <sup>77</sup><https://www.djangoproject.com/en/2.0/ref/settings/#allowany>

## Код

---

```
# django_project/settings.py
REST_FRAMEWORK = {
    "DEFAULT_PERMISSION_CLASSES":
        [ "rest_framework.permissions.AllowAny", # новый
    ],
}
```

---

На самом деле есть четыре встроенных параметра разрешений на уровне проекта, которые мы можем использовать:

- **Разрешить любой**<sup>78</sup> - любой пользователь, аутентифицированный или нет, имеет полный доступ
- **IsAuthenticated**<sup>79</sup> - доступ имею только аутентифицированные зарегистрированные пользователи •
- IsAdminUser**<sup>80</sup> - доступ имею только администраторы/суперпользователи • **IsAuthenticatedOrReadOnly**<sup>81</sup> - неавторизованные пользователи могут просматривать любую страницу, но только авторизованные отмеченные пользователи имеют права на запись, редактирование или удаление

Реализация любого из этих четырех параметров требует обновления параметра `DEFAULT_PERMISSION_CLASSES` и обновления нашего веб-браузера. Вот и все!

Давайте переключимся на `IsAuthenticated`, чтобы только аутентифицированные или вошедшие в систему пользователи могли просматривать API.

Обновите файл `django_project/settings.py` следующим образом:

---

<sup>78</sup><http://www.django-rest-framework.org/api-guide/permissions/#allowany>

<sup>79</sup><http://www.django-rest-framework.org/api-guide/permissions/#isauthenticated>

<sup>80</sup><http://www.django-rest-framework.org/api-guide/permissions/#isadminuser>

<sup>81</sup><http://www.django-rest-framework.org/api-guide/permissions/#isauthenticatedилитолько для чтения>

## Код

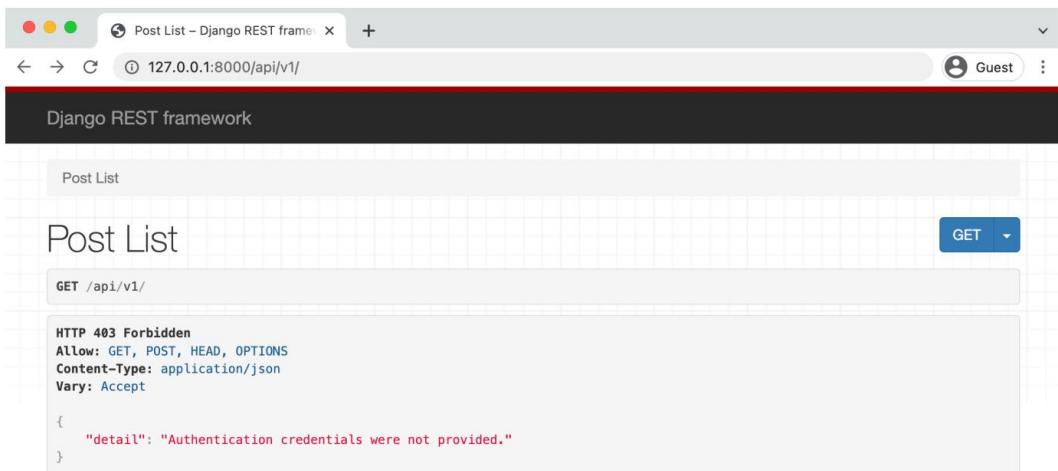
---

```
# django_project/settings.py
REST_FRAMEWORK = {
    "DEFAULT_PERMISSION_CLASSES": [
        "rest_framework.permissions.IsAuthenticated", # новый
    ],
}
```

---

Если вы обновите свой веб-браузер, ничего не изменится потому что мы уже вошли в систему с нашей учетной записью суперпользователя. Он должен приставать в правом верхнем углу вашего докладного для просмотра API. Чтобы выйти, войдите в админку по адресу <http://127.0.0.1:8000/admin/> и нажмите ссылку «Выход» в правом верхнем углу.

Если вы вернетесь к <http://127.0.0.1:8000/api/v1/>, он отобразит ошибку HTTP 403 Forbidden, поскольку учетные данные для аутентификации не были предоставлены. Это то, что мы хотим.



403 Ошибка

[Создать новых пользователей](#)

Нам нужно создать нового пользователя, чтобы проверить, что обычный пользователь, а не только суперпользователь-администратор, имеет доступ к API. Есть два способа сделать это: создать пользователя в командной оболочке с помощью `python`

## Глава 7: Разрешения

manage.py создает суперпользователя, или мы можем войти в систему администратора и таким образом добавить пользователя. Давай поговорим маршрут администратора.

Вернитесь к администратору по адресу <http://127.0.0.1:8000/admin/> и войдите в систему, используя ваши учетные данные суперпользователя.

Затем нажмите «+ Добавить» рядом с «Пользователи». Введите имя пользователя и пароль для нового пользователя

и нажмите на кнопку «Сохранить». Я выбрал здесь имя пользователя `testuser`. Обратите внимание, что поле Имя

доступно, но требуется ялаг ода я нашей пользовательской модели пользователя

Add user

First, enter a username and password. Then, you'll be able to edit more user options.

**Username:** testuser  
Required. 150 characters or fewer. Letters, digits and @./+/-/\_ only.

**Password:** \*\*\*\*\*  
Your password can't be too similar to your other personal information.  
Your password must contain at least 8 characters.  
Your password can't be a commonly used password.  
Your password can't be entirely numeric.

**Password confirmation:** \*\*\*\*\*  
Enter the same password as before, for verification.

**Name:**

**Actions:**

- Save and add another
- Save and continue editing
- SAVE**

Админ страница добавления пользователя

Следующим экраном является страница меню пользователя администратора. Я назвал своего пользователя `testuser`, и здесь я могу

добавить дополнительную информацию включая новую модель пользователя по умолчанию такую как имя, фамилия,

адрес электронной почты и т. д. Но для наших целей все это не нужно: нам просто нужны имя пользователя и пароль для тестирования.

The screenshot shows the Django administration interface for a user named 'testuser'. At the top, there is a success message: 'The user "testuser" was added successfully. You may edit it again below.' Below this, the 'Change user' form is displayed. The 'Username' field contains 'testuser'. The 'Password' field shows a long hash value: 'algorithm: pbkdf2\_sha256 iterations: 320000 salt: Ca9o81\*\*\*\*\* hash: FsLTNz\*\*\*\*\*'. A note below says, 'Raw passwords are not stored, so there is no way to see this user's password, but you can change the password using this form.' Under the 'Personal info' tab, there are fields for 'First name', 'Last name', and 'Email address', all of which are empty. In the 'Permissions' tab, the 'Active' checkbox is checked, and a note says, 'Designates whether this user should be treated as active. Unselect this instead of deleting accounts.' There are also checkboxes for 'Staff status' and 'Superuser status', both of which are unchecked. Below these, there is a 'Groups' section with two tabs: 'Available groups' and 'Chosen groups'. The 'Available groups' tab has a 'Filter' input field. The 'Chosen groups' tab is currently selected and is empty. At the bottom of the page, there is a footer note: 'С меню пользователя администратора'.

Прокрутите вниз страницу и нажмите на кнопку «Сохранить». Он будет перенаправлен обратно на главную страницу пользователей по адресу <http://127.0.0.1:8000/admin/auth/user/82>.

<sup>82</sup><http://127.0.0.1:8000/admin/авторизация/>

The screenshot shows the Django administration interface for the 'CustomUser' model. At the top, there's a success message: "The user "testuser" was changed successfully." On the right, there's an "ADD USER" button. The main area displays a table with columns: EMAIL ADDRESS, USERNAME, NAME, and STAFF STATUS. The table contains two rows: one for 'testuser' and one for 'wsv'. The 'testuser' row has a red asterisk icon in the 'STAFF STATUS' column, while the 'wsv' row has a green checkmark icon. To the right of the table is a "FILTER" sidebar with sections for "By staff status" (All, Yes, No), "By superuser status" (All, Yes, No), and "By active" (All, Yes, No).

## Админ два пользователя

Мы видим, что два наших пользователя перечислены. Обратите внимание, что «Статус персонала» показывает, что только одна из учетных записей предназначена для суперпользователя. В качестве последнего шага нажмите ссылку «Выход» в правом верхнем углу веб-страницы.

**ОСТАВИТЬ АДМИН.**

The screenshot shows the Django administration logout page. It displays the message "Logged out" and a link "Log in again".

Выход из администратора

## Добавить вх од и вых од

С этой настройкой, как наш новый пользователь может войти в дистривьютер для просмотра API? Как оказалось, мы можем сделать это, обновив нашу URLconf на уровне проекта. Обновите django\_project/urls.py следующим образом, указав новый путь для входа.

Код

---

```
# django_project/urls.py из
django.contrib import admin from
django.urls путь импорта, включая

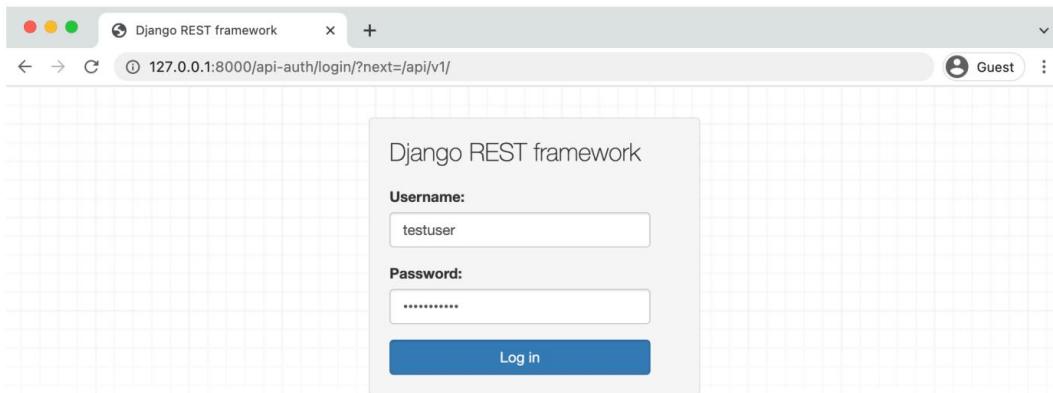
urlpatterns =
    [ path("admin/", admin.site.urls), path("api/
v1/", include("posts.urls")), path("api-auth/",
include("rest_framework. URL")), # новый
]
```

---

Теперь перейдите к нашему дистривьютеру для просмотра API по адресу <http://127.0.0.1:8000/api/v1/>. Есть тонкое изменение: ссылка «Войти» в правом верхнем углу. Нажмите на нее, чтобы войти

Ссылка для входа в API

Используйте новую учетную запись testuser для входа в систему.



Страница входа в API

Это перенаправляет обратно на страницу с списком сообщений, где в правом верхнем углу присутствует testuser вместе с стрелкой, которая показывает раскрывающую ярусную ссылку «Выход».

## Г лава 7: Разрешения

Post List

**OPTIONS** **GET**

`GET /api/v1/`

```
HTTP 200 OK
Allow: GET, POST, HEAD, OPTIONS
Content-Type: application/json
Vary: Accept

[
    {
        "id": 1,
        "author": 1,
        "title": "1st Blog Post (edited)",
        "body": "Hello, World!",
        "created_at": "2022-02-07T17:34:00.892277Z"
    }
]
```

**Raw data** **HTML form**

**Author**: wsv

**Title**:

**Body**:

**POST**

Вход в API Testuser

## Разрешения на уровне просмотра

Разрешения также могут быть добавлены на уровне представления для более детального контроля. Давайте обновим наше представление `PostDetail`, чтобы оно могло просматривать только пользователи с правами администратора. Если мы сделаем это правильно, пользователь, вышедший из системы, вообще не сможет просматривать API, пользователь, вошедший в систему, может просматривать страницу списка, но только администратор может видеть страницу сведений.

В файле `posts/views.py` импортируйте разрешения из Django REST Framework, а затем добавьте

поле permission\_classes в PostDetail , которое устанавливает его в IsAdminUser.

Код

---

```
# posts/views.py из
rest_framework import generics, разрешения# new

из .models import Post
из .serializers import PostSerializer

class PostList(generics.ListCreateAPIView): queryset =
    Post.objects.all() serializer_class = PostSerializer
```

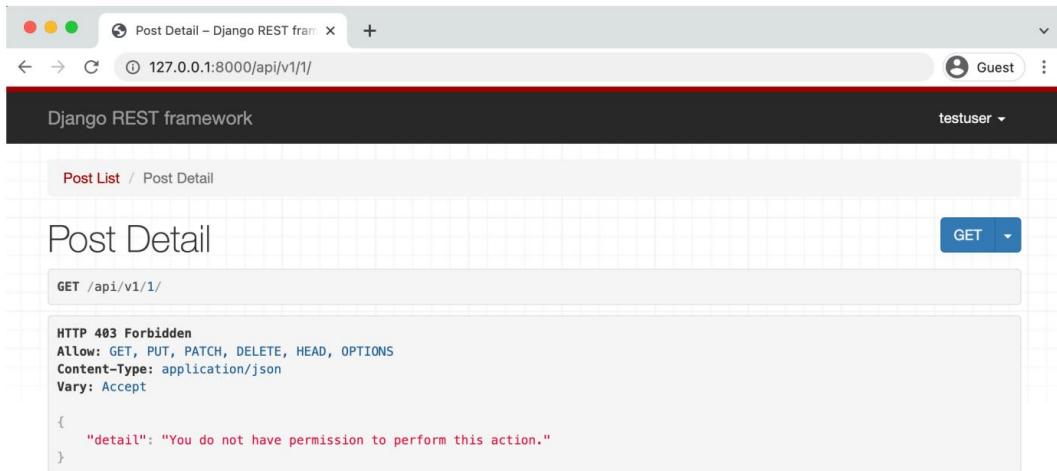
---

```
класс PostDetail(generics.RetrieveUpdateDestroyAPIView):
    permission_classes = (permissions.IsAdminUser,) # новый набор запросов
    = Post.objects.all() serializer_class = PostSerializer
```

---

Это все, что нам нужно. Обновите дистринг для просмотра API по адресу <http://127.0.0.1:8000/api/v1/> и публикацию

Страница списка по-прежнему доступна для просмотра. Однако, если вы перейдете по адресу <http://127.0.0.1:8000/api/v1/1/> , чтобы увидеть страницу сведений о публикации , отобразится код состояния HTTP 403 Forbidden .



