

Django для API

**Создавайте веб-API с помощью
Python и Django.**

William S. Vincent

Переведено не
профессионально.



django_kg

Django для API.

Создавайте веб-API с помощью Python и Django.

William S. Vincent

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

Эта версия была опубликована 13 августа 2020 г.



Это книга Leanpub. Leanpub предоставляет авторам и издателям возможность использовать процесс Lean Publishing. Lean Publishing — это процесс публикации незавершенной электронной книги с использованием облегченных инструментов и множества итераций для получения отзывов читателей, поворота до тех пор, пока у вас не будет нужной книги, и создания тяги после того, как вы это сделаете.

© 2018 - 2020 William S. Vincent

**Перевод сделала: Жумабекова Карлыгач
2022г**

Содержание



- Введение
- Предпосылки
- Почему API
- **Django REST Framework**
- Почему эта книга
- Вывод

- **Глава 1. Веб-API**
- Всемирная паутина
- URL-адреса
- Пакет интернет-протокола
- Методы HTTP
- Конечные точки
- HTTP
- Коды состояния
- Statelessness
- REST
- Вывод

- **Глава 2: Веб-сайт библиотеки и API**
- Традиционный **Джанго**
- Первое приложение
- Модели (models)
- Админ (admin)
- Представления (views)
- URL-адреса (urls)
- Веб-страница

- **Django REST Framework**
 - URL-адреса (urls)
 - Представления(views)
 - Сериализаторы (serializers)
 - cURL
 - Доступный для просмотра API
 - Вывод
-
- **Глава 3: Todo API**
 - Первоначальная настройка
 - Модели (Models)
 - Django REST Framework
 - URL-адреса (urls)
 - Сериализаторы (Serializers)
 - Представления(views)
 - Использование API
 - Доступный для просмотра API
 - CORS (cors)
 - Тесты (tests)
 - Вывод
-
- **Глава 4: Todo React**
 - Установить Node
 - Установите React
 - Фиктивные данные
 - **Django REST Framework + React**
 - Вывод

- **Глава 5: API блога**
 - Первоначальная настройка
 - Модель (model)
 - Тесты (tests)
 - **Django REST Framework**
 - URL-адреса (urls)
 - Сериализаторы (Serializers)
 - Представления(views)
 - Использование API
 - Доступный для просмотра API
 - Вывод
-
- **Глава 6: Права доступа (Permissions)**
 - Создать нового пользователя
 - Добавить вход в доступный для просмотра API
 - AllowAny- (Доступ любому)
 - Права доступа на уровне просмотра
 - Права доступа на уровне проекта
 - Вывод
-
- **Глава 7: Аутентификация пользователя**
 - Базовая аутентификация (Basic Authentication)
 - Аутентификация сеанса (Session Authentication)
 - Аутентификация с помощью токена (Token Authentication)
 - Аутентификация по умолчанию (Default Authentication)
 - Реализация аутентификации по токену
 - Конечные точки (Endpoints)
 - dj-rest-auth
 - Регистрация пользователя
 - Токены
 - Вывод

- **Глава 8. Наборы представлений и маршрутизаторы**
 - Конечные точки пользователя (User endpoints)
 - Набор представлений
 - Маршрутизаторы
 - Вывод
-
- **Глава 9. Схемы и документация.**
 - Схемы
 - Документация
 - Выводы
-
- **Заключение**
 - Следующие шаги
 - Спасибо

Введение

Интернет работает на основе **RESTful API**. За кулисами даже самой простой онлайн-задачи стоят несколько компьютеров, взаимодействующих друг с другом.

API (программный интерфейс приложения) - это формальный способ общения двух компьютеров, взаимодействующих друг с другом напрямую. И хотя существует множество способов создания API, веб-интерфейсы, которые позволяют передавать данные через всемирную паутину, в подавляющем большинстве случаев структурированы по принципу **RESTful (REpresentational State Transfer)**.

В этой книге вы узнаете, как построить несколько **RESTful** веб-интерфейсов с возрастающей сложностью с нуля, используя **Django** и **Django REST Framework**, один из самых популярных и настраиваемых способов создания веб-интерфейсов, используемый многими крупнейшими технологическими компаниями в мире, включая **Instagram**, **Mozilla**, **Pinterest** и **Bitbucket**. Этот подход также отлично хорошо подходит для начинающих, поскольку подход **Django** "батареи в комплекте" маскирует большую часть базовой сложности и рисков безопасности, связанных с созданием любого веб-API.

Предпосылки

Если вы новичок в веб-разработке с **Django**, я рекомендую сначала прочитать мою предыдущую книгу "**Django для начинающих**". Первые несколько глав доступны бесплатно онлайн и охватывают правильную настройку, приложение **Hello World** и приложение **Pages**. Полная версия книги более глубока и охватывает веб-сайт **Blog** с формами и учетными записями пользователей, а также готовый к производству сайт **Newspaper** с кастомной моделью пользователя, полным потоком аутентификации пользователей, электронной почтой, разрешениями, развертыванием, переменными окружения и многим другим.

Этот опыт работы с традиционным **Django** очень важен, поскольку **Django REST Framework** намеренно имитирует многие соглашения **Django**.

Также рекомендуется, чтобы читатели имели базовые знания о самом языке **Python**. Полное освоение языка **Python** занимает годы, но, обладая лишь небольшими знаниями, вы можете сразу же погрузиться в него и начать создавать свои программы.

Почему API?

Django был впервые выпущен в 2005 году, и в то время большинство веб-сайтов состояли из одной большой монолитной кодовой базы. "**Back-End**" состоял из моделей баз данных, URL и представлений, которые взаимодействовали с "**Front-End**" шаблонами HTML, CSS и JavaScript, которые управляли презентационным оформлением каждой веб-страницы.

Однако в последние годы подход "API-first" стал, пожалуй, доминирующей парадигмой в веб-разработке. Этот подход подразумевает формальное отделение **back-end** от **front-end**. Это означает, что **Django** становится мощной базой данных и API, а не просто каркасом для веб-сайта.

Сегодня **Django**, пожалуй, чаще используется в крупных компаниях как просто API для бэкенда, а не как полноценное монолитное решение для веб-сайта!

На этом этапе возникает очевидный вопрос: "Зачем это нужно?".

Традиционный **Django** достаточно хорошо работает сам по себе, и превращение **Django**-сайта в веб-интерфейс кажется большой дополнительной работой. К тому же, как разработчику, вам придется писать специализированный фронтенд на другом языке программирования.

Такой подход к разделению сервисов на различные компоненты, кстати, широко известен как сервис-ориентированная архитектура.
(**Service-oriented architecture**.)



Однако оказалось, что разделение фронтенда и бэкенда имеет множество преимуществ. Во-первых, это, вероятно, гораздо более "перспективно", поскольку API бэкенда может быть использован любым фронтенном **JavaScript**. Учитывая быстрые темпы изменения библиотек для фронтенда - **React** был выпущен только в 2013 году, а **Vue** 6 - в 2014! Когда в ближайшие годы текущие фронтенд-фреймворки будут заменены еще более новыми, **бэкенд API** может остаться прежним. Никакой серьезной переработки не потребуется.

Во-вторых, API может поддерживать несколько фронтендов, написанных на разных языках и в разных фреймворках.

Подумайте, что **JavaScript** используется для веб-фронтэндов, в то время как для приложений **Android** требуется язык программирования **Java**, а для приложений **iOS** - язык программирования **Swift**. При традиционном монолитном подходе веб-сайт **Django** не может поддерживать эти различные фронт-энды. Но с помощью внутреннего **API** все три могут взаимодействовать с одной и той же базовой версией базы данных!

В-третьих, подход, основанный на **API**, можно использовать как внутри компании, так и за ее пределами. Когда я работал в **Quizlet** в 2010 году, у нас не было ресурсов для разработки собственных приложений для **iOS** или **Android**. Но у нас был доступен внешний **API**, который использовали более 30 разработчиков для создания собственных приложений для работы с флеш-картами на основе базы данных **Quizlet**. Некоторые из этих приложений были загружены более миллиона раз, обогащая разработчиков и одновременно увеличивая охват **Quizlet**. Сейчас **Quizlet** входит в двадцатку лучших сайтов в США в течение учебного года.

Основной недостаток подхода **API-first** заключается в том, что он требует больше конфигурации, чем традиционное приложение **Django**. Однако, как мы увидим в этой книге, фантастическая библиотека **Django REST Framework** устраняет большую часть этой сложности.

Django REST Framework

Существуют сотни и сотни сторонних приложений, которые добавляют дополнительную функциональность **Django**. Вы можете посмотреть полный список с возможностью поиска на **Django Packages**, а также список в репозитории **awesome-django**.

Однако, среди всех сторонних приложений, Django REST Framework является, пожалуй, самым лучшим приложением для Django. Он зрелый, полный возможностей, настраиваемый, тестируемый и чрезвычайно хорошо документированный. Он также целенаправленно имитирует многие традиционные соглашения Django, что значительно ускоряет его изучение. И он написан на языке программирования Python, замечательном, популярном и доступном языке.

Если вы уже знакомы с Django, то изучение Django REST Framework будет логичным следующим шагом. С минимальным количеством кода он может превратить любое существующее приложение Django в веб-Api интерфейс

Почему эта книга?

Я написал эту книгу, потому что существует явный недостаток хороших ресурсов, доступных для разработчиков-новичков **Django REST Framework**. Предполагается, что все уже знают всё об API, HTTP, REST и тому подобном. Мой собственный путь в изучении того, как создавать веб-интерфейсы API, был разочаровывающим... и я уже знал **Django** достаточно хорошо, чтобы написать по нему книгу!

Эта книга - руководство, о существовании которого я мечтал, когда начинал работать с **Django REST Framework**.

Глава 1 начинается с краткого введения в веб-API и протокол HTTP. Во **главе 2** мы рассматриваем различия между традиционным **Django** и **Django REST Framework**, создавая сайт библиотеки книг, а затем добавляя к нему API. Затем в **главах 3-4** мы создадим API Todo и подключим его к **React front-end**. Этот же процесс можно использовать для подключения любого специализированного фронтенда (Web, iOS, Android, Desktop или другого) к бэкенду веб-интерфейса API.

В **главах 5-9** мы создаем готовый к производству Blog API, который включает в себя полную функциональность **CRUD**.

Мы также рассмотрим подробные разрешения, аутентификацию пользователей, наборы представлений, маршрутизаторы, документацию и многое другое.

Полный исходный код всех глав можно найти на Github.

Вывод

Django и Django REST Framework - это мощный и доступный способ создания веб-интерфейсов API. К концу этой книги вы сможете создавать свои собственные веб-API с нуля, правильно используя современные передовые методы. И вы сможете расширить любой существующий веб-сайт Django до веб-API с минимальным количеством кода.

Давайте начнем!



Глава 1: Веб-API

Прежде чем мы начнем создавать собственные веб-API, важно рассмотреть, как на самом деле работает Интернет.

В конце концов, "веб-интерфейс" в буквальном смысле слова располагается поверх существующей архитектуры всемирной паутины и опирается на множество технологий, включая HTTP, TCP/IP и другие.

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

World Wide Web (Всемирная паутина)

Интернет - это система взаимосвязанных компьютерных сетей, существующая по крайней мере с 1960-х годов. Однако на ранних этапах использования Интернета он был ограничен небольшим количеством изолированных сетей, в основном правительственные, военных или научных, которые обменивались информацией в электронном виде. К 1980-м годам многие исследовательские институты и университеты использовали Интернет для обмена данными. В Европе крупнейший узел Интернета находился в CERN (Европейская организация ядерных исследований) в Женеве, Швейцария, где работает крупнейшая в мире лаборатория физики частиц.

В ходе этих экспериментов генерируется огромное количество данных, которыми необходимо удаленно обмениваться с учеными по всему миру. Однако, по сравнению с сегодняшним днем, в 1980-х годах общее использование Интернета было мизерным. Большинство людей не имели к нему доступа и даже не понимали, почему он так важен. Небольшое количество узлов интернета обеспечивало весь трафик, а компьютеры, использующие его, находились в основном в тех же небольших сетях.

Все изменилось в 1989 году, когда ученый-исследователь из CERN Тим Бернерс-Ли изобрел HTTP и положил начало современной Всемирной паутине.

Его великая идея заключалась в том, что существующую гипертекстовую систему, в которой текст, отображаемый на экране компьютера, содержит ссылки (гиперссылки) на другие документы, можно перенести в Интернет.

Его изобретение, протокол передачи гипертекста (HTTP), стал первым стандартным, универсальным способом обмена документами через Интернет. Он положил начало концепции веб-страниц: отдельных документов с URL-адресом, ссылками и ресурсами, такими как изображения, аудио или видео.

Сегодня, когда большинство людей думают об Интернете, они думают о Всемирной паутине, которая в настоящее время является основным способом общения миллиардов людей и компьютеров в сети.

URL-адреса

URL (Uniform Resource Locator) - это адрес ресурса в Интернете. Например, домашняя страница Google находится по адресу <https://www.google.com>. Когда вы хотите перейти на домашнюю страницу Google, вы вводите полный адрес URL в веб-браузер.

Затем ваш браузер посыпает запрос через Интернет и волшебным образом подключается (мы рассмотрим, что происходит на самом деле, в ближайшее время) к серверу, который отвечает данными, необходимыми для отображения домашней страницы Google в вашем браузере.

Эта схема запроса и ответа является основой всех веб-коммуникаций. Клиент (обычно веб-браузер, но также и нативное приложение или вообще любое устройство, подключенное к Интернету) запрашивает информацию, а сервер отвечает на запрос.

Поскольку веб-коммуникация происходит по протоколу HTTP, формально они известны как HTTP-запросы и HTTP-ответы.

Внутри одного 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.

Она указывает веб-браузеру, как получить доступ к ресурсам по данному адресу. Для веб-сайта это обычно http или https, но это может быть и ftp для файлов, smtp для электронной почты и так далее. Следующий раздел, www.google.com , - это имя хоста или фактическое имя сайта. Каждый URL содержит схему и хост.

Многие веб-страницы также содержат необязательный путь. Если вы перейдете на домашнюю страницу Python по адресу https://www.python.org и нажмете на ссылку для страницы "About", вы будете перенаправлены на https://www.python.org/about/ . Часть /about/ - это путь. В общем, каждый URL, подобный https://python.org/about/, имеет три потенциальные части:

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

Набор интернет-протоколов

Как только мы узнаем фактический URL ресурса, целая коллекция других технологий должна работать правильно (вместе), чтобы соединить клиента с сервером и загрузить фактическую веб-страницу. Все это в целом называется набором интернет-протоколов, и существуют целые книги, написанные только на эту тему. Для наших целей, однако, мы можем придерживаться общих основ.

Когда пользователь набирает в браузере адрес https://www.google.com и нажимает Enter, происходит несколько вещей. Сначала браузеру нужно найти нужный сервер где-то на просторах Интернета. Он использует службу доменных имен (DNS) для преобразования доменного имени "google.com" в IP-адрес, который представляет собой уникальную последовательность цифр, обозначающую каждое подключенное устройство в Интернете. Доменные имена используются потому, что человеку легче запомнить доменное имя типа "google.com", чем IP-адрес типа "172.217.164.68".

После того, как браузер получил IP-адрес определенного домена, ему необходимо установить последовательное соединение с нужным сервером. Это происходит с помощью протокола управления передачей (TCP), который обеспечивает надежную, упорядоченную и проверенную на ошибки доставку байтов между двумя приложениями.

Чтобы установить TCP-соединение между двумя компьютерами, между клиентом и сервером происходит трехстороннее "рукопожатие":

1. Клиент посыпает SYN с запросом на установление соединения.
2. Сервер отвечает SYN-ACK, подтверждая запрос и передавая параметр соединения.
3. Клиент отправляет ACK обратно на сервер, подтверждая соединение.

После установления TCP-соединения два компьютера могут начать взаимодействие через HTTP.

HTTP Методы

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

Рассмотрим веб-сайт Facebook. После входа в систему вы можете прочитать свою временную шкалу, создать новый пост или отредактировать/удалить существующий. Эти четыре действия "Create-Read-Update-Delete"(создать-читать-обновить-удалить)известны под общим названием CRUD-функциональность и представляют собой подавляющее большинство действий, выполняемых в Интернете.

Протокол HTTP содержит ряд методов запроса , которые могут быть использованы при запросе информации с сервера. Четыре наиболее распространенных метода связаны с функцией CRUD. Это POST, GET, PUT и DELETE.

Diagram:

CRUD	HTTP -методы
-----	-----
Create	<-----> POST
Read	<-----> GET
Update	<-----> PUT
Delete	<-----> DELETE

Для создания контента используется POST, для чтения - GET, для обновления - PUT, а для удаления - DELETE.

Endpoints

Веб-сайт состоит из веб-страниц, содержащих HTML, CSS, изображения, JavaScript и многое другое.

https://en.wikipedia.org/wiki/Hypertext_Transfer_Protocol#Request_method

Эндпоинт (Endpoint - конечная точка) — это само обращение к маршруту отдельным HTTP методом. Эндпоинт выполняют конкретную задачу, принимают параметры и возвращают данные Клиенту.

Но веб-интерфейс API имеет конечные точки, которые представляют собой URL-адреса со списком доступных действий (HTTP-методы), которые предоставляют данные (обычно в формате JSON , который является наиболее распространенным форматом данных в наши дни и используется по умолчанию в Django REST Framework).

Например, мы можем создать следующие конечные точки API для нового сайта под названием mysite.

Diagram

`https://www.mysite.com/api/users` # GET возвращает всех пользователей

`https://www.mysite.com/api/users/<id>` # GET возвращает одного пользователя

В первой конечной точке, /api/users , доступный запрос GET возвращает список всех доступных пользователей. Этот тип конечной точки, которая возвращает несколько ресурсов данных, известен как коллекция.

Вторая конечная точка /api/users/<id> представляет одного пользователя. Запрос GET возвращает информацию только об этом одном пользователе. Если мы добавим POST к первой конечной точке, то сможем создать нового пользователя, а добавление DELETE ко второй конечной точке позволит нам удалить одного пользователя.

В ходе работы над этой книгой мы познакомимся с конечными точками API гораздо ближе, но в конечном итоге создание API подразумевает создание серии конечных точек: URL-адреса с соответствующими HTTP-методами запроса.

Веб-страница состоит из HTML, CSS, изображений и многое другого. Но конечная точка - это просто способ доступа к данным с помощью доступных HTTP-методов запроса.

HTTP

Мы уже много говорили о HTTP в этой главе, но здесь мы опишем, что это такое на самом деле и как это работает.

HTTP - это протокол запроса-ответа между двумя компьютерами, имеющими существующее TCP-соединение. Компьютер, делающий запрос, называется клиентом, а компьютер, отвечающий на запрос, называется сервером. Обычно клиентом является веб-браузер, но это также может быть приложение для iOS или любое другое устройство, подключенное к Интернету. Сервер - это необычное название любого компьютера, оптимизированного для работы через Интернет. Для превращения обычного ноутбука в сервер необходимо лишь специальное программное обеспечение и постоянное подключение к Интернету.

Каждое сообщение HTTP состоит из строки состояния, заголовков и необязательных данных тела. Например, HTTP-сообщения, которое браузер может отправить для запроса домашней страницы Google, расположенной по адресу <https://www.google.com>.

Diagram

GET / HTTP/1.1

Host: google.com

Accept-Language: en-US

Верхняя строка известна как строка запроса, и в ней указывается метод HTTP для использования (GET), путь (/) и конкретная версия HTTP для использования (HTTP/1.1).

Две последующие строки - это HTTP-заголовки: Host - имя домена и Accept-Language - используемый язык, в данном случае американский английский. Существует множество заголовков HTTP.

HTTP-сообщения также имеют необязательный третий раздел, известный как тело. Однако тело мы видим только в HTTP-ответах, содержащих данные. Для простоты предположим, что домашняя страница Google содержит только HTML "Hello, World!".

Вот как может выглядеть сообщение HTTP-ответа от сервера Google.

Diagram

HTTP/1.1 200 OK

Date: Mon, 03 Aug 2020 23:26:07 GMT

Server: gws

Accept-Ranges: bytes

Content-Length: 13

Content-Type: text/html; charset=UTF-8

Hello, world!

Верхняя строка - это строка ответа, в ней указано, что мы используем HTTP/1.1. Код состояния 200 OK означает, что запрос клиента был успешным (подробнее о кодах состояния в ближайшее время).

Следующие восемь строк - это заголовки HTTP. И, наконец, после переноса строки находится собственно содержимое тела "Hello, world!".

Поэтому каждое сообщение HTTP, будь то запрос или ответ, имеет следующий формат:

Diagram

Response/request line

Headers...

(optional) Bod

Большинство веб-страниц содержат множество ресурсов, которые требуют нескольких циклов HTTP-запросов/ответов. Если веб-страница содержит HTML, один CSS-файл и изображение, то для того, чтобы веб-страница полностью отобразилась в браузере, потребуется три отдельных цикла обмена данными между клиентом и сервером.

Коды состояния (status codes)

После того, как ваш веб-браузер выполнил HTTP-запрос на URL-адресе, нет никакой гарантии, что все будет работать! Поэтому существует довольно длинный список из 19 кодов состояния HTTP, сопровождающих каждый ответ HTTP.

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

- 2xx Успешно - действие, запрошенное клиентом, было получено, понято и принято
- 3xx Перенаправление - запрошенный URL переместился
- 4xx Ошибка клиента - произошла ошибка, как правило, неверный запрос URL клиентом.
- 5xx Ошибка сервера - серверу не удалось разрешить запрос

Нет необходимости запоминать все доступные коды состояния. С практикой вы освоите наиболее распространенные из них, такие как 200 (OK), 201 (Создан), 301 (Перемещен постоянно), 404 (Не найден) и 500 (Ошибка сервера).

Важно помнить, что, говоря в общем, существует только четыре потенциальных результата любого данного HTTP-запроса: он сработал (2xx), он был каким-то образом перенаправлен (3xx), клиент допустил ошибку (4xx) или сервер допустил ошибку (5xx). Эти коды состояния автоматически помещаются в строку запроса/ответа в верхней части каждого HTTP-сообщения.

Statelessness

(Протокол передачи без сохранения состояния)

Последний важный момент, который необходимо отметить в отношении HTTP, заключается в том, что это протокол без статических данных. Это означает, что каждая пара запрос/ответ полностью независима от предыдущей. Нет никакой памяти о прошлых взаимодействиях, что в информатике известно как состояние .

Отсутствие состояния дает много преимуществ HTTP. Поскольку во всех электронных системах связи со временем происходит потеря сигнала, если бы у нас не было протокола без состояния, все бы постоянно ломалось, если бы один цикл запроса/ответа не проходил. В результате HTTP известен как очень устойчивый распределенный протокол.

Однако обратная сторона заключается в том, что управление состоянием очень, очень важно в веб-приложениях. Состояние - это то, как веб-сайт запоминает, что вы вошли в систему, и как сайт электронной коммерции управляет вашей корзиной. Это фундаментальная составляющая того, как мы используем современные веб-сайты, однако она не поддерживается в самом HTTP.

Исторически состояние поддерживалось на сервере, но в современных фронтенд-фреймворках, таких как React, Angular и Vue, оно все больше и больше перемещается на клиент, в веб-браузер. Мы узнаем больше о состоянии, когда будем рассматривать аутентификацию пользователей, но помните, что HTTP не имеет состояния. Это делает его очень хорошим для надежной передачи информации между двумя компьютерами, но плохим для запоминания чего-либо вне каждой отдельной пары запрос/ответ.

REST

REpresentational State Transfer (REST) 21 - это архитектура, впервые предложенная в 2000 году Роем Филдингом в его диссертационной работе. Это подход к созданию API поверх Интернета, то есть поверх протокола HTTP.

Написаны целые книги о том, что делает API фактически RESTful или нет. Но есть три основных признака, на которых мы сосредоточимся для наших целей. Каждый RESTful API:

- не имеет статусов, как HTTP
- поддерживает общие методы HTTP (GET, POST, PUT, DELETE и т.д.)
- возвращает данные в формате JSON или XML
-

Любой RESTful API должен, как минимум, обладать этими тремя принципами. Стандарт важен поскольку он обеспечивает последовательный способ разработки и использования веб-интерфейсов API.

Вывод

Хотя в основе современного всемирного интернета лежит множество технологий, нам, разработчикам, не обязательно внедрять их с нуля. Прекрасное сочетание [Django](#) и [Django REST Framework](#) должным образом справляется с большинством сложностей, связанных с веб-интерфейсами API . Однако важно иметь хотя бы общее представление о том, как все эти части сочетаются друг с другом.

В конечном итоге веб-интерфейс API представляет собой набор конечных точек, которые открывают определенные части базовой базы данных. Как разработчики мы контролируем URL-адреса для каждой конечной точки, какие базовые данные доступны, и какие действия возможны с помощью HTTP-глаголов. Используя HTTP-заголовки, мы можем устанавливать различные уровни аутентификации и разрешения, как мы увидим далее в книге.

Глава 2: Веб-сайт библиотеки и API

Django REST Framework работает вместе с веб-фреймворком Django для создания веб-API. Мы не можем создать веб-интерфейс с помощью только Django Rest Framework; его всегда нужно добавлять в проект после установки и настройки самого Django.

В этой главе мы рассмотрим сходства и различия между традиционным Django и Django REST Framework. Наиболее важным моментом является то, что Django создает веб-сайты, содержащие веб-страницы, в то время как Django REST Framework создает веб-API, которые представляют собой набор конечных точек URL, 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.

Убедитесь, что на вашем компьютере уже установлены Python 3 и Pipenv. Полные инструкции можно найти здесь, если вам нужна помощь.

