



1ST EDITION

Создание веб-API Python с ПОМОЩЬЮ FastAPI

Быстрое руководство по созданию
высокопроизводительных и надежных веб-API с очень
небольшим количеством шаблонного кода

Абдулазиз Абдулазиз Адешина

Создание веб-API Python с помощью FastAPI

Быстрое руководство по созданию высокопроизводительных и надежных веб-API с очень небольшим количеством шаблонного кода.

Абдулазиз Абдулазиз Адешина



BIRMINGHAM—MUMBAI

Создание веб-API Python с помощью FastAPI

Copyright © 2022 Packt Publishing

Все права защищены. Никакая часть этой книги не может быть воспроизведена, сохранена в поисковой системе или передана в любой форме и любыми средствами без предварительного письменного разрешения издателя, за исключением случаев, когда краткие цитаты включены в критические статьи или обзоры.

При подготовке этой книги были приложены все усилия для обеспечения точности представленной информации. Однако информация, содержащаяся в этой книге, продается без явно выраженных или подразумеваемых гарантий. Ни автор, ни издательство Packt Publishing, ни его дилеры и распространители не несут ответственности за любой ущерб, причиненный или предположительно вызванный прямо или косвенно этой книгой.

Packt Publishing постарался предоставить информацию о товарных знаках обо всех компаниях и продуктах, упомянутых в этой книге, с надлежащим использованием заглавных букв. Однако Packt Publishing не может гарантировать точность этой информации.

Ассоциированный менеджер по продуктам группы: Паван Рамчандани

Менеджер по издательскому продукту: Аарон Танна

Главный редактор: Марк Дсуза

Редактор по разработке контента: Дивья Виджаян

Технический редактор: Шубхам Шарма

Редактор текста: Сафис Эдитинг

Координатор проекта: Рашика Ба

Корректор: Сафис Эдитинг

Индексатор: Пратик Широкар

Художник-постановщик: Виджай Камбл

Координаторы по маркетингу: Анамика Сингх и Мэрилу Де Мелло

Впервые опубликовано: июль 2022 г

Производственный номер: 1150722

Опубликовано Packt Publishing

ООО Ливери Плейс

Улица Ливери 35

Бирмингем

В3 2PB, Великобритания.

ISBN 978-1-80107-663-0

www.packt.com

Моей матери и памяти моего отца за их жертвы, веру и постоянную поддержку на протяжении многих лет. Моим удивительным сестрам Амидат и Аминат за то, что они являются постоянным источником радости и счастья.

Моему дяде Бако за его постоянную поддержку на протяжении многих лет. Моим лучшим друзьям Абдулрахману и Амине за то, что они всегда были рядом.

– Абдулазиз Абдулазиз Адешина

Авторы

Об авторе

Абдулазиз Абдулазиз Адешина — опытный разработчик Python, инженер-программист и технический писатель с широким набором технических навыков в своем арсенале.

Его опыт привел его к созданию приложений командной строки, серверных приложений в FastAPI и инструментов для поиска сокровищ на основе алгоритмов. Ему также нравится преподавать Python и решать математические задачи в своем блоге. Абдулазиз в настоящее время находится на предпоследнем курсе программы по водным ресурсам и охране окружающей среды. Его опыт работы в качестве приглашенного технического автора включает такие проекты, как Auth0, LogRocket, Okteto, и TestDriven.

.

Я хочу поблагодарить Аллаха (СВТ) за его бесконечную милость и Себастьяна Рамиреса за создание FastAPI. Я хочу поблагодарить Прешэс Ндубуезе за то, что познакомил меня с FastAPI и настоял на том, чтобы я ознакомился с фреймворком, и Боладжи Оладжиде за помощь в обзоре первого главы. Наконец, я также хочу поблагодарить каждого из моих близких сотрудников, особенно моего дядю, Тосина Олаянджу, за их поддержку на протяжении всей работы над этой книгой — я невероятно благодарен и польщен.