Деталь с сообщением API 403

## Глава 7: Разрешения

Если вы выйдете из доступного для просмотра администратора, а затем войдете в свою учетную запись администратора, с правами сведений о публикации все равно будет видна. Таким образом, мы эффективно применили разрешение на уровне просмотра.

Как вы можете видеть, стандартные типы разрешений для установки: разрешить любой для полного доступа, ограничить для пользователей, прошедших проверку подлинности, ограничить пользователей-администраторов или разрешить пользователям, прошедшим проверку подлинности, выполнять любой запрос, но только для чтения для других пользователей. То, как вы настраиваете разрешения, зависит от вашего проекта потребности.

Прежде чем мы продолжим, уделите поле permission\_classes в PostDetail. Для наших целей достаточно ограничить доступ для аутентифицированных пользователей, что мы сделали в django\_project/settings.py с конфигурацией DEFAULT\_PERMISSION\_CLASSES .

## Пользовательские разрешения

Для нашего первого пользовательского разрешения мы хотим ограничить доступ, чтобы только автор сообщения в блоге мог редактировать или удалять его. Суперпользователь-администратор будет иметь полный доступ к нему, но обычный пользователь может только обновлять/удалять свой собственный контент.

Внутри Django REST Framework находится якорь класса BasePermission , от которого наследуются остальные классы разрешений. Все встроенные параметры разрешений, такие как AllowAny или IsAuthenticated , просто расширяют BasePermission . Вот фактический исходный код, который доступен на GitHub<sup>83</sup>:

### Код

---

#### Класс BasePermission (объект):

Базовый класс, от которого должны наследоваться классы разрешений.

```
def has_permission(self, request, view):
```

Возвратите «True», если разрешение предоставлено, «False» в противном случае.

вернуть Истина

```
def has_object_permission(self, request, view, obj):
```

Возвратите «True», если разрешение предоставлено, «False» в противном случае.

---

<sup>83</sup><https://github.com/encode/django-rest-framework>

|||||

[вернуть Истина](#)

---

Для пользовательского класса с разрешений вы можете определить один или оба этих метода. `has_permission` работает с представлениями списка, в то время как подробные представления выполняют оба сначала `has_permission`, а затем, если это проходит один, `has_object_permission`. Насколько рекомендуется явно указывать оба метода явно, потому что каждый из них по умолчанию имеет значение `True`, что означает, что они будут разрешать доступ неявно, если они не установлены явно.

В нашем случае мы хотим, чтобы только автор с сообщения в блоге имел права на запись для редактирования или удаления. Мы также хотим ограничить просмотр списка только для чтения аутентифицированными пользователями. Для этого мы создадим новый файл с именем `posts/permissions.py` и заполним его следующим кодом:

Код

---

```
# posts/permissions.py из
разрешений на импорт rest_framework
```

класс с `IsAuthorOrReadOnly` (разрешения `BasePermission`):

```
def has_permission(я запрос, просмотра):
    # Аутентифицированные пользователи могут видеть
    # список только если request.user.is_authenticated:
    вернуть Истина
    вернуть ложь
```

def `has_object_permission`(я запрос, просмотра, объект):

```
# Разрешение на чтение разрешены для любого запроса, поэтому мы всегда #
разрешаем запросы GET, HEAD или OPTIONS, если request.method в
разрешениях .SAFE_METHODS:
    вернуть Истина
```

```
# Разрешение на запись разрешено только автору с сообщения return obj.author ==
request.user
```

---

Мы импортируем разрешения вверху, а затем создаем собственный класс `IsAuthorOrReadOnly`, который расширяет `BasePermission`. Первый метод `has_permission`, требует, чтобы пользователь вошел в систему, или аутентифицирован, чтобы иметь доступ. Второй метод `has_object_permission` позволяет запросы только для чтения, но ограничивает права на запись только автором сообщения в блоге. Мы получаем доступ к автору через `obj.author` и текущего пользователя через `request.user`.

Вернувшись в файл views.py, мы можем удалить импорт разрешений, потому что мы заменим разрешения PostDetail.IsAdminUser на импорт нашего пользователя с его разрешениями IsAuthorOrReadOnly. Добавьте новое разрешение в классы разрешений как для PostDetail, так и для PostList.

Код

---

```
# posts/views.py из
rest_framework import generics

from .models import Post
from .permissions import IsAuthorOrReadOnly # new from .serializers
import PostSerializer

класс PostList(generics.ListCreateAPIView):
    permission_classes = (IsAuthorOrReadOnly,) # новый набор
    запросов = Post.objects.all() serializer_class = PostSerializer

класс PostDetail(generics.RetrieveUpdateDestroyAPIView): permission_classes
    = (IsAuthorOrReadOnly,) # новый набор запросов = Post.objects.all()
    serializer_class = PostSerializer
```

---

И мы закончили. Чтобы проверить это, нам нужно создать запись в блоге с testuser в качестве автора и подтвердить, что testuser может получить к ней доступ. Наша текущая учетная запись суперпользователя может видеть и делать все по умолчанию. Перейдите по адресу <http://127.0.0.1:8000/admin/> и войдите в систему как суперпользователь. Затем создайте новый пост с testuser в качестве автора.

The screenshot shows the Django administration interface for adding a new post. The browser title is "Add post | Django site admin". The URL in the address bar is "127.0.0.1:8000/admin/posts/post/add/". The top right shows a "Guest" user. The main header says "Django administration" and "WELCOME, WSV. VIEW SITE / CHANGE PASSWORD / LOG OUT". Below it, the breadcrumb navigation shows "Home > Posts > Posts > Add post". The form has fields for "Title" (containing "Second blog post"), "Body" (containing "Testing out our custom permissions."), and "Author" (set to "testuser"). At the bottom, there are three buttons: "Save and add another", "Save and continue editing", and a large blue "SAVE" button.

После его создания перейдите к конечной точке Post Detail по адресу <http://127.0.0.1:8000/api/v1/2/>.

Используйте раскрытое меню в правом верхнем углу, чтобы «Выйти» из своей учетной записи и суперпользователя и войти в систему как тестовый пользователь.

Django REST framework

testuser

Post List / Post Detail

## Post Detail

[DELETE](#) [OPTIONS](#) [GET](#)

`GET /api/v1/2/`

```
HTTP 200 OK
Allow: GET, PUT, PATCH, DELETE, HEAD, OPTIONS
Content-Type: application/json
Vary: Accept

{
    "id": 2,
    "author": 2,
    "title": "Second blog post",
    "body": "Testing out our custom permissions.",
    "created_at": "2022-02-08T18:44:29.008882Z"
}
```

[Raw data](#) [HTML form](#)

Author	testuser
Title	Second blog post
Body	Testing out our custom permissions.

[PUT](#)

Сведения о сообщении TestUser

Да! Есть варианты редактирования или удаления записи, поскольку testuser является автором. Однако если вы перейдете на страницу с информацией о первом сообщении в блоге по адресу <http://127.0.0.1:8000/api/v1/1/>, оно будет доступно только для чтения поскольку testuser не был автором.

```

HTTP 200 OK
Allow: GET, PUT, PATCH, DELETE, HEAD, OPTIONS
Content-Type: application/json
Vary: Accept

{
    "id": 1,
    "author": 1,
    "title": "1st Blog Post (edited)",
    "body": "Hello, World!",
    "created_at": "2022-02-07T17:34:00.892277Z"
}

```

Деталь с обложкой TestUser не является автором

Чтобы убедиться что наши элементы управления аутентификацией работают правильно, выйдите из системы в правом верхнем углу.

Затем перейдите к конечной точке «Список публикаций» и к двум конечным точкам «Сведения о публикации», чтобы подтвердить, что вышедший из системы пользователь не доступен.

Чтобы закончить, мы должны закоммитить нашу новую работу в Git.

Ракурс

---

```
(.venv) > git status (.venv)
> git add -A (.venv) > git
commit -m "добавить разрешения"
```

---

## Заключение

Установка правильных разрешений — очень важная часть любого API. В качестве общей стратегии рекомендуется установить строгую политику разрешений на уровне проекта, чтобы только пользователи, прошедшие проверку подлинности, могли просматривать API. Затем сделайте разрешения на уровне представления или настраиваемые разрешения более доступными по мере необходимости на определенных конечных точках API.

## Г лава 8: Аутентификац ия пользователя

В предыдущей главе мы обновили наши разрешения API, которые также называются авторизацией. В этой главе мы реализуем аутентификацию, которая представляет собой процесс, с помощью которого пользователь может зарегистрировать новую учетную запись, войти с ее помощью или выйти из нее.

В традиционном монолитном веб-сайте Django аутентификация проще и включает шаблон cookie на основе сессии, который мы рассмотрим ниже. А вот с API все немного сложнее.

Помните, что HTTP — это протокол без сохранения состояния, поэтому нет встроенных способов запомнить, аутентифицирован ли пользователь от одного запроса к другому. Каждый раз, когда пользователь запрашивает ограниченный ресурс, он должен подтвердить себя.

Решение состоит в том, чтобы передавать уникальный идентификатор с каждым HTTP-запросом. Как ни странно, не существует общепринятого подхода к форме этого идентификатора, и он может принимать несколько форм. Django REST Framework предоставляет ясные [четырьмя различными встроенными вариантами аутентификации](#): базовый, сессии и по умолчанию есть много других сторонних пакетов, которые предлагают дополнительные функции, такие как веб-токены JSON (JWT).

В этой главе мы подробно рассмотрим, как работает аутентификация API, рассмотрим плюсы и минусы каждого подхода, а затем сделаем осознанный выбор в пользу нашего Blog API. К концу мы создадим конечные точки API для регистрации, входа и выхода.

### Базовая аутентификация

Наиболее распространенная форма HTTP-аутентификации известна как [«базовая аутентификация»](#). Когда клиент делает HTTP-запрос, он вынужден отправить установленные учетные данные для проверки подлинности, прежде чем будет предоставлен доступ.

Полный поток запроса/ответа выглядит следующим образом:

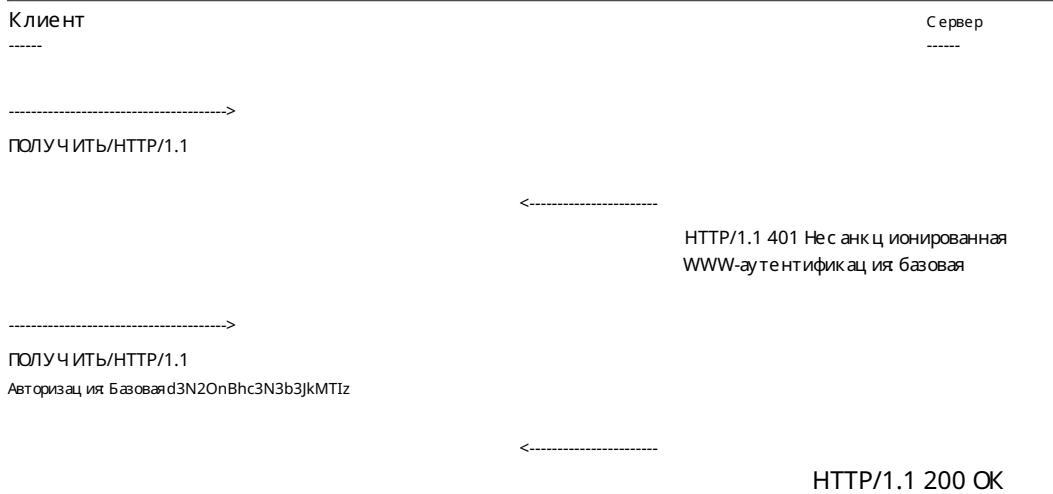
---

<sup>84</sup><https://www.django-rest-framework.org/api-guide/authentication/#api-reference> <sup>85</sup><https://tools.ietf.org/html/rfc7617>

1. Клиент делает HTTP-запрос
2. Сервер отвечает HTTP-ответом, с кодом состояния 401 (Unauthorized) и HTTP-заголовок WWW-Authenticate с подробной информацией о том, как авторизоваться
3. Клиент отправляет учетные данные обратно через Authorization<sup>86</sup>. HTTP-заголовок
4. Сервер проверяет учетные данные и отвечает кодом состояния 200 OK или 403 Forbidden.

После утверждения клиент отправляет все будущие запросы с учетными данными HTTP-заголовка авторизации. Мы также можем визуализировать этот обмен следующим образом:

#### Диаграмма



Обратите внимание, что отправленные учетные данные авторизации являются явлением незашифрованными [base64 encoded](#)<sup>87</sup>. Весьма <имя пользователя>:<пароль>. Итак, в моем случае это wsv:password123, который с кодировкой base64 d3N2OnBhc3N3b3JkMTIz.

Основным преимуществом этого подхода является его простота. Но есть несколько существенных минусов. Во-первых, при каждом отдельном запросе сервер должен искать и проверять имя пользователя и пароль, что неэффективно. Было бы лучше выполнить поиск один раз, а затем передать какой-либо токен, который говорит, что этот пользователь одобрен. Во-вторых, учетные данные пользователя передаются в открытом виде, а не в зашифрованном виде, через Интернет. Это невероятно небезопасно. Любой интернет-трафик, не

<sup>86</sup><https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Авторизация><sup>87</sup><https://en.wikipedia.org/wiki/Base64>

зашифрованные могут быть легкоперевачены и использованы повторно. Таким образом, базовую аутентификацию следует использовать только через [HTTPS<sup>88</sup>](#), безопасная версия HTTP.

## Аутентификация сессии

Монолитные веб-сайты, такие как традиционный Django, уже давно используют альтернативную ему аутентификации, которая представляет собой комбинацию сессий и файлов cookie. На высоком уровне клиент аутентифицируется с помощью своих учетных данных (имя пользователя/пароль), а затем получает идентификатор сессии от сервера, который со временем хранится в виде файла cookie. Затем этот идентификатор сессии передается в заголовке каждого будущего HTTP-запроса.

Когда передается идентификатор сессии сервер использует его для поиска объекта сессии, содержащего всю информацию о пользователе, включая учетные данные.

Этот подход учитывает состояние, поскольку запись должна храниться и поддерживаться как на сервере (объект сессии), так и на клиенте (идентификатор сессии).