<https://pipenv-fork.readthedocs.io/en/latest/>

<https://djangoforbeginners.com/initial-setup/>

Традиционный Django

Во-первых, нам нужен специальный каталог на нашем компьютере для хранения кода. Он может находиться где угодно, но для удобства, если вы работаете на Mac, мы можем поместить его в папку Desktop. Расположение не имеет значения, просто он должен быть легко доступен.

Командная строка:

\$ cd ~/Desktop

\$ mkdir code && cd code

Эта папка code будет местом хранения всего кода в этой книге.

Следующим шагом будет создание выделенного каталога для нашего библиотечного сайта, установка Django через Pipenv, а затем вход в виртуальную среду с помощью команды shell. Вы всегда должны использовать выделенную виртуальную среду для каждого нового проекта Python.

<https://pipenv-fork.readthedocs.io/en/latest/>

<https://djangoforbeginners.com/initial-setup/>

командная строка

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

Pipenv создает Pipfile и Pipfile.lock в нашем текущем каталоге. Знак (library) в скобках перед командной строкой показывает, что наша виртуальная среда активна.

Традиционный **Django**-сайт состоит из одного проекта и одного (или нескольких) приложений, представляющих отдельные функциональные возможности. Давайте создадим новый проект с помощью команды startproject. Не забудьте включить точку . в конце, которая устанавливает код в наш текущий каталог. Если вы не включите точку, Django по умолчанию создаст дополнительный каталог.

Командная строка :

```
(library) $ django-admin startproject config .
```

Django автоматически генерирует для нас новый проект, который мы можем увидеть с помощью команды tree.

(Примечание: Если tree не работает на Mac, установите его с помощью Homebrew 24: brew install tree .)

Command Line

```
(library) $ tree
```

```
.  
├── Pipfile  
├── Pipfile.lock  
└── config  
    ├── __init__.py  
    ├── asgi.py  
    ├── settings.py  
    ├── urls.py  
    └── wsgi.py  
└── manage.py
```

1 directory, 8 files

<https://brew.sh/>

Файлы имеют следующие роли:

- **__init__.py** - это способ Python рассматривать каталог как пакет; он пустой
- **asgi.py** означает Asynchronous Server Gateway Interface и является новой опцией в Django 3.0+
- **settings.py** содержит всю конфигурацию для нашего проекта
- **urls.py** управляет URL-маршрутами верхнего уровня
- **wsgi.py** означает Web Server Gateway Interface и помогает Django обслуживать конечные веб-страницы
- **manage.py** выполняет различные команды Django, такие как запуск локального веб-сервера или создание нового приложения.

Запустите `migrate` для синхронизации базы данных с настройками Django по умолчанию и запустите локальный Django веб-сервер.

Командная строка

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

Откройте веб-браузер на `http://127.0.0.1:8000/`, чтобы убедиться, что наш проект успешно установлен.

#от переводчика: (если вы используете pip, virtualenv):

команды в терминале будут такими :

```
mkdir django && cd django  
virtualenv venv  
source/venv/bin/activate  
pip3 install django=3.1.0  
django-admin startproject config .
```

если успешно всё запустилось , то по этому адресу откроется такая веб-страница:

django

[View release notes for Django 3.1](#)



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.

Первое приложение

Типичный следующий шаг - начать добавлять приложения, которые представляют собой отдельные области функциональности. Один проект **Django** может поддерживать несколько приложений.

Остановите локальный сервер, набрав Control+c, а затем создайте приложение books

командная строка:

```
(library) $ python manage.py startapp books
```

Теперь давайте посмотрим, какие файлы сгенерировал Django.

командная строка :

```
(library) $ tree
```

```
.
├── Pipfile
├── Pipfile.lock
└── books
    ├── __init__.py
    ├── admin.py
    ├── apps.py
    ├── migrations
        └── __init__.py
    ├── models.py
    ├── tests.py
    └── views.py
└── config
    ├── __init__.py
    ├── asgi.py
    ├── settings.py
    ├── urls.py
    └── wsgi.py
└── db.sqlite3
└── manage.py
```

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

- admin.py - это файл конфигурации для встроенного приложения Django Admin
- apps.py - конфигурационный файл для самого приложения
- в каталоге migrations/ хранятся файлы миграций для изменений в базе данных
- models.py - это место, где мы определяем наши модели базы данных
- tests.py - для наших тестов, специфичных для приложения
- views.py - это место, где мы обрабатываем логику запросов/ответов для нашего веб-приложения

Как правило, разработчики также создают файл urls.py в каждом приложении для маршрутизации.

Давайте создадим файлы так, чтобы наш проект Library выводил список всех книг на главной странице. Откройте в текстовом редакторе файл config/settings.py. Первым шагом будет добавление нового приложения в нашу конфигурацию INSTALLED_APPS.

Code

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

Для каждой веб-страницы в традиционном **Django** требуется несколько файлов: представление, url и шаблон. Но сначала нам нужна модель базы данных, поэтому давайте начнем с нее.

Models

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

code

```
# books/models.py
from django.db import models
class Book(models.Model):
    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):
        return self.title
```

Это базовая модель Django, в которой мы импортируем модели из Django в верхней строке, а затем создаем класс Book, который расширяет его. Есть четыре поля: название, подзаголовок, автор и isbn. Мы также включаем метод __str__, чтобы название книги в дальнейшем отображалось в админке.

Обратите внимание, что ISBN - это уникальный 13-символьный идентификатор, присваиваемый каждой опубликованной книге.

Поскольку мы создали новую модель базы данных, нам необходимо создать файл миграции. Указание имени приложения необязательно, но рекомендуется. Мы могли бы просто набрать `python manage.py makemigrations`, но если бы было несколько приложений с изменениями в базе данных, оба были бы добавлены в файл миграций, что усложнило бы отладку в будущем. Сохраняйте файлы миграций как можно более конкретными.

Затем запустите `migrate`, чтобы обновить нашу базу данных.

`python3 manage.py migrate`

Командная строка

(library) \$ python manage.py makemigrations books

Migrations for 'books':

 books/migrations/0001_initial.py

 - Create model Book

(library) \$ python manage.py migrate

Operations to perform:

 Apply all migrations: admin, auth, books, contenttypes, sessions

Running migrations:

 Applying books.0001_initial... OK

Пока все хорошо. Если что-то из этого кажется вам новым, я советую вам сделать паузу и просмотреть "**Django для начинающих**", чтобы получить более подробное объяснение традиционного Django.

Admin

Мы можем начать вводить данные в нашу новую модель через встроенное приложение **Django**. Но сначала мы должны сделать две вещи: создать учетную запись суперпользователя и обновить файл **admin.py**, чтобы отображалось приложение **books**.

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

командная строка:

(library) \$ python manage.py createsuperuser

Следуйте подсказкам для ввода имени пользователя, электронной почты и пароля. Обратите внимание, что в целях безопасности текст не будет отображаться на экране при вводе пароля.

Теперь обновите файл **admin.py** нашего книжного приложения.

code

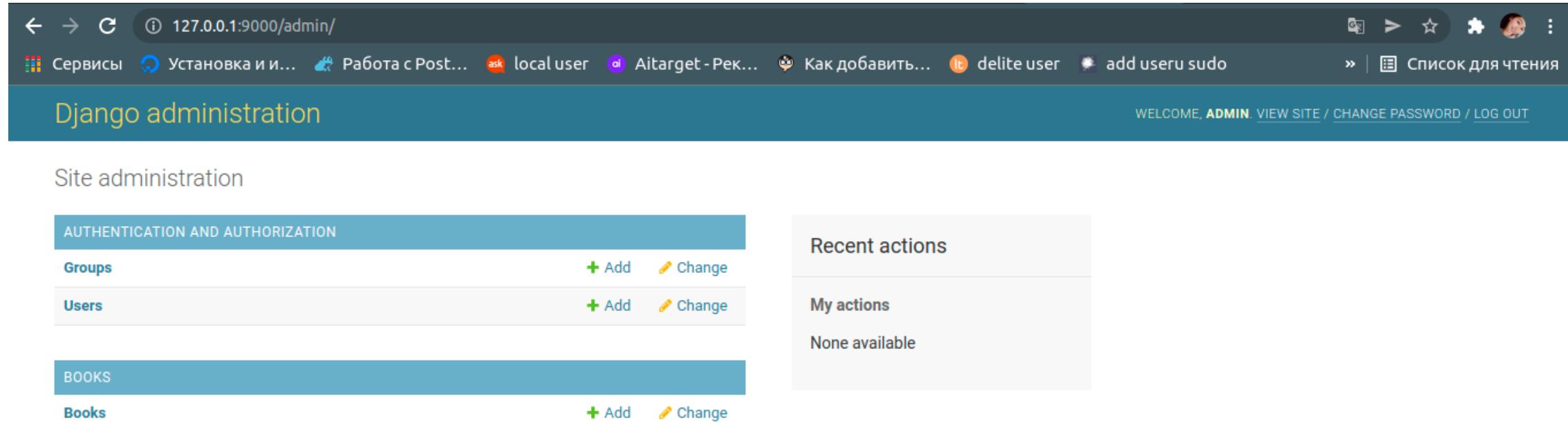
```
# books/admin.py
from django.contrib import admin
from .models import Book
admin.site.register(Book)
```

Это все, что нам нужно! Снова запустите локальный сервер.

командная строка

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

Перейдите по адресу <http://127.0.0.1:8000/admin> и войдите в систему. Вы будете перенаправлены на главную страницу администратора.



Нажмите на ссылку "+ add " рядом с Books .

Django administration

Home › Books › Books › Add book

AUTHENTICATION AND AUTHORIZATION

Groups + Add

Users + Add

BOOKS

Books + Add

Add book

Title: Записки Шерлока Холмса

Subtitle: Приключения Шерлока Холмса и доктора

Author: Артур Конан Дойл

ISBN: 13

Save and add another Save and continue editing SAVE

Администратор добавляет книгу

Я ввел данные для примера. Вы можете ввести здесь любой текст, какой захотите. Это чисто в демонстрационных целях. После нажатия кнопки "SAVE" мы перенаправляемся на страницу "Книги", где перечислены все текущие записи.

Select book to change

ADD BOOK +

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

BOOK

Записки Шерлока Холмса

1 book

Список книг администратора

В нашем традиционном Django-Проекте уже 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.

Представления (Views)

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`.

Обновите файл `books/views.py`

code

```
# books/views.py
from django.views.generic import ListView
from .models import Book
class BookListView(ListView):
    model = Book
    template_name = 'book_list.html'
```

В верхних строках мы импортировали `ListView` и нашу модель `Book`. Затем мы создаем класс `BookListView`, в котором указываем используемую модель и шаблон (еще не созданный).

Еще два шага, прежде чем у нас будет рабочая веб-страница: создание шаблона и настройка URL-адресов.

Давайте начнем с URL-адресов.

URLs

Нам нужно настроить как файл `urls.py` на уровне проекта, так и файл в приложении `books`. Когда пользователь заходит на наш сайт, он сначала взаимодействует с файлом `config/urls.py`, поэтому давайте сначала настроим его. Добавьте импорт `include` во второй строке, а затем новый путь для нашего приложения `books`.

Code

```
# config/urls.py
from django.contrib import admin
from django.urls import path, include # new
urlpatterns = [
    path('admin/', admin.site.urls),
    path('', include('books.urls')), # new
]
```

Две верхние строки импортируют встроенное приложение администратора, путь для наших маршрутов и `include`, который будет использоваться с нашим приложением `books`. Если пользователь перейдет по адресу `/admin/`, он будет перенаправлен в приложение администратора. Мы используем пустую строку " для маршрута приложения `books`, что означает, что пользователь на главной странице будет перенаправлен непосредственно в приложение `books`.

Теперь мы можем настроить наш файл `books/urls.py`. Но, ой! Django по какой-то причине не включает файл `urls.py` по умолчанию в приложения, поэтому нам нужно создать его самостоятельно. Если вы работаете на Mac, вы можете использовать команду `touch`; пользователи Windows должны создать файл в текстовом редакторе.

командная строка

```
(library) $ touch books/urls.py
```

Теперь в текстовом редакторе обновите новый файл.

Code

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

Мы импортируем наш файл представлений, настроим `BookListView` на пустую строку '' , и добавим именованный URL `home` в качестве лучшей практики.

Как работает Django, сейчас, когда пользователь заходит на главную страницу нашего сайта, он сначала попадает в файл config/urls.py, затем перенаправляется на файл books/urls.py, который определяет использование BookListView. В этом файле представления, модель Book используется вместе с ListView для вывода списка всех книг.

Последний шаг - создание нашего файла шаблона, который управляет макетом на реальной веб-странице.

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

Какой из них вы в конечном итоге используете в своих проектах - это ваше личное предпочтение. Здесь мы будем использовать структуру по умолчанию. Если вам интересно узнать о втором подходе, обратитесь к книге "**Django Для начинающих**".

Начните с создания новой папки templates в приложении books, затем внутри нее папки books и, наконец, файла book_list.html.

Командная строка:

```
(library) $ mkdir books/templates  
(library) $ mkdir books/templates/books  
(library) $ touch books/templates/books/book_list.html
```

Теперь обновите файл шаблона.

HTML

```
<!-- books/templates/books/book_list.html -->  
<h1>All books</h1>  
{% for book in object_list %}  
    <ul>  
        <li>Title: {{ book.title }}</li>  
        <li>Subtitle: {{ book.subtitle }}</li>  
        <li>Author: {{ book.author }}</li>  
        <li>ISBN: {{ book.isbn }}</li>  
    </ul>  
{% endfor %}
```

Django поставляется с языком шаблонов , который позволяет использовать базовую логику. Здесь мы используем тег for для перебора всех доступных книг. Теги шаблона должны быть включены в открывающие/закрывающие скобки и круглые скобки. Таким образом, формат всегда {`% for ... %`}, а затем мы должны закрыть наш цикл с помощью {`% endfor %`}.

То, над чем мы работаем в цикле, - это объект, содержащий все доступные книги в нашей модели, созданной с помощью ListView. Имя этого объекта - object_list . Поэтому, чтобы перебрать каждую книгу, мы напишем {`% for book in object_list %`}. А затем отобразим каждое поле из нашей модели.

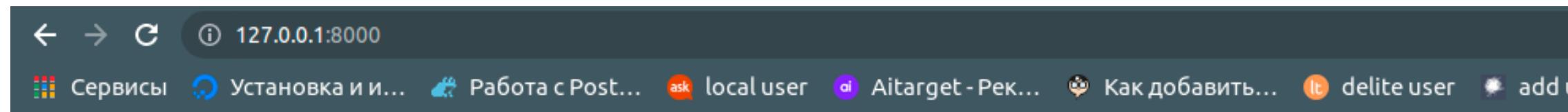
Web-page (Веб-страница)

Теперь мы можем запустить локальный сервер Django и посмотреть нашу веб-страницу.

Командная строка

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

Перейдите на домашнюю страницу, которая находится по адресу <http://127.0.0.1:8000/>.



All books

- Title: Записки Шерлока Холмса
- Subtitle: Приключения Шерлока Холмса и доктора Ватсона: Собака Баскервилей
- Author: Артур Конан Дойл
- ISBN: 13

Веб-страница книги

Если мы добавим дополнительные книги в админке, они тоже появятся здесь.

Это была очень быстрая проработка традиционного Django-сайта. Теперь давайте добавим к нему API!

Django REST Framework

Django REST Framework добавляется так же, как и любое другое стороннее приложение. Обязательно завершите работу локального сервера **Control+c**, если он все еще запущен. Затем в командной строке введите следующий текст.

Командная строка

```
(library) $ pipenv install djangorestframework~=3.11.0
```

кто просто использует pip и venv то введите :

```
(venv) $ pip3 install djangorestframework~=3.11.0
```

Добавьте **rest_framework** в конфигурацию **INSTALLED_APPS** в нашем файле **config/settings.py**. Мне нравится делать различие между сторонними приложениями и локальными приложениями следующим образом, поскольку в большинстве проектов количество приложений быстро растет.

Code

```
# config/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', # new
    # Local
    'books',
```

1

В конечном итоге, наш API будет представлять одну конечную точку, которая выводит список всех книг в JSON. Поэтому нам понадобится новый URL-маршрут, новое представление и новый файл сериализатора (подробнее об этом в ближайшее время).

Существует несколько способов организации этих файлов, однако я предпочитаю создать специальное API приложение. Таким образом, даже если в будущем мы добавим еще несколько приложений, каждое из них сможет содержать модели, представления, шаблоны и URL, необходимые для отдельных веб-страниц, но все специфические для API файлы для всего проекта будут находиться в специальном API приложении.

Давайте сначала создадим новое API приложение.

Командная строка

```
(library) $ python manage.py startapp api
```

Затем добавьте его в INSTALLED_APPS .

Code

```
# config/settings.py
INSTALLED_APPS = [
    # Local
    'books.apps.BooksConfig',
    'api.apps.ApiConfig', # new
    # 3rd party
    'rest_framework',
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
]
```

Приложение API не будет иметь собственных моделей базы данных, поэтому нет необходимости создавать файл миграции и запускать migrate для обновления базы данных.

URL-адреса

Давайте начнем с конфигурации наших URL. Добавление конечной точки API происходит точно так же, как и настройка маршрутов традиционного приложения Django. Сначала на уровне проекта нам нужно включить приложение API и настроить его URL-маршрут, который будет API/ .

Code

```
# config/urls.py
from django.contrib import admin
from django.urls import path, include
urlpatterns = [
    path('admin/', admin.site.urls),
    path('api/', include('api.urls')), # new
    path(' ', include('books.urls')),
]
```

Затем создайте файл urls.py в приложении api.

командная строка:

```
(library) $ touch api/urls.py
```

И обновите его следующим образом:

Code

```
# api/urls.py
from django.urls import path
from .views import BookAPIView
urlpatterns = [
    path(' ', BookAPIView.as_view()),
]
```

Все готово.

Views (представления)

Следующим идет наш файл **views.py**, который полагается на встроенные в **Django REST Framework** общие представления классов. По формату они намеренно имитируют традиционные представления Django на основе общих классов, но это не одно и то же.

Чтобы избежать путаницы, некоторые разработчики называют файл представлений API **apiviews.py** или **api.py**. Лично я, работая в рамках выделенного **api** приложения, не нахожу путаницы в том, чтобы просто называть файл представлений **views.py** из **Django REST Framework**, но мнения по этому поводу расходятся.

В нашем файле `views.py` обновите его, чтобы он выглядел следующим образом:

Code

```
# api/views.py
from rest_framework import generics
from books.models import Book
from .serializers import BookSerializer
class BookAPIView(generics.ListAPIView):
    queryset = Book.objects.all()
    serializer_class = BookSerializer
```

В верхних строках мы импортируем класс представлений `generics` из **Django REST Framework**, модели из нашего приложения `books` и сериализаторы из нашего приложения `api` (сериализаторы мы сделаем позже).

Затем мы создаем представление `BookAPIView`, которое использует `ListAPIView` для создания конечной точки "только для чтения" для всех экземпляров книг. Существует множество типовых представлений, и мы рассмотрим их подробнее в последующих главах.

Единственные два шага, необходимые в нашем представлении, - это указать набор запросов, который представляет собой все доступные книги, а затем класс_сериализатора, которым будет `BookSerializer`.

Serializers(Сериализаторы)

Сериализатор переводит данные в формат, который легко использовать через интернет, обычно это `JSON`, и отображается на конечной точке API. Мы также более подробно рассмотрим сериализаторы и `JSON` в следующих главах. Пока же я хочу продемонстрировать, как легко создать сериализатор с помощью Django REST Framework для преобразования моделей Django в `JSON`.

Создайте файл `serializers.py` в нашем `api` приложении.

командная строка:

```
(library) $ touch api/serializers.py
```

Затем обновите его в текстовом редакторе следующим образом.

Code

```
# api/serializers.py
from rest_framework import serializers
from books.models import Book
class BookSerializer(serializers.ModelSerializer):
    class Meta:
        model = Book
        fields = ('title', 'subtitle', 'author', 'isbn')
```

В верхней строке мы импортируем класс сериализаторов **Django REST Framework** и модель Book из нашего приложения books. Мы расширяем ModelSerializer из **Django REST Framework** в класс BookSerializer, который определяет нашу модель базы данных Book и поля базы данных, которые мы хотим раскрыть: title, subtitle, author и isbn.

Вот и все! Мы закончили.

cURL

Мы хотим посмотреть, как выглядит наша конечная точка API. Мы знаем, что она должна возвращать JSON по адресу URL <http://127.0.0.1:8000/api/>. Давайте убедимся, что наш локальный сервер Django запущен:

командная строка:

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

Теперь откройте новую, вторую консоль командной строки. Мы будем использовать ее для доступа к API, запущенному в существующей консоли командной строки.

Мы можем использовать популярную программу cURL для выполнения HTTP-запросов через командную строку. Все, что нам нужно для базового GET-запроса, это указать curl и URL, который мы хотим вызвать.

командная строка:

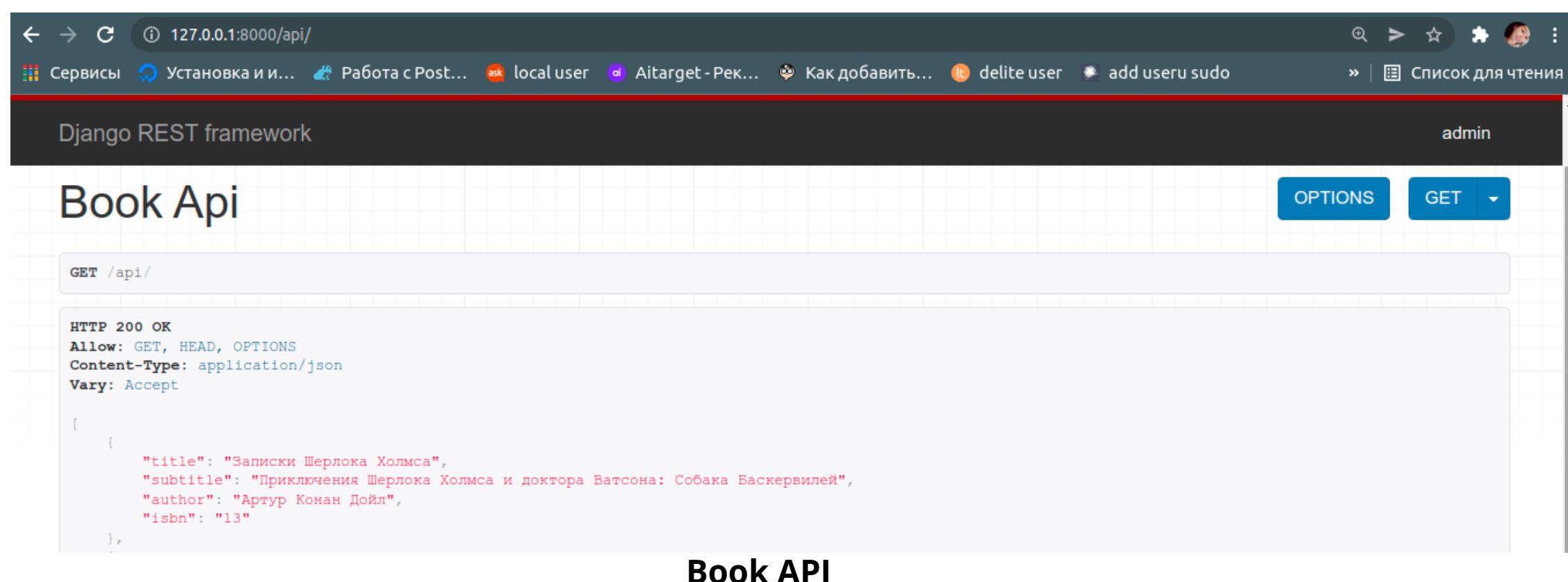
```
$ curl http://127.0.0.1:8000/api/
[
    {
        "title": "Записки Шерлока Холмса",
        "subtitle": "Приключения Шерлока Холмса и доктора Ватсона:  
Собака Баскервилей",
        "author": "Артур Конан Дойл",
        "isbn": "9781735467207"
    }
]
```

Все данные есть, в формате JSON, но они плохо отформатированы, и в них трудно разобраться.

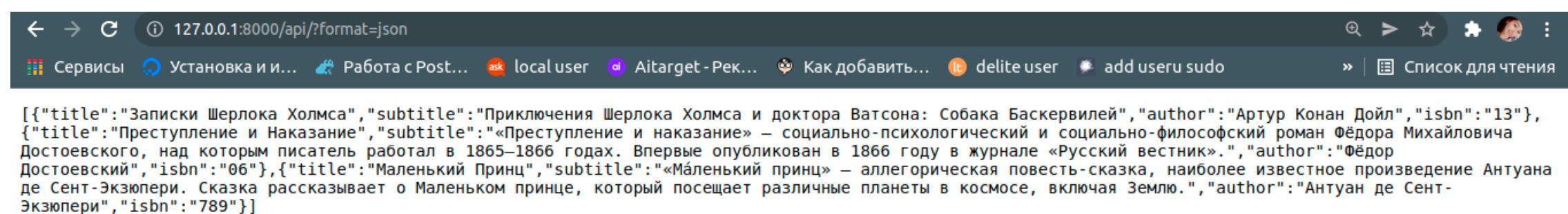
К счастью, **Django REST Framework** подготовил для нас еще один сюрприз: мощный визуальный режим для наших конечных точек API.

Просматриваемый API

Поскольку локальный сервер все еще запущен в первой консоли командной строки, перейдите к нашей конечной точке API в веб-браузере по адресу <http://127.0.0.1:8000/api/>.