О рецензенте

Акаш Ранджан — профессионал Python с более чем 6-летним опытом работы в отрасли. Он разрабатывал, развертывал и управлял крупномасштабными корпоративными приложениями. У него большой опыт в создании API и разработке архитектуры приложений на основе микросервисов

Оглавление

Предисловие

Часть 1: Введение в FastAPI

1

Начало работы с FastAPI

Технические требования	4	Управление пакетами с помощью pip	11
Основы Git	4	Установка pip	12
Установка Git	4	Настройка Docker	13
Git-операции	5	Dockerfile	14
Git-ветки	8	Создание простого приложения FastAPI	15
Создание изолированных сред разработки с помощью Virtualenv	9	Резюме	16
Создание виртуальной среды	9		
Активация и деактивация виртуальной среды	9		

2

Маршрутизация в FastAPI

Технические требования	18	Валидация тела запроса с использованием моделей Pydantic	23
Понимание маршрутизации в FastAPI	18	Вложенные модели	26
Пример маршрутизации	18	Путь и параметры запроса	27
Маршрутизация с APIRouter class	19	Параметры пути	27
		Параметры запроса	30

Тело запроса	30	Создание простого CRUD-приложения	37
Автоматические документы FastAPI	31	Резюме	42

3

Модели ответов и обработка ошибок

Технические требования	44	Коды состояния	44
Понимание ответов в FastAPI	44	Построение моделей ответа	45
Что такое заголовок ответа?	44	Обработка ошибок	48
Что такое тело ответа?	44	Резюме	52

4

Шаблоны в FastAPI

Технические требования	53	Макросы	57
Понимание Jinja	54	Наследование шаблонов	58
Фильтры	54	Использование шаблонов Jinja в FastAPI	58
Использование операторов if	56	Резюме	67
Циклы	56		

Часть 2: Создание и защита приложений FastAPI

5

Структурирование приложений FastAPI

Технические требования	72	Реализация моделей	74
Структурирование в приложениях FastAPI	72	Реализация маршрутов	78
Создание приложения для планирования мероприятий	73	Резюме	87

6

Подключение к базе данных

Технические требования	90	Документ	104
Настройка SQLAlchemy	90	Инициализация базы данных	106
Таблицы	90	CRUD операции	110
Строки	91	Создать	110
Сессии	92	Читать	110
Создание базы данных	92	Обновить	111
Создание событий	97	Удалить	112
Чтение событий	99	routes/events.py	112
Обновление событий	101	routes/users.py	114
Удалить событие	102	Резюме	120
Настройка MongoDB	104		

7

Защита приложений FastAPI

Технические требования	122	Обработка аутентификации пользователя	132
Методы аутентификации в FastAPI	122	Обновление приложения	133
Внедрение зависимости	123	Обновление маршрута входа пользователя	133
Создание и использование зависимости	123	Обновление маршрутов событий	136
Защита приложения с помощью OAuth2 и JWT	124	Обновление класса документа события и маршрутов	140
Хэширование паролей	125	Настройка CORS	145
Создание и проверка токенов доступа	129	Резюме	146

Часть 3: Тестирование и развертывание приложений FastAPI

8

Тестирование приложений FastAPI

Технические требования	150	Тестирование маршрута регистрации	157
Модульное тестирование с помощью pytest	150	Тестирование маршрута входа	158
Устранение повторения с помощью фикстур pytest	153	Тестирование конечных точек CRUD	160
Настройка тестовой среды	154	Тестирование конечных точек READ	161
Написание тестов для конечных точек REST API	156	Тестирование конечной точки CREATE	163
		Тестирование конечной точки UPDATE	166
		Тестирование конечной точки DELETE	168
		Покрытие тестами	171
		Резюме	174

9

Развертывание приложений FastAPI

Технические требования	176	Развертывание приложения локально	181
Подготовка к развертыванию	176	Запускаем приложение	183
Управление зависимостями	176	Развертывание Docker образов	186
Настройка переменных среды	177	Развертывание баз данных	187
Развертывание с помощью Docker	178	Резюме	187
Написание Dockerfile	178		
Создание Docker образа	180		

Другие книги, которые могут вам понравиться

Предисловие

FastAPI — это быстрая и эффективная веб-инфраструктура для создания API с помощью Python. Эта книга представляет собой подробное руководство по созданию приложения с помощью среды FastAPI.