Давайте рассмотрим основной поток:

1. Пользователь вводит свои учетные данные для входа (обычно имя пользователя/пароль)
2. Сервер проверяет правильность учетных данных и создает объект сессии, который затем хранится в базе данных
3. Сервер отправляет клиенту идентификатор сессии, а не сам объект сессии, который со временем хранится в виде куки в браузере
4. Во все будущие запросы идентификатор сессии включается в качестве заголовка HTTP и, если он базовый, запрос продолжается
5. Как только пользователь выходит из приложения, идентификатор сессии уничтожается как клиентом, так и сервером
6. Если позже пользователь снова войдет в систему, новый идентификатор сессии будет сгенерирован и со временем хранен в виде файла cookie на клиенте

Настойка по умолчанию Django REST Framework на самом деле представляет собой комбинацию базовой аутентификации и аутентификации сессии. Используется стандартная тема аутентификации Django на основе сессий, и идентификатор сессии передается в заголовке HTTP при каждом запросе через обычную аутентификацию.

---

<sup>88</sup><https://en.wikipedia.org/wiki/HTTPS>

Преимущество этого подхода заключается в том, что он более безопасен, поскольку учетные данные пользователя отправляются только один раз, а не при каждом цикле запроса/ответа, как в базовой аутентификации. Это также более эффективно, поскольку серверу не нужно каждый раз проверять учетные данные пользователя, он просто сопоставляет идентификатор сессии с объектом сессии, что является ябыстрым присоединением.

Однако есть несколько недостатков. Во-первых, идентификатор сессии действителен только в браузере, в котором был выполнен вход; он не будет работать в нескольких доменах. Это очевидная проблема, когда API должен поддерживать несколько интерфейсов, таких как веб-сайт и мобильное приложение. Во-вторых, объект сессии должен поддерживаться в актуальном состоянии, что может быть затруднительно на больших сайтах с несколькими серверами. Как вы поддерживаете точность объекта сессии на каждом сервере? И в-третьих, файл cookie отправляется для каждого отдельного запроса, даже для тех, которые не требуют аутентификации, что неэффективно.

В результате обычно не рекомендуется использовать сессию аутентификации на основе сессии для любого API, который будет иметь несколько внешних интерфейсов.

## Токен Аутентификации

Третий основной подход, который мы реализуем в нашем Blog API, — это использование аутентификации с помощью токена. Это самый популярный подход в последние годы из-за роста числа односервичных приложений.

Аутентификация на основе токенов не имеет состояния, так как только клиент отправляет первоначальные учетные данные пользователя на сервер, генерируя уникальный токен, который затем сохраняется клиентом либо в виде файла cookie, либо в локальном хранилище<sup>89</sup>. Затем этот токен передается в заголовке каждого запроса на HTTP-запрос, и сервис использует его для проверки подлинности пользователя. С сервером не ведется записи о пользователе, а только о том, действителен ли токен или нет.

Файлы cookie против локального хранилища

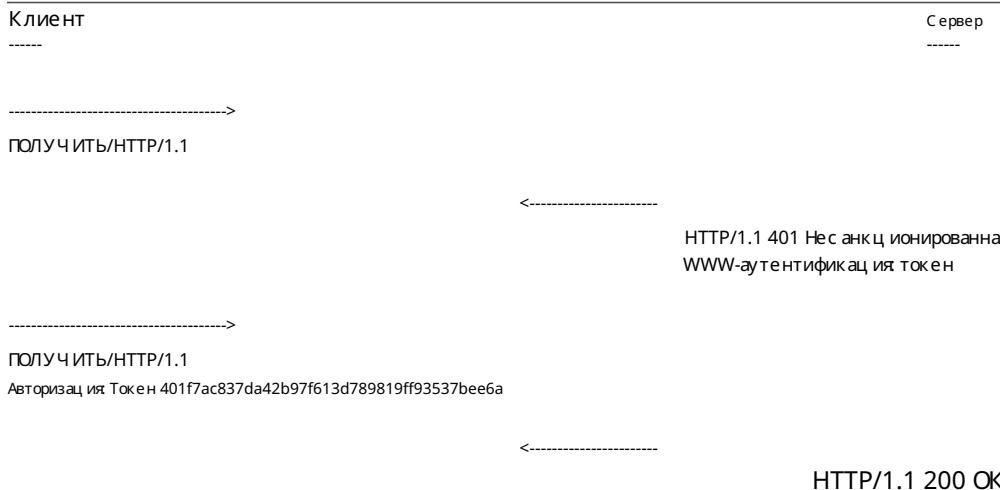
Файлы cookie используются для хранения информации на стороне сервера. Они меньше (4 КБ) по размеру и автоматически отправляются с каждым HTTP-запросом. LocalStorage предназначен для хранения информации на стороне клиента. Он намного больше (5120 КБ), и его содержимое по умолчанию отправляется с каждым HTTP-запросом.

<sup>89</sup><https://developer.mozilla.org/en-US/docs/Web/API/Window/localStorage>

Токены, хранящиеся как в файлах cookie, так и в localStorage, уязвимы для XSS-атак. В настоящее время рекомендуется хранить токены в файле cookie с флагами httpOnly и Secure cookie.

Давайте посмотрим на простую версию настолько HTTP-сообщений в этом потоке вызов/ответ. Обратите внимание, что заголовок HTTP WWW-Authenticate указывает на использование токена, который используется в ответном запросе заголовка авторизации.

Диаграмма



Этот подход имеет множество преимуществ. Поскольку токены хранятся на клиенте, масштабирование серверов для поддержания актуальных объектов гораздо проще, чем для баз данных. И токены могут использоваться не только для интерфейсов API: один и тот же токен может представлять пользователя на веб-сайте и одновременно на пользователе в мобильном приложении. Один и тот же идентификатор не может быть разделен между различными внешними интерфейсами, что является серьезным ограничением.

Поговорим о недостатком: это то, что токены могут вырасти довольно большими. Маркер содержит всю информацию о пользователе, а не только идентификатор, как в случае с идентификатором объекта сущности. Поскольку токен отправляется при каждом запросе, управление его размером может стать проблемой производительности.

То, как именно реализуется токен, также может существенно отличаться в Django REST фреймворки

встроенный `TokenAuthentication`<sup>90</sup> предназначен довольно просто. В результате он не поддерживает установку с рекомендацией токенов, что является улучшением безопасности, которое можно добавить. Он также генерирует только один токен для каждого пользователя по тому пользователю на веб-сайте, а затем в мобильном приложении будет использовать один и тот же токен. Поскольку информация о пользователе хранится локально, это может вызвать проблемы с поддержкой и обновлением двух наборов информации о клиенте.

Веб-токены JSON (JWT) — это новая форма токена, содержащая криптографические подписанные JSON. JWT изначально были разработаны для использования в `OAuth`<sup>91</sup>, открытый стандартный способ обмена паролями пользователей. JWT могут быть сгенерированы на сервере с помощью стороннего пакета, такого как `djangorestframework-simplejwt`<sup>92</sup>. или через сторонний сервис, например Auth0. Однако среди разработчиков продолжают споры о плюсах и минусах использования JWT для аутентификации пользователей, и их надлежащее осуществление выходит за рамки этой книги. Вот почему в этой книге мы будем придерживаться встроенной `TokenAuthentication`.

## Аутентификация по умолчанию

Первый шаг — настроить наши новые параметры аутентификации. Django REST Framework поставляется с рядом настроек<sup>93</sup>, которые задаются явно. Например, для `DEFAULT_PERMISSION_CLASSES` было задано значение `AllowAny` до того, как мы обновили его до `IsAuthenticated`.

`DEFAULT_AUTHENTICATION_CLASSES` установлены по умолчанию по тому, давайте явно добавим как `SessionAuthentication`, так и `BasicAuthentication` в наш файл `django_project/settings.py`.

---

<sup>90</sup><http://www.django-rest-framework.org/api-guide/authentication/#tokenauthentication> <sup>91</sup><https://en.wikipedia.org/wiki/OAuth> <sup>92</sup><https://github.com/jazzband/djangorestframework-simplejwt> <sup>93</sup><http://www.django-rest-framework.org/api-guide/settings/>

Код

---

```
# django_project/settings.py
REST_FRAMEWORK =
    { "DEFAULT_PERMISSION_CLASSES":
        [ "rest_framework.permissions.IsAuthenticated",
        ],
    "DEFAULT_AUTHENTICATION_CLASSES": [ # new
        "rest_framework.authentication.SessionAuthentication",
        "rest_framework.authentication.BasicAuthentication",
    ],
}
```

---

Зачем использовать оба метода? Ответ заключается в том, что они служат разным целям. Сеансы ис пользуются для обес печения работы Browsable API и возможнос ти вх оди и вых оди из него. BasicAuthentication ис пользуетс я для прохождения идентификатор сеанса в заголовках HTTP для с амог о API.

Если вы с нова посетите API для просмотра по адресу <http://127.0.0.1:8000/api/v1/>, он будет работать так же, как и раньше.

Технически ничего не изменилось, мы просто сделали настройки по умолчанию явными.

## Внедрение аутентификации по токену

Теперь нам нужно обновить нашу систему аутентификации, чтобы использовать токены. Первый шаг — обновить наш параметр DEFAULT\_AUTHENTICATION\_CLASSES для использования TokenAuthentication следующим образом:

Код

---

```
# django_project/settings.py
REST_FRAMEWORK =
    { "DEFAULT_PERMISSION_CLASSES":
        [ "rest_framework.permissions.IsAuthenticated",
        ],
    "DEFAULT_AUTHENTICATION_CLASSES":
        [ "rest_framework.authentication.SessionAuthentication",
        "rest_framework.authentication.TokenAuthentication", # новый
    ],
}
```

---

Мы сняли SessionAuthentication, так как он нам по-прежнему нужен для нашего Browsable API, но теперь мы используем токены для передачи учетных данных аутентификации туда и обратно в наших HTTP-заголовках. Нам также необходимо добавить

приложение `authtoken`, которое генерирует токены на сервере. Он входит в состав Django REST Framework, но его необходимо добавить в нашу настройку `INSTALLED_APPS`:

Код

---

```
# django_project/settings.py
INSTALLED_APPS =
    [ "django.contrib.admin",
      "django.contrib.auth",
      "django.contrib.contenttypes",
      "django.contrib.sessions",
      "django.contrib.messages",
      "django.contrib.staticfiles",

      # Сторонние
      # приложения
      "rest_framework",
      "corsheaders", "rest_framework.authtoken", # новые

      # Местный
      "accounts.apps.AccountsConfig",
      "posts.apps.PostsConfig",
  ]
```

---

Поскольку мы внесли изменения в наши `INSTALLED_APPS`, нам необходимо синхронизировать нашу базу данных.

Откройте консоль сервера, нажав `Ctrl+C`. Затем выполните следующую команду.

Ракушка

---

```
(.venv) > python manage.py migrate Операции
для выполнения
```

Применить все миграции: учетные записи, администрирование, аутентификация, авторизация, типы контента, сообщения, сеансы.

Запуск миграций:

Применение `authtoken.0001_initial...` OK

Применение `authtoken.0002_auto_20160226_1747...` OK

Применение `authtoken.0003_tokenproxy...` OK

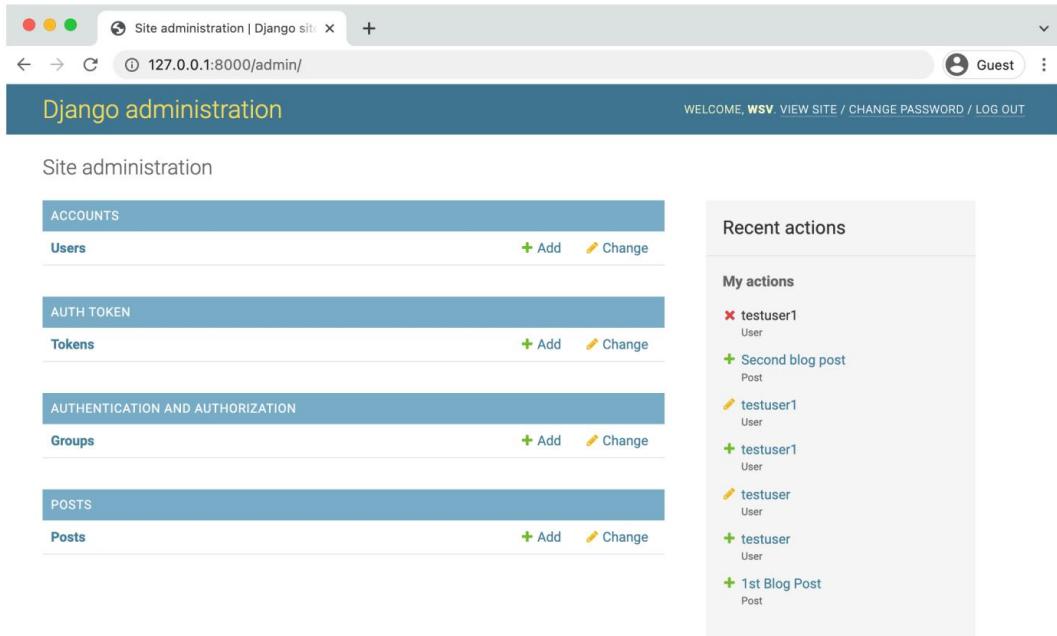
---

Теперь снова запустите сервер.

Ракурска

(.venv) &gt; сервер запуска python manage.py

Если вы перейдете к администратору Django по адресу <http://127.0.0.1:8000/admin/>, вы увидите, что теперь вверху есть раздел «Токены». Убедитесь, что вы вошли в свою учетную запись суперпользователя чтобы дос тул.



[Домашняя страница администратора с токенами](#)

Нажмите на ссылку для токенов. В настоящий момент нет токенов, что может удивить.



Ведь у нас есть существующие пользователи. Однако токены генерируются только после вызова API для всех пользователей в системе.

Мы еще не делали, поэтому токенов нет. Мы скоро!

## Конечные точки

Нам также необходимо создать конечные точки, чтобы пользователи могли выходить из системы. Мы могли бы создать специальное пользовательское приложение для этой цели, а затем добавить сюда собственные URL-адреса представления и сериализаторы. Однако аутентификация пользователя — это область, в которой мы действительно не хотим ошибаться. И поскольку почти все API требует этой функциональности, имеет смысл использовать несколько отличных и проверенных сторонних пакетов, которые мы можем использовать вместе.

В частности, мы будем использовать [dj-rest-auth<sup>94</sup>](#), в сочетании с [django-allauth<sup>95</sup>](#) упростить вещи. Не расстраивайтесь из-за использования сторонних пакетов. Они существуют не просто так, и даже лучшие профессионалы Django все время полагаются на них. Нет смысла изобретать велосипед, если вы это не делаете.