Посмотрите на это! Django REST Framework предоставляет эту визуализацию по умолчанию. И в эту страницу встроено множество функций, которые мы будем изучать на протяжении всей книги. Сейчас я хочу, чтобы вы сравнили эту страницу с необработанной конечной точкой JSON. Нажмите на кнопку "GET" и выберите "json" из выпадающего меню.



The screenshot shows a browser window with the URL `127.0.0.1:8000/api/?format=json`. The page title is "Book API JSON". The content area displays a JSON array of book records:

```
[{"title": "Записки Шерлока Холмса", "subtitle": "Приключения Шерлока Холмса и доктора Ватсона: Собака Баскервилей", "author": "Артур Конан Дойл", "isbn": "13"}, {"title": "Преступление и Наказание", "subtitle": "«Преступление и наказание» – социально-психологический и социально-философский роман Фёдора Михайловича Достоевского, над которым писатель работал в 1865–1866 годах. Впервые опубликован в 1866 году в журнале «Русский вестник».", "author": "Фёдор Достоевский", "isbn": "06"}, {"title": "Маленький Принц", "subtitle": "«Маленький принц» – аллегорическая повесть-сказка, наиболее известное произведение Антуана де Сент-Экзюпери. Сказка рассказывает о Маленьком принце, который посещает различные планеты в космосе, включая Землю.", "author": "Антуан де Сент-Экзюпери", "isbn": "789"}]
```

Book API JSON

Вот как выглядит необработанный JSON из конечной точки нашего API. Я думаю, мы можем согласиться, что версия **Django REST Framework** более привлекательна.

Заключение

В этой главе мы рассмотрели много материала, поэтому не волнуйтесь, если сейчас все кажется немного запутанным. Сначала мы создали традиционный сайт библиотеки Django. Затем мы добавили **Django REST Framework** и смогли добавить конечную точку API с минимальным количеством кода.

В следующих двух главах мы создадим наш собственный бэкэнд API Todo и соединим его с фронтэндом на базе React, чтобы продемонстрировать полный рабочий пример, который поможет подтвердить, как вся эта теория сочетается на практике!

Код из этой главы можете найти тут:



<https://github.com/jumabekova06/Code-of-DjangoApiBook>

Глава 3 : Todo API

В течение следующих двух глав мы создадим бэкэнд Todo API, а затем соединим его с фронтэндом React. Мы уже создали наш первый API и рассмотрели, как абстрактно работают HTTP и REST, но, скорее всего, вы еще не совсем понимаете, как все это сочетается. К концу этих двух глав вы поймете.

Поскольку мы создаем отдельный back-end и front-end, мы разделим наш код на подобную структуру. В существующем каталоге кода мы создадим каталог todo, содержащий наш внутренний код Django Python и наш внешний код React JavaScript.

В конечном итоге схема будет выглядеть следующим образом.

Diagram

todo

|

 └── frontend

|

 └── React...

|

 └── backend

|

 └── Django...

Эта глава посвящена внутреннему интерфейсу, а глава 4 - внешнему интерфейсу.

Начальная настройка

Первым шагом для любого Django API всегда является установка Django и последующее добавление Django REST Framework поверх него. Сначала создайте специальный каталог todo в нашем каталоге кода на рабочем столе.

Откройте новую консоль командной строки и введите следующие команды:

Командная строка

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

Примечание: Убедитесь, что вы деактивировали виртуальную среду из предыдущей главы. Это можно сделать, набрав exit или же deactivate . Перед командной строкой больше нет круглых скобок? Хорошо. Значит, вы находитесь не в существующей виртуальной среде.

Внутри этой папки todo будут наши каталоги backend и frontend. Давайте создадим папку backend, установим Django и активируем новую виртуальную среду.

Командная строка (от Автора)

```
$ mkdir backend && cd backend  
$ pipenv install django~=3.1.0  
$ pipenv shell
```

Командная строка (от Переводчика)

```
$ mkdir backend && cd backend  
$ virtualenv backend  
$ source backend/bin/activate  
$ pip install django~=3.1.0
```

Вы должны увидеть скобки в командной строке, подтверждающие, что виртуальная среда (бэкенд) активирована.

Теперь, когда Django установлен, мы должны начать с создания традиционной конфигурации проекта Django, добавления в нее наших первых приложений, а затем переноса нашей начальной базы данных.

Командная строка

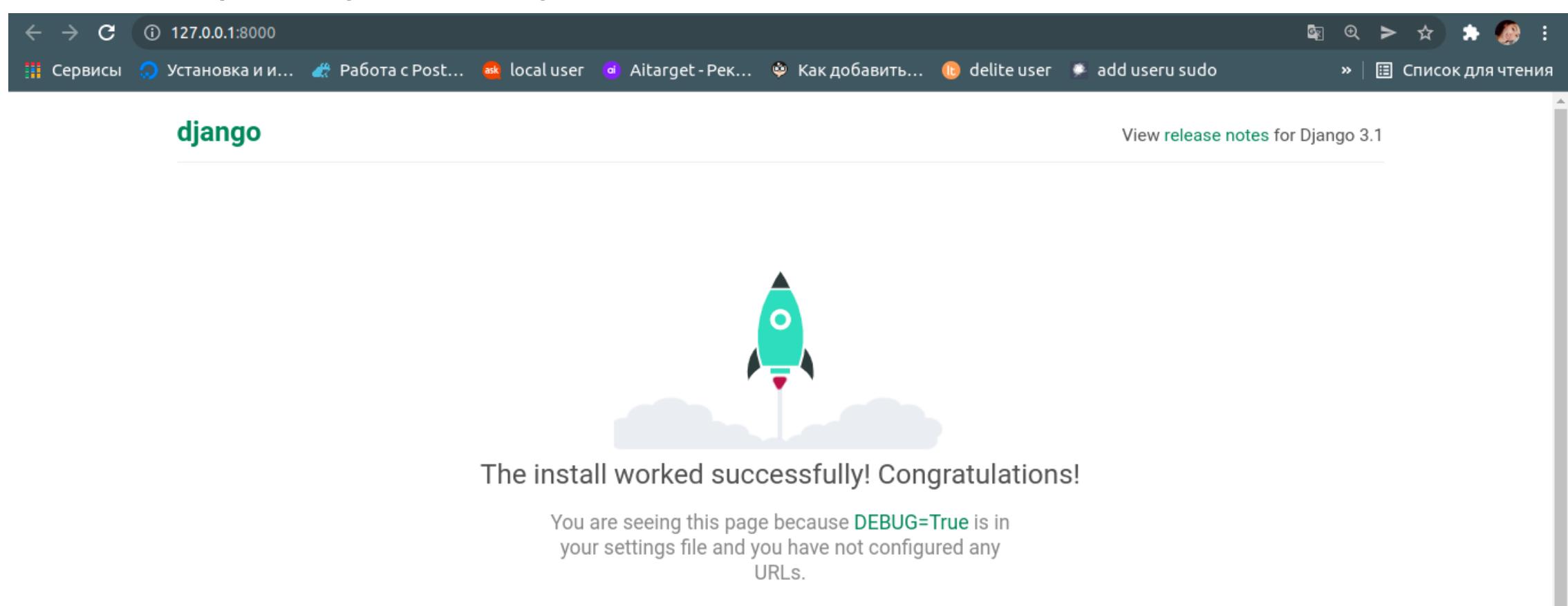
```
(backend) $ django-admin startproject config .  
(backend) $ python manage.py startapp todos  
(backend) $ python manage.py migrate
```

В Django нам всегда нужно добавлять новые приложения в настройки INSTALLED_APPS, поэтому сделайте это сейчас. Откройте config/settings.py в текстовом редакторе. В нижней части файла добавьте todos .

Code

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

Если теперь запустить `python manage.py runserver` в командной строке и перейти в веб-браузере по адресу `http://127.0.0.1:8000/`, то можно увидеть, что наш проект успешно установлен.



Мы готовы к работе!

Models (Модели)

Следующим шагом будет определение нашей модели базы данных Todo в приложении todos. Мы будем придерживаться базового подхода и иметь только два поля: title и body.

Code

```
# todos/models.py

from django.db import models

class Todo(models.Model):
    title = models.CharField(max_length=200)
    body = models.TextField()
    def __str__(self):
        return self.title
```

Мы импортируем модели в верхней части, а затем подклассифицируем ее, чтобы создать нашу собственную модель Todo. Мы также добавляем `__str__`, чтобы обеспечить человекочитаемое имя для каждого будущего экземпляра модели.

Поскольку мы обновили нашу модель, настало время для двухэтапного танца Django - создания нового файла миграции и затем синхронизировать базу данных с изменениями каждый раз. В командной строке наберите Control+c, чтобы остановить наш локальный сервер. Затем выполните эти две команды:

Командная строка:

```
(backend) $ python manage.py makemigrations todos
Migrations for 'todos':
    todos/migrations/0001_initial.py
        - Create model Todo
(backend) $ python manage.py migrate
Operations to perform:
    Apply all migrations: admin, auth, contenttypes, sessions, todos
Running migrations:
    Applying todos.0001_initial... OK
```

Добавлять конкретное приложение, для которого мы хотим создать файл миграции, необязательно - вместо этого мы можем набрать просто `python manage.py makemigrations` - тем не менее, это хорошая практика.

Миграционные файлы - это фантастический способ отладки приложений, и вы должны стремиться создавать миграционный файл для каждого небольшого изменения. Если бы мы обновили модели в двух разных приложениях, а затем запустили `python manage.py makemigrations`, то получившийся единый файл миграции содержал бы данные по обоим приложениям. Это только усложняет отладку. Страйтесь, чтобы ваши миграции были как можно меньше.

Теперь мы можем использовать встроенное приложение Django admin 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` следующим образом.

Code

```
# todos/admin.py
from django.contrib import admin
from .models import Todo
admin.site.register(Todo)
```

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

командная строка:

```
(backend) $ python manage.py createsuperuser
```

А затем снова запустите локальный сервер:

Командная строка:

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

Если вы перейдете по адресу `http://127.0.0.1:8000/admin/`, вы сможете войти в систему. Нажмите на "+ Добавить" рядом с Todos и создайте 3 новых пункта, обязательно добавив заголовок и тело для обоих. Вот как выглядит мой пункт:

The screenshot shows the Django Admin interface for the 'Todos' model. At the top, there's a success message: "The todo 'William S. Vincent' was added successfully." Below the message, there's a table with one row containing the text "William S. Vincent". At the bottom of the table, it says "1 todo". On the left side, there are navigation links for 'Home', 'Todos', and 'Todos'. There are also buttons for 'Groups', 'Users', 'Todos', and 'ADD TODO'.

На этом мы фактически закончили с традиционной Django-частью нашего Todo API. Поскольку мы не собираемся создавать веб-страницы для этого проекта, нет необходимости в URL сайта, представлениях или шаблонах. Все, что нам нужно - это модель, а **Django REST Framework** позаботится обо всем остальном.

Django REST Framework

Остановите локальный сервер Control+c и установите Django REST Framework через pipenv . (или же через pip)

Командная строка:

```
(backend) $ pipenv install djangorestframework~=3.11.0
```

Затем добавьте rest_framework в настройки INSTALLED_APPS, как и любое другое стороннее приложение. Мы также хотим начать настройку специфических параметров Django REST Framework, которые все существуют в разделе REST_FRAMEWORK . Для начала, давайте явно установим разрешения на AllowAny 35 . Эта строка находится в самом низу файла.

code

```
INSTALLED_APPS = [  
    'django.contrib.admin',  
    'django.contrib.auth',  
    'django.contrib.contenttypes',  
    'django.contrib.sessions',  
    'django.contrib.messages',  
    'django.contrib.staticfiles',  
    # 3rd party  
    'rest_framework', # new  
    # Local  
    'todos',  
]  
# new  
REST_FRAMEWORK = {  
    'DEFAULT_PERMISSION_CLASSES': [  
        'rest_framework.permissions.AllowAny',  
    ]  
}
```

Django REST Framework имеет длинный список неявно заданных параметров по умолчанию. Полный список можно посмотреть здесь <http://www.django-rest-framework.org/api-guide/settings/>. AllowAny является одним из них, что означает, что когда мы устанавливаем его явно, как мы сделали выше, эффект будет точно таким же, как если бы у нас не было настроек DEFAULT_PERMISSION_CLASSES.

Изучение настроек по умолчанию требует времени. Мы познакомимся с некоторыми из них в течение книги. Главное, что нужно запомнить, это то, что настройки по умолчанию в `implci` созданы для того, чтобы разработчики могли быстро включиться и начать работать в локальной среде разработки. Однако настройки по умолчанию не подходят для производства. Поэтому, как правило, мы будем вносить в них ряд изменений по ходу проекта.

Итак, Django REST Framework установлен. Что дальше? В отличие от проекта Library в предыдущей главе, где мы создавали и веб-страницу, и API, здесь мы создаем только API. Поэтому нам не нужно создавать никаких файлов шаблонов или традиционных представлений Django.

Вместо этого мы обновим три файла, специфичных для Django REST Framework, чтобы преобразовать нашу модель базы данных в веб-API: `urls.py`, `views.py`, и `serializers.py`.

URLs

Мне нравится начинать с URL, поскольку они являются точкой входа для наших конечных точек API. Как и в традиционном проекте Django, файл `urls.py` позволяет нам настроить маршрутизацию.

Начните с файла уровня проекта Django - `config/urls.py`. Мы импортируем `include` во второй и добавим маршрут для нашего приложения `todos` по адресу `api/`.

Code

```
# config/urls.py
from django.contrib import admin
from django.urls import include, path # new
urlpatterns = [
    path('admin/', admin.site.urls),
    path('api/', include('todos.urls')), # new
]
```

Затем создайте наш файл todos/urls.py на уровне приложения.

командная строка:

(backend) \$ touch todos/urls.py

И обновите его с помощью приведенного ниже кода.

code

```
# todos/urls.py
from django.urls import path
from .views import ListTodo, DetailTodo
urlpatterns = [
    path('<int:pk>/', DetailTodo.as_view()),
    path('', ListTodo.as_view()),
]
```

Обратите внимание, что мы ссылаемся на два представления - ListTodo и DetailTodo, которые нам еще предстоит создать. Но теперь маршрутизация завершена. По адресу пустой строки "", другими словами по адресу api/, будет находиться список всех todo. А каждый отдельный todo будет доступен по его первичному ключу, который Django автоматически устанавливает в каждой таблице базы данных. Первая запись будет 1, вторая - 2 и так далее. Поэтому наш первый todo в конечном итоге будет расположен в конечной точке API api/1/ .

Serializers (Сериализаторы)

Давайте рассмотрим, на чем мы остановились. Мы начали с традиционного Django проекта и приложения, где мы создали модель базы данных и добавили данные. Затем мы установили Django REST Framework и настроили наши URL. Теперь нам нужно преобразовать наши данные из моделей в JSON, который будет выводиться на URL-адреса. Поэтому нам нужен сериализатор.

Django REST Framework поставляется с мощным встроенным классом сериализаторов, который мы можем быстро расширить с помощью небольшого количества кода. Именно это мы и сделаем здесь. Сначала создайте новый файл serializers.py в приложении todos.

командная строка:

(backend) \$ touch todos/serializers.py

Затем обновите его следующим кодом.

code

```
# todos/serializers.py
from rest_framework import serializers
from .models import Todo
class TodoSerializer(serializers.ModelSerializer):
    class Meta:
        model = Todo
        fields = ('id', 'title', 'body')
```

В верхней части мы импортировали сериализаторы из Django REST Framework, а также из нашего файла models.py. Далее мы создаем класс TodoSerializer . Формат здесь очень похож на то, как мы создаем классы моделей или форм в самом Django. Мы указываем, какую модель использовать и какие именно поля в ней мы хотим раскрыть. Помните, что id создается Django автоматически, поэтому нам не нужно было определять его в нашей модели Todo, но мы будем использовать его в нашем детальном представлении.

Вот и все. Django REST Framework теперь волшебным образом преобразует наши данные в JSON, раскрывая поля для id, title и body из нашей модели Todo.

Последнее, что нам нужно сделать, это настроить наш файл views.py.

Views (Представления)

В традиционном Django представления используются для настройки того, какие данные отправлять в шаблоны. В Django REST Framework представления делают то же самое, но для наших сериализованных данных.

Синтаксис представлений Django REST Framework намеренно очень похож на обычные представления Django, и, как и в обычном Django, Django REST Framework поставляется с типовыми представлениями для общих случаев использования. Именно их мы и будем использовать здесь.

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

Code

```
# todos/views.py
from rest_framework import generics
from .models import Todo
from .serializers import TodoSerializer

class ListTodo(generics.ListAPIView):
    queryset = Todo.objects.all()
    serializer_class = TodoSerializer

class DetailTodo(generics.RetrieveAPIView):
    queryset = Todo.objects.all()
    serializer_class = TodoSerializer
```

В верхней части мы импортируем представления generics от Django REST Framework и оба наших файла models.py и serializers.py.

Вспомните из нашего файла todos/urls.py, что у нас есть два маршрута и, следовательно, два разных представления. Мы будем использовать ListAPIView для отображения всех todos и RetrieveAPIView для отображения одного экземпляра модели.

Внимательные читатели заметят, что здесь в коде есть немного избыточности. По сути, мы повторяем queryset и serializer_class для каждого представления, даже несмотря на то, что общее представление расширено по-разному. Позже в книге мы узнаем о наборах представлений и маршрутизаторах, которые решают эту проблему и позволяют нам создавать те же представления API и URL с гораздо меньшим количеством кода.

Но на данный момент мы закончили! Наш API готов к использованию. Как вы видите, единственное реальное различие между Django REST Framework и Django заключается в том, что в Django REST Framework нам нужно добавить файл serializers.py и нам не нужен файл templates. В остальном файлы urls.py и views.py действуют аналогичным образом.

Потребление API

Традиционно потребление API было сложной задачей. Просто не существовало хороших визуализаций для всей информации, содержащейся в теле и заголовке данного HTTP-ответа или запроса. Вместо этого большинство разработчиков использовали HTTP-клиент командной строки, например cURL , который мы рассматривали в предыдущей главе, или HTTP.

В 2012 году был выпущен сторонний программный продукт Postman 41 , который сегодня используется миллионами разработчиков по всему миру, желающих получить визуальный, многофункциональный способ взаимодействия с API.

Но одна из самых удивительных вещей в Django REST Framework - это то, что он поставляется с мощным просматриваемым API, который мы можем использовать сразу же. Если вы обнаружите, что вам нужна дополнительная настройка для работы с API, то для этого есть такие инструменты, как Postman. Но зачастую встроенного просматриваемого API более чем достаточно.

Просматриваемый API

Давайте теперь воспользуемся просматриваемым API для взаимодействия с нашими данными. Убедитесь, что локальный сервер запущен.

командная строка

(backend) \$ python manage.py runserver

Затем перейдите по адресу <http://127.0.0.1:8000/api/>, чтобы увидеть нашу рабочую конечную точку представления списка API.

List Todo

OPTIONS

GET

GET /api/

```

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

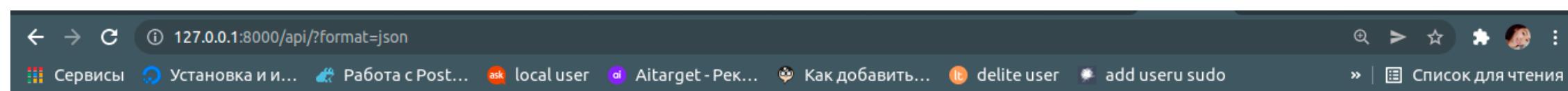
[
    {
        "id": 1,
        "title": "William S. Vincent",
        "body": "Автор нескольких книг по джанго"
    },
    {
        "id": 2,
        "title": "Antonio Mele",
        "body": "Автор книги Django by example"
    }
]

```

API List

На этой странице отображаются три todos, которые мы создали ранее в модели базы данных. Конечная точка API известна как коллекция, потому что она показывает несколько элементов.

Мы можем многое сделать с нашим API с возможностью просмотра. Для начала давайте посмотрим необработанный вид JSON - то, что на самом деле будет передано через интернет. Нажмите на кнопку "GET" в правом верхнем углу и выберите JSON .



API JSON

Если вернуться на страницу просмотра списка `http://127.0.0.1:8000/api/`, то можно увидеть, что там есть дополнительная информация. Напомним, что HTTP- метод GET используется для чтения данных, а POST - для обновления или создания данных.

Под "List Todo" написано GET /api/, что говорит нам о том, что мы выполнили GET на этой конечной точке.

Ниже написано HTTP 200 OK, что является нашим кодом состояния, все работает. Очень важно, что ниже показано ALLOW: GET, HEAD, OPTIONS. Обратите внимание, что здесь не указан POST, поскольку это конечная точка только для чтения, мы можем выполнять только GET.

Мы также создали представление DetailTodo для каждой отдельной модели. Оно известно как экземпляр и видно на сайте `http://127.0.0.1:8000/api/1/`. По сути это детальная информация об одном объекте.

The screenshot shows a browser window with the URL `127.0.0.1:8000/api/1/`. The page title is "Django REST framework" and the user is logged in as "admin". The main content is titled "Detail Todo". There are two buttons at the top right: "OPTIONS" and "GET ▾". Below these are sections for "GET /api/1/" and "HTTP 200 OK". The "GET /api/1/" section shows the response headers: `Allow: GET, HEAD, OPTIONS`, `Content-Type: application/json`, and `Vary: Accept`. The "HTTP 200 OK" section shows a JSON response with one item:

```
{  
    "id": 1,  
    "title": "William S. Vincent",  
    "body": "Автор нескольких книг по джанго"  
}
```

API Detail

CORS

Остался последний шаг, который нам необходимо сделать, это разобраться с технологией современных браузеров (CORS). Всякий раз, когда клиент взаимодействует с API, размещенным на другом домене (`mysite.com` против `yoursite.com`) или порту (`localhost:3000` против `localhost:8000`), возникают потенциальные проблемы безопасности.

В частности, CORS требует от сервера включения определенных HTTP-заголовков, которые позволяют клиенту определить, разрешены ли междоменные запросы и когда они должны быть разрешены.

Наш Django API back-end будет взаимодействовать с выделенным front-end, который расположен на другом порту для локальной разработки и на другом домене после развертывания.

Самый простой способ справиться с этим - и тот, который рекомендуется Django REST Framework - использовать промежуточное ПО, которое будет автоматически включать соответствующие HTTP-заголовки на основе наших настроек.

Мы будем использовать пакет `django-cors-headers`, который можно легко добавить в наш существующий проект.

Сначала выйдите из нашего сервера с помощью `Control+c`, а затем установите `django-cors-headers` с помощью `Pipenv` .(или `pip`)

командная строка :

(backend) \$ `pipenv install django-cors-headers==3.4.0`

<https://developer.mozilla.org/en-US/docs/Web/HTTP/CORS>

<http://www.django-rest-framework.org/topics/ajax-csrf-cors/>

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

Далее обновите наш файл config/settings.py в трех местах:

- добавить corsheaders в INSTALLED_APPS
- добавить CorsMiddleware над CommonMiddleware в MIDDLEWARE
- создать CORS_ORIGIN_WHITELIST

Code

```
# config/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',
    'corsheaders', # new

# Local
    'todos',
]

MIDDLEWARE = [
    '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',
]

# new

CORS_ORIGIN_WHITELIST = (
    'http://localhost:3000',
    'http://localhost:8000',
)
```

Очень важно, чтобы corsheaders.middleware.CorsMiddleware появился в правильном месте. То есть выше django.middleware.common.CommonMiddleware в настройке MIDDLEWARE поскольку промежуточные программы загружаются сверху вниз. Также обратите внимание, что мы внесли в белый список два домена: localhost:3000 и localhost:8000 .

Первый является портом по умолчанию для React, который мы будем использовать для нашего фронтенда в следующей главе. Второй - это порт по умолчанию для Django.

Тесты

Вы всегда должны писать тесты для своих проектов Django. Небольшое количество времени, потраченное заранее, сэкономит вам огромное количество времени и усилий на отладке ошибок. Давайте добавим два основных теста, чтобы убедиться, что заголовок и содержимое тела ведут себя так, как ожидается.

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

Code

```
# todos/tests.py
from django.test import TestCase
from .models import Todo
class TodoModelTest(TestCase):
    @classmethod
    def setUpTestData(cls):
        Todo.objects.create(title='first todo', body='a body here')
    def test_title_content(self):
        todo = Todo.objects.get(id=1)
        expected_object_name = f'{todo.title}'
        self.assertEqual(expected_object_name, 'first todo')
    def test_body_content(self):
        todo = Todo.objects.get(id=1)
        expected_object_name = f'{todo.body}'
        self.assertEqual(expected_object_name, 'a body here')
```

Для этого используется встроенный в Django класс TestCase. Сначала мы устанавливаем наши данные в setUpTestData, а затем пишем два новых теста. Затем запускаем тесты с помощью команды python manage.py test.

Первый является портом по умолчанию для React, который мы будем использовать для нашего фронтенда в следующей главе. Второй - это порт по умолчанию для Django.

Тесты

Вы всегда должны писать тесты для своих проектов Django. Небольшое количество времени, потраченное заранее, сэкономит вам огромное количество времени и усилий на отладке ошибок. Давайте добавим два основных теста, чтобы убедиться, что заголовок и содержимое тела ведут себя так, как ожидается.

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

Code

```
# todos/tests.py
from django.test import TestCase
from .models import Todo
class TodoModelTest(TestCase):
    @classmethod
    def setUpTestData(cls):
        Todo.objects.create(title='first todo', body='a body here')
    def test_title_content(self):
        todo = Todo.objects.get(id=1)
        expected_object_name = f'{todo.title}'
        self.assertEqual(expected_object_name, 'first todo')
    def test_body_content(self):
        todo = Todo.objects.get(id=1)
        expected_object_name = f'{todo.body}'
        self.assertEqual(expected_object_name, 'a body here')
```

Для этого используется встроенный в Django класс TestCase. Сначала мы устанавливаем наши данные в setUpTestData, а затем пишем два новых теста. Затем запускаем тесты с помощью команды python manage.py test.