Начинается с основ структуры FastAPI и других технологий, используемых в этой книге. Затем вы узнаете о различных аспектах фреймворка: системе маршрутизации, моделировании ответов, обработке ошибок и шаблонах.

В этой книге вы узнаете, как создавать быстрые, эффективные и масштабируемые приложения на Python с помощью FastAPI. Вы начнете с приложения *Hello World* к полноценному API, использующему базу данных, аутентификацию и шаблоны. Вы узнаете, как структурировать свое приложение для повышения эффективности, удобочитаемости и масштабируемости. Благодаря интеграции с другими библиотеками в вашем приложении вы узнаете, как подключить ваше приложение как к базе данных SQL, так и к базе данных NoSQL, интегрировать шаблоны и создать аутентификацию. Ближе к концу этой книги вы узнаете, как писать тесты, контейнеризовать приложение, создавать конвейер непрерывной интеграции и доставки с помощью действий GitHub, а также разворачивать приложение в нескольких облачных службах. Все это будет преподаваться с помощью теоретического и практического подхода.

К концу этой книги вы будете владеть необходимыми знаниями для создания и развертывания надежного веб-API с использованием инфраструктуры FastAPI.

Для кого эта книга

Основная аудитория этой книги — любой разработчик Python, заинтересованный в создании веб-API. Идеальный читатель знаком с основами языка программирования Python.

Что охватывает эта книга

Глава 1 «Начало работы с FastAPI» знакомит с основами FastAPI и других технологий, используемых в книге. В главе также подробно описаны шаги, связанные с настройкой среды разработки для вашего приложения FastAPI.

В главе 2 «Маршрутизация в FastAPI» подробно рассказывается о процессе создания конечных точек с использованием системы маршрутизации в FastAPI. Компоненты системы маршрутизации, включая тело запроса и параметры пути, также обсуждаются наряду с их проверкой с помощью классов `pydantic`.

Глава 3 «Модели ответов и обработка ошибок», знакомит с ответами в FastAPI, моделированием ответов, обработкой ошибок и кодами состояния.

В главе 4 «Шаблоны в FastAPI» обсуждается, как можно использовать шаблоны для создания представлений и вывода ответов из API.

Глава 5 «Структурирование приложений FastAPI», знакомит со структурированием приложений, а также кратко описывает приложение, которое будет создано в следующих главах.

В главе 6 «Подключение к базе данных» обсуждаются два класса баз данных (SQL и NoSQL) и демонстрируется, как вы можете подключить свое приложение FastAPI к любому из них. Мы расскажем, как подключиться к базе данных SQL и использовать ее с помощью `SQLModel`, а также как работать с MongoDB с помощью средства сопоставления объектов и документов `Beanie`.

В главе 7 «Защита приложений FastAPI» рассказывается о том, что влечет за собой обеспечение безопасности вашего приложения — авторизация и аутентификация, реализация аутентификации и ограничение доступа к конечным точкам приложения.

Глава 8 «Тестирование приложений FastAPI», объясняет, что такое тестирование и как тестировать конечные точки API.

В главе 9 «Развертывание приложений FastAPI» обсуждаются шаги, необходимые для развертывания вашего приложения FastAPI.

Чтобы получить максимальную отдачу от этой книги

Вам понадобится последняя версия Python, установленная на вашем компьютере. Вы также должны быть знакомы с языком программирования Python, чтобы извлечь максимальную пользу из этой книги.

Software/hardware covered in the book	Operating system requirements
Python 3.10	Windows, macOS, or