придетс я

## dj-rest-auth

Сначала мы добавим конечные точки API в базу данных и сброс пароля. Они поставляются из коробки с популярным пакетом [dj-rest-auth](#). Установите сервер с помощью `Control+C`, а затем установите его.

---

<sup>94</sup><https://github.com/jazzband/dj-rest-auth> <sup>95</sup><https://github.com/pennersr/django-allauth>

Ракурска

---

```
(.venv) > python -m pip install dj-rest-auth==2.1.11
```

---

Добавьте новое приложение в конфигурацию INSTALLED\_APPS в нашем файле django\_project/settings.py .

Код

---

```
# django_project/settings.py
INSTALLED_APPS =
    [ "django.contrib.admin",
      "django.contrib.auth",
      "django.contrib.contenttypes",
      "django.contrib.sessions",
      "django.contrib.messages",
      "django.contrib.staticfiles",

      # Сторонние
      # приложения
      "rest_framework",
      "corsheaders",
      "rest_framework.authtoken", "dj_rest_auth", # new

      # Местный
      "accounts.apps.AccountsConfig",
      "posts.apps.PostsConfig",
  ]
```

---

Обновите наш файл django\_project/urls.py с помощью пакета dj\_rest\_auth . Мы устанавливаем URL-маршруты на api/v1/dj-rest-auth . Обратите внимание, что URL-адреса должны иметь тире, а не подчеркивать \_, чтобы избежать ошибок .

## Код

---

```
# django_project/urls.py из
django.contrib import admin from
django.urls путь импорта, включая

urlpatterns =
    [ path("admin/", admin.site.urls), path("api/
v1/", include("posts.urls")), path("api-auth/",
include("rest_framework. urls")), path("api/v1/dj-rest-auth/",
include("dj_rest_auth.urls")), # новый
]
```

---

И мы закончили! Если вы когда-либо пытались внедрить сюда собственные конечные точки аутентификации пользователей, вы действительно удивитесь, сколько времени и головной боли нам это потребует dj-rest-auth. Теперь мы можем запустить сервер, чтобы увидеть, что предоставил dj-rest-auth.

## Ракушка

---

```
(.venv) > сервер запуска python manage.py
```

---

У нас есть рабочий журнал в конечной точке по адресу <http://127.0.0.1:8000/api/v1/dj-rest-auth/login/>.

Django REST framework

Post List / Login

## Login

Check the credentials and return the REST Token if the credentials are valid and authenticated.  
Calls Django Auth login method to register User ID in Django session framework

Accept the following POST parameters: username, password  
Return the REST Framework Token Object's key.

**OPTIONS**

**GET /api/v1/dj-rest-auth/login/**

```
HTTP 405 Method Not Allowed
Allow: POST, OPTIONS
Content-Type: application/json
Vary: Accept

{
    "detail": "Method \"GET\" not allowed."
}
```

**Raw data** **HTML form**

Username

Email

Password

**POST**

Конечная точка входа в API

И конечная точка входа по адресу <http://127.0.0.1:8000/api/v1/dj-rest-auth/logout/>.

The screenshot shows a browser window with the title "Logout - Django REST framework". The URL in the address bar is "127.0.0.1:8000/api/v1/dj-rest-auth/logout/". The page header includes a user icon labeled "Guest" and a dropdown menu. The main content area is titled "Django REST framework" and "Logout". Below the title, it says "Calls Django logout method and delete the Token object assigned to the current User object." A note indicates "Accepts/Returns nothing." There are two buttons at the top right: "OPTIONS" and "GET". The "GET" button is highlighted. A code block shows the response for a GET request:

```
HTTP 405 Method Not Allowed
Allow: GET, POST, HEAD, OPTIONS
Content-Type: application/json
Vary: Accept

{
    "detail": "Method \"GET\" not allowed."
}
```

Below this, there is a form for a POST request. It has a "Media type:" dropdown set to "application/json" and a "Content:" text area. A "POST" button is located at the bottom right of the form.

Конечная точка вьюхи из API

Существуют также конечные точки для сброса пароля, расположенные по адресу:

<http://127.0.0.1:8000/api/v1/dj-rest-auth/пароль/сброс>

The screenshot shows a browser window displaying the Django REST framework's browsable API interface. The URL is `127.0.0.1:8000/api/v1/dj-rest-auth/password/reset/`. The page title is "Password Reset – Django REST". The main content area is titled "Password Reset" and contains the following information:

- Calls Django Auth PasswordResetForm save method.
- Accepts the following POST parameters: email
- Returns the success/fail message.

A "GET /api/v1/dj-rest-auth/password/reset/" button is shown, which triggers an "HTTP 405 Method Not Allowed" response with the following headers and body:

```
HTTP 405 Method Not Allowed
Allow: POST, OPTIONS
Content-Type: application/json
Vary: Accept

{
    "detail": "Method \"GET\" not allowed."
}
```

Below this, there is a form to enter an "Email" address, with "Raw data" and "HTML form" tabs above it, and a "POST" button to its right.

Сброс пароля API

И для подтверждения сброса пароля

<http://127.0.0.1:8000/api/v1/dj-rest-auth/пароль/сброс/подтверждение>

Post List / Password Reset / Password Reset Confirm

## Password Reset Confirm

OPTIONS

Password reset e-mail link is confirmed, therefore this resets the user's password.

Accepts the following POST parameters: token, uid, new\_password1, new\_password2  
Returns the success/fail message.

`GET /api/v1/dj-rest-auth/password/reset/confirm/`

**HTTP 405 Method Not Allowed**  
**Allow:** POST, OPTIONS  
**Content-Type:** application/json  
**Vary:** Accept

```
{
    "detail": "Method \"GET\" not allowed."
}
```

New password1

New password2

Uid

Token

Raw data    HTML form

POST

Сброс пароля API Подтвердить

## Регистрация пользователя

Далее идет регистрация пользователя или регистрация конечной точки. Традиционный Django не поставляется с овстроенными представлениями или URL-адресами для регистрации пользователей, равно как и Django REST Framework. Это означает, что нам нужно написать собственный код с нуля, не сколько рискованный подход, учитывая серьезность и последствия для безопасности неправильного понимания.

Популярным подходом является использование стороннего пакета [django-allauth](#)<sup>96</sup>, который поставляется с исходным кодом пользователем, а также рядом дополнительных функций системы аутентификации Django, таких как социальная аутентификация через Facebook, Google, Twitter и т. д. Если мы добавим dj\_rest\_auth.registration из пакета dj-rest-auth, у нас будет регистрация пользователя. Концы тоже!

Установите локальный сервер с помощью Control+C и установите django-allauth.

Ракушка

---

```
(.venv) > python -m pip install django-allauth~=0.48.0
```

---

Затем обновите нашу настройку INSTALLED\_APPS. Мы должны добавить несколько новых конфигов:

- django.contrib.sites
- аллаут
- allauth.аккаунт
- allauth.socialaccount
- dj\_rest\_auth.registration

Код

---

```
# django_project/settings.py
INSTALLED_APPS =
    [
        "django.contrib.admin",
        "django.contrib.auth",
        "django.contrib.contenttypes",
        "django.contrib.sessions",
        "django.contrib.messages",
        "django.contrib.staticfiles",
        "django.contrib.sites", # новый

        # Сторонние
        # приложения
        "rest_framework",
        "corsheaders",
        "rest_framework.authtoken",
        "allauth", # new "allauth.account", #
        new "allauth.socialaccount", # new
        "dj_rest_auth",
```

---

<sup>96</sup><https://github.com/pennersr/django-allauth>

```
"dj_rest_auth.registration", # новый  
# Местный  
"accounts.apps.AccountsConfig",  
"posts.apps.PostsConfig",  
]
```

---

django-allauth необх одимо добавить в конфиг урац июTEMPLATES после сущес твующих проце ссров контекста, а также настрайтъ EMAIL\_BACKEND на консоль и добавить SITE\_ID , равный 1 .

Код

```
# django_project/settings.py ШАБЛОНЫ  
=[  
 {  
 "BACKEND": "django.template.backends.django.DjangoTemplates", "DIRS": [],  
 "APP_DIRS": True, "OPTIONS": {  
  
 "КОНТЕКСТНЫЕ_ПРОЦЕССОРЫ": [  
 "django.template.context_processors.debug",  
 "django.template.context_processors.request",  
 "django.contrib.auth.context_processors.auth",  
 "django.contrib.messages.context_processors.messages",  
 "django.template.context_processors.запрос ", # новый  
 ],  
 },  
 },  
 ],  
EMAIL_BACKEND = "django.core.mail.backends.console.EmailBackend" # новый  
SITE_ID = 1 # новый
```

---

Необх одима внутрення конфигурац ияэ лектронной почты, пос кольку по умолчанию при ре гистрац ии нового пользователя будет отправлено э лектронное письмо с просьбой подтвердить свою учетную запись. Вместо этого, чтобы также настрайтъ с ервер э лектронной почты, мы будем выводить э лектронные письма на консоль с настрайкой console.EmailBackend .

SITE\_ID является ячаком троеной с реды «сайтов» Django97, э то с пос об размес тить нес колько веб-сайтов из одног о проекта Django. У нас есть только один сайт, над которым мы работаем, но django-allauth ис пользует структуру сайтов, пос тому мы должны указать настройку по умолчанию

<sup>97</sup><https://docs.djangoproject.com/en/4.0/ref/contrib/sites/>

Хорошо. Мы добавили новые приложения, так что пришло время обновить базу данных.

Ракушка

---

(.venv) > перенос python manage.py

---

Затем добавьте новый маршрут URL для регистрации.

Код

---

```
# django_project/urls.py из
django.contrib import admin from
django.urls путь импорта, включая

urlpatterns =
    [ path("admin/", admin.site.urls), path("api/
        v1/", include("posts.urls")), path("api-auth/",
        include("rest_framework.urls")), path("api/v1/dj-rest-auth/",
        include("dj_rest_auth.urls")), path("api/v1/dj-rest-auth/registration/", # new
        include ("dj_rest_auth.registration.urls")),
```

]

---

И мы закончили. Мы можем запустить локальный сервер.

Ракушка

---

(.venv) > сервер запуска python manage.py

---

Теперь конечная точка регистрации пользователей находится по адресу:

<http://127.0.0.1:8000/api/v1/dj-rest-auth/registration/>.

## Токены

Чтобы убедиться, что все работает, создайте третью учетную запись пользователя через новую конечную точку регистрации пользователя. Я назвал ее тестовым пользователем testuser1.

Post List / Register

## Register

**OPTIONS**

**GET /api/v1/dj-rest-auth/registration/**

```
HTTP 405 Method Not Allowed
Allow: POST, OPTIONS
Content-Type: application/json
Vary: Accept

{
    "detail": "Method \"GET\" not allowed."
}
```

**Raw data** **HTML form**

Username	testuser1
Email	testuser1@email.com
Password1	testpass123
Password2	testpass123

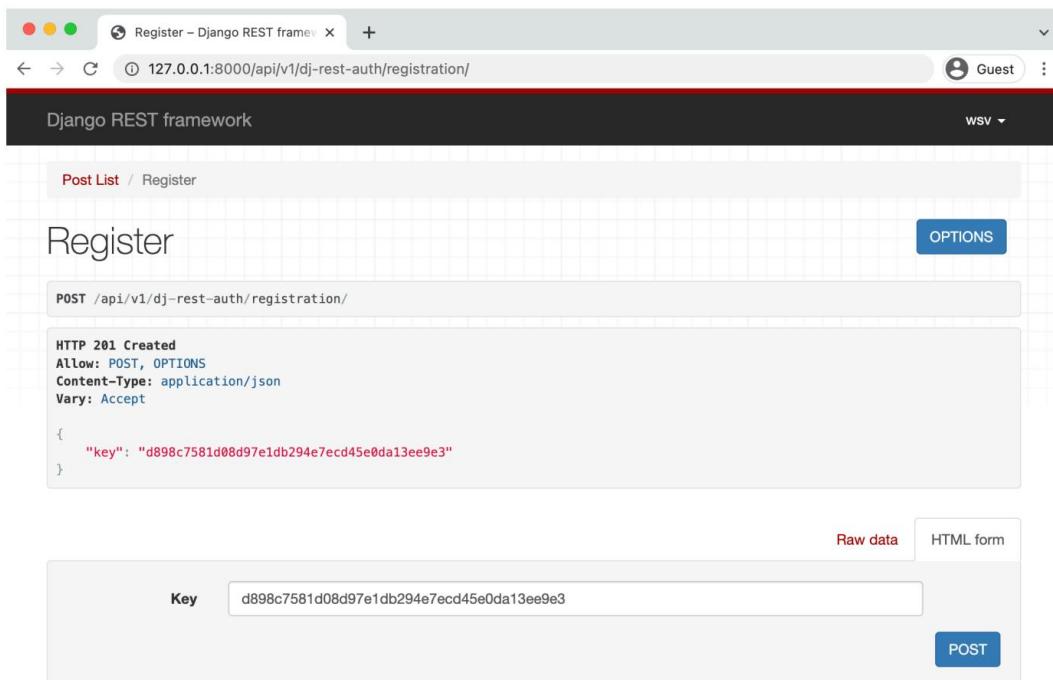
**POST**

API Зарегистрировать нового пользователя

После нажатия на кнопку «POST» на следующем экране отображается HTTP-ответ от сервера.

Наша регистрация пользователя POST прошла успешно, поэтому код состояния HTTP 201 Created вверху.

Клиент возвращает значение — это токен аутентификации для этого нового пользователя.



Если вы посмотрите на консоль командной строки, это электронное письмо было автоматически сгенерировано django-allauth.  
Этот текст по умолчанию можно обновить и добавить SMTP-сервера электронной почты с дополнительной настройкой, описанной в книге Django для начинающих.

Content-Type: текстовый/обычный; кодировка="utf-8"

MIME version 1.0

Content Transfer Encoding: 7bit TeMoX [example.com]

Пожалуйста, подтвердите свой адрес электронной почты. От: webmaster@localhost. Ком:

testuser1@email.com Дата: Среда, 09 февраля 2022 г. 16:22:38 -0000 Message-ID:

Привет с example.com!

Вы получили это электронное письмо, поскольку пользователь testuser1 предоставил ваш адрес электронной почты для регистрации учетной записи на сайте example.com.