▼ TERMINAL

```
(backend) → backend git:(master) ✘ python3 manage.py test
Creating test database for alias 'default'...
System check identified no issues (0 silenced).
.
-----
Ran 2 tests in 0.004s

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

Вот и все! Теперь наша внутренняя часть завершена. Убедитесь, что сервер запущен, так как мы будем использовать его в следующей главе.

командная строка

(backend) \$ `python manage.py runserver`

Вывод

С минимальным количеством кода Django REST Framework позволил нам создать Django API с нуля. Из традиционного Django нам понадобились только файл `models.py` и наши маршруты `urls.py`. Файлы `views.py` и `serializers.py` были полностью специфичны для Django REST Framework.

В отличие от нашего примера в предыдущей главе, мы не создавали никаких веб-страниц для этого проекта, поскольку нашей целью было только создание API. Однако в любой момент в будущем мы легко сможем это сделать!

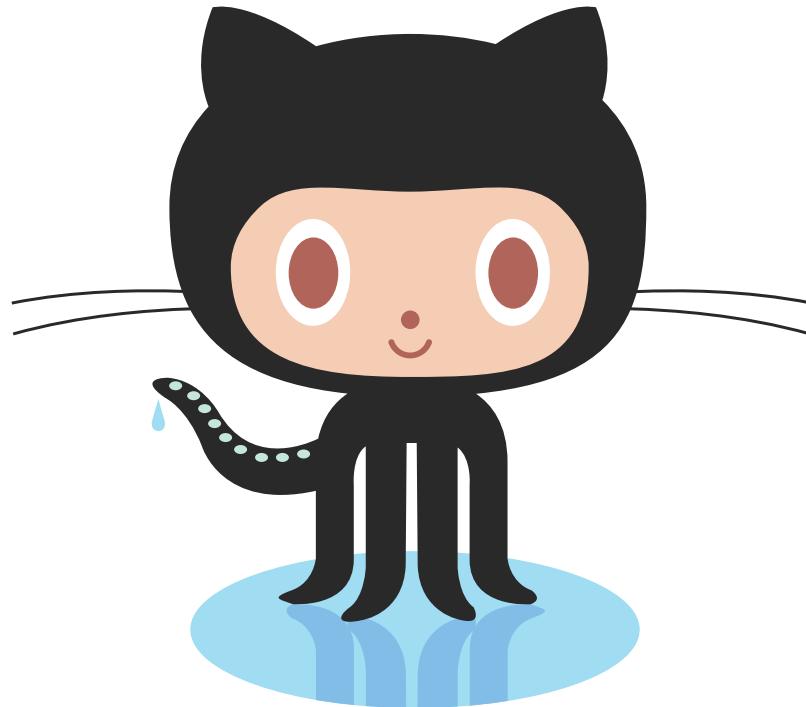
Для этого нужно будет просто добавить новое представление, URL и шаблон для демонстрации существующей модели базы данных.

Важным моментом в этом примере является то, что мы добавили CORS-заголовки и явно указали только домены `localhost:3000` и `localhost:8000` для доступа к нашему API. Правильная установка CORS-заголовков - это то, в чем легко запутаться, когда вы только начинаете создавать API.

Есть много других настроек, которые мы можем и будем делать позже, но в конце дня создание Django API сводится к созданию модели, написанию некоторых URL-маршрутов, а затем добавлению небольшого количества магии, предоставляемой сериализаторами и представлениями Django REST Framework.

В следующей главе мы создадим React front-end и подключим его к нашему Todo API backend.

Код из этой главы можете найти тут:



<https://github.com/jumabekova06/Code-of-DjangoApiBook>

Глава 4: Todo React Front-end

API существует для взаимодействия с другой программой. В этой главе мы будем использовать API Todo из предыдущей главы через фронтенд React , чтобы вы могли увидеть, как все работает вместе на практике.

Я решил использовать React, так как в настоящее время это самая популярная библиотека JavaScript для фронтенда, но описанные здесь техники будут работать и с любым другим популярным фреймворком для фронтенда, включая Vue , Angular или Ember . Они будут работать даже с мобильными приложениями для iOS или Android, десктопными приложениями или вообще с чем угодно. Процесс подключения к внутреннему API удивительно похож.

Если вы застряли или хотите узнать больше о том, что на самом деле происходит с React, ознакомьтесь с официальным учебником , который довольно хорош.

Установка Node

Мы начнем с настройки приложения React в качестве нашего front-end. Сначала откройте новую консоль командной строки, чтобы теперь было две консоли. Это важно. Нам нужно, чтобы наш существующий бэк-энд Todo из предыдущей главы все еще работал на локальном сервере. И мы будем использовать вторую консоль для создания и последующего запуска нашего React front-end на отдельном локальном порту. Таким образом мы локально имитируем то, как будет выглядеть производственная установка выделенного и развернутого фронтенда/бэкенда.

В новой, второй консоли командной строки установите NodeJS , который представляет собой движок выполнения JavaScript.

Он позволяет нам запускать JavaScript вне веб-браузера.

На компьютере Mac мы можем использовать Homebrew , который уже должен быть установлен, если вы следовали инструкциям "Django для начинающих" по настройке вашего локального компьютера.

<https://reactjs.org/>
<https://vuejs.org/>
<https://angular.io/>
<https://emberjs.com/>
<https://reactjs.org/tutorial/tutorial.html>
<https://nodejs.org/en/>
<https://brew.sh/>

командная строка:

```
$ brew install node
```

На компьютере с Windows существует несколько подходов, но популярным является использование nvm-windows . Его репозиторий содержит подробные и актуальные инструкции по установке.

Если вы работаете на Linux, используйте nvm . На момент написания этой статьи команда может быть выполнена с помощью cURL:

командная строка:

```
$ curl -o- https://raw.githubusercontent.com/nvm-sh/nvm/v0.35.3/install.sh | bash
```

или с помощью Wget

командная строка:

```
$ wget -qO- https://raw.githubusercontent.com/nvm-sh/nvm/v0.35.3/install.sh | bash
```

командная строка:

```
$ command -v nvm
```

Закройте текущую консоль командной строки и откройте ее снова для завершения установки.

Установка React

Мы будем использовать превосходный пакет "create-react-app" для быстрого запуска нового проекта React. Это сгенерирует наш шаблон проекта и установит необходимые зависимости одной командой!

Для установки React мы будем полагаться на npm , который является менеджером пакетов JavaScript. Подобно pipenv для Python, npm значительно упрощает управление и установку множества программных пакетов. Последние версии npm также включают npx , который представляет собой улучшенный способ локальной установки пакетов без загрязнения глобального пространства имен. Это рекомендуемый способ установки React, и именно его мы будем использовать здесь!

Полезная ссылка по установке на Linux:

<https://phoenixnap.com/kb/update-node-js-version>

<https://djangoforbeginners.com/initial-setup/>

<https://github.com/coreybutler/nvm-windows>

<https://github.com/creationix/nvm>

<https://github.com/facebookincubator/create-react-app>

<https://www.npmjs.com/>

<https://medium.com/@maybekatz/introducing-npx-an-npm-package-runner-55f7d4bd282b>

Убедитесь, что вы находитесь в правильном каталоге, перейдя на Рабочий стол (если это Mac), а затем в папку todo.

командная строка:

```
$ cd ~/Desktop  
$ cd todo
```

Создайте новое приложение React под названием frontend .

командная строка:

```
$ npx create-react-app frontend
```

Теперь структура ваших каталогов должна выглядеть следующим образом:

Diagram

todo

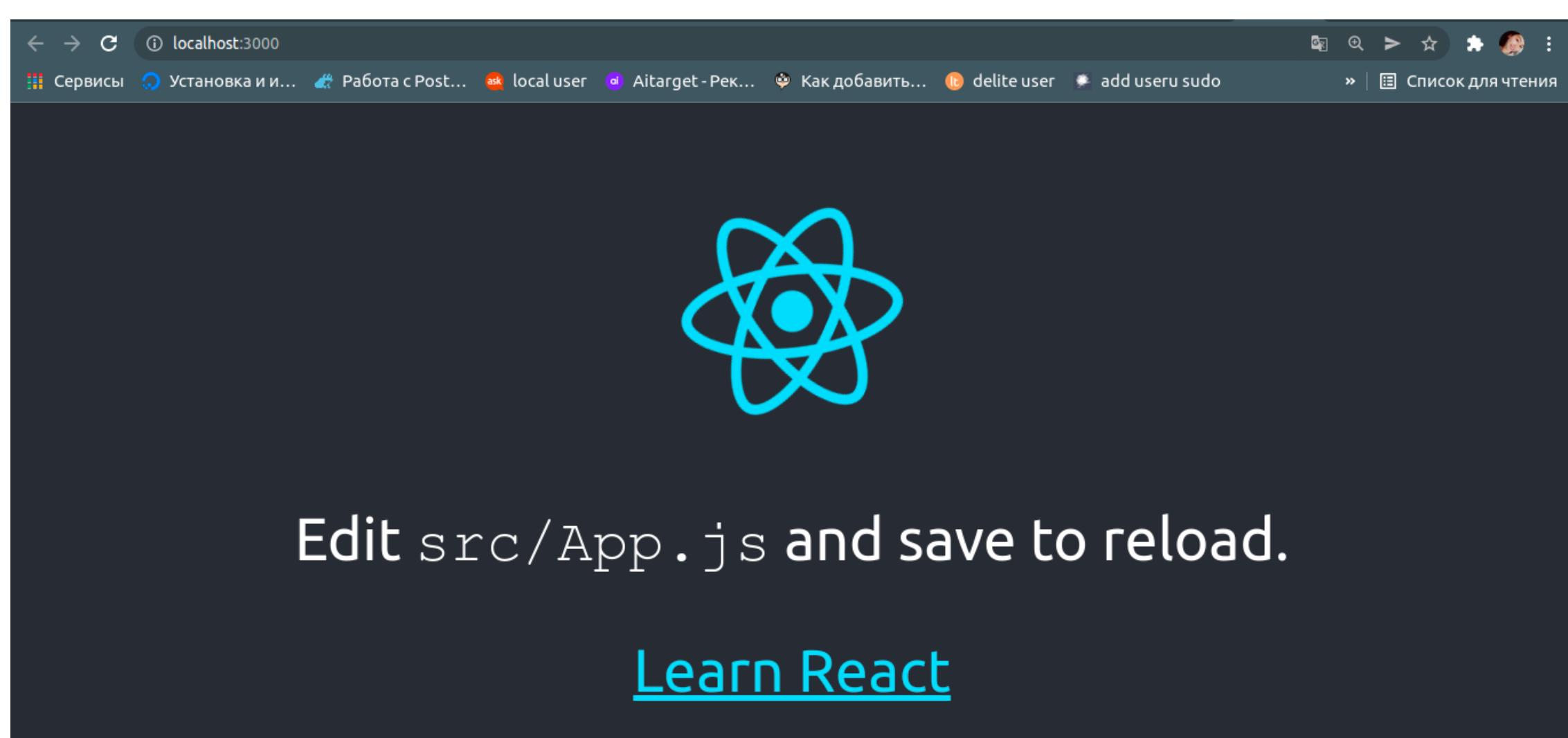
```
|--- frontend  
|     |--- React...  
  
|--- backend  
|     |--- Django...
```

Перейдите в наш фронтенд-проект и запустите приложение React командой npm start .

командная строка:

```
$ cd frontend  
$ npm start
```

Если вы перейдете по адресу http://localhost:3000/, вы увидите домашнюю страницу create-react-app по умолчанию.



Макетные данные

Если вы вернетесь к нашей конечной точке API, вы можете увидеть необработанный JSON в браузере по адресу: <http://127.0.0.1:8000/api/?format=json>

Code

```
[  
  {  
    "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."  
  }  
]
```

Он возвращается при каждом GET-запросе к конечной точке API. В конечном итоге мы будем использовать API напрямую, но хорошим начальным шагом будет сначала сымитировать данные, а затем настроить наш вызов API.

Единственный файл, который нам нужно обновить в нашем приложении React, это `src/App.js`. Давайте начнем с имитации данных API в переменной с именем `list`, которая на самом деле является массивом с тремя значениями.

Code

```
// src/App.js
import React, { Component } from 'react';
const list = [
  {
    "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."
  }
]
```

Далее мы загружаем список в состояние нашего компонента, а затем используем метод массива JavaScript map() для отображения всех элементов.

Я намеренно двигаюсь здесь быстро, поэтому если вы никогда раньше не использовали React, просто скопируйте код, чтобы вы могли увидеть, как "будет" работать подключение React front-end к нашему Django back-end.

Вот полный код, который теперь нужно включить в файл src/App.js.

Code

```
// src/App.js
import React, { Component } from 'react';
const list = [
  {
    "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."
  }
]
class App extends Component {
  constructor(props) {
    super(props);
    this.state = { list };
  }
  render() {
    return (
      <div>
        {this.state.list.map(item => (
          <div key={item.id}>
            <h1>{item.title}</h1>
            <p>{item.body}</p>
          </div>
        )));
      </div>
    );
  }
}
export default App;
```



1st todo

Learn Django properly.

Second item

Learn Python.

Learn HTTP

It's important.

Фиктивные данные

Примечание: Если вы хоть какое-то время работаете с React, то, скорее всего, в какой-то момент вы увидите сообщение об ошибке sh: react-scripts: command not found while running npm start . Не пугайтесь.

Это очень, очень распространенная проблема при разработке JavaScript. Исправление обычно заключается в том, чтобы запустить npm install, а затем снова попробовать npm start. Если это не сработало, удалите папку node_modules и запустите npm install . Это решает проблему в 99% случаев. Добро пожаловать в современную разработку JavaScript :).

Django REST Framework + React

Теперь давайте подключимся к нашему Todo API по-настоящему, а не используя имитацию данных в переменной list. В другой консоли командной строки наш сервер Django запущен, и мы знаем, что конечная точка API, содержащая список всех Todo, находится по адресу <http://127.0.0.1:8000/api>. Поэтому нам нужно отправить к ней GET-запрос. Существует два популярных способа выполнения HTTP-запросов: с помощью встроенного Fetch API или с помощью axios , который имеет несколько дополнительных возможностей. В этом примере мы будем использовать axios. Остановите запущенное приложение React в командной строке с помощью Control+c . Затем установите axios .

командная строка:

\$ npm install axios

Снова запустите приложение React с помощью npm start.

командная строка:

\$ npm start

Then in your text editor at the top of the App.js file import Axios.

https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API
<https://github.com/axios/axios>

Code

```
// src/App.js
import React, { Component } from 'react';
import axios from 'axios'; // new
...
```

Осталось два шага. Во-первых, мы будем использовать axios для нашего GET-запроса. Для этого мы можем сделать специальную функцию getTodos.

Во-вторых, мы хотим убедиться, что этот вызов API будет выполнен в правильное время в жизненном цикле React. HTTP-запросы должны выполняться с помощью componentDidMount , поэтому мы вызовем getTodos там.

Мы также можем удалить список mock, поскольку он больше не нужен. Теперь наш полный файл App.js будет выглядеть следующим образом.

Теперь наш полный файл App.js будет выглядеть следующим образом.

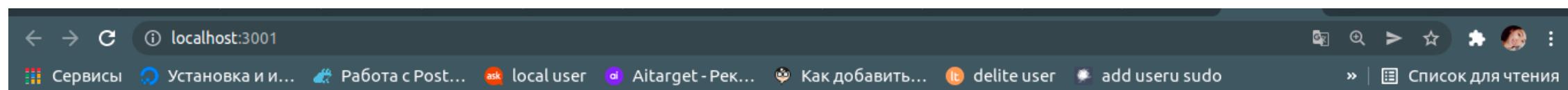
code

```
// src/App.js

import React, { Component } from 'react';
import axios from 'axios'; // new

class App extends Component {
  state = {
    todos: []
  };
  // new
  componentDidMount() {
    this.getTodos();
  }
  // new
  getTodos() {
    axios
      .get('http://127.0.0.1:8000/api/')
      .then(res => {
        this.setState({ todos: res.data });
      })
      .catch(err => {
        console.log(err);
      });
  }
  render() {
    return (
      <div>
        {this.state.todos.map(item => (
          <div key={item.id}>
            <h1>{item.title}</h1>
            <span>{item.body}</span>
          </div>
        ))}
      </div>
    );
  }
}
export default App;
```

Если вы снова посмотрите на <http://localhost:3000/>, страница останется прежней, несмотря на то, что у нас больше нет жестко закодированных данных. Теперь все данные поступают из конечной точки и запроса нашего API. (Смотрите какой порт запустили при npm start иногда может запустить <http://localhost:3001/>)
<https://reactjs.org/docs/state-and-lifecycle.html>



William S. Vincent

Автор нескольких книг по джанго

Antonio Mele

Автор книги Django by example

Данные API

И вот как это делается с помощью React!

Вывод

Теперь мы подключили наш внутренний API Django к внешнему интерфейсу React. Что еще лучше, у нас есть возможность обновить наш фронтенд в будущем или полностью заменить его по мере изменения требований проекта.

Вот почему подход, основанный на использовании API, - это отличный способ "защитить" свой сайт от будущего. Это может потребовать немного больше работы на начальном этапе, но обеспечивает гораздо большую гибкость. В последующих главах мы усовершенствуем наши API, чтобы они поддерживали множество HTTP-методов, таких как POST (добавление новых записей), PUT (обновление существующих записей) и DELETE (удаление записей).

В следующей главе мы начнем создавать надежный API для блогов, поддерживающий полную функциональность CRUD (Create-Read-Update-Delete), а позже добавим к нему аутентификацию пользователей, чтобы пользователи могли входить, выходить и регистрировать аккаунты через наш API.

Код из этой главы можете найти тут:



<https://github.com/jumabekova06/Code-of-DjangoApiBook>

Глава 5: API для блога

Наш следующий проект - API для блога, использующий полный набор возможностей Django REST Framework. Он будет иметь пользователей, права доступа и обеспечивать полную функциональность CRUD (Create-Read-Update-Delete). Мы также изучим наборы представлений, маршрутизаторы и документацию. В этой главе мы создадим базовый раздел API. Как и в случае с API Library и Todo, мы начнем с традиционного Django, а затем добавим Django REST Framework. Основное отличие заключается в том, что наши конечные точки API будут с самого начала поддерживать CRUD, что, как мы увидим, Django REST Framework позволяет сделать довольно легко.

Начальная установка

Наша начальная установка такая же, как и раньше. Перейдите в наш каталог кода и в нем создайте один каталог для этого проекта под названием blogapi . Затем установите Django в новой виртуальной среде, создайте новый проект Django (config) и приложение для записей блога (posts).

командная строка:

```
$ cd ~/Desktop && cd code  
$ mkdir blogapi && cd blogapi  
$ pipenv install django~=3.1.0  
$ pipenv shell  
(blogapi) $ django-admin startproject config .  
(blogapi) $ python manage.py startapp posts
```

Поскольку мы добавили новое приложение, нам нужно сообщить об этом Django. Поэтому не забудьте добавить приложение (posts) посты в наш список INSTALLED_APPS в файле config/settings.py.

Code

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

Теперь запустите migrate в первый раз, чтобы синхронизировать нашу базу данных с настройками Django по умолчанию и новым приложением.

командная строка:

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

Model(Модель)

Наша модель базы данных будет иметь пять полей: author , title , body , created_at , и updated_at . В качестве автора мы можем использовать встроенную в Django модель User при условии, что мы импортируем ее во второй строке сверху.

Code

```
# posts/models.py
from django.db import models
from django.contrib.auth.models import User
class Post(models.Model):
    author = models.ForeignKey(User, on_delete=models.CASCADE)
    title = models.CharField(max_length=50)
    body = models.TextField()
    created_at = models.DateTimeField(auto_now_add=True)
    updated_at = models.DateTimeField(auto_now=True)
    def __str__(self):
        return self.title
```

Обратите внимание, что мы также определяем, каким должно быть __str__ представление модели, что является лучшей практикой Django. Таким образом, мы увидим название в нашей админке Django позже.

Теперь обновим нашу базу данных, сначала создав новый файл миграции, а затем запустив migrate, чтобы синхронизировать базу данных с изменениями нашей модели.

командная строка:

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

```
Migrations for 'posts':
```

```
  posts/migrations/0001_initial.py
```

```
    - Create model Post
```

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

```
Operations to perform:
```

```
  Apply all migrations: admin, auth, contenttypes, posts, sessions
```

```
Running migrations:
```

```
  Applying posts.0001_initial... OK
```

Отлично! Мы хотим просматривать наши данные в отличном встроенным админ-приложении Django, поэтому давайте добавим его в posts/admin.py следующим образом.

Code

```
# posts/admin.py
from django.contrib import admin
from .models import Post
admin.site.register(Post)
```

командная строка:

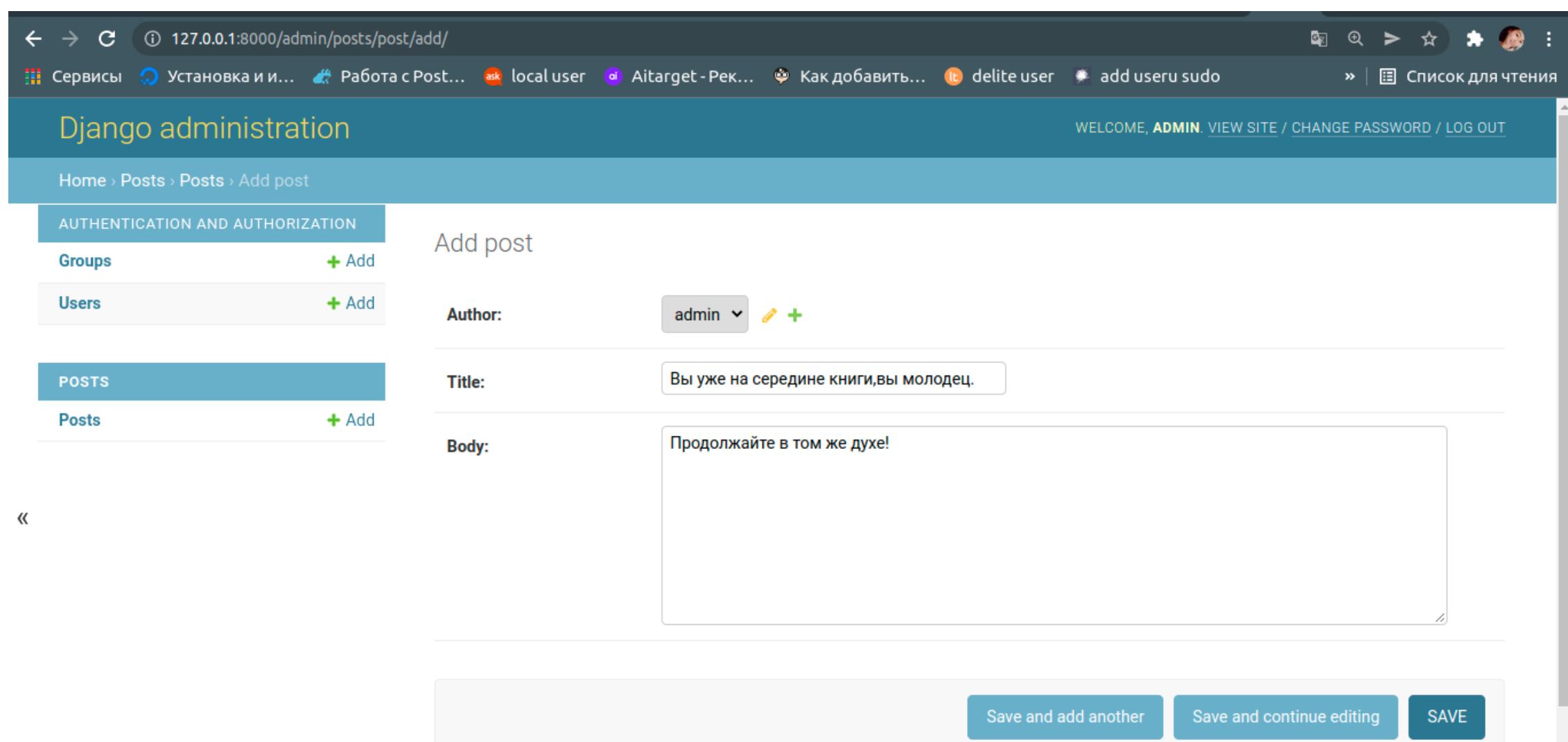
(blogapi) \$ python manage.py createsuperuser

Теперь мы можем запустить локальный веб-сервер.

командная строка:

(blogapi) \$ python manage.py runserver

Перейдите на сайт <http://127.0.0.1:8000/admin/> и войдите в систему с учетными данными суперпользователя. Нажмите на кнопку "+ Add" рядом с Posts и создайте новую запись в блоге. Рядом с "Автор" будет выпадающее меню, в котором будет указана ваша учетная запись суперпользователя (моя называется admin). Убедитесь, что выбран автор. Добавьте заголовок и основное содержание, затем нажмите на кнопку "Сохранить"-SAVE.



The screenshot shows the Django Admin interface at the URL <http://127.0.0.1:8000/admin/posts/post/add/>. The top navigation bar includes links for 'Сервисы', 'Установка и и...', 'Работа с Post...', 'local user', 'Aitarget - Рек...', 'Как добавить...', 'delete user', 'add user sudo', and 'Список для чтения'. The main title is 'Django administration'. The left sidebar has sections for 'AUTHENTICATION AND AUTHORIZATION' (Groups, Users), 'POSTS' (Posts), and a 'Posts' link with a '+ Add' button. The current page is 'Add post'. The form fields are: 'Author:' dropdown set to 'admin', 'Title:' input field containing 'Вы уже на середине книги, вы молодец.', and 'Body:' text area containing 'Продолжайте в том же духе!'. At the bottom are buttons for 'Save and add another', 'Save and continue editing', and 'SAVE'.

Администратор добавляет запись в блог

Вы будете перенаправлены на страницу Posts, где отображаются все существующие записи блога.

The screenshot shows the Django admin interface at the URL `127.0.0.1:8000/admin/posts/post/`. The top navigation bar includes links for 'Сервисы', 'Установка и ин...', 'Работа с Post...', 'local user', 'Aitarget - Рек...', 'Как добавить...', 'delete user', 'add user sudo', 'Список для чтения', 'WELCOME, ADMIN', 'VIEW SITE / CHANGE PASSWORD / LOG OUT', and a user profile icon.

The main title is 'Django administration' and the subtitle is 'Posts'. A success message 'The post "Вы уже на середине книги, вы молодец." was added successfully.' is displayed. The left sidebar has tabs for 'AUTHENTICATION AND AUTHORIZATION', 'Groups' (+ Add), 'Users' (+ Add), and 'POSTS' (selected). The 'POSTS' tab shows a list with one item: 'Posts' (+ Add) and 'Вы уже на середине книги, вы молодец.' (with a checkbox). Below the list is a note '1 post'.

Записи в блоге администратора

Тесты

Давайте напишем базовый тест для нашей модели Post. Мы хотим убедиться, что вошедший в систему пользователь может создать запись в блоге с заголовком и телом.

Code

```
# posts/tests.py
from django.test import TestCase
from django.contrib.auth.models import User
from .models import Post
class BlogTests(TestCase):
    @classmethod
    def setUpTestData(cls):
        # Create a user
        testuser1 = User.objects.create_user(
            username='testuser1', password='abc123')
        testuser1.save()
        # Create a blog post
        test_post = Post.objects.create(
            author=testuser1, title='Blog title', body='Body content...')
        test_post.save()
    def test_blog_content(self):
        post = Post.objects.get(id=1)
        author = f'{post.author}'
        title = f'{post.title}'
        body = f'{post.body}'
        self.assertEqual(author, 'testuser1')
        self.assertEqual(title, 'Blog title')
        self.assertEqual(body, 'Body content...')
```

Чтобы убедиться, что наши тесты работают, выйдите из локального сервера `Control+c`. Затем запустите наши тесты.

командная строка.

(blogapi) \$ `python manage.py test`

Вы должны увидеть результат, подобный следующему, который подтверждает, что все работает так, как ожидалось.