Чтобы убедиться, что это правильно, перейдите по адресу [http://127.0.0.1:8000/api/v1/dj-rest-auth/registration/account-confirm-email/MQ:1nHrjq:D1vZokItkCU2bqKO\\_g9cmA\\_hf2fThyl6vgtC7CpNdfI/](http://127.0.0.1:8000/api/v1/dj-rest-auth/registration/account-confirm-email/MQ:1nHrjq:D1vZokItkCU2bqKO_g9cmA_hf2fThyl6vgtC7CpNdfI/).

Спасибо за использование example.com!  
пример.com

---

[09/фев/2022 16:22:38] "POST /api/v1/dj-rest-auth/registration/HTTP/1.1" 201 7828

---

Переключитесь на администратора Django в веб-браузере по адресу <http://127.0.0.1:8000/admin/>. Для этого вам нужно будет использовать свою учетную запись суперпользователя. Здесь есть несколько новых полей, которые добавил django-allauth. Нажмите на ссылку для токенов, и вы будете перенаправлены на токены.

страница.

KEY	USER	CREATED
d898c7581d08d97e1db294e7ecd45e0da13ee9e3	testuser1	Feb. 9, 2022, 4:22 p.m.

Django REST Framework сгенерировал один токен для пользователя testuser1. При создании дополнительных пользователей через API их токены также будут отображаться здесь.

**Логичный вопрос:** Почему нет токенов для нашей учетной записи суперпользователя или тестового пользователя? Ответ заключается в том, что мы создали эти учетные записи до того, как была добавлена аутентификация по токену. Но не беспокойтесь, как только мы войдем в любую учетную запись через API, токен будет автоматически добавлен и досгенерирован.

Давайте дальше, давайте войдем в нашу новую учетную запись testuser1. Обязательно выйдите из системы администратора, а затем в веб-браузере перейдите по адресу <http://127.0.0.1:8000/api/v1/dj-rest-auth/login/>. Введите информацию для нашей учетной записи testuser1. Нажмите на кнопку «РАЗМЕСТИТЬ».

Check the credentials and return the REST Token if the credentials are valid and authenticated.  
Calls Django Auth login method to register User ID in Django session framework

Accept the following POST parameters: username, password  
Return the REST Framework Token Object's key.

```
GET /api/v1/dj-rest-auth/login/
HTTP 405 Method Not Allowed
Allow: POST, OPTIONS
Content-Type: application/json
Vary: Accept

{
    "detail": "Method \"GET\" not allowed."
}
```

<b>Raw data</b>	<b>HTML form</b>
<input type="text" value="testuser1"/> <input type="text" value="testuser1@email.com"/> <input type="password" value="*****"/>	<b>POST</b>

Вход в API testuser1

Произошли две вещи. В правом верхнем углу видна наша учетная запись пользователя `testuser1`, подтверждая, что мы вошли в систему. Также сервер отправил ответ HTTP с токеном.

Django REST framework

testuser1 ▾

Post List / Login

## Login

Check the credentials and return the REST Token if the credentials are valid and authenticated.  
Calls Django Auth login method to register User ID in Django session framework

Accept the following POST parameters: username, password  
Return the REST Framework Token Object's key.

**POST** /api/v1/dj-rest-auth/login/

```
HTTP 200 OK
Allow: POST, OPTIONS
Content-Type: application/json
Vary: Accept

{
    "key": "d898c7581d08d97e1db294e7ecd45e0da13ee9e3"
}
```

Raw data    HTML form

Key: d898c7581d08d97e1db294e7ecd45e0da13ee9e3

POST

## Токен вх ода в API

В нашей интерфейсной среде нам нужно было бы захватить и схранить этот токен. Традиционно это происходит на клиенте либо в `localStorage`<sup>98</sup> или как файл cookie, а затем все будущие запросы включают токен в заголовок как способ аутентификации пользователя. Обратите внимание, что в этой теме есть дополнительные проблемы с безопасностью, поэтому вам следует позаботиться о том, чтобы внедрить лучшие практики вашего внешнего интерфейса.

Чтобы закончить, мы должны закоммитить нашу новую работу в Git.

<sup>98</sup><https://developer.mozilla.org/en-US/docs/Web/API/Window/localStorage>

Ракушка

---

```
(.venv) > git status (.venv)
> git add -A (.venv) > git
commit -m "добавить аутентификацию пользователя"
```

---

## Заключение

Аутентификация пользователей — одна из самых сложных областей для понимания при первой работе с веб-API. Без преимущества монолитной структуры мы, как разработчики, должны глубоко понимать и соответствие таким образом настраивать наши циклы запросов/ответов HTTP.

Django REST Framework имеет множество встроенных средств поддержки этого процесса, включая встроенную TokenAuthentication. Однако разработчики должны сами настроить дополнительные области, такие как регистрация пользователей и выделенные URL-адреса представления. В результате популярным, мощным и безопасным подходом является использование сторонних пакетов dj-rest-auth и django-allauth для минимизации объема кода, который мы должны написать с нуля.

## Глава 9. Наборы представлений и маршрутизаторы

Наборы просмотров<sup>99</sup> и роутеры<sup>100</sup> — это инструменты в рамках Django REST Framework, которые могут упростить разработку API. Они представляют собой дополнительный уровень абстракции поверх представлений и URL-адресов. Основное преимущество заключается в том, что один набор представлений может заменить несколько связанных представлений. И маршрутизатор может автоматически генерировать URL-адреса для разработчика. В более крупных проектах с многими конечными точками это означает, что разработчику приходится писать меньше кода. Кроме того, возможно, опытному разработчику легче понять и обосновать небольшое количество комбинаций наборов представлений и маршрутизаторов, чем длинный список отдельных комбинаций просмотров и URL-адресов.

В этой главе мы добавим две новые конечные точки API в наш существующий проект и посмотрим, как переключение с представлениями и URL-адресами на наборы представлений и маршрутизаторы может обес печить ту же функциональность с гораздо меньшими затратами. Код:

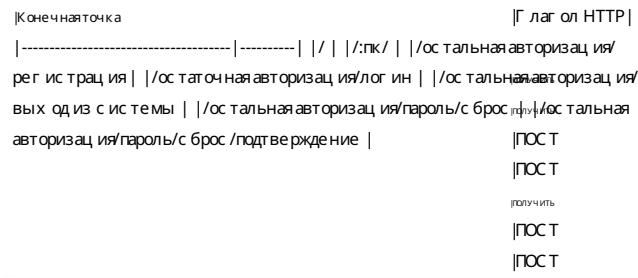
### Пользовательские конечные точки

В настоящее время в нашем проекте есть следующие конечные точки API. Все они имеют префикс `api/v1/`, который для краткости не показан:

---

<sup>99</sup><http://www.djangoproject.com/en/1.7/ref/api-guide/viewsets/> <sup>100</sup><http://www.djangoproject.com/en/1.7/ref/api-guide/routers/>

## Диаграмма



Первые две конечные точки были созданы нами, `adj-rest-auth` предоставил пять других. Давайте теперь добавим две дополнительные конечные точки, чтобы вывести список всех пользователей и отдельных пользователей. Это общая функция многих API, и она проясняет, почему рефакторинг наших представлений и URL-адресов в наборы представлений и маршрутизаторы могут иметь смысл.

Процесс подключения новых конечных точек ведет к следующим трем шагам:

- новый класс сериализатора для модели
- новые представления для каждой конечной точки
- новые маршруты URL для каждой конечной точки

Начните с нашего сериализатора. Нам нужно импортировать модель `CustomUser`, а затем создать класс `UserSerializer`, который ее использует. Затем добавьте его в наш существующий файл `posts/serializers.py`.

## Код

```
# posts/serializers.py from
django.contrib.auth import get_user_model # new from rest_framework import
сериалайзеры

из поста импорта .models

класс c PostSerializer (сериалайзеры.ModelSerializer):
```

## Метакласс:

```
модель = Пост
fields = ("id", "автор", "название", "тело", "created_at",)
```

```
класс UserSerializer(serializers.ModelSerializer): # новый
```

#### Метакласс:

```
model = get_user_model() fields  
= ("id", "username",)
```

---

Стоит отметить, что хотя мы исользовали `get_user_model` для ссылки на модель `CustomUser` здесь, на самом деле существует [три разных способа](#) ссылающихся на текущую модель пользователя в Django.

Используя `get_user_model`, мы гарантируем, что имеем в виду правильную модель пользователя будь то пользователь по умолчанию или [настраиваемая модель пользователя](#)<sup>102</sup>, как `CustomUser` в нашем случае.

Двигаясь дальше, нам нужно определить представления для каждой конечной точки. Сначала добавьте `UserSerializer` в список импорта. Затем создайте класс `UserList`, в котором перечислены все пользователи, и класс `UserDetail`, представляющий подробное представление об отдельном пользователе. Как и в случае с нашими представлениями с общением, здесь мы можем использовать `ListCreateAPIView` и `RetrieveUpdateDestroyAPIView`. Нам также нужно склоняться на модель пользователей через `get_user_model`, чтобы она была импортирована в верхней строке.

#### Код

---

```
# posts/views.py из  
django.contrib.auth import get_user_model # новое из rest_framework  
import generics
```

```
from .models import Post  
from .permissions import IsAuthorOrReadOnly  
from .serializers import PostSerializer, UserSerializer # new
```

```
класс PostList(generics.ListCreateAPIView):  
    permission_classes = (IsAuthorOrReadOnly,) queryset =  
        Post.objects.all() serializer_class = PostSerializer
```

```
класс PostDetail (generics.RetrieveUpdateDestroyAPIView):  
    классы_разрешения = (IsAuthorOrReadOnly,) queryset =  
        Post.objects.all() serializer_class = PostSerializer
```

<sup>101</sup><https://docs.djangoproject.com/en/4.0/topics/auth/customizing/#reference-the-user-model>

<sup>102</sup><https://docs.djangoproject.com/en/4.0/topics/auth/customizing/#specifying-a-custom-user-model>

```
class UserList(generics.ListCreateAPIView): # новый набор запросов
    queryset = get_user_model().objects.all() serializer_class = UserSerializer
```

```
класс UserDetail(generics.RetrieveUpdateDestroyAPIView): # новый
    queryset = get_user_model().objects.all() serializer_class
    = UserSerializer
```

---

Если вы заметили, здесь довольно много повторений. И представления Post , и представления User имеют один и тот же набор запросов и serializer\_class . Может быть, их можно как-то совместить, чтобы сократить код?

Наконец, у нас есть наши URL-маршруты. Обязательно импортируйте наши новые представления UserList и UserDetail .

Затем мы можем использовать префикс users/ для каждого.

Код

---

```
# posts/urls.py из
пути импорта django.urls

из .views импортировать PostList, PostDetail, UserList, UserDetail # новый

urlpatterns =
    [ path("users/", UserList.as_view()), # new path("users/
        <int:pk>/", UserDetail.as_view()), # new path("", PostList.as_view( )),
    путь("<int:pk>", PostDetail.as_view()),

]
```

---

И мы закончили. Убедитесь, что локальный сервер работает, и перейдите к дистальному для просмотра API, чтобы убедиться, что все работает должным образом.

Наша конечная точка списка пользователей находится по адресу <http://127.0.0.1:8000/api/v1/users/>.

```

HTTP 200 OK
Allow: GET, POST, HEAD, OPTIONS
Content-Type: application/json
Vary: Accept

[
    {
        "id": 1,
        "username": "wsv"
    },
    {
        "id": 2,
        "username": "testuser"
    },
    {
        "id": 3,
        "username": "testuser1"
    }
]

```

Код состояния 200 ОК, что означает, что все работает. Мы видим трех наших существующих пользователей. Конечная точка сведений о пользователе доступна в первичном ключе для каждого пользователя. Итак, наша учетная запись суперпользователя находиться по адресу: <http://127.0.0.1:8000/api/v1/users/1/>.

The screenshot shows the Django REST framework's User Detail view at `127.0.0.1:8000/api/v1/users/1/`. The top navigation bar includes the Django logo, a user icon labeled "Guest", and a dropdown menu. The main title is "User Detail". Below it, a breadcrumb trail shows "Post List / User List / User Detail". On the right, there are three buttons: "DELETE" (red), "OPTIONS" (blue), and "GET" (blue). A "Raw data" tab is selected, showing the response body:

```
HTTP 200 OK
Allow: GET, PUT, PATCH, DELETE, HEAD, OPTIONS
Content-Type: application/json
Vary: Accept

{
    "id": 1,
    "username": "wsv"
}
```

Below this, an "HTML form" tab is visible. The "Username" field contains "wsv". A note below the field states: "Required. 150 characters or fewer. Letters, digits and @/./+/-/\_ only." To the right is a blue "PUT" button.

### Экземпляр пользователя API

## Наборы представлений

Набор представлений — это способ объединить логику для нескольких связанных представлений в один класс. Другими словами, один набор представлений может заменить несколько представлений. В настоящем времени есть четыре представления для постов в блоге и два для пользователей. Вместо этого мы можем имитировать ту же функциональность с двумя наборами представлений: один для сообщений в блогах и один для пользователей. Компромисс заключается в потере удобочитаемости для коллег-разработчиков, которые не очень хорошо знакомы с наборами представлений.

Вот как выглядит код в нашем обновленном файле `posts/views.py`, когда мы меняем наборы представлений.

## Код

---

```
# posts/views.py из
django.contrib.auth импортировать get_user_model из
rest_framework импортировать наборы представлений # новый

from .models import Post
from .permissions import IsAuthorOrReadOnly
from .serializers import PostSerializer, UserSerializer

класс PostViewSet(viewsets.ModelViewSet): # новый
    permission_classes = (IsAuthorOrReadOnly,) queryset =
        Post.objects.all() serializer_class = PostSerializer

    class UserViewSet(viewsets.ModelViewSet): # новый набор
        запросов = get_user_model ().objects.all() serializer_class =
            UserSerializer
```

---

Вверху вместе с импортом дженериков из `rest_framework` мы теперь импортируем наборы представлений во второй строке. Затем мы ис пользуем [ModelViewSet103](#), который предоставляет нам как представление списка, так и подробное представление. И нам больше не нужно повторять один и тот же набор запросов и класс сериализатора для каждого представления, как мы это делали раньше!

В этот момент локальный веб-сервер становится похожим на `Django` жалуется на отсутствие соответствующих URL-адресов. Давайте установим их рядом.

## Маршрутизаторы

[Маршрутизаторы104](#) работать напрямую с наборами представлений, чтобы автоматически генерировать для нас шаблоны URL. Наш текущий файл `posts/urls.py` содержит четыре шаблона URL: два для сообщений в блогах и два для пользователей. Вместо этого мы можем использовать один маршрут для каждого набора представлений. Итак, два маршрута вместо четырех шаблонов URL. Это звучит лучше, верно?

---

<sup>103</sup><http://www.djangoproject.org/api-guide/viewsets/#modelviewset> <sup>104</sup><http://www.djangoproject.org/api-guide/routers/>

Django REST Framework имеет два маршрутизатора по умолчанию `SimpleRouter`<sup>105</sup> и `DefaultRouter`<sup>106</sup>. Мы будем использовать `SimpleRouter`, но также можно создавать собственные маршрутизаторы для более продвинутой функциональности.

Вот как выглядит обновленный код:

#### Код

```
# posts/urls.py из
пути импорта django.urls из
rest_framework.routers import SimpleRouter

из .views импортировать UserViewSet, PostViewSet

router = SimpleRouter()
router.register("пользователи", UserViewSet, basename="users")
router.register("", PostViewSet, basename="posts")

urlpatterns = router.urls
```

В верхней строке импортируется `SimpleRouter` вместе с нашими представлениями. Маршрутизатор настроен на `SimpleRouter`, и мы «регистрируем» каждый набор представлений для пользователей и сообщений. Наконец, мы устанавливаем наши URL-адреса для использования нового маршрутизатора. Идите вперед и проверьте наши конечные точки, запустив локальный сервер с помощью `python manage.py runserver`. Сначала идет список пользователей по адресу `http://127.0.0.1:8000/api/v1/users/`, который одинаковый.

Однако подробное представление по адресу `http://127.0.0.1:8000/api/v1/users/1/` немного отличается. Теперь он называется я «Экземпляр пользователя» вместо «Сведения о пользователе», есть дополнительная опция «удалить», встроенная в `ModelViewSet`<sup>107</sup>.

<sup>105</sup><http://www.djangoproject.org/api-guide/routers/#simplerouter> <sup>106</sup><http://www.djangoproject.org/api-guide/routers/#defaultrouter> <sup>107</sup><http://www.djangoproject.org/api-guide/viewsets/#modelviewset>

Django REST framework

User Instance

[DELETE](#) [OPTIONS](#) [GET](#)

[Post List](#) / [User List](#) / [User Instance](#)

HTTP 200 OK

Allow: GET, PUT, PATCH, DELETE, HEAD, OPTIONS

Content-Type: application/json

Vary: Accept

```
{
    "id": 1,
    "username": "wsv"
}
```

Raw data

HTML form

Username: wsv

Required. 150 characters or fewer. Letters, digits and @./+/-/\_ only.

PUT

Сведение о пользователе API

## Разрешения

Если мы смотрим на наш API в данный момент, то на самом деле возникнет одна проблема с безопасностью любой пользователь, прошедший проверку подлинности, может добавить нового пользователя на страницу списка пользователей или изменить/удалить/обновить пользователя на странице каждого пользователя поскольку для `UserViewSet` нет явных разрешений. Это большая проблема!

Важно подумать о разрешениях для любой конечной точки API, но особенно когда речь идет о пользовательской информации. В этом случае мы хотим ограничить доступ только куперзователям. Если мы посмотрим на страницу разрешений в официальной документации, там есть соответствующий раздел с правами `IsAdminUser`. Чтобы мы хотим добавление его в `UserViewSet` на самом деле довольно

<sup>108</sup><https://www.django-rest-framework.org/api-guide/permissions/#isadminuser>

прос то й.

В файле posts/views.py импортируйте IsAdminUser вверх у, а затем в классе UserViewSet :  
установите для permission\_classes значение [IsAdminUser].

Код

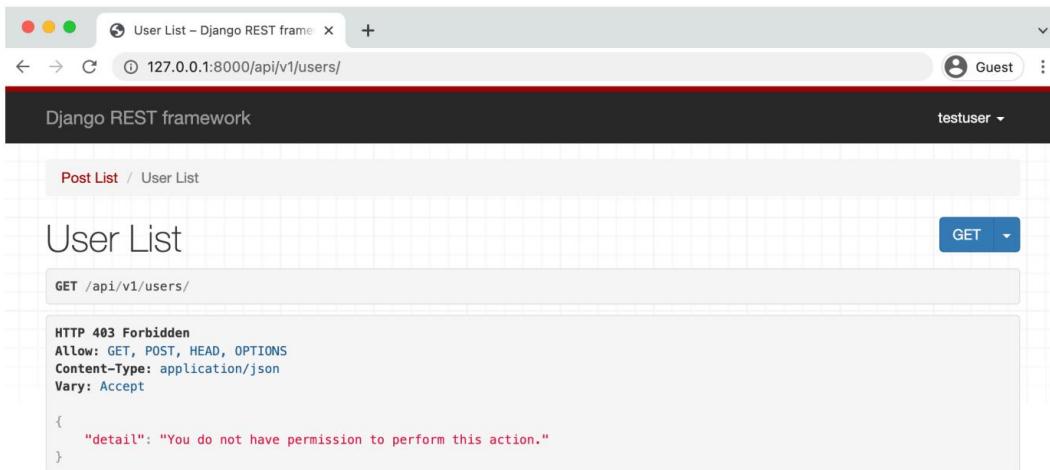
```
# posts/views.py из
django.contrib.auth import get_user_model из rest_framework
import viewsets из rest_framework.permissions import
IsAdminUser # new

from .models import Post
from .permissions import IsAuthorOrReadOnly
from .serializers import PostSerializer, UserSerializer

класс с PostViewSet(viewsets.ModelViewSet):
    permission_classes = (IsAuthorOrReadOnly,) queryset =
        Post.objects.all() serializer_class = PostSerializer
```

```
класс с UserViewSet (viewsets.ModelViewSet):
    permission_classes = [IsAdminUser] # новый набор
    запрос ов = get_user_model () .objects.all() serializer_class
        = UserSerializer
```

Локальный веб-сервер должен автоматически перезапуститься с измененным кодом, чтобы мы могли восстановить конечную точку списка пользователей по адресу <http://127.0.0.1:8000/api/v1/users/>.



Несмотря на то, что мы все еще вошли в систему как `testuser`, эта конечная точка недоступна, и то же самое верно для каждой конечной точки в квадратах пользователя.

При настройке разрешений все же рекомендуется иметь ограничительные настройки на уровне проекта и открывать доступ для каждой конечной точки по мере необходимости. Также важно проверить, что существующие конечные точки не остаются широкими открытыми, как это было раньше с нашими пользователями, что довольно легко сделать!

Чтобы закончить, мы должны закоммитить нашу новую работу в Git.

Ракушка

---

```
(.venv) > git status (.venv)
> git add -A (.venv) > git
commit -m "добавить скрипту и документацию"
```

---

## Заключение

Наборы представлений и маршрутизаторы — это мощные абстракции, которые сокращают объем кода, который мы, как разработчики, должны писать. Однако это также означает, что вы должны быть знакомы с первоначальным обучением. Первые несколько раз, когда вы будете использовать наборы представлений и маршрутизаторы вместе с представлениями и шаблонами URL, вы получите практику с ними.

В конечном счете решение о том, когда добавлять наборы представлений и маршрутизаторы в ваш проект, является субъективным. Хорошее эмпирическое правило — начинать с просмотров и URL-адресов. По мере усложнения вашего API, если вы обнаружите, что повторяете одни и те же шаблоны конечных точек снова и снова, обратите внимание на наборы представлений и маршрутизаторы. Дотех пор сократите просмотр.

## Г лава 10: Схемы и документация

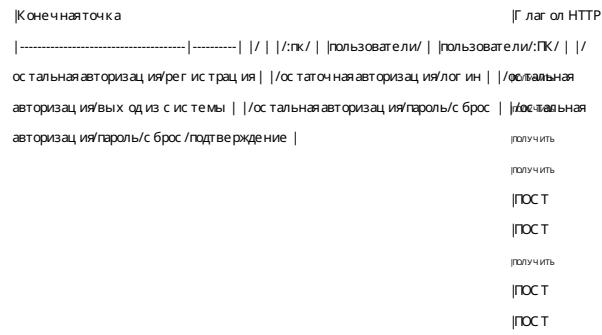
Теперь, когда у нас есть готовый API, нам нужен способ обобщить и документировать его функциональность для других. В конце концов, в большинстве компаний и команд разработчик, который использует API, — это не тот же разработчик, который изначально его создавал. Если API создан для общности, для его использования определенно требуется оно документированное руководство.

Схема — это машиночитаемый документ, в котором перечислены все доступные конечные точки API, URL-адреса и команды HTTP (GET, POST, PUT, DELETE и т. д.). Это здорово, но не очень понятно для человека.

Ведите документацию, которая добавляется к схеме, облегчая людям читать и потреблять.

Напоминаем, вот полный список наших текущих конечных точек API:

Диаграмма



В этой главе мы добавим в наш проект Blog как машиночитаемую схему, так и удобочитаемую документацию. Более того, мы автоматизируем создание каждого из них, чтобы они всегда соответствовали последней версии нашего API.

## Схема

[OpenAPI109](#) с пецификацией — это текущий с пособием документирования API по умолчанию. В нем описаны общие правила, касающиеся формата доступных конечных точек, ввода данных, методов аутентификации, контактной информации и т. д. На момент написания этой статьи [drf-spectacular110](#) — рекомендуемый сторонний пакет для создания схемы OpenAPI 3 для Django REST Framework.

Для начала установите drf-spectacular с помощью pip обычным способом.

Редактор

---

```
(.venv) > python -m pip install drf-spectacular~=0.21.0
```

---

Добавьте его в конфигурацию INSTALLED\_APPS в файле django\_project/settings.py .

Код

---

```
# django_project/settings.py
INSTALLED_APPS =
    [ "django.contrib.admin",
      "django.contrib.auth",
      "django.contrib.contenttypes",
      "django.contrib.sessions",
      "django.contrib.messages",
      "django.contrib.staticfiles",
      "django.contrib.sites",

      # Сторонние
      # приложения
      "rest_framework",
      "corsheaders",
      "rest_framework.authtoken",
      "allauth", "allauth.account",
      "allauth.socialaccount", "dj_rest_auth",
      "dj_rest_auth.registration",
      "drf_spectacular", # новый

      # Местный
      "accounts.apps.AccountsConfig",
```

---

<sup>109</sup><https://www.openapi.org/>

<sup>110</sup><https://github.com/tfranzel/drf-spectacular/>

```
"posts.apps.PostsConfig",
]
```

---

Затем зарегистрируйте drf-spectacular в разделе REST\_FRAMEWORK файла django\_project/settings.py файла.

Код

```
# django_project/settings.py
REST_FRAMEWORK =
    { "DEFAULT_PERMISSION_CLASSES":
        [ "rest_framework.permissions.IsAuthenticated", ],
        "DEFAULT_AUTHENTICATION_CLASSES":
            [ "rest_framework.authentication.SessionAuthentication",
              "rest_framework.authentication.TokenAuthentication",
            ],
        "DEFAULT_SCHEMA_CLASS": "drf_spectacular.openapi.AutoSchema", # новый
    }
```

---

Последним шагом является добавление некоторых метаданных, таких как заголовок, описание и версия настройкам по [умолчанию](#)<sup>11</sup>. Создайте новый раздел в django\_project/settings.py и добавьте следующее:

Код

```
# django_project/settings.py
SPECTACULAR_SETTINGS = {
    "TITLE": "Проект Blog API",
    "DESCRIPTION": "Образец блога для изучения DRF", "VERSION":
    "1.0.0",
    # Другие настройки
}
```

---

Чтобы сгенерировать схему как отдельный файл, мы можем использовать команду управления и указать имя файла, которое будет schema.yml.

<sup>11</sup><https://drf-spectacular.readthedocs.io/en/latest/settings.html>

Редактор

---

```
(.venv) > python manage.py generate-schema --file schema.yml
```

---

В корневом каталоге проекта создан новый файл schema.yml. Если вы откроете этот файл в текстовом редакторе, он будет довольно длинным и не очень удобным для человека. Но компьютер отличнно форматируется

## Динамическая схема

Более динамичный подход заключается в том, чтобы обслуживать схему непосредственно из нашего API в качестве URL-маршрута. Мы сделаем это, импортировав SpectacularAPIView, а затем добавив новый путь URL-адреса api/schema/ для его отображения.

Код

---

```
# django_project/urls.py from
django.contrib import admin from
django.urls import path, include from
drf_spectacular.views import SpectacularAPIView # new

urlpatterns =
    [ path("admin/", admin.site.urls), path("api/
v1/", include("posts.urls")), path("api-auth/", include("rest_framework.urls")), path("api/v1/dj-rest-auth/", include("dj_rest_auth.urls")), path( "api/v1/dj-rest-auth/registration/", include("dj_rest_auth.registration.urls"),

),
    path("api/schema/", SpectacularAPIView.as_view(), name="schema"), # новый
]
```

---

Снова запустите локальный сервер с помощью `python manage.py runserver` и перейдите к конечной точке URL-адреса новой схемы по адресу `http://127.0.0.1:8000/api/schema/`. Автоматически сгенерированный файл схемы все еще останется доступен и будет загружен в виде файла.

Лично я предпочитаю динамический подход в проектах, а не генерировать файл schema.yml каждый раз при изменении API.

## Документация

Схема описывает для пользователей компьютером, но люди обычно предпочитают документацию для пользователей API. drf-spectacular поддерживает два инструмента документации API: [Redoc](#)<sup>112</sup> и [SwaggerUI](#)<sup>113</sup>. К счастью, преобразование нашей схемы в любую из них является относительно безболезненным процессом.

Начнем с Редока. Чтобы добавить его, импортируйте SpectacularRedocView вверху django\_project/urls.py, а затем добавьте URL-адрес в api/schema/redoc/.