```
(blogapi) → Глава-5 git:(master) ✘ ./manage.py test
Creating test database for alias 'default'...
System check identified no issues (0 silenced).

.
-----
Ran 1 test in 0.204s

OK
Destroying test database for alias 'default'...
(blogapi) → Глава-5 git:(master) ✘
```

Теперь мы закончили с обычной частью Django для 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.

Django REST Framework

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

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

В командной строке используйте `pipenv` для установки Django REST Framework.

командная строка:

```
(blogapi) $ pipenv install djangorestframework~=3.11.0
```

Затем добавьте его в раздел `INSTALLED_APPS` нашего файла `config/settings.py`. Также неплохо явно установить наши разрешения, которые по умолчанию в Django REST Framework настроены на `AllowAny`. Мы обновим их в следующей главе.

Code

```
# config/settings.py
INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
# 3rd-party apps
    'rest_framework', # new
# Local
    'posts.apps.PostsConfig',
]
# new
REST_FRAMEWORK = {
    'DEFAULT_PERMISSION_CLASSES': [
        'rest_framework.permissions.AllowAny',
    ]
}
```

Теперь нам нужно создать наши URL, представления и сериализаторы.

URL-адреса

Давайте начнем с маршрутов URL для фактического расположения конечных точек. Обновите файл urls.py на уровне проекта, добавив импорт include во второй строке и новый маршрут api/v1/ для нашего приложения posts.

Code

```
# config/urls.py
from django.contrib import admin
from django.urls import include, path # new
urlpatterns = [
    path('admin/', admin.site.urls),
    path('api/v1/', include('posts.urls')), # new
]
```

Хорошей практикой является постоянное обновление версий API - v1/, v2/, и т.д., поскольку при внесении больших изменений может возникнуть некоторая задержка, прежде чем различные потребители API также смогут обновить его. Таким образом, вы можете поддерживать версию API v1 в течение определенного периода времени, одновременно выпуская новую, обновленную версию v2 и избегая поломки других приложений, которые полагаются на бэкэнд вашего API.

Обратите внимание, что поскольку единственным нашим приложением на данный момент являются посты, мы можем включить его непосредственно сюда. Если бы у нас было несколько приложений в проекте, возможно, имело бы смысл создать специальное api приложение, а затем включить в него все остальные маршруты API url. Но для таких базовых проектов, как этот, я предпочитаю избегать api приложения, которое используется только для маршрутизации. Мы всегда можем добавить его позже, если потребуется.

Далее создайте файл urls.py нашего приложения posts.

командная строка:

(blogapi) \$ touch posts/urls.py

А затем включите приведенный ниже код.

Code

```
# posts/urls.py
from django.urls import path
from .views import PostList, PostDetail
urlpatterns = [
    path('<int:pk>/', PostDetail.as_view()),
    path('', PostList.as_view()),
]
```

Все маршруты блога будут находиться на api/v1/, поэтому наше представление PostList (которое мы напишем в ближайшее время) с пустой строкой " будет находиться на api/v1/, а представление PostDetail (также будет написано) на api/v1/#, где # представляет собой первичный ключ записи. Например, первая запись блога имеет первичный идентификатор 1, поэтому она будет находиться по маршруту api/v1/1, вторая запись - по api/v1/2, и так далее.

Serializers(Сериализаторы)

Теперь о наших сериализаторах. Создайте новый файл serializers.py в нашем приложении posts.

командная строка:

(blogapi) \$ touch posts/serializers.py

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

Возможность так легко включать/исключать поля в нашем API является замечательной особенностью. Чаще всего базовая модель базы данных содержит гораздо больше полей, чем нужно.

Мощный класс сериализатора Django REST Framework позволяет управлять этим очень просто.

Code

```
# posts/serializers.py
from rest_framework import serializers
from .models import Post
class PostSerializer(serializers.ModelSerializer):
    class Meta:
        fields = ('id', 'author', 'title', 'body', 'created_at',)
        model = Post
```

В верхней части файла мы импортировали класс сериализаторов Django REST Framework и наши собственные модели. Затем мы создали `PostSerializer` и добавили класс `Meta`, где мы указали, какие поля включать и явно задали модель для использования. Существует множество способов настройки сериализатора, но для обычных случаев использования, таких как базовый блог, это все, что нам нужно.

VIEWS (Представления)

Последний шаг - создание представлений. Django REST Framework имеет несколько общих представлений, которые могут быть полезны. Мы уже использовали `ListAPIView` в API Library и `Todos` для создания коллекции конечных точек, доступных только для чтения, по сути, списка всех экземпляров модели. В API `Todos` мы также использовали `RetrieveAPIView` для единственной конечной точки, доступной только для чтения, что аналогично представлению деталей в традиционном Django.

Для нашего Blog API мы хотим получить список всех доступных записей блога в качестве конечной точки для чтения и записи, что означает использование `ListCreateAPIView`, который похож на `ListAPIView`, который мы использовали ранее, но позволяет делать записи. Мы также хотим сделать отдельные записи блога доступными для чтения, обновления или удаления. И, конечно же, для этого существует встроенное общее представление Django REST Framework: `RetrieveUpdateDestroyAPIView`. Именно его мы и будем использовать здесь.

Обновите файл `views.py` следующим образом.

Code

```
# posts/views.py
from rest_framework import generics
from .models import Post
from .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
```

В верхней части файла мы импортируем `generics` из Django REST Framework, а также наши модели и файлы сериализаторов. Затем мы создаем два представления. `PostList` использует общий `ListCreateAPIView`, а `PostDetail` - `RetrieveUpdateDestroyAPIView`.

Удивительно, что для радикального изменения поведения конечной точки API нам достаточно обновить наше общее представление. В этом и заключается преимущество использования полнофункционального фреймворка, такого как Django REST Framework: вся эта функциональность доступна, протестирована и просто работает. Нам, как разработчикам, не нужно изобретать колесо.

Фух. Наш API теперь завершен, и нам действительно не пришлось писать много кода самостоятельно. В следующих главах мы внесем дополнительные улучшения в наш API, но стоит оценить, что он уже выполняет основные функции работы со списками и CRUD, которые мы хотим. Настало время протестировать работу с просматриваемым API Django Rest Framework.

Просматриваемый API

Запустите локальный сервер для взаимодействия с нашим API.

Командная строка:

(blogapi) \$ python manage.py runserver

Затем перейдите на <http://127.0.0.1:8000/api/v1/>, чтобы увидеть конечную точку Post List.

The screenshot shows the Django REST framework's built-in API documentation for a 'Post List' endpoint. At the top, there's a header bar with 'Django REST framework' on the left and 'admin' on the right. Below the header, the title 'Post List' is displayed, along with 'OPTIONS' and 'GET' buttons. A 'Raw data' tab is selected, showing the JSON response for a GET request to '/api/v1/'. The response includes HTTP headers (Allow: GET, POST, HEAD, OPTIONS; Content-Type: application/json; Vary: Accept) and a single JSON object representing a post. The post has fields: id (1), author (1), title ("Вс уже на середине книги, вы молодец."), body ("Продолжайте в том же духе!"), and created_at ("2022-02-06T14:48:59.033307Z"). Below the JSON response, there's a form for creating a new post. The form has three fields: 'Author' (dropdown menu with 'admin'), 'Title' (text input), and 'Body' (text area). There are 'Raw data' and 'HTML form' tabs at the top of the form section, and a 'POST' button at the bottom.

На странице отображается список записей нашего блога - только одна на данный момент - в формате JSON. Обратите внимание, что разрешены методы GET и POST.

Теперь давайте подтвердим, что конечная точка экземпляра нашей модели, которая относится к одному посту, а не к списку всех постов, существует.

Перейдите на сайт <http://127.0.0.1:8000/api/v1/1/>

The screenshot shows the 'Post Detail' page of a Django REST framework application. At the top, there are navigation links 'Post List' and 'Post Detail', and a user 'admin'. Below the header are four buttons: 'DELETE' (red), 'OPTIONS' (blue), 'GET' (blue), and a dropdown menu. A code block displays a successful GET request response:

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

{
    "id": 1,
    "author": 1,
    "title": "Вы уже на середине книги, вы молодец.",
    "body": "Продолжайте в том же духе!",
    "created_at": "2022-02-06T14:48:59.033307Z"
}
```

Below the response is an 'HTML form' section with fields for 'Author' (set to 'admin'), 'Title' (set to 'Вы уже на середине книги, вы молодец.'), and 'Body' (set to 'Продолжайте в том же духе!'). There are 'Raw data' and 'HTML form' tabs at the top of this section. A blue 'PUT' button is located at the bottom right.

Детальная информация об одном объекте

В заголовке видно, что поддерживаются GET, PUT, PATCH и DELETE, но не POST. И на самом деле вы можете использовать HTML-форму ниже для внесения изменений или даже использовать красную кнопку "DELETE" для удаления экземпляра.

Давайте попробуем это сделать. Обновите наш заголовок с дополнительным текстом (отредактированным) в конце. Затем нажмите на кнопку "PUT".

The screenshot shows the 'Post Detail' page again. The 'Title' field now contains the updated value 'Почти закончили! Осталось совсем чуть-чуть...'. The 'PUT' button has been clicked, and a message 'Детальная информация об одном объекте изменена' (Detailed information about one object has been changed) is displayed at the bottom.

Вернитесь к просмотру списка постов, нажав на ссылку для него в верхней части страницы или перейдя непосредственно на <http://127.0.0.1:8000/api/v1/>, и вы сможете увидеть обновленный текст и там.

The screenshot shows the Django REST framework's Post List API endpoint. At the top, there is a header bar with "Django REST framework" and "admin". Below it, the URL "Post List" is shown, along with "OPTIONS" and "GET" buttons. A "Raw data" tab is selected, showing the response body:

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

[
    {
        "id": 1,
        "author": 1,
        "title": "Почти закончили! Осталось совсем чуть-чуть...",
        "body": "Продолжайте в том же духе!",
        "created_at": "2022-02-06T14:48:59.033307Z"
    }
]
```

Below the raw data, there is a form with fields for "Author" (set to "admin"), "Title", and "Body". There are "Raw data" and "HTML form" tabs above the form. A "POST" button is at the bottom right of the form area.

Список постов API отредактирован

Вывод

На данный момент наш Blog API полностью функционален. Однако есть большая проблема: любой может обновить или удалить существующую запись в блоге! Другими словами, у нас нет никаких разрешений. В следующей главе мы узнаем, как применить разрешения для защиты нашего API.

Код из этой главы можете найти тут:



<https://github.com/jumabekova06/Code-of-DjangoApiBook>

Глава 6: Permissions Разрешения

Безопасность является важной частью любого веб-сайта, но она вдвойне важна для веб-интерфейсов API. В настоящее время API нашего блога предоставляет полный доступ любому пользователю. Нет никаких ограничений; любой пользователь может делать все, что угодно, что крайне опасно. Например, анонимный пользователь может создавать, читать, обновлять или удалять любые записи в блоге. Даже ту, которую они не создавали! Очевидно, что мы этого не хотим.

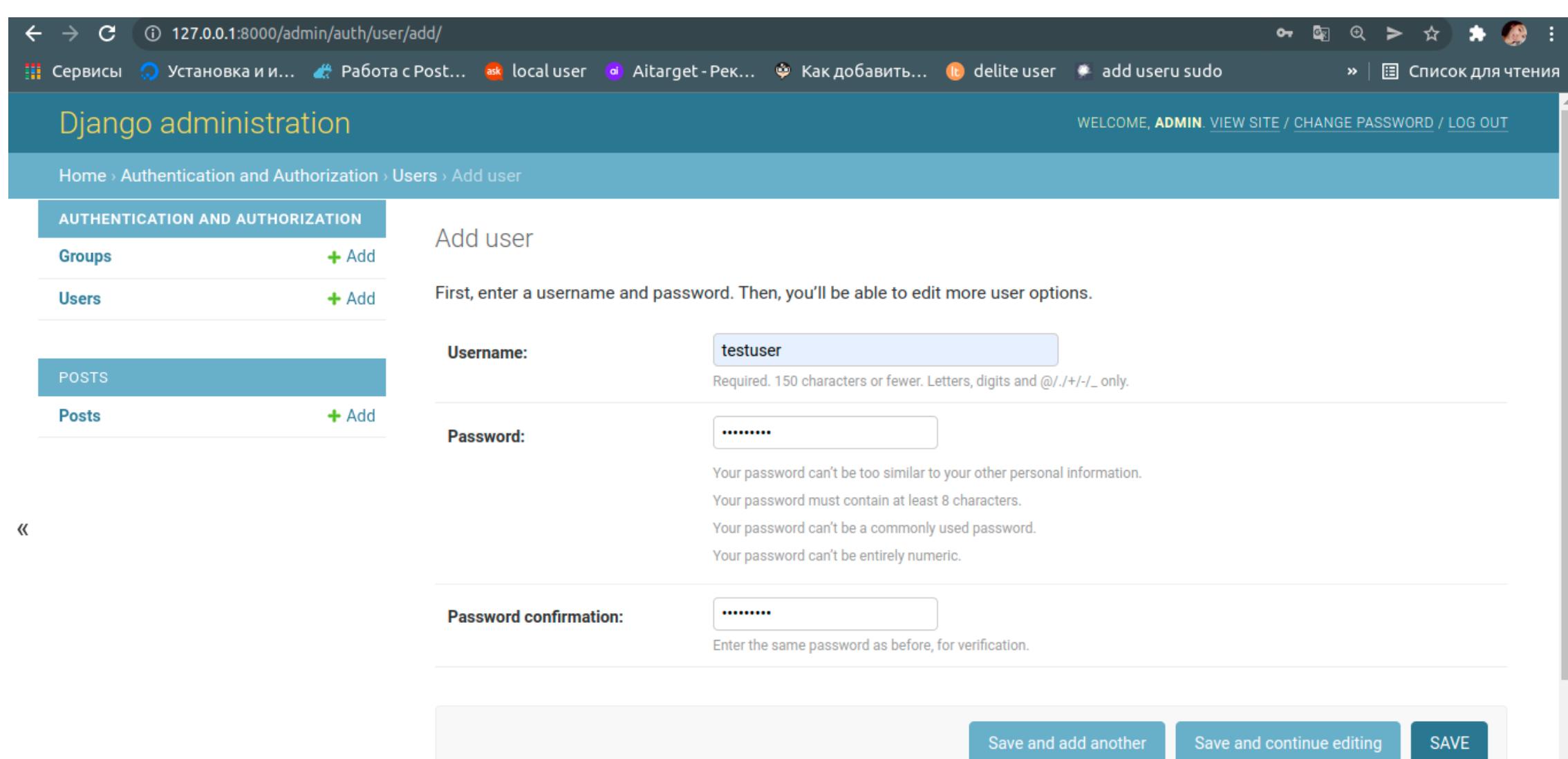
Django REST Framework поставляется с несколькими готовыми настройками разрешений, которые мы можем использовать для защиты нашего API. Их можно применять на уровне проекта, на уровне представления или на уровне отдельной модели.

В этой главе мы добавим нового пользователя и поэкспериментируем с некоторыми настройками разрешений. Затем мы создадим собственное пользовательское разрешение, чтобы только автор записи в блоге имел возможность обновлять или удалять ее.

Создание нового пользователя.

Давайте начнем с создания второго пользователя. Таким образом, мы сможем переключаться между двумя учетными записями пользователей для проверки наших настроек разрешений.

Перейдите в админку по адресу `http://127.0.0.1:8000/admin/`. Затем нажмите на "+ Добавить" рядом с разделом Пользователи . Введите имя пользователя и пароль для нового пользователя и нажмите на кнопку "Сохранить". Здесь я выбрал имя пользователя `testuser`.



127.0.0.1:8000/admin/auth/user/add/

Сервисы Установка и и... Работа с Post... local user Aitarget - Рек... Как добавить... delete user add user sudo Список для чтения

WELCOME, ADMIN. VIEW SITE / CHANGE PASSWORD / LOG OUT

Django administration

Home > Authentication and Authorization > Users > Add user

AUTHENTICATION AND AUTHORIZATION

Groups + Add

Users + Add

POSTS

Posts + Add

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.

Save and add another Save and continue editing SAVE

Страница добавления пользователя администратором

Следующий экран - это страница Admin User Change.(изменения) Я назвал своего пользователя testuser, и здесь я мог бы добавить дополнительную информацию, включенную в модель пользователя по умолчанию, такую как имя, фамилия, адрес электронной почты и т. д. Но ничего из этого не нужно для наших целей: нам просто нужно имя пользователя и пароль для тестирования.

The screenshot shows the Django Admin interface for changing a user. The URL in the browser is `127.0.0.1:8000/admin/auth/user/2/change/`. The page title is "Django administration". The top navigation bar includes links for "Home", "Authentication and Authorization", "Users", and "testuser". On the left, there's a sidebar with sections for "AUTHENTICATION AND AUTHORIZATION" (Groups, Users), "POSTS" (Posts), and "HISTORY". The main content area has a success message: "The user "testuser" was added successfully. You may edit it again below." Below this, the "Change user" form is displayed. It includes fields for "Username" (set to "testuser") and "Password" (showing a hashed value). Under "Personal info", there are fields for "First name", "Last name", and "Email address", all of which are empty. In the "Permissions" section, the "Active" checkbox is checked. There are also checkboxes for "Staff status" and "Superuser status", both of which are unchecked. The "Groups" section shows a list of available groups on the left and chosen groups on the right. The "User permissions" section shows a list of available permissions on the left and chosen permissions on the right. At the bottom, there are "Choose all" and "Remove all" buttons for both groups and permissions.

Изменение пользователя администратора

Прокрутите страницу вниз и нажмите кнопку "Сохранить" ('SAVE'). Это приведет к перенаправлению на главную страницу пользователей <http://127.0.0.1:8000/admin/auth/user/>.

The screenshot shows the Django administration interface at the URL <http://127.0.0.1:8000/admin/auth/user/>. The top navigation bar includes links for 'Сервисы', 'Установка и и...', 'Работа с Post...', 'local user', 'Aitarget - Рек...', and 'Список для чтения'. The main title is 'Django administration' with a welcome message for 'ADMIN'. Below it, the breadcrumb navigation shows 'Home > Authentication and Authorization > Users'. A green success message at the top right says 'The user "testuser" was changed successfully.' On the left, there are two tabs: 'Groups' and 'Users', with 'Users' currently selected. An 'ADD USER' button is located in the top right. The main content area displays a table titled 'Select user to change' with columns: Action, USERNAME, EMAIL ADDRESS, FIRST NAME, LAST NAME, and STAFF STATUS. It lists two users: 'admin' (selected) and 'testuser'. The 'STAFF STATUS' column shows a green checkmark for 'admin' and a red X for 'testuser'. The table has a total count of '2 users'. To the right of the table is a 'FILTER' sidebar with three sections: 'By staff status' (All, Yes, No), 'By superuser status' (All, Yes, No), and 'By active' (All, Yes).

Администратор и Все пользователи

Мы видим, что в списке есть два наших пользователя.

Добавьте вход в систему в просматриваемый API

В дальнейшем, когда мы захотим переключаться между учетными записями пользователей, нам придется заходить в админку Django, выходить из одной учетной записи и входить в другую. Каждый раз. Затем снова переключаться на конечную точку нашего API.

Это настолько частое явление, что в Django REST Framework есть односторонняя настройка для добавления входа и выхода непосредственно в сам просматриваемый API. Сейчас мы это реализуем. В файле `urls.py` на уровне проекта добавьте новый маршрут URL, который включает `rest_framework.urls`.

Несколько смущает тот факт, что на самом деле указанный маршрут может быть каким угодно; главное, чтобы `rest_framework.urls` был куда-то включен. Мы будем использовать маршрут `api-auth`, так как это соответствует официальной документации, но мы можем с тем же успехом использовать любой маршрут, какой захотим, и все будет работать точно так же.

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

Code

```
# config/urls.py
from django.contrib import admin
from django.urls import include, path
urlpatterns = [
    path('admin/', admin.site.urls),
    path('api/v1/', include('posts.urls')),
    path('api-auth/', include('rest_framework.urls')), # new
]
```

Теперь перейдите к нашему просматриваемому API по адресу <http://127.0.0.1:8000/api/v1/>. Есть небольшое изменение: рядом с именем пользователя в правом верхнем углу появилась маленькая стрелка, направленная вниз.

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": "Почти закончили! Осталось совсем чуть-чуть...",  
    "body": "Продолжайте в том же духе!",  
    "created_at": "2022-02-06T14:48:59.033307Z"  
  }  
]
```

Raw data HTML form

Вход в систему API

Поскольку в этот момент мы вошли в систему под учетной записью суперпользователя - для меня это admin - появится это имя. Нажмите на ссылку, и появится выпадающее меню с надписью "Выйти". Нажмите на него.

Верхняя правая ссылка теперь меняется на "Войти". Нажимаем на нее. Мы перенаправляемся на страницу входа в Django REST Framework. Используйте здесь нашу учетную запись тестового пользователя. В конце концов, это перенаправит нас обратно на главную страницу API, где testuser присутствует в правом верхнем углу.

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": "Почти закончили! Осталось совсем чуть-чуть...",  
    "body": "Продолжайте в том же духе!",  
    "created_at": "2022-02-06T14:48:59.033307Z"  
  }  
]
```

Вход в систему API через testuser

В качестве последнего шага выйдите из учетной записи тестового пользователя.

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

[
    {
        "id": 1,
        "author": 1,
        "title": "Почти закончили! Осталось совсем чуть-чуть...",
        "body": "Продолжайте в том же духе!",
        "created_at": "2022-02-06T14:48:59.033307Z"
    }
]
```

Ссылка для входа в систему API

Вы должны снова увидеть ссылку "Войти" в правом верхнем углу.

AllowAny

В настоящее время любой анонимный неавторизованный пользователь может получить доступ к нашей конечной точке PostList. Мы знаем это, потому что даже если мы не вошли в систему, мы можем видеть единственную запись нашего блога. Что еще хуже, любой пользователь имеет полный доступ к созданию, редактированию, обновлению или удалению сообщения!

И на странице подробностей <http://127.0.0.1:8000/api/v1/1/> эта информация также видна, и любой случайный пользователь может обновить или удалить существующую запись в блоге. Не очень хорошо.

The screenshot shows the Django REST framework's browsable API interface at the URL `127.0.0.1:8000/api/v1/`. The title bar includes links for 'Сервисы', 'Установка и и...', 'Работа с Post...', 'local user', 'Aitarget - Рек...', and 'Список для чтен...'. The main header says 'Django REST framework' and has a 'Log in' button. Below it, a 'Post List' section is shown with a 'OPTIONS' button and a 'GET' button. A code block shows the response for a GET request:

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

[
    {
        "id": 1,
        "author": 1,
        "title": "Почти закончили! Осталось совсем чуть-чуть...",
        "body": "Продолжайте в том же духе!",
        "created_at": "2022-02-06T14:48:59.033307Z"
    }
]
```

Причина, по которой мы все еще видим конечную точку Post List, а также конечную точку Detail List, заключается в том, что мы ранее установили разрешения на уровне проекта AllowAny в нашем файле config/settings.py. В качестве краткого напоминания, это выглядело следующим образом:

Code

```
# config/settings.py
REST_FRAMEWORK = {
    'DEFAULT_PERMISSION_CLASSES': [
        'rest_framework.permissions.AllowAny',
    ]
}
```

Разрешения на уровне представления

Сейчас мы хотим ограничить доступ к API для аутентифицированных пользователей. Это можно сделать на уровне проекта, представления или объекта, но поскольку у нас сейчас только два представления, давайте начнем с этого и добавим разрешения к каждому из них.

В вашем файле posts/views.py импортируйте permissions из Django REST Framework, а затем добавьте поле permission_classes к каждому представлению.

Code

```
# posts/views.py
from rest_framework import generics, permissions
from .models import Post
from .serializers import PostSerializer
# new
class PostList(generics.ListCreateAPIView):
    permission_classes = (permissions.IsAuthenticated,)
    queryset = Post.objects.all()
    serializer_class = PostSerializer
# new
class PostDetail(generics.RetrieveUpdateDestroyAPIView):
    permission_classes = (permissions.IsAuthenticated,) # new
    queryset = Post.objects.all()
    serializer_class = PostSerializer
```

Это все, что нам нужно. Обновите просматриваемый API на сайте <http://127.0.0.1:8000/api/v1/>. Посмотрите, что произошло!

Django REST framework

Log in

Post List

Post List

GET

GET /api/v1/

```
HTTP 403 Forbidden
Allow: GET, POST, HEAD, OPTIONS
Content-Type: application/json
Vary: Accept

{
    "detail": "Authentication credentials were not provided."
}
```

Мы больше не видим страницу со списком сообщений. Вместо этого нас встречает недружелюбный код состояния HTTP 403 Forbidden(Запрещено), поскольку мы не вошли в систему. И в просматриваемом API нет форм для редактирования данных, поскольку у нас нет прав.

Если вы используете URL-адрес для Post Detail <http://127.0.0.1:8000/api/v1/1/>, вы увидите аналогичное сообщение и также не увидите доступных форм для редактирования.

Post Detail – Django REST fram

Guest

Django REST framework

Post List / Post Detail

Post Detail

GET

GET /api/v1/1/

```
HTTP 403 Forbidden
Allow: GET, PUT, PATCH, DELETE, HEAD, OPTIONS
Content-Type: application/json
Vary: Accept

{
    "detail": "Authentication credentials were not provided."
}
```

Поэтому на данный момент только вошедшие в систему пользователи могут просматривать наш API. Если вы снова войдете в систему под учетной записью суперпользователя или тестового пользователя, конечные точки API будут доступны.

Но подумайте о том, что будет происходить по мере усложнения API. Вполне вероятно, что в будущем у нас будет гораздо больше представлений и конечных точек.

Добавление специальных permission_classes для каждого представления кажется повторяющимся, если мы хотим установить одинаковые настройки разрешений для всего нашего API.

Не лучше ли изменить наши разрешения один раз, в идеале на уровне проекта, а не делать это для каждого представления?

Разрешения на уровне проекта

На этом этапе вы должны кивнуть головой в знак согласия. Гораздо проще и безопаснее установить строгую политику разрешений на уровне проекта и ослаблять ее по мере необходимости на уровне представления. Именно так мы и поступим.

К счастью, Django REST Framework поставляется с рядом встроенных настроек разрешений на уровне проекта, которые мы можем использовать, включая:

- **AllowAny** - любой пользователь, аутентифицированный или нет, имеет полный доступ
- **IsAuthenticated** - доступ имеют только аутентифицированные, зарегистрированные пользователи
- **IsAdminUser** - доступ имеют только администраторы/суперпользователи
- **IsAuthenticatedOrReadOnly** - неавторизованные пользователи могут просматривать любую страницу, но только авторизованные пользователи имеют права на запись, редактирование или удаление.

Для реализации любой из этих четырех настроек требуется обновить параметр DEFAULT_PERMISSION_CLASSES и обновить наш веб-браузер. Вот и все!

Давайте переключимся на IsAuthenticated, чтобы только аутентифицированные или вошедшие в систему пользователи могли просматривать API. Обновите файл config/settings.py следующим образом:

<http://www.djangoproject.org/api-guide/permissions/#allowany>
<http://www.djangoproject.org/api-guide/permissions/#isauthenticated>
<http://www.djangoproject.org/api-guide/permissions/#isadminuser>
<http://www.djangoproject.org/api-guide/permissions/#isauthenticatedorreadonly>

Code

```
# config/settings.py
REST_FRAMEWORK = {
    'DEFAULT_PERMISSION_CLASSES': [
        'rest_framework.permissions.IsAuthenticated', # new
    ]
}
```

Теперь вернитесь в файл `posts/views.py` и удалите изменения разрешений, которые мы только что сделали.

Code

```
# posts/views.py
from rest_framework import generics
from .models import Post
from .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
```

Если вы обновите страницы API Post List и Detail List, вы все равно увидите тот же код состояния 403.

Сейчас мы потребовали от всех пользователей аутентификации перед доступом к API, но при необходимости мы всегда можем внести дополнительные изменения на уровне представления.

Пользовательские разрешения

Настало время для нашего первого пользовательского разрешения. В качестве краткого обзора того, где мы сейчас находимся: у нас есть два пользователя, `testuser` и суперпользователь. В нашей базе данных есть одна запись блога, которая была создана суперпользователем.

Мы хотим, чтобы только автор конкретного сообщения мог редактировать или удалять его; в противном случае сообщение должно быть доступно только для чтения; запись в блоге должна быть доступна только для чтения. Таким образом, учетная запись суперпользователя должна иметь полный CRUD-доступ к индивидуальному экземпляру блога, а обычный пользователь `testuser` не должен.

Остановите локальный сервер с помощью Control+c и создайте новый файл permissions.py в нашем приложении posts.

Командная строка:

(blogapi) \$ touch posts/permissions.py

Внутри Django REST Framework опирается на класс BasePermission, от которого наследуются все остальные классы разрешений. Это означает, что встроенные параметры разрешений, такие как AllowAny, IsAuthenticated и другие, расширяют его. Вот фактический исходный код, который доступен на Github :

Code

```
class BasePermission(object):
    """
    A base class from which all permission classes should inherit.
    """

    def has_permission(self, request, view):
        """
        Return 'True' if permission is granted, 'False' otherwise.
        """
        return True

    def has_object_permission(self, request, view, obj):
        """
        Return 'True' if permission is granted, 'False' otherwise.
        """
        return True
```

Чтобы создать свое собственное разрешение, мы переопределим метод has_object_permission. В частности, мы хотим разрешить только чтение для всех запросов, но для любых запросов на запись, таких как редактирование или удалить, автор должен быть тем же, что и текущий зарегистрированный пользователь.

Вот как выглядит наш файл posts/permissions.py.

Code

```
# posts/permissions.py
from rest_framework import permissions
class IsAuthorOrReadOnly(permissions.BasePermission):
    def has_object_permission(self, request, view, obj):
        if request.method in permissions.SAFE_METHODS:
            return True
        return obj.author == request.user
```

Мы импортируем разрешения сверху, а затем создаем пользовательский класс IsAuthorOrReadOnly, который расширяет BasePermission . Затем мы переопределяем has_object_permission . Если запрос содержит HTTP-методы, включенные в SAFE_METHODS - кортеж, содержащий GET, OPTIONS и HEAD - то это запрос только для чтения, и разрешение предоставляется.

В противном случае запрос предназначен для какой-то записи, что означает обновление ресурса API с функциями создания, удаления или редактирования. В этом случае мы проверяем, совпадает ли автор рассматриваемого объекта, которым является наша запись в блоге obj.author, с пользователем, делающим запрос request.user .

Вернувшись в файл views.py, мы должны импортировать IsAuthorOrReadOnly и затем мы можем добавить permission_classes для PostDetail.

Code

```
# posts/views.py
from rest_framework import generics
from .models import Post
from .permissions import IsAuthorOrReadOnly # new
from .serializers import PostSerializer

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

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

И мы закончили. Давайте проверим, как это работает. Перейдите на страницу Подробности поста, которая находится по адресу <http://127.0.0.1:8000/api/v1/1/>. Убедитесь, что вы вошли в систему под учетной записью суперпользователя, который является автором сообщения. Имя пользователя должно быть видно в правом верхнем углу страницы.

The screenshot shows the 'Post Detail' page for a post with ID 1. The URL in the browser is <http://127.0.0.1:8000/api/v1/1/>. The top navigation bar includes links for 'Сервисы', 'Установка и и...', 'Работа с Post...', 'local user', 'Aitarget - Рек...', and 'Список для'. The user 'admin' is logged in. The main content area shows the post details with a red 'DELETE' button, a blue 'OPTIONS' button, and a blue 'GET' button. Below these buttons is a 'Raw data' section containing the JSON response from a GET request. The JSON data is as follows:

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

{
    "id": 1,
    "author": 1,
    "title": "Почти закончили! Осталось совсем чуть-чуть...",
    "body": "Продолжайте в том же духе!",
    "created_at": "2022-02-06T14:48:59.033307Z"
}
```

Below the raw data is a form for updating the post. It has three fields: 'Author' (set to 'admin'), 'Title' (set to 'Почти закончили! Осталось совсем чуть-чуть...'), and 'Body' (set to 'Продолжайте в том же духе!'). There are 'Raw data' and 'HTML form' tabs at the top of the form, and a 'PUT' button at the bottom right.

API Detail Superuser

Однако если выйти из системы, а затем войти в нее под учетной записью testuser, страница изменится.

```
GET /api/v1/1/
```

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

{
    "id": 1,
    "author": 1,
    "title": "Почти закончили! Осталось совсем чуть-чуть...",
    "body": "Продолжайте в том же духе!",
    "created_at": "2022-02-06T14:48:59.033307Z"
}
```

API Detail Testuser

Мы можем просматривать эту страницу, поскольку разрешены разрешения только на чтение. Однако мы не можем делать запросы PUT или DELETE из-за нашего пользовательского класса разрешений IsAuthorOrReadOnly.

Обратите внимание, что общие представления будут проверять разрешения на уровне объекта только для представлений, которые получают один экземпляр модели. Если вам требуется фильтрация на уровне объектов для представлений списка - для коллекции экземпляров - вам придется фильтровать, переопределив исходный набор запросов (queryset).

Вывод

Установка надлежащих разрешений - очень важная часть любого API. В качестве общей стратегии, хорошей идеей будет установить строгую политику разрешений на уровне проекта, чтобы только аутентифицированные пользователи могли просматривать API. Затем, по мере необходимости, сделать разрешения на уровне представления или пользовательские разрешения более доступными для определенных конечных точек API.

<http://www.djangoproject.org/api-guide/filtering/#overriding-the-initial-queryset>

Код из этой главы можете найти тут:



<https://github.com/jumabekova06/Code-of-DjangoApiBook>

Глава 7: Аутентификация пользователей

В предыдущей главе мы обновили разрешения наших API, которые также называются авторизацией. В этой главе мы реализуем аутентификацию, которая представляет собой процесс, с помощью которого пользователь может зарегистрировать новый аккаунт, войти в него и выйти из него.

В рамках традиционного, монолитного Django-сайта аутентификация проще и включает в себя шаблон cookie на основе сеанса, который мы рассмотрим ниже. Но с API все немного сложнее.

Помните, что HTTP - это протокол без статических данных, поэтому нет встроенного способа запомнить, аутентифицирован ли пользователь от одного запроса к другому. Каждый раз, когда пользователь запрашивает ограниченный ресурс, он должен подтвердить свою подлинность.

Решение заключается в передаче уникального идентификатора при каждом HTTP-запросе. Смущает то, что не существует общепринятого подхода к форме этого идентификатора, и он может принимать различные формы. Django REST Framework поставляется с четырьмя различными встроенными вариантами **аутентификации** : basic, session, token и default. Кроме того, существует множество сторонних пакетов, которые предлагают дополнительные возможности, такие как JSON Web Tokens (JWTs).

В этой главе мы подробно изучим, как работает аутентификация API, рассмотрим плюсы и минусы каждого подхода, а затем сделаем осознанный выбор для API нашего блога. В конце мы создадим конечные точки API для регистрации, входа и выхода.

Базовая аутентификация

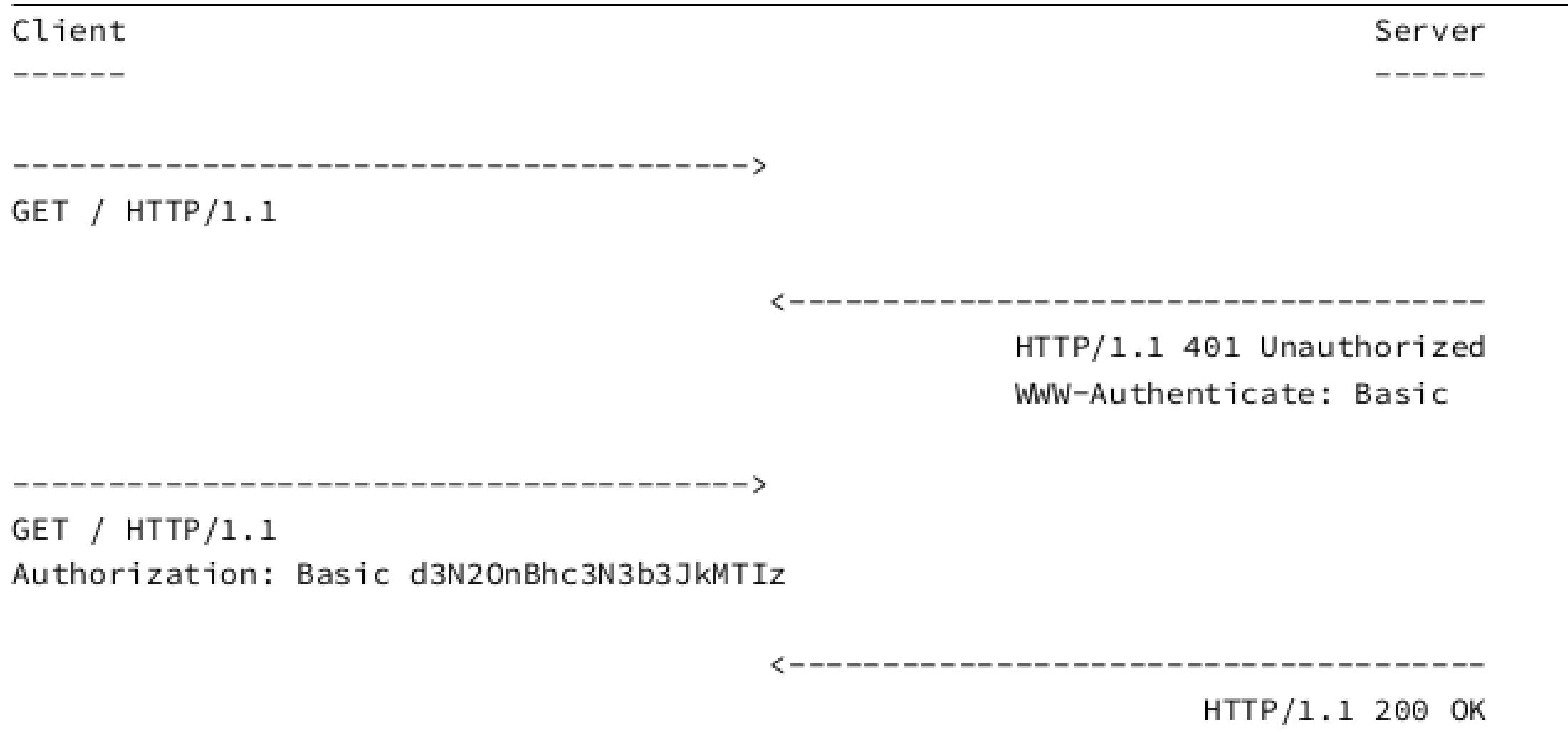
Наиболее распространенная форма аутентификации HTTP известна как "базовая" аутентификация. Когда клиент делает HTTP-запрос, он вынужден отправить подтвержденный мандат аутентификации, прежде чем ему будет предоставлен доступ.

Полный поток запросов/ответов выглядит следующим образом:

- 1. Клиент делает HTTP-запрос**
- 2. Сервер отвечает HTTP-ответом, содержащим код состояния 401 (неавторизованный) и WWW-Authenticate HTTP заголовок с подробной информацией о том, как авторизоваться**
- 3. Клиент отправляет учетные данные обратно через HTTP-заголовок Authorization .**
- 4. Сервер проверяет учетные данные и отвечает либо кодом состояния 200 OK, либо 403 Forbidden.(Запрещено)**

После одобрения клиент отправляет все последующие запросы с данными авторизационного HTTP-заголовка. Мы также можем представить этот обмен следующим образом:

Diagram



Обратите внимание, что отправленные учетные данные авторизации - это незашифрованная **base64** кодированная версия <username>:<password>. Так что в моем случае это wsv:password123, который в кодировке base64 имеет вид d3N2OnBhc3N3b3JkMTIz .

Основным преимуществом этого подхода является его простота. Но есть и несколько серьезных недостатков.

Во-первых, при каждом запросе сервер должен искать и проверять имя пользователя и пароль, что неэффективно. Было бы лучше выполнить поиск один раз, а затем передать некий маркер, который скажет, что этот пользователь одобрен. Во-вторых, учетные данные пользователя передаются открытым текстом - нисколько не зашифрованным - через Интернет. Это невероятно небезопасно. Любой интернет-трафик, который не зашифрован, может быть легко перехвачен и использован повторно. Поэтому базовая аутентификация должна использоваться только через HTTPS , защищенную версию HTTP.

<https://en.wikipedia.org/wiki/HTTPS>

<https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Authorization>

<https://en.wikipedia.org/wiki/Base64>

Аутентификация по сеансам(Session Authentication)

Монолитные веб-сайты, такие как традиционный Django, уже давно используют альтернативную схему аутентификации, которая представляет собой комбинацию сессий и cookies. На высоком уровне клиент аутентифицируется с помощью своих учетных данных (имя пользователя/пароль), а затем получает от сервера идентификатор сессии, который хранится в виде cookie. Этот идентификатор сессии затем передается в заголовке каждого будущего HTTP-запроса.

Когда идентификатор сессии передан, сервер использует его для поиска объекта сессии, содержащего всю доступную информацию для данного пользователя, включая учетные данные.

Этот подход является статичным, поскольку запись должна храниться и поддерживаться как на сервере (объект сессии), так и на клиенте (идентификатор сессии).

Давайте рассмотрим основную последовательность действий:

1. Пользователь вводит свои учетные данные для входа в систему (обычно имя пользователя/пароль).
2. Сервер проверяет правильность учетных данных и генерирует объект сессии, который затем сохраняется в базе данных.
3. Сервер посыпает клиенту идентификатор сессии, а не сам объект сессии, который хранится в браузере в виде cookie.
4. При всех последующих запросах идентификатор сессии включается в HTTP-заголовок, и, если он подтвержден базой данных, запрос выполняется.
5. Как только пользователь выходит из приложения, идентификатор сессии уничтожается как клиентом, так и сервером.
6. Если пользователь снова входит в приложение, генерируется новый идентификатор сессии, который сохраняется на клиенте в виде cookie.

Настройки по умолчанию в Django REST Framework на самом деле представляют собой комбинацию базовой аутентификации и сеансовой аутентификации. В Django используется традиционная система аутентификации на основе сеанса, и идентификатор сеанса передается в HTTP-заголовке при каждом запросе через Basic Authentication.

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 при базовой аутентификации. Он также более эффективен, поскольку серверу не нужно каждый раз проверять учетные данные пользователя, он просто сопоставляет идентификатор сессии с объектом сессии, что является быстрым поиском.

Однако есть несколько недостатков. Во-первых, идентификатор сессии действителен только в том браузере, в котором был выполнен вход; он не будет работать в нескольких доменах. Это очевидная проблема, когда API должен поддерживать несколько внешних интерфейсов, например, веб-сайт и мобильное приложение. Во-вторых, объект сессии необходимо поддерживать в актуальном состоянии, что может оказаться сложной задачей на крупных сайтах с несколькими серверами. Как поддерживать точность объекта сессии на каждом сервере? И в-третьих, cookie-файл отправляется на каждый запрос, даже на те, которые не требуют аутентификации, что неэффективно.

В результате, как правило, не рекомендуется использовать схему аутентификации на основе сеанса для любого API, который будет иметь несколько внешних интерфейсов.

Аутентификация с помощью токенов

Третий основной подход - и тот, который мы реализуем в API нашего блога, - заключается в использовании аутентификации с помощью токенов. Это наиболее популярный подход в последние годы в связи с ростом числа одностраничных приложений.

Аутентификация на основе токенов не имеет статичности: как только клиент отправляет на сервер начальные учетные данные пользователя, генерируется уникальный токен, который затем сохраняется клиентом в виде cookie или в локальном хранилище . Затем этот токен передается в заголовке каждого входящего HTTP-запроса, и сервер использует его для проверки подлинности пользователя. Сам сервер не хранит данные о пользователе, только о том, действителен ли токен или нет.

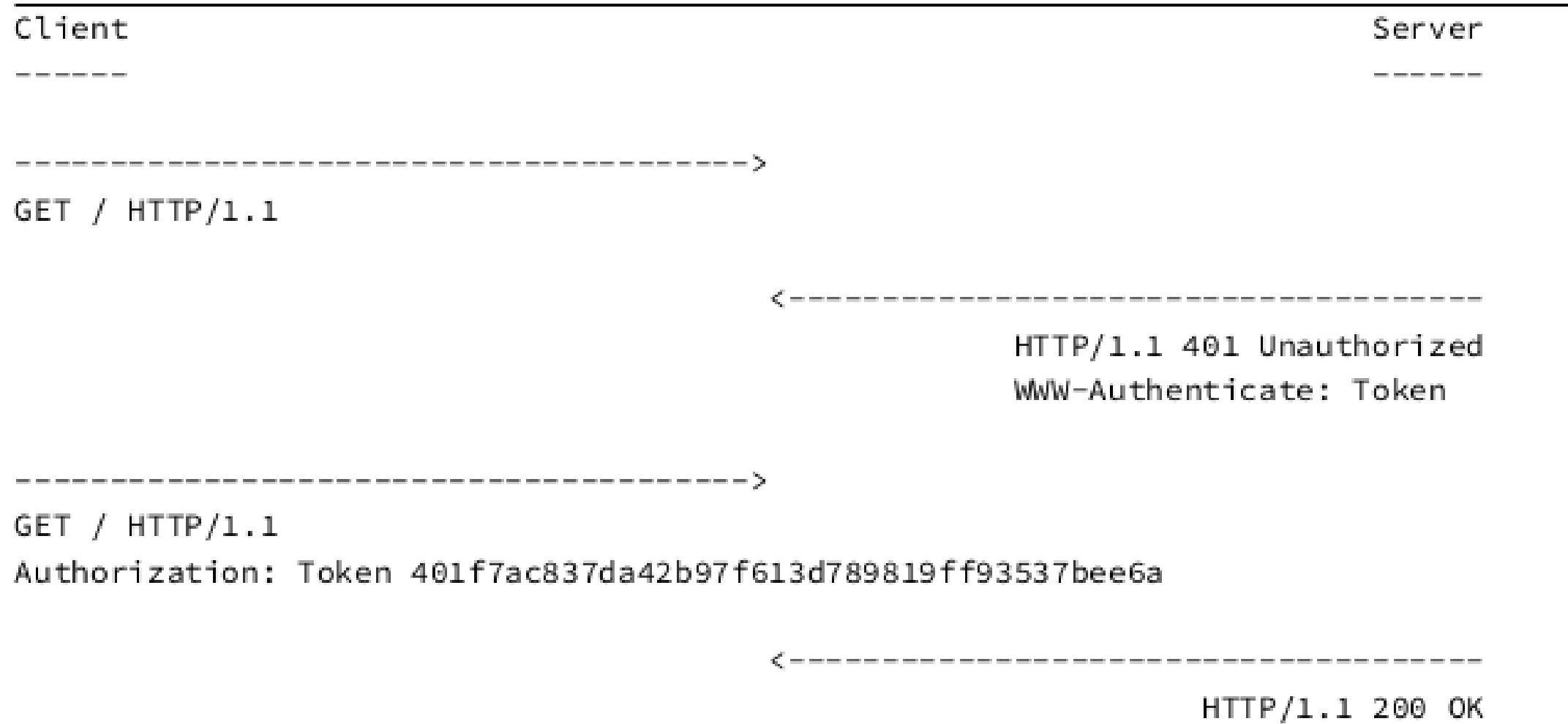
Cookies против localStorage

Cookies используются для чтения информации на стороне сервера. Они имеют меньший размер (4 КБ) и автоматически отправляются с каждым HTTP-запросом. LocalStorage предназначен для хранения информации на стороне клиента. Его размер гораздо больше (5120 КБ), и его содержимое не отправляется по умолчанию с каждым HTTP-запросом.

Токены, хранящиеся как в cookie, так и в localStorage, уязвимы для XSS-атак. В настоящее время лучшей практикой является хранение токенов в cookie с флагами httpOnly и Secure.

Давайте рассмотрим простую версию фактических HTTP-сообщений в этом потоке вызова/ответа. Обратите внимание, что HTTP-заголовок WWW-Authenticate указывает на использование токена, который используется в ответном запросе заголовка Authorization.

Diagram



Этот подход имеет множество преимуществ. Поскольку токены хранятся на клиенте, масштабирование серверов для поддержания актуальных объектов сессии больше не является проблемой. Кроме того, токены могут быть общими для нескольких фронтендов: один и тот же токен может представлять пользователя на сайте и пользователя в мобильном приложении. Один и тот же идентификатор сессии не может быть общим для разных фронтендов, что является серьезным ограничением.

Потенциальным недостатком является то, что токены могут вырасти довольно большими. Токен содержит всю информацию о пользователе, а не только идентификатор, как в случае с идентификатором сессии/объектом сессии. Поскольку токен отправляется при каждом запросе, управление его размером может стать проблемой производительности.

То, как именно реализован токен, также может существенно отличаться. Встроенный TokenAuthentication в Django REST Frameworks намеренно довольно прост.

В результате он не поддерживает установку срока действия токенов, что является улучшением безопасности, которое можно добавить. Он также генерирует только один токен для каждого пользователя, поэтому пользователь на сайте, а затем в мобильном приложении будет использовать один и тот же токен.

Поскольку информация о пользователе хранится локально, это может вызвать проблемы с поддержанием и обновлением двух наборов информации о клиенте.

JSON Web Tokens (JWTs) - это новая, улучшенная версия токенов, которые могут быть добавлены в Django REST Framework с помощью нескольких сторонних пакетов. JWT имеют ряд преимуществ, включая возможность генерировать уникальные клиентские токены и срок действия токенов. Они могут генерироваться либо на сервере, либо с помощью стороннего сервиса, например Auth0 . Кроме того, JWT могут быть зашифрованы, что делает их более безопасными для отправки через незащищенные HTTP-соединения.

В конечном счете, наиболее безопасным для большинства веб-интерфейсов API является использование схемы аутентификации на основе токенов. JWT являются хорошим, современным дополнением, хотя они требуют дополнительной настройки. В результате в этой книге мы будем использовать встроенный TokenAuthentication .

Аутентификация по умолчанию(Default Authentication)

Первым шагом будет настройка наших новых параметров аутентификации. Django REST Framework поставляется с рядом настроек, которые заданы неявно. Например, DEFAULT_PERMISSION_CLASSES был установлен на AllowAny до того, как мы обновили его на IsAuthenticated . DEFAULT_AUTHENTICATION_CLASSES установлены по умолчанию, поэтому давайте явно добавим SessionAuthentication и BasicAuthentication в наш файл config/settings.py.

Code

```
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. Сессии используются для работы просматриваемого API и возможности входа и выхода из него. BasicAuthentication используется для передачи идентификатора сессии в HTTP-заголовках для самого API.

Если вы перейдете к просматриваемому API 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.

Реализация аутентификации с помощью токенов

Теперь нам нужно обновить нашу систему аутентификации для использования токенов. Первым шагом будет обновление нашего параметра `DEFAULT_AUTHENTICATION_CLASSES` на использование

`TokenAuthentication` следующим образом:

Code

```
# config/settings.py
REST_FRAMEWORK = {
    'DEFAULT_PERMISSION_CLASSES': [
        'rest_framework.permissions.IsAuthenticated'],
    'DEFAULT_AUTHENTICATION_CLASSES': [
        'rest_framework.authentication.SessionAuthentication',
        'rest_framework.authentication.TokenAuthentication', # new
    ],
}
```

Мы сохраняем `SessionAuthentication`, поскольку она все еще нужна нам для нашего Browsable API, но теперь используем токены для передачи учетных данных аутентификации туда и обратно в HTTP-заголовках.

Нам также нужно добавить приложение auth token, которое генерирует токены на сервере. Оно поставляется в комплекте с Django REST Framework, но должно быть добавлено в наш параметр INSTALLED_APPS:

Code

```
# config/settings.py
INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    # 3rd-party apps
    'rest_framework',
    'rest_framework.authtoken', # new
    # Local
    'posts',
]
```

Поскольку мы внесли изменения в наш INSTALLED_APPS, нам необходимо синхронизировать нашу базу данных. Остановите сервер с помощью Control+c . Затем выполните следующую команду.

командная строка :

```
(blogapi) ~ Глава-7 git:(master) ✘ python3 manage.py migrate
Operations to perform:
  Apply all migrations: admin, auth, authtoken, contenttypes, posts, sessions
Running migrations:
  Applying auth_token.0001_initial... OK
  Applying auth_token.0002_auto_20160226_1747... OK
```

Теперь снова запустите сервер.

командная строка:

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

Если вы перейдете в админку Django по адресу <http://127.0.0.1:8000/admin/>, вы увидите, что теперь в верхней части есть раздел Tokens. Убедитесь, что вы вошли в систему под учетной записью суперпользователя, чтобы иметь к нему доступ.

The screenshot shows the Django administration interface at the URL 127.0.0.1:8000/admin/. The top navigation bar includes links for Сервисы, Установка и..., Работа с Post..., local user, Aitarget - Рек..., Как добавить..., delete user, add user sudo, and WELCOME, ADMIN. The main content area is titled "Site administration". It features three main sections: "AUTH TOKEN" (Tokens, Add, Change), "AUTHENTICATION AND AUTHORIZATION" (Groups, Add, Change; Users, Add, Change), and "POSTS" (Posts, Add, Change). A sidebar on the right lists "Recent actions" and "My actions" with "None available".

Нажмите на ссылку для токенов. В настоящее время нет токенов, что может удивить.

The screenshot shows the "Tokens" list page under the "AUTH TOKEN" section. The URL is 127.0.0.1:8000/admin/authtoken/token/. The top navigation bar is identical to the previous screenshot. The main content area shows a message "Select Token to change" and "0 Tokens". A "ADD TOKEN" button is visible. The title of the page is "Страница токенов администратора".

В конце концов, у нас есть существующие пользователи. Однако токены генерируются только после того, как есть вызов API для входа пользователя в систему. Мы еще не сделали этого, поэтому токенов нет. Мы сделаем это в ближайшее время!

Конечные точки(Endpoints)

Нам также необходимо создать конечные точки, чтобы пользователи могли входить и выходить из системы. Для этого мы можем создать специальное приложение для пользователей, а затем добавить наши собственные url, представления и сериализаторы. Однако аутентификация пользователей - это та область, где мы действительно не хотим допустить ошибку. И поскольку почти все API требуют этой функциональности, логично, что есть несколько отличных и проверенных пакетов сторонних разработчиков, которые мы можем использовать вместо нее.

В частности, мы будем использовать **dj-rest-auth** в сочетании с **django-allauth** для упрощения работы. Не расстраивайтесь из-за использования сторонних пакетов. Они существуют не просто так, и даже лучшие профессионалы Django постоянно полагаются на них. Нет смысла изобретать колесо, если в этом нет необходимости!

dj-rest-auth

Сначала мы добавим конечные точки API для входа, выхода иброса пароля. Они поставляются в комплекте с популярным пакетом dj-rest-auth. Остановите сервер с помощью Control+c, а затем установите его.

командная строка:

```
(blogapi) $ pipenv install dj-rest-auth==1.1.0
```

Добавьте новое приложение в конфигурацию INSTALLED_APPS в нашем файле config/settings.py.

Code

```
# config/settings.py
INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    # 3rd-party apps
    'rest_framework',
    'rest_framework.authtoken',
    'dj_rest_auth', # new
    # Local
    'posts',
```

1

<https://github.com/jazzband/dj-rest-auth>

<https://github.com/pennersr/django-allauth>

Обновите наш файл config/urls.py с помощью пакета dj_rest_auth. Мы устанавливаем URL маршрутов на api/v1/dj-rest-auth . Обязательно обратите внимание, что URL-адреса должны содержать тире, а не знак подчеркивания _ , что является простейшей ошибкой.

Code

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

И все готово! Если вы когда-нибудь пытались реализовать собственные конечные точки аутентификации пользователей, то просто удивительно, сколько времени и головной боли сэкономил нам dj-rest-auth. Теперь мы можем запустить сервер, чтобы посмотреть, что предоставил dj-rest-auth.

командная строка:

(blogapi) \$ python manage.py runserver

У нас есть рабочий пункт входа в систему по адресу

<http://127.0.0.1:8000/api/v1/dj-rest-auth/login/> .

The screenshot shows a browser window displaying the Django REST framework's API documentation for the login endpoint. The URL in the address bar is `127.0.0.1:8000/api/v1/dj-rest-auth/login/`. The page title is "Login - Django REST framework". The main content area is titled "Login" and contains the following text:

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." }
```

Below this, there is a form for logging in with fields for "Username", "Email", and "Password", and a "POST" button. There are also tabs for "Raw data" and "HTML form".

Конечная точка входа в систему API

И конечная точка выхода из системы на
`http://127.0.0.1:8000/api/v1/dj-rest-auth/logout/ .`

The screenshot shows a browser window displaying the Django REST framework's API documentation for the 'Logout' endpoint. The URL in the address bar is `127.0.0.1:8000/api/v1/dj-rest-auth/logout/`. The page title is 'Logout' under the 'Django REST framework' header. On the right, there are buttons for 'OPTIONS' and 'GET'. Below the title, it says 'Calls Django logout method and delete the Token object assigned to the current User object.' It also notes 'Accepts/Returns nothing.' 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 with 'Media type:' set to 'application/json' and a large empty 'Content:' field. A 'POST' button is at the bottom right.

Конечная точка выхода из системы API

Существуют также конечные точки для сброса пароля, которые расположены по адресу:

`http://127.0.0.1:8000/api/v1/dj-rest-auth/password/reset`

The screenshot shows a browser window titled "Password Reset - Django REST". The URL is "127.0.0.1:8000/api/v1/dj-rest-auth/password/reset/". The page is titled "Post List / Password Reset" and displays the "Password Reset" endpoint. It says "Calls Django Auth PasswordResetForm save method." and "Accepts the following POST parameters: email". It also says "Returns the success/fail message." Below this, there is a "GET /api/v1/dj-rest-auth/password/reset/" section which shows an error response: "HTTP 405 Method Not Allowed" with "Allow: POST, OPTIONS", "Content-Type: application/json", and "Vary: Accept". The response body is: { "detail": "Method \"GET\" not allowed." }. At the bottom, there is a form with a "Raw data" tab selected and an "HTML form" tab. The form has a "Email" input field and a "POST" button.

Сброс пароля API

И для сброса пароля подтверждения:

<http://127.0.0.1:8000/api/v1/dj-rest-auth/password/reset/confirm>

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

Подтверждение сброса пароля API

Регистрация пользователей

Следующей будет конечная точка регистрации пользователя, или sign up. Традиционный Django не поставляется со встроенными представлениями или URL для регистрации пользователей, как и Django REST Framework. Это означает, что нам придется писать свой собственный код с нуля; несколько рискованный подход, учитывая серьезность и последствия для безопасности, которые может повлечь за собой неправильная работа. Популярным подходом является использование стороннего пакета django-allauth, который поставляется с регистрацией пользователей, а также рядом дополнительных функций к системе Django auth, таких как социальная аутентификация через Facebook, Google, Twitter и т.д. Если мы добавим dj_rest_auth.registration из пакета dj-rest-auth, то у нас также появятся конечные точки регистрации пользователей!

Остановите локальный сервер с помощью Control+c и установите django-allauth .

командная строка:

(blogapi) \$ pipenv install django-allauth~=0.42.0

Затем обновите наш параметр INSTALLED_APPS. Мы должны добавить несколько новых настроек:

- **django.contrib.sites**
- **allauth**
- **allauth.account**
- **allauth.socialaccount**
- **dj_rest_auth.registration**

Не забудьте также включить EMAIL_BACKEND и SITE_ID . Технически не имеет значения, где в файле config/settings.py они размещены, но обычно принято добавлять дополнительные конфигурации внизу.

Code

```
# config/settings.py
INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    'django.contrib.sites', # new
    # 3rd-party apps
    'rest_framework',
    'rest_framework.authtoken',
    'allauth', # new
    'allauth.account', # new
    'allauth.socialaccount', # new
    'dj_rest_auth',
    'dj_rest_auth.registration', # new
    # Local
    'posts',
]
EMAIL_BACKEND = 'django.core.mail.backends.console.EmailBackend' # new
SITE_ID = 1 # new
```

Конфигурация email back-end необходима, поскольку по умолчанию при регистрации нового пользователя будет отправляться письмо с просьбой подтвердить учетную запись. Вместо того, чтобы настраивать почтовый сервер, мы будем выводить письма в консоль с помощью параметра `console.EmailBackend`.

`SITE_ID` является частью встроенного фреймворка Django "sites", который представляет собой способ размещения нескольких сайтов из одного проекта Django. Очевидно, что здесь мы работаем только с одним сайтом, но django-allauth использует фреймворк sites, поэтому мы должны указать параметр по умолчанию.

Хорошо. Мы добавили новые приложения, поэтому пришло время обновить базу данных.

командная строка:

(blogapi) \$ python manage.py migrate

Затем добавьте новый URL-маршрут для регистрации.

Code

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

командная строка:

(blogapi) \$ python manage.py runserver

Теперь существует конечная точка регистрации пользователя по адресу <http://127.0.0.1:8000/api/v1/dj-rest-auth/registration/>.

Django REST framework

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

Email

Password1

Password2

POST

API Register

Токены

Чтобы убедиться, что все работает, создайте учетную запись третьего пользователя через конечную точку API с возможностью просмотра. Я назвал своего пользователя testuser2 . Затем нажмите на кнопку "POST".

The screenshot shows the Django REST framework's browsable API interface. The URL is `127.0.0.1:8000/api/v1/dj-rest-auth/registration/`. The page title is "Register". There is a "Post List / Register" link and an "OPTIONS" button. A "GET" method is listed with the URL `/api/v1/dj-rest-auth/registration/`. The response is an **HTTP 405 Method Not Allowed** error message. It includes headers: `Allow: POST, OPTIONS`, `Content-Type: application/json`, and `Vary: Accept`. The JSON response is:

```
{ "detail": "Method \"GET\" not allowed." }
```

. Below this, there is a form for a POST request with fields: Username (testuser2), Email (testuser2@email.com), Password1 (testpass123), and Password2 (testpass123). There are "Raw data" and "HTML form" tabs, and a "POST" button.

API Register New User

На следующем экране показан HTTP-ответ от сервера. Наша регистрация пользователя POST прошла успешно, отсюда и код состояния HTTP 201 Created в верхней части. Возвращаемое значение - это токен аутентификации для этого нового пользователя.

Post List / Register

Register

OPTIONS

POST /api/v1/dj-rest-auth/registration/

HTTP 201 Created
Allow: POST, OPTIONS
Content-Type: application/json
Vary: Accept

{
 "key": "b8f2d999ffcb48a3f78e60f408ba08318e036357"
}

Raw data HTML form

Key: b8f2d999ffcb48a3f78e60f408ba08318e036357

POST

API Auth Key

Если вы посмотрите на консоль командной строки, электронное письмо было автоматически сгенерировано django-allauth. Этот текст по умолчанию можно обновить и добавить почтовый SMTP-сервер с помощью дополнительной конфигурации, которая рассматривается в книге "Django для начинающих" .

Командная строка:

```
Content-Type: text/plain; charset="utf-8"
MIME-Version: 1.0
Content-Transfer-Encoding: 7bit
Subject: [example.com] Please Confirm Your E-mail Address
From: webmaster@localhost
To: aika2000tls@mail.ru
Date: Mon, 07 Feb 2022 19:57:40 -0000
Message-ID: <164426386042.235843.12095298803433072043@karlygach>

Hello from example.com!

You're receiving this e-mail because user testuser2 has given yours as an e-mail address to connect their account.

To confirm this is correct, go to http://127.0.0.1:8000/api/v1/dj-rest-auth/registration/account-confirm-email/Mg:lnHA8q:64MnVSB9s9-lMHS3xxp46xqQ9hD2SbSevn89Kk8_yBk/

Thank you from example.com!
example.com
[07/Feb/2022 19:57:40] "POST /api/v1/dj-rest-auth/registration/ HTTP/1.1" 201 7840
```

Перейдите в админку Django в веб-браузере по адресу <http://127.0.0.1:8000/admin/>. Для этого вам нужно будет использовать учетную запись суперпользователя. Затем нажмите на ссылку **Tokens** в верхней части страницы. Вы будете перенаправлены на страницу Tokens.

The screenshot shows the Django administration interface at the URL http://127.0.0.1:8000/admin/auth_token/token/. The title bar says "Select Token to change | Django". The main content area is titled "Django administration" and "WELCOME, wsv. VIEW SITE / CHANGE PASSWORD / LOG OUT". Below this, the breadcrumb navigation shows "Home > Auth Token > Tokens". The page title is "Select Token to change". On the right, there is a button labeled "ADD TOKEN +". A table lists one token entry:

Action	KEY	USER	CREATED
<input type="checkbox"/>	b8f2d999ffcb48a3f78e60f408ba08318e036357	testuser2	July 29, 2020, 8:54 p.m.

Below the table, it says "1 Token".

Admin Tokens

Один токен был сгенерирован Django REST Framework для пользователя testuser2. По мере создания дополнительных пользователей через API, их токены также будут появляться здесь.

Возникает логичный вопрос, почему нет токенов для учетной записи суперпользователя или testuser? Ответ заключается в том, что мы создали эти учетные записи до того, как была добавлена аутентификация с помощью токенов. Но не беспокойтесь, как только мы войдем в систему с помощью API, токен будет автоматически добавлен и доступен.

Идем дальше, давайте войдем в систему с новой учетной записью testuser2. В веб-браузере перейдите по адресу <http://127.0.0.1:8000/api/v1/dj-rest-auth/login/>. Введите информацию для нашей учетной записи testuser2. Нажмите на кнопку "POST".

The screenshot shows a browser window with the title "Login - Django REST framework". The URL in the address bar is "127.0.0.1:8000/api/v1/dj-rest-auth/login/". The page header includes "Guest" and "WSV" dropdown menus. The main content area is titled "Login" and describes the endpoint as checking credentials and returning a REST Token if valid. It notes that it calls the Django Auth login method to register User ID in the session framework. Below this, instructions state to accept POST parameters: username, password, and return the REST Framework Token Object's key. A "GET /api/v1/dj-rest-auth/login/" button is shown, leading to a response message: "HTTP 405 Method Not Allowed" with allowed methods POST, OPTIONS, Content-Type: application/json, and Vary: Accept. The response body contains a JSON object with a "detail": "Method \"GET\" not allowed." message. At the bottom, there are "Raw data" and "HTML form" tabs, and a "POST" button.

API Log In testuser2

Произошли две вещи. В правом верхнем углу отображается наша учетная запись пользователя testuser2, что подтверждает, что мы вошли в систему. Также сервер отправил ответ HTTP с токеном.

The screenshot shows a browser window for a Django REST framework application. The URL is 127.0.0.1:8000/api/v1/dj-rest-auth/login/. The page title is "Login - Django REST framework". The user is logged in as "testuser2". The main content is a "Login" section with a "OPTIONS" button. Below it, a note says "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". It also specifies "Accept the following POST parameters: username, password" and "Return the REST Framework Token Object's key". A "POST /api/v1/dj-rest-auth/login/" button is shown. The response details show "HTTP 200 OK" with headers: Allow: POST, OPTIONS; Content-Type: application/json; Vary: Accept. The response body is a JSON object with a single key "key": "b8f2d999ffcb48a3f78e60f408ba08318e036357". At the bottom, there are "Raw data" and "HTML form" buttons, and a "Key" input field containing the token value.

API Log In Token

В нашем front-end фреймворке нам нужно будет перехватить и сохранить этот токен. Традиционно это происходит на клиенте, либо в localStorage, либо в виде cookie, а затем все последующие запросы включают токен в заголовок как способ аутентификации пользователя. Обратите внимание, что существуют дополнительные проблемы безопасности в этой области, поэтому вам следует позаботиться о внедрении лучших практик выбранного вами front-end фреймворка.

<https://developer.mozilla.org/en-US/docs/Web/API/Window/localStorage>

Вывод

Аутентификация пользователей - одна из самых сложных областей, которую трудно понять при работе с веб-интерфейсами API. Не имея преимущества монолитной структуры, мы, разработчики, должны глубоко понимать и соответствующим образом настраивать наши циклы HTTP запросов/ответов. Django REST Framework поставляется с большим количеством встроенной поддержки для этого процесса, включая встроенную TokenAuthentication. Однако разработчики должны сами настраивать дополнительные области, такие как регистрация пользователей и выделенные url/views. В результате, популярным, мощным и безопасным подходом является использование сторонних пакетов dj-rest-auth и django-allauth для минимизации объема кода, который нам приходится писать с нуля.

Код из этой главы можете найти тут:



<https://github.com/jumabekova06/Code-of-DjangoApiBook>

Глава 8: Наборы представлений и маршрутизаторы (Viewsets and Routers)

Наборы представлений⁸⁹ и маршрутизаторы⁹⁰ - это инструменты в Django REST Framework, которые могут ускорить разработку API. Они представляют собой дополнительный уровень абстракции поверх представлений и URL. Основное преимущество заключается в том, что один набор представлений может заменить несколько связанных представлений. А маршрутизатор может автоматически генерировать URL-адреса для разработчика. В больших проектах с большим количеством конечных точек это означает, что разработчику приходится писать меньше кода. Кроме того, опытному разработчику легче понять и рассуждать о небольшом количестве комбинаций наборов представлений и маршрутизаторов, чем о длинном списке отдельных представлений и URL.

В этой главе мы добавим две новые конечные точки API в наш существующий проект и посмотрим, как переход от представлений и URL к наборам представлений и маршрутизаторам позволяет достичь той же функциональности с гораздо меньшим количеством кода.

Конечные точки пользователя

В настоящее время в нашем проекте есть следующие конечные точки API. Все они имеют префикс `api/v1/`, который для краткости не показан:

http://www.djangoproject.com/en/1.7/_/api-guide/viewsets/

http://www.djangoproject.com/en/1.7/_/api-guide/routers/

Diagram

Endpoint	HTTP Verb
/	GET
/:pk/	GET
/rest-auth/registration	POST
/rest-auth/login	POST
/rest-auth/logout	GET
/rest-auth/password/reset	POST
/rest-auth/password/reset/confirm	POST

Первые две конечные точки были созданы нами, а пять остальных предоставил dj-rest-auth. Давайте теперь добавим две дополнительные конечные точки для списка всех пользователей и отдельных пользователей. Это обычная функция для многих API, и это сделает более понятным, почему рефакторинг наших представлений и URL в наборы представлений и маршрутизаторы может иметь смысл.

Традиционный Django имеет встроенный класс модели User, который мы уже использовали в предыдущей главе для аутентификации. Поэтому нам не нужно создавать новую модель базы данных. Вместо этого нам просто нужно подключить новые конечные точки. Этот процесс всегда включает в себя следующие три шага:

- **новый класс сериализатора для модели**
- **новые представления для каждой конечной точки**
- **новые маршруты URL для каждой конечной точки**

Начнем с нашего сериализатора. Нам нужно импортировать модель User, а затем создать класс UserSerializer, который использует ее. Затем добавьте его в наш существующий файл posts/serializers.py.

Code

```
# posts/serializers.py
from django.contrib.auth import get_user_model # new
from rest_framework import serializers
from .models import Post
class PostSerializer(serializers.ModelSerializer):
    class Meta:
        model = Post
        fields = ('id', 'author', 'title', 'body', 'created_at',)
class UserSerializer(serializers.ModelSerializer): # new
    class Meta:
        model = get_user_model()
        fields = ('id', 'username',)
```

Стоит отметить, что хотя мы использовали `get_user_model` для ссылки на модель `User` здесь, на самом деле существует три различных способа ссылки на модель `User` в Django.

Используя `get_user_model`, мы гарантируем, что ссылаемся на правильную модель пользователя, будь то пользователь по умолчанию `User` или собственная модель пользователя, как это часто бывает в новых проектах Django.

Далее нам нужно определить представления для каждой конечной точки. Сначала добавьте `UserSerializer` в список импорта. Затем создайте класс `UserList`, который выводит список всех пользователей, и класс `UserDetail`, который предоставляет детальное представление отдельного пользователя. Как и в случае с нашими представлениями сообщений, мы можем использовать здесь `ListCreateAPIView` и `RetrieveUpdateDestroyAPIView`. Нам также нужно ссыльаться на модель пользователей через `get_user_model`, чтобы она была импортирована в верхней строке.

Code

```
# posts/views.py
from django.contrib.auth import get_user_model # new
from rest_framework import generics
from .models import Post
from .permissions import IsAuthorOrReadOnly
from .serializers import PostSerializer, UserSerializer # new
class PostList(generics.ListCreateAPIView):
    queryset = Post.objects.all()
    serializer_class = PostSerializer
class PostDetail(generics.RetrieveUpdateDestroyAPIView):
    permission_classes = (IsAuthorOrReadOnly,)
    queryset = Post.objects.all()
    serializer_class = PostSerializer
class UserList(generics.ListCreateAPIView): # new
    queryset = get_user_model().objects.all()
    serializer_class = UserSerializer
class UserDetail(generics.RetrieveUpdateDestroyAPIView): # new
    queryset = get_user_model().objects.all()
    serializer_class = UserSerializer
```

Если вы заметили, здесь довольно много повторений. И представления Post, и представления User имеют точно такой же набор запросов и класс serializer_class. Может быть, их можно как-то объединить для экономии кода?

Наконец, у нас есть наши URL-маршруты. Не забудьте импортировать наши новые представления UserList , и UserDetail .
Затем мы можем использовать префикс users/ для каждого из них.

Code

```
# posts/urls.py
from django.urls import path
from .views import UserList, UserDetail, PostList, PostDetail # new
urlpatterns = [
    path('users/', UserList.as_view()), # new
    path('users/<int:pk>', UserDetail.as_view()), # new
    path('', PostList.as_view()),
    path('<int:pk>', PostDetail.as_view()),
]
```

И мы закончили. Убедитесь, что локальный сервер все еще работает, и перейдите к просматриваемому API, чтобы убедиться, что все работает так, как ожидалось.

Наша конечная точка списка пользователей находится по адресу <http://127.0.0.1:8000/api/v1/users/>.

The screenshot shows a browser window titled "User List - Django REST framework" at the URL "127.0.0.1:8000/api/v1/users/". The page is titled "User List" and includes a "GET /api/v1/users/" button. The response is an "HTTP 200 OK" JSON object:

```
[{"id": 1, "username": "wsv"}, {"id": 2, "username": "testuser"}, {"id": 3, "username": "testuser2"}]
```

Below the JSON, there is a form with a "Username" field and a "POST" button. The "Raw data" tab is selected.

API Users List

The status code is 200 OK which means everything is working. We can see our three existing users.

A user detail endpoint is available at the primary key for each user. So our superuser account is located at: **http://127.0.0.1:8000/api/v1/users/1/**.

The screenshot shows a browser window titled "User Detail - Django REST framework" with the URL "127.0.0.1:8000/api/v1/users/1". The page displays the "User Detail" section of the Django REST framework. At the top, there are buttons for "DELETE", "OPTIONS", and "GET". Below that, a "Raw data" tab is selected, showing the response from a "GET /api/v1/users/1" request. The response is an "HTTP 200 OK" with headers: "Allow: GET, PUT, PATCH, DELETE, HEAD, OPTIONS", "Content-Type: application/json", and "Vary: Accept". The JSON data returned is: { "id": 1, "username": "wsv" }. Below this, there is a "HTML form" tab. A "PUT" button is visible. A "Username" input field contains "wsv", with the placeholder "Required. 150 characters or fewer. Letters, digits and @/./+/-/_ only.".

API User Instance

Наборы представлений

Набор представлений - это способ объединения логики для нескольких связанных представлений в одном классе. Другими словами, один набор представлений может заменить несколько представлений. В настоящее время у нас есть четыре представления: два для записей блога и два для пользователей. Вместо этого мы можем имитировать ту же функциональность с помощью двух наборов представлений: один для записей блога и один для пользователей.

Компромисс заключается в том, что при этом ухудшается читаемость для разработчиков, которые не знакомы с наборами представлений. Так что это компромисс.

Вот как выглядит код в нашем обновленном файле `posts/views.py`, когда мы меняем местами наборы представлений.

Code

```
# posts/views.py

from django.contrib.auth import get_user_model
from rest_framework import viewsets # new
from .models import Post
from .permissions import IsAuthorOrReadOnly
from .serializers import PostSerializer, UserSerializer

class PostViewSet(viewsets.ModelViewSet): # new
    permission_classes = (IsAuthorOrReadOnly,)
    queryset = Post.objects.all()
    serializer_class = PostSerializer

class UserViewSet(viewsets.ModelViewSet): # new
    queryset = get_user_model().objects.all()
    serializer_class = UserSerializer
```

В верхней части вместо импорта `generics` из `rest_framework` мы теперь импортируем наборы представлений во второй строке. Затем мы используем `ModelViewSet`, который предоставляет нам как представление списка, так и представление деталей. И нам больше не нужно повторять один и тот же `queryset` и `serializer_class` для каждого представления, как мы делали раньше!

На этом этапе локальный веб-сервер остановится, так как Django жалуется на отсутствие соответствующих URL-путей. Давайте зададим их дальше.

Маршрутизаторы

Маршрутизаторы работают непосредственно с наборами представлений, чтобы автоматически генерировать для нас паттерны URL. Наш текущий файл `posts/urls.py` содержит четыре паттерна URL: два для записей блога и два для пользователей. Вместо этого мы можем использовать один маршрут для каждого набора представлений. Итак, два маршрута вместо четырех URL-паттернов. Это звучит лучше, верно?

<http://www.djangoproject.com/en/1.8/intro/tutorial02/>
<http://www.djangoproject.com/en/1.8/intro/tutorial03/>

Django REST Framework имеет два маршрутизатора по умолчанию: SimpleRouter и DefaultRouter . Мы будем использовать SimpleRouter, но также можно создавать пользовательские маршрутизаторы для более продвинутой функциональности.

Вот как выглядит обновленный код:

Code

```
# posts/urls.py
from django.urls import path
from rest_framework.routers import SimpleRouter
from .views import UserViewSet, PostViewSet
router = SimpleRouter()
router.register('users', UserViewSet, basename='users')
router.register('', PostViewSet, basename='posts')
urlpatterns = router.urls
```

В верхней строке импортируется SimpleRouter, а также наши представления. Маршрутизатор устанавливается на SimpleRouter, и мы "регистрируем" каждый набор представлений для пользователей и постов. Наконец, мы устанавливаем наши URL-адреса для использования нового маршрутизатора. Теперь проверьте наши четыре конечные точки, запустив локальный сервер с помощью python manage.py runserver .

<http://www.django-rest-framework.org/api-guide/routers/#simplerouter>

<http://www.django-rest-framework.org/api-guide/routers/#defaultrouter>

The screenshot shows a browser window titled "User List - Django REST framework" at the URL "127.0.0.1:8000/api/v1/users/". The page is titled "User List" and includes a navigation bar with "Post List / User List", "OPTIONS", "GET", and a dropdown for "testuser2". A "Raw data" tab is selected, showing the following JSON response:

```
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": "testuser2"
    }
]
```

Below the response, there is a form with a "Username" field and a "POST" button. The "HTML form" tab is selected.

API User List

Обратите внимание, что список пользователей остался прежним, однако представление деталей немного отличается. Теперь он называется "User Instance" вместо "User Detail", и есть дополнительная опция "delete", которая встроена в ModelViewSet.

The screenshot shows a browser window titled "User Instance – Django REST fr" with the URL "127.0.0.1:8000/api/v1/users/1". The page is titled "User Instance" and displays a single user record with ID 1 and username "wsv". It includes standard RESTful actions: DELETE, OPTIONS, and GET. Below the record, there's a form to update the user's username, with "wsv" entered. Buttons for "Raw data" and "HTML form" are visible, along with a "PUT" button.

API User Detail

Настройки наборов представлений возможны, но важным преимуществом в обмен на написание немного меньшего количества кода с наборами представлений является то, что настройки по умолчанию могут потребовать некоторой дополнительной конфигурации, чтобы соответствовать именно тому, что вы хотите.

Переходя к списку постов на <http://127.0.0.1:8000/api/v1/>, мы видим, что здесь все то же самое:

The screenshot shows a browser window titled "Post List - Django REST framework" with the URL "127.0.0.1:8000/api/v1/". The page is titled "Post List" and displays a single post entry:

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

[
    {
        "id": 1,
        "author": 1,
        "title": "Hello world! (edited)",
        "body": "This is my first blog post.",
        "created_at": "2020-07-29T17:37:03.350702Z"
    }
]
```

Below the JSON response is a form for creating a new post. The "Raw data" tab is selected. The form fields are:

Author	wsv
Title	<input type="text"/>
Body	<input type="text"/>

A "POST" button is located at the bottom right of the form.

API Post List

И, что важно, наши разрешения по-прежнему работают. Когда мы вошли в систему под учетной записью testuser2, экземпляр Post по адресу `http://127.0.0.1:8000/api/v1/1/` доступен только для чтения.

The screenshot shows a browser window titled "Post Instance - Django REST fr". The address bar displays "127.0.0.1:8000/api/v1/1/". The user is logged in as "testuser2". The page title is "Django REST framework". Below the title, there's a breadcrumb navigation: "Post List / Post Instance". The main content area is titled "Post Instance". On the right, there are two buttons: "OPTIONS" and "GET". A dropdown menu indicates the current method is "GET". Below these buttons, the URL "GET /api/v1/1/" is shown. The response status is "HTTP 200 OK". The response headers are: "Allow: GET, PUT, PATCH, DELETE, HEAD, OPTIONS", "Content-Type: application/json", and "Vary: Accept". The response body is a JSON object:

```
{  
    "id": 1,  
    "author": 1,  
    "title": "Hello world! (edited)",  
    "body": "This is my first blog post.",  
    "created_at": "2020-07-29T17:37:03.350702Z"  
}
```

API Post Instance Not Owner

Однако если мы войдем в систему под учетной записью суперпользователя, который является автором единственной записи в блоге, то у нас будут полные привилегии на **чтение-запись-редактирование-удаление**.

The screenshot shows a browser window titled "Post Instance - Django REST fr" with the URL "127.0.0.1:8000/api/v1/1/". The page is titled "Django REST framework" and "Post Instance". It displays a single post instance with the following details:

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

{
    "id": 1,
    "author": 1,
    "title": "Hello world! (edited)",
    "body": "This is my first blog post.",
    "created_at": "2020-07-29T17:37:03.350702Z"
}
```

Below the JSON response, there is a form for editing the post. The "Raw data" tab is selected. The form fields are:

	Value
Author	wsv
Title	Hello world! (edited)
Body	This is my first blog post.

A "PUT" button is located at the bottom right of the form area.

API Post Instance Owner

Вывод

Наборы представлений и маршрутизаторы - это мощная абстракция, которая сокращает объем кода, который мы, разработчики, должны писать. Однако за эту компактность приходится платить начальной кривой обучения. В первые несколько раз, когда вы будете использовать наборы представлений и маршрутизаторы вместо представлений и паттернов URL, это будет казаться странным.

В конечном итоге решение о том, когда добавлять наборы представлений и маршрутизаторы в ваш проект, довольно субъективно. Хорошее эмпирическое правило - начинать с представлений и URL. По мере роста сложности вашего API, если вы обнаружите, что повторяете одни и те же шаблоны конечных точек снова и снова, тогда обратитесь к наборам представлений и маршрутизаторам. До тех пор сохраняйте простоту.

Глава 9: Схемы и документация

Теперь, когда у нас есть готовый API, нам нужен способ быстро и точно документировать его функциональность для других. В конце концов, в большинстве компаний и команд, разработчик использующий API, - это не тот же разработчик, который изначально его создал. К счастью, существуют автоматизированные инструменты, позволяющие сделать это за нас.

Схема - это машиночитаемый документ, в котором описаны все доступные конечные точки API, URL-адреса и HTTP-глаголы (GET, POST, PUT, DELETE и т.д.), которые они поддерживают. Документация - это то, что добавляется в схеме, что делает ее более удобной для чтения и потребления человеком. В этой главе мы добавим схему в наш проект Blog, а затем добавим два различных способа документирования. В конце мы внедрим автоматизированный способ документирования любых текущих и будущих изменений в нашем API.

В качестве напоминания, вот полный список наших текущих конечных точек API:

Diagram

Endpoint	HTTP Verb
/	GET
/:pk/	GET
/users/	GET
/users/:pk/	GET
/rest-auth/registration	POST
/rest-auth/login	POST
/rest-auth/logout	GET
/rest-auth/password/reset	POST
/rest-auth/password/reset/confirm	POST

Схемы

До версии 3.9 , Django REST Framework полагался на Core API для схем, но теперь он перешел на схему OpenAPI (ранее известную как Swagger).

Первым шагом будет установка PyYAML и uritemplate . PyYAML преобразует нашу схему в формат YAML-basd OpenAPI, а uritemplate добавит параметры в пути URL.

командная строка:

(blogapi) \$ pipenv install pyyaml==5.3.1 uritemplate==3.0.1

Далее нам предлагается выбор: сгенерировать статическую схему или динамическую схему. Если ваш API меняется нечасто, статическую схему можно генерировать периодически и обслуживать из статических файлов для высокой производительности. Однако, если ваш API меняется довольно часто, вы можете рассмотреть вариант с динамической схемой. Здесь мы реализуем оба варианта.

Во-первых, это подход со статической схемой, который использует команду управления generateschema. Мы можем вывести результат в файл openapi-schema.yml .

командная строка:

(blogapi) \$ python manage.py generateschema > openapi-schema.yml

Если вы откроете этот файл, он будет довольно длинным и не очень удобным для человека. Но для компьютера он отлично отформатирован. Для динамического подхода обновите config/urls.py, импортировав get_schema_view в верхней части, а затем создайте выделенный путь в openapi. Название, описание и версия могут быть настроены по необходимости.

<https://www.djangoproject.org/community/3.9-announcement>

<http://www.coreapi.org/>

<https://www.openapi.org/>

<https://pyyaml.org/>

<https://github.com/python-hyper/uritemplate>

Code

```
# config/urls.py

from django.contrib import admin
from django.urls import include, path
from rest_framework.schemas import get_schema_view # 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('openapi', get_schema_view( # new
        title="Blog API",
        description="A sample API for learning DRF",
        version="1.0.0"
    ), name='openapi-schema'),
]
```

Если вы снова запустите локальный сервер с помощью `python manage.py runserver` и перейдете к нашей новой конечной точке URL схемы по адресу `http://127.0.0.1:8000/openapi`, то автоматически сгенерированная схема всего нашего API будет доступна.

The screenshot shows a web browser window titled "Schema - Django REST framework". The URL is "127.0.0.1:8000/openapi". The page displays the API schema for a "Blog API" version 1.0.0. It includes sections for "openapi", "info", "paths", and "responses". The "paths" section shows a "get" operation for "/api/v1/users/" which returns a list of users. The "responses" section for this operation specifies a JSON schema with an array of items, each having an "id" (integer, read-only) and a "username" (string, required, max length 150, pattern matching [a-zA-Z][a-zA-Z0-9._]+). The "post" operation for the same path is also defined.

```
HTTP 200 OK
Allow: GET, HEAD, OPTIONS
Content-Type: application/vnd.oai.openapi
Vary: Accept

openapi: 3.0.2
info:
  title: Blog API
  version: 1.0.0
  description: A sample API for learning DRF
paths:
  /api/v1/users/:
    get:
      operationId: listUsers
      description: ''
      parameters: []
      responses:
        '200':
          content:
            application/json:
              schema:
                type: array
                items:
                  properties:
                    id:
                      type: integer
                      readOnly: true
                    username:
                      type: string
                      description: Required. 150 characters or fewer. Letters, digits and @/./+-/_ only.
                      pattern: ^[a-zA-Z][a-zA-Z0-9._]+[a-zA-Z]$
                      maxLength: 150
                  required:
                    - username
                  description: ''
    post:
      operationId: createUser
```

API Schema

Лично я предпочитаю динамический подход в проектах.

Документация

Django REST Framework также поставляется со встроенной функцией документирования API, которая переводит схему в более удобный для разработчиков формат.

В настоящее время существует три популярных подхода: использование SwaggerUI, ReDoc или стороннего пакета drf-yasg. Поскольку drf-yasg довольно популярен и поставляется с большим количеством встроенных функций, мы будем использовать его здесь.

Шаг первый - установить последнюю версию drf-yasg .

<https://www.django-rest-framework.org/topics/documenting-your-api/>

<https://swagger.io/tools/swagger-ui/>

<https://github.com/Rebilly/ReDoc>

командная строка:

(blogapi) \$ pipenv install drf-yasg==1.17.1

Шаг второй: добавьте его в конфигурацию INSTALLED_APPS в config/settings.py .

Code

```
# config/settings.py

INSTALLED_APPS = [
...
# 3rd-party apps
    'rest_framework',
    'rest_framework.authtoken',
    'allauth',
    'allauth.account',
    'allauth.socialaccount',
    'rest_auth',
    'rest_auth.registration',
    'drf_yasg', # new

# Local
    'posts.apps.PostsConfig',
]
```

Шаг третий: обновите наш файл urls.py на уровне проекта. В верхней части файла мы можем заменить get_schema_view из DRF на get_schema_view из drf_yasg, а также импортировать openapi . Мы также добавим разрешение DRF для дополнительных опций.

Переменная schema_view обновляется и включает дополнительные поля, такие как terms_of_service , contact , и license . Затем в разделе urlpatterns мы добавляем пути для Swagger и ReDoc.

Code

```
# config/urls.py
from django.contrib import admin
from django.urls import include, path
from rest_framework import permissions # new
from drf_yasg.views import get_schema_view # new
from drf_yasg import openapi # new
schema_view = get_schema_view( # new
    openapi.Info(
        title="Blog API",
        default_version="v1",
        description="A sample API for learning DRF",
        terms_of_service="https://www.google.com/policies/terms/",
        contact=openapi.Contact(email="hello@example.com"),
        license=openapi.License(name="BSD License"),
    ),
    public=True,
    permission_classes=(permissions.AllowAny,),
)
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('swagger/', schema_view.with_ui( # new
        'swagger', cache_timeout=0), name='schema-swagger-ui'),
    path('redoc/', schema_view.with_ui( # new
        'redoc', cache_timeout=0), name='schema-redoc'),
]
```

Убедитесь, что локальный сервер запущен. Конечная точка Swagger теперь доступна по адресу:

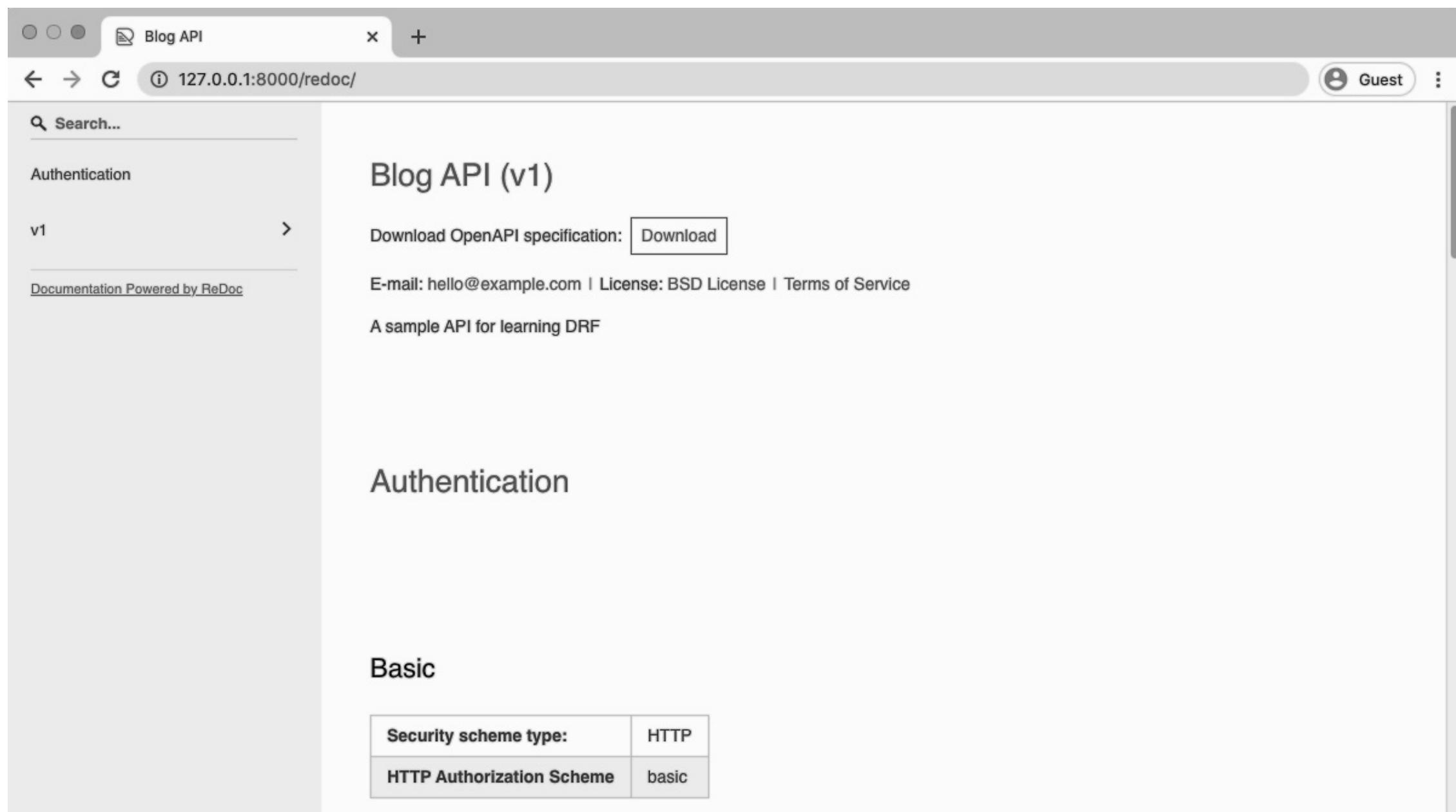
<http://127.0.0.1:8000/swagger/>

The screenshot shows the Swagger UI interface for a 'Blog API v1'. The top navigation bar includes tabs for 'Guest' and 'Explore'. The main title is 'Blog API v1' with a subtitle '[Base URL: 127.0.0.1:8000/api] http://127.0.0.1:8000/swagger/?format=openapi'. Below the title, there's a note: 'A sample API for learning DRF' followed by links to 'Terms of service', 'Contact the developer', and 'BSD License'. On the left, there's a dropdown for 'Schemes' set to 'HTTP'. On the right, there are buttons for 'Django wsv' (selected), 'Django Logout', and 'Authorize'. A 'Filter by tag' input field is present. The main content area is titled 'v1' and lists several API endpoints:

- GET /v1/** **v1_list** (locked)
- POST /v1/** **v1_create** (locked)
- POST /v1/dj-rest-auth/login/** **v1_dj-rest-auth_login_create** (locked)
- GET /v1/dj-rest-auth/logout/** Calls Django logout method and delete the Token object assigned to the current User object. **v1_dj-rest-auth_logout_list** (locked)
- POST /v1/dj-rest-auth/logout/** Calls Django logout method and delete the Token object assigned to the current User object. **v1_dj-rest-auth_logout_create** (locked)
- POST /v1/dj-rest-auth/password/change/** Calls Django Auth SetPasswordForm save method. **v1_dj-rest-auth_password_change_create** (locked)

Swagger view

Затем подтвердите, что представление ReDoc также работает на сайте <http://127.0.0.1:8000/redoc/>.



ReDoc view

Документация drf-yasg является достаточно полной и описывает множество других настроек, которые могут быть сделаны в зависимости от потребностей вашего API.

Вывод

Добавление документации является жизненно важной частью любого API. Как правило, это первое, на что смотрят коллеги-разработчики в команде или в проектах с открытым исходным кодом. Благодаря автоматизированным инструментам, рассмотренным в этой главе, для обеспечения вашего API точной и актуальной документацией требуется лишь небольшое количество настроек.

Вывод

Мы подошли к концу книги, но это только начало того, чего можно достичь с помощью Django REST Framework. На протяжении трех различных проектов - API библиотеки, API Todo и API блога - мы создавали все более сложные веб-интерфейсы с нуля. И не случайно что на каждом этапе этого пути Django REST Framework предоставляет встроенные функции, облегчающие нам жизнь.

Если вы никогда раньше не создавали веб-интерфейсы с помощью другого фреймворка, предупреждаем, что вы избалованы. Если же вы это делали, то будьте уверены, что эта книга лишь поверхностно описывает возможности Django REST Framework. Официальная документация является отличным ресурсом для дальнейшего изучения, когда вы уже имеете представление об основах.

Следующие шаги

Самой большой областью, заслуживающей дальнейшего изучения, является тестирование. Традиционные тесты Django можно и нужно применять к любому проекту веб-API, но в Django REST Framework есть целый набор инструментов для тестирования запросов API.

Хорошим следующим шагом будет реализация API pastebin, описанного в официальном учебнике по DRF . Я даже написал обновленное руководство для начинающих, в котором есть пошаговые инструкции.

Сторонние пакеты так же важны для разработки Django REST Framework, как и для самого Django. Полный список можно найти на Django Packages или в репозитории awesome-django на Github.

<http://www.django-rest-framework.org/>

<http://www.django-rest-framework.org/api-guide/testing/>

<http://www.django-rest-framework.org/tutorial/1-serialization/>

[1https://learndjango.com/tutorials/official-django-rest-framework-tutorial-beginners](https://learndjango.com/tutorials/official-django-rest-framework-tutorial-beginners)

<https://djangopackages.org/>

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

Благодарность

В то время как сообщество Django довольно велико и опирается на упорный труд многих людей, Django REST Framework гораздо меньше по сравнению с ним. Изначально он был создан Томом Кристи , английским инженером-программистом, который теперь работает над ним полный рабочий день благодаря финансированию open-source. Он по-прежнему ведет активную разработку. Если вам нравится работать с Django REST Framework, пожалуйста, уделите время, чтобы лично поблагодарить его в Twitter .

И спасибо, что читаете и поддерживаете мою работу. Если вы купили книгу на Amazon, пожалуйста, подумайте о том, чтобы оставить честный отзыв: они оказывают огромное влияние на продажи книг и помогают мне продолжать выпускать книги и бесплатный контент по Django, что я очень люблю делать.

<http://www.tomchristie.com/>

https://twitter.com/_tomchristie

От переводчика...

Перевод книги посвящается моему ментору (А Э), который 24/7 на связи, поддерживает, и всегда готов помочь. В мире программирования люди более отзывчивые, помогут с кодом, если хоть готовый ответ не скинут то отправят подсказки, также хочу поблагодарить одного ментора(Д А) который помогал с терминами и с переводом.

На перевод книги меня подтолкнуло отсутствие реально нужной литературы по Django Rest Framework на русском... Когда я учила этот framework... так сложно было понимать английские статьи, книги, и тд ...

Но спустя время, уже через не хочу стала понимать техническую литературу.... Потом подумала а ведь действительно нужной литературы на русском нету. И начала перевод, закончила через неделю.... Мне понравилось В дальнейшем буду переводить остальные хорошие книги по django...

Пожелайте удачи.

ITcoder

Если есть хорошие книги на английском отправляйте в телеграм:

<https://t.me/kjumabekovva> 

подписывайтесь на канал в телеграм: https://t.me/python_itkg