Код

---

```
# django_project/urls.py from
django.contrib import admin from
django.urls import path, include from
drf_spectacular.views import (SpectacularAPIView,
    SpectacularRedocView, # new
)
urlpatterns =
    [ path("admin/", admin.site.urls), path("api/
v1/", include("posts.urls")), path("api-auth/",
    include("rest_framework.urls")), path("api/v1/dj-rest-auth/",
    include("dj_rest_auth.urls")), path("api/v1/dj-rest-auth/registration/",
        включить("dj_rest_auth.registration.urls")
    ),
    path("api/schema/", SpectacularAPIView.as_view(), name="schema"), path("api/schema/
redoc/", SpectacularRedocView.as_view(
        url_name="schema"), name="redoc", # новый
]

```

---

Если локальный сервер все еще работает, вы можете перейти непосредственно к <http://127.0.0.1:8000/api/schema/redoc/> чтобы увидеть нашу новую документацию

<sup>112</sup><https://redoc.ly/redoc/>

<sup>113</sup><https://swagger.io/tools/swagger-ui/>

Blog API Project (1.0.0)

Authentication

schema > Download OpenAPI specification: [Download](#)

v1 > A sample blog to learn about DRF

Documentation Powered by ReDoc

## Authentication

### cookieAuth

Security Scheme Type	API Key
Cookie parameter name:	sessionid

### tokenAuth

Token-based authentication with required prefix "Token"

Security Scheme Type	API Key
Header parameter name:	Authorization

## schema

Схема редактора

Процесс добавления SwaggerUI очень похож. Импортируйте SpectacularSwaggerView вверху файла, а затем добавьте для него URL-адрес в api/schema/swagger-ui/.

Код

---

```
# django_project/urls.py from
django.contrib import admin from
django.urls import path, include from
drf_spectacular.views import ( SpectacularAPIView,
    SpectacularRedocView, SpectacularSwaggerView,
    # new

)

urlpatterns =
    [ path("admin/", admin.site.urls), path("api/
v1/", include("posts.urls")), path("api-auth/",
include("rest_framework.urls")), path("api/v1/dj-rest-auth/",
include("dj_rest_auth.urls")), path("api/v1/dj-rest-auth/registration/",

    включить("dj_rest_auth.registration.urls")
),
    path("api/schema/", SpectacularAPIView.as_view(), name="schema"), path("api/schema/
redoc/", SpectacularRedocView.as_view( url_name="schema"), name="redoc ",), path("api/
schema/swagger-ui/", SpectacularSwaggerView.as_view( url_name="schema"),
name="swagger-ui"), # новый

]

```

---

Затем перейдите в веб-браузер, чтобы увидеть вывод по адресу:

<http://127.0.0.1:8000/api/schema/swagger-ui/>.

**Blog API Project 1.0.0 OAS3**

/api/schema/

A sample blog to learn about DRF

Authorize

### schema

GET /api/schema/ ✓ 🔒

### v1

GET /api/v1/ ✓ 🔒

POST /api/v1/ ✓ 🔒

GET /api/v1/{id}/ ✓ 🔒

PUT /api/v1/{id}/ ✓ 🔒

PATCH /api/v1/{id}/ ✓ 🔒

DELETE /api/v1/{id}/ ✓ 🔒

POST /api/v1/dj-rest-auth/login/ ✓ 🔒

POST /api/v1/dj-rest-auth/logout/ ✓ 🔒

POST /api/v1/dj-rest-auth/password/change/ ✓ 🔒

Схема SwaggerUI

## Заключение

Добавление схемы и документации является важной частью любого API. Обычно это первое, на что обращает внимание коллега разработчик, будь то в команде или в проектах с открытым исходным кодом. Благодаря автоматизированному

## Глава 10: Схемы и документация

инструменты, описанные в этой главе, чтобы убедиться, что ваш API имеет точную актуальную документацию. Требуется лишь небольшая настройка. Последним шагом является правильное развертывание Blog API, которое мы рассмотрим в следующей главе.

## Г лава 11: Развёртывание в производственной среде

Последним шагом для любого проекта веб-API является развертывание. Внедрение его в производство очень похоже на традиционное развертывание Django, но с некоторыми дополнительными проблемами. В этой главе мы рассмотрим добавление переменных среды, настройку наших параметров для повышения безопасности, переключение на базу данных PostgreSQL в производственной среде и выполнение сброса производственного контролльного списка развертывания Django, чтобы убедиться, что мы ничего не упустили.

### Переменные среды

Переменные среды могут быть загружены в кодовую базу во время выполнения, но не сохранены в исходном коде. Это делает их идеальным способом переключения между локальными и производственными настройками. Они также являются хорошим местом для хранения конфиденциальной информации, которая не должна присутствовать в системе управления версиями. При использовании Git любые изменения сохраняются в истории Git, поэтому, даже если что-то в последствии будет удалено из кодовой базы, если оно было проверено в коммите в любое время, оно останется там навсегда, если кто-то умеет смотреть.

Существует несколько пакетов, позволяющих работать с переменными окружения в Python, но для этого проекта мы будем использовать [enviro114](#), потому что он позволяет ясно дополнительной конфигурацией Django, которая очень полезна.

Начнем с установки среды [django]. Если вы используете Zsh в качестве терминальной оболочки, необходимо добавить одинарные кавычки " вокруг имени пакета, поэтому запустите `python -m pip install 'окружение [джанго] == 9.3.5'`.

---

<sup>114</sup><https://github.com/sloria/enviro>

Ракурска

---

```
(.venv)> python -m pip install 'environs [django] == 9.5.0'
```

---

В верхней части файла django\_project/settings.py нужно добавить три строки импорта.

Код

---

```
# django_project/settings.py из pathlib
import Путь из environs import Env #
new

env = Env() # новый
env.read_env() # новый
```

---

Затем создайте новый скрытый файл с именем .env в корневом каталоге проекта, в котором будут храниться переменные среды. Пока он пуст, но будет использован в следующем разделе. Последний шаг — добавить .env в наш существующий файл .gitignore. Нет смысла использовать переменные окружения, если они все равно будут храниться в Git!

---

```
.gitignore
.venv/
.env
```

---

И пока мы обновляем файл, мы могли бы также добавить все файлы \*.rs и каталог \_\_ruscache\_\_. Если вы работаете на Mac, нет необходимости отслеживать .DS\_Store, в котором хранится информация о настройках папки. Наконец, не рекомендуется передавать локальный файл db.sqlite3 вместе с темой управления версиями.

Он содержит всю базу данных, так что любой, кто имеет доступ к ним, сможет на них держаться. Мы продолжим использовать его локально для удобства и все коре увидим, как вместе нечто можно использовать PostgreSQL в производственной среде.

Вот что должен содержать окончательный файл .gitignore:

```
.gitignore
```

---

```
.venv/  
.env  
__pycache__/  
db.sqlite3 .DS_Store  
# Только для Mac
```

---

Перед фиксацией запустите `git status`, чтобы убедиться что все эти файлы игнорируются как предполагалось. Затем добавляем нашу новую работу и сдвигаем коммит.

Ракушка

---

```
(.venv) > git status (.venv) >  
git add -A (.venv) > git  
commit -m "добавить переменные среды"
```

---

## ОТЛАДКА И SECRET\_KEY

В файле Django по умолчанию `settings.py` по умолчанию автоматически установлены локальные производственные настройки, которые упрощают запуск проектов, но есть несколько конфигураций, которые необходимо настроить перед развертыванием в производственной среде. Если вы посмотрите на конфигурацию DEBUG в `django_project/settings.py`, она в настоящее время имеет значение `True`. Это сдвигает очень подробную страницу ошибок и трассировку стека. Например, запустите локальный веб-сервер с помощью `python manage.py runserver` и посмотрите конечную точку API, которая не существует, например `http://127.0.0.1:8000/99`.

**Page not found (404)**

Request Method: GET  
Request URL: http://127.0.0.1:8000/99

Using the URLconf defined in `django_project.urls`, Django tried these URL patterns, in this order:

1. admin/
2. api/v1/
3. api-auth/
4. api/v1/dj-rest-auth/
5. api/v1/dj-rest-auth/registration/
6. api/schema/ [name='schema']
7. api/schema/redoc/ [name='redoc']
8. api/schema/swagger-ui/ [name='swagger-ui']

The current path, `99`, didn't match any of these.

You're seeing this error because you have `DEBUG = True` in your Django settings file. Change that to `False`, and Django will display a standard 404 page.

404 Страница

Мы хотим, чтобы DEBUG имел значение True для локальной разработки и False для рабочей среды. Если есть какие-либо трудности с загрузкой переменных среды, мы хотим, чтобы DEBUG по умолчанию имел значение False, чтобы мы были в большей безопасности. Чтобы реализовать это, начните с добавления DEBUG=True в файл .env.

.env

ОТЛАДКА=Истина

Затем в `django_project/settings.py` измените настройку DEBUG, чтобы прочитать переменную «DEBUG» из файла .env, но со значением по умолчанию False.

Код

---

```
# django_project/settings.py DEBUG =
env.bool("DEBUG", по умолчанию=False)
```

---

Если вы обновите веб-страницу по адресу `http://127.0.0.1:8000/99`, вы увидите, что полная локальная страница ошибок все еще существует. Всё работает правильно.

Следующим параметром, который нужно изменить, является `SECRET_KEY`, представляющий собой случайную строку из 50 символов, генерируемую при каждом запуске startproject. Если вы посмотрите на текущее значение в `django_project/settings.py`, оно начинается ясно `django-insecure`, чтобы указать, что текущее значение небезопасно. Почему это небезопасно? Потому что `SECRET_KEY` легкозадокументирован в системе управления версиями, что мы, собственно, уже и сделали. Даже если мы перестанем использовать это значение в переменной окружения, теперь оно все равно станет явным в истории Git проекта.

Решение состоит в том, чтобы сгенерировать новый секретный ключ и сюда занести это значение в переменной среды, чтобы оно никогда не касалось системы управления версиями. Один из способов создать новый — вызвать встроенные [секреты Python](#)<sup>115</sup>. модуль, запустив `python -c 'import secrets; print(secrets.token_urlsafe())'` в командной строке.

Ракурс

---

```
(.venv) > python -c "import secrets; print(secrets.token_urlsafe())"
KBI3sX5kLrd2zxj-pAichjT0EZJKMS0cXzhWI7Cydqc
```

---

Скопируйте и вставьте это новое значение в файл .env под переменной SECRET\_KEY.

.env

---

```
отладка=истина
SECRET_KEY=KBI3sX5kLrd2zxj-pAichjT0EZJKMS0cXzhWI7Cydqc
```

---

Наконец, переключитесь на SECRET\_KEY в файле django\_project/settings.py для чтения из переменной окружения сейчас.

Код

---

```
# django_project/settings.py SECRET_KEY
= env.str("SECRET_KEY")
```

---

Чтобы убедиться, что все работает правильно, перезапустите локальный сервер с помощью `python manage.py runserver` и обновите любую конечную точку API на нашем сайте. Он должен нормально работать.

## РАЗРЕШЕННЫЕ ХОСТИ

Далее следует конфигурация ALLOWED\_HOSTS в нашем файле django\_project/settings.py, которая предстаетименем хостов/доменов, которые может обслуживать наш проект Django. Здесь мы добавим три хоста: `.herokuapp.com` для развертывания на Heroku, а также локальный хост `127.0.0.1` для локальной разработки.

---

<sup>115</sup><https://docs.python.org/3/library/secrets.html>

## Код

---

```
# django_project/settings.py
ALLOWED_HOSTS = [".herokuapp.com", "localhost", "127.0.0.1"] # новый
```

---

Если вы повторно запустите команду `python manage.py runserver` и обновите `http://127.0.0.1:8000/1`, после изменения она должна отображаться normally.

## БАЗЫ ДАННЫХ

Наш текущий конфигурация DATABASES предназначен для SQLite, но мы хотим иметь возможность переключиться на PostgreSQL для производства на Heroku. Когда мы ранее устанавливали `enviro[n]django`, среди «ключей» Django был легантный [dj-database-url](#)<sup>116</sup>. Пакет, который принимает все конфигурации базы данных, необходимые для нашей базы данных, SQLite или PostgreSQL, и созывает среди DATABASE\_URL переменная.

Чтобы реализовать это, обновите конфигурацию DATABASES с помощью `dj_db_url` из `enviro[n]django`, чтобы облегчить анализ DATABASE\_URL.

## Код

---

```
# django_project/settings.py
DATABASES = {
    "по умолчанию": env.dj_db_url("DATABASE_URL") # новый
}
```

---

Вот и все! Все, что нам нужно сделать сейчас, это указать SQL в качестве локального значения DATABASE\_URL в файле .env.

## .env

---

```
ОТЛАДКА=Истина
SECRET_KEY=KB13sX5kLrd2zxj-pAichjT0EZJKMS0cXzhWI7CydqC
DATABASE_URL=sqlite:///db.sqlite3
```

---

Это кажется довольно волшебным, не так ли? Причина, по которой это работает, заключается в том, что всякий раз, когда Heroku созывает новую базу данных PostgreSQL, он автоматически созывает для нее переменную конфигурации DATABASE\_URL с именем DATABASE\_URL.

Поскольку файл .env не предназначен для производства, наш проект Django на Heroku вместе с тем что будет использовать эту конфигурацию PostgreSQL. Довольно легально, не так ли?

---

<sup>116</sup><https://github.com/jacobian/dj-database-url>

## Статические файлы

Как мы видели в предыдущих развертываниях, для работы достаточно пингод для просмотра веб-API необходимо настроить статические файлы. Сначала создайте новый каталог уровня проекта с именем static.

Ракушка

---

(.venv) > mkdir статический

---

С помощью текстового редактора создайте пустой файл .keep в статическом каталоге, чтобы Git подхватил его. Затем установите whitenoise для обработки статических файлов в рабочей среде.

Ракушка

---

(.venv) > python -m pip install whitenoise == 5.3.0

---

WhiteNoise необходимо добавить в django\_project/settings.py в следующих местах:

- белый шум над django.contrib.staticfiles в INSTALLED\_APPS
- WhiteNoiseMiddleware выше CommonMiddleware
- Конфигурация STATICFILES\_STORAGE, указывающая на WhiteNoise.

Код

---

```
# django_project/settings.py
INSTALLED_APPS = [
    ...
    "whitenoise.runserver_nostatic", # новый
    "django.contrib.staticfiles",
]
[...]
"django.middleware.security.SecurityMiddleware",
"django.contrib.sessions.middleware.SessionMiddleware",
"whitenoise.middleware.WhiteNoiseMiddleware", # new
"corsheaders.middleware.CorsMiddleware",
...
]

STATIC_URL = "/статический/"
```

```
STATICFILES_DIRS = [BASE_DIR / "static"] # новый
STATIC_ROOT = BASE_DIR / "staticfiles" # новый
STATICFILES_STORAGE =
    "whitenoise.storage.CompressedManifestStaticFilesStorage" # новый
```

---

Последним шагом является запуск команды `collectstatic`, чтобы все статические каталоги и файлы были скомпилированы в одном месте для целей развертывания.

Ракушка

```
(.venv) > python manage.py collectstatic
```

---

## Письмо ГУННИКОРН

Есть два последних пакета, которые необходимо установить для надлежащей производственной среды.

[Письмо 117](#) — это адаптер базы данных, который позволяет приложению Python взаимодействовать с базами данных PostgreSQL. Если вы используете macOS, сначала необходимо установить PostgreSQL через Homebrew, а затем psycopg2.

Ракушка

```
# Ok на
(.venv)> python -m pip установить psycopg2 == 2.9.3

# Mac OS
(.venv) % brew установить postgresql (.venv)
% python -m pip установить psycopg2==2.9.3
```

---

Мы можем использовать этот подход, потому что Django ORM (Object Relational Mapper) переводит наш код `models.py` из Python в выбранную первоначальную часть базы данных. Это работает почти всегда без ошибок. Возможны странные ошибки, и в профессиональном проекте рекомендуется также установить PostgreSQL локально, чтобы избежать их.

Gunicorn — это производственный веб-сервер, который также необходимо установить, чтобы заменить текущий веб-сервер Django, который подходит только для локальной разработки.

---

<sup>117</sup><https://www.psycopg.org/docs/>

Ракушка

---

```
(.venv) > python -m pip install gunicorn == 20.1.0
```

---

## требования.txt

Как мы видели ранее в этой книге, необх одим файл requirements.txt , в котором перечислены все пакеты, установленные в нашей локальной виртуальной среде. Мы также можем с делать это с помощью леду `pip` команда.

Ракушка

---

```
(.venv)> python -m pip замораживание> требования.txt
```

---

Вот как выглядит содержимое моего файла requirements.txt . Ваш может выглядеть немного иначе: например, Django, скорее всего, будет в версии 4.1.1 или более поздней, потому что мы установили его с помощью `=` что означает, что установлена последняя версия 4.0.x.

требования.txt

---

```
asgiref==3.5.0
attrs==21.4.0
сертификат == 2021.10.8
cffi==1.15.0
кодировка нормализатор == 2.0.11
криптография== 36.0.1 defusedxml
== 0.7.1
dj-database-url==0.5.0 dj-email-
url==1.0.5 dj-rest-auth==2.1.11
Django==4.0.2 django-allauth==0.48.0
django-cache-url= 3.2.3 django-cors-
headers==3.10.1
djangorestframework==3.13.1 drf-
spectacular==0.21.2 с рефы==9.5.0
```

---

pushka == 20.1.0 идна == 3.3

перегиб == 0,5,1
jsonschema == 4.4.0

```

зефир==3.14.1
оутлиб==3.2.0
psycopg2==2.9.3
rusparser==2.21
PyJWT==2.3.0
pyrsistent==0.18.1 python-
dotenv==0.19.2 python3-
openid==3.2.0 pytz==2021.3
PyYAML==6.0 запросы==2.27.1
запросы-oauthlib==1.3.1
sqlparse==0.4.2 uritemplate==4.1.1
urllib3==1.26.8

```

белый шум==5.3.0

---

## Procfile и runtime.txt

Heroku полагается на пользовательский файл Procfile , который описывает, как запускать проекты в производстве.

Он должен быть создан в корневом каталоге проекта рядом с файлом manage.py . Сделайте это прямо сейчас в текстовом редакторе и добавьте следующую строку, чтобы использовать Gunicorn в качестве производственного веб-сервера.

Procfile

---

web: gunicorn django\_project.wsgi --log-file -

---

Последний шаг — указать, какая версия Python должна работать на Heroku, с помощью файла runtime.txt .

В текстовом редакторе создайте этот новый файл runtime.txt на уровне проекта, что означает, что он находится в том же каталоге, что и manage.py и Procfile . Нам нужна версия Python 3.10.2.

время выполненияtxt

---

питон-3.10.2

---

## Контрольный список развертывания

Мы прошли много шагов. Слишком много, чтобы помнить большинство разработчиков, поэтому существою контрольные списки развертывания. Подводя итог, вот что мы сделали:

- добавить переменные окружения через `environs[django]`
- установите для `DEBUG` значение `False`
- установить `ALLOWED_HOSTS`
- ис пользовать переменную окружения для `SECRET_KEY`
- обновить базы данных, чтобы ис пользовать SQLite локально и PostgreSQL в рабочей среде.
- настроить статические файлы и установить белый шум
- установить `gunicorn` для производственного веб-сервера • создать файл `requirements.txt`
- создать `Procfile` для Heroku
- создайте `runtime.txt`, чтобы установить версию Python на Heroku

Помимо файла `Procfile`, созданного для Heroku, эти шаги развертывания практически одинаковы для любой платформы.

Обязательно зафиксируйте все эти изменения в Git, прежде чем мы фактически развернем проект.

Ракушка

---

```
(.venv) > git status (.venv)
> git add -A (.venv) > git
commit -m "контрольный список развертывания"
```

---

## Развёртывание Героку

Чтобы развернуть на Heroku, убедитесь, что вы вошли в систему через оболочку терминала.

Ракушка

---

```
(.venv) > логин на гироку
```

---

Команда `heroku create` создает новый контейнер для нашего приложения и по умолчанию Heroku присваивает ему лучшее имя. Вы можете указать собственное имя, как мы делаем здесь, но оно должно быть уникальным на Heroku.

Мой называется `dfa-blog-api`, так что это имя уже занято; тебе нужен другой счётчик букв и цифр!

Ракушка

---

```
(.venv) > heroku create dfa-blog-api Сздание
dfa-blog-api... сделано https://dfa-blog-
api.herokuapp.com/ | https://git.heroku.com/dfa-blog-api.git
```

---

Всё идет нормально. Новым шагом на этом этапе является создание базы данных PostgreSQL на самом Heroku, чего мы раньше не делали. У Heroku есть собственные размещенные базы данных PostgreSQL, которые мы можем использовать на нескольких уровнях. Для подобного учебного проекта уровня без платного обработчика более чем достаточно. Выполните следующую команду, чтобы создать эту новую базу данных. Замените `dfa-blog-api` своим собственным пользовательским именем.

Ракушка

---

```
(.venv) > heroku addons:create heroku-postgresql:hobby-dev Сздание heroku-
postgresql:hobby-dev на dfa-blog-api... без платной базы данных создана и доступна
```

---

! Эта база данных пуста. При обновлении можно перенести данные из другой базы данных с помощью `pg:copy`. Создан `postgresql-angular-74744` как `DATABASE_URL`. Используйте `heroku addons:docs heroku-postgresql` для просмотра документации.

Вы видели, что Heroku создал собственный `DATABASE_URL` для доступа к базе данных? Для меня здесь это `postgresql-angular-74744`. Это автоматически досчитано как переменная конфигурации в Heroku после развертывания. Вот почему нам не нужно устанавливать переменную среды для `DATABASE_URL` в рабочей среде. Нам также не нужно устанавливать для `DEBUG` значение `False`, потому что это значение по умолчанию в нашем файле `django_project/settings.py`. Единственная переменная среды, которую нужно вручную добавить в Heroku, — это `SECRET_KEY`, поэтому скопируйте ее значение из файла `.env` и запустите команду `config:set`, поместив значение самого `SECRET_KEY` в двойные кавычки «».

Ракурска

---

```
(.venv) > конфиг урал ияг ероя set SECRET_KEY="KBl3sX5kLrd2zxj-pAichjT0EZJKMS0cXzhWI7Cydqc"
Установка SECRET_KEY и перезапуск dfa-blog-api... сделано, v5 SECRET_KEY:
KBl3sX5kLrd2zxj-pAichjT0EZJKMS0cXzhWI7Cydqc
```

---

Теперь пришло время отправить наш код на Heroku и запустить веб-процесс, чтобы наш динамометр Heroku работал.

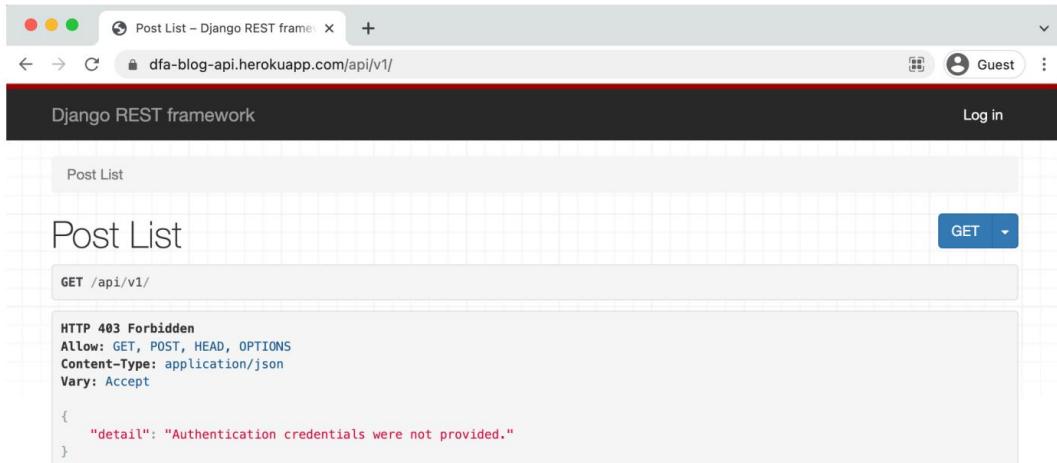
Ракурска

---

```
(.venv) > git push heroku main (.venv) >
heroku ps:scale web=1
```

---

URL-адрес вашего нового приложения будет в выводе командной строки, или вы можете запустить `heroku open`, чтобы найти его. У нас нет стандартной домашней страницы для этого API, поэтому вам нужно перейти к конечной точке, например `/api/v1/`.



Развёрнутая конечная точка списка общих

Если вы нажмете на ссылку «Войти» в правом верхнем углу, он ответит с сообщением об ошибке сервера 500! Это потому, что баз данных PostgreSQL существует, но еще не настроена



## Server Error (500)

500 Ошибка сервера

Ранее мы исользовали производственное SQLite, который основан на файлах и уже был настроен локально, а затем добавлен в Heroku. Но эта наша база данных PostgreSQL совершенно новая! У Heroku есть весь наш код, но мы еще не настроили эту производственную базу данных.

Тот же процесс, который используется локально для запуска миграций, с создания учетной записи и суперпользователя и ввода сообщений в блог в администраторе, необходимо повторить снова. Чтобы запустить команду с помощью Heroku, а не локально, добавьте к ней префикс heroku run.

Ракурска

---

```
(.venv) > heroku run python manage.py migrate (.venv) >
heroku run python manage.py createsuperuser
```

---

Вам нужно будет войти на сайт администратора, чтобы добавить записи в блог и пользователей, поскольку это совершенно новая база данных, не связанная с нашей локальной базой данных SQLite.

Обновите ваш живой веб-сайт, и он должен работать правильно. Обратите внимание: поскольку рабочий сервер будет постоянно работать в фоновом режиме, вам не нужно использовать команду runserver на Heroku.

## Заключение

Мы подошли к концу книги, но это только начало того, что можно сделать с помощью Django REST Framework. В ходе трех разных проектов — Library API, Todo API и Blog API — мы с нуля создавали, тестировали и развертывали все более сложные веб-API. И не с лучайно на каждом этапе пути Django REST Framework обеспечивает

встроенные функции, облегчающие нашу жизнь.

Если вы никогда раньше не создавали веб-API с другим фреймворком, будьте предупреждены, что вас избаловали. И если да, будьте уверены, что эта книга а лишь поверхности описывает возможности Django REST Framework.

Официальная документация<sup>118</sup> — это отличный ресурс для дальнейшего изучения, когда вы разобрались с основами.

## Расширенные темы

Помимо простых веб-API появляется множество дополнительных тем, которые стоит изучить, но которые мы не рассмотрели в книге. [Логинация<sup>119</sup>](#) — это полезный способ управления отображением данных на отдельных конечных точках API. [Фильтрация<sup>120</sup>](#) также становится неотъемлемым во многих проектах, особенно в сочетании с отличным [django-filter<sup>121</sup>](#) библиотекой.

[Десerialизация<sup>122</sup>](#) часто не обходит API как более продвинутая форма разрешений. Например, общедоступная сторона API может иметь ограничительные ограничения для запросов без проверки подлинности, в то время как запросы с проверкой подлинности с талливаются с гораздо более мягким регулированием.

[Последняя дополнительная область для изучения — кэширование<sup>123</sup>.](#) API с ображением производительности. Это работает очень похоже на то, как кэширование обрабатывается в традиционных проектах Django.

---

<sup>118</sup><http://www.djangoproject.org/>

<sup>119</sup><https://www.djangoproject.org/api-guide/pagination/>

<sup>120</sup><https://www.djangoproject.org/api-guide/filtering/>

<sup>121</sup><https://github.com/carltongibson/django-filter>

<sup>122</sup><https://www.djangoproject.org/api-guide/throttling/>

<sup>123</sup><https://www.djangoproject.org/api-guide/caching/>

## Следующие шаги

Хорошим следующим шагом является реализация API pastebin, описанного в официальном руководстве по DRF<sup>124</sup>.

Это не должно быть так сложно после прочтения этой книги и демонстрации еще нескольких сторон DRF.

Сторонние пакеты также важны для разработки Django REST Framework, как и для самого Django. Полный список можно найти на [Django Packages](#)<sup>125</sup>, или кураторский список на [awesome-django](#)<sup>126</sup> репозиторий на Github.

В конечном счете, то, как вы будете использовать Django и Django REST Framework, зависит от того, что вы хотите построить. Является ли это мобильным приложением для iOS или Android? Работать в координатах с полноценным интерфейсом JavaScript? Для внутреннего использования или для отображения общедоступного контента? Лучший способ учиться — работать в обратном направлении от большого проекта и разбираться его по частям.

## Спасибо

В то время как сообщество Django довольно велико и зависит от тяжелой работы многих людей, Django REST Framework намного меньше по сравнению с ним. Первоначально он был создан Томом Кристи<sup>127</sup>, английским инженером-программистом, который теперь работает над ним на постоянной основе благодаря финансированию открытых исходных кодов.

Спасибо, что читаете и поддерживаете мою работу. Если вы приобрели книгу на Amazon, подумайте о том, чтобы оставить честный отзыв: они оказывают огромное влияние на продажи книг и помогают мне продолжать выпускать как книги, так и бесплатный контент Django, который я люблю делить.

---

<sup>124</sup><http://www.django-rest-framework.org/tutorial/1-serialization/>

<sup>125</sup><https://djangopackages.org/> <sup>126</sup><https://github.com/wsvincent/awesome-django> <sup>127</sup><http://www.tomchristie.com/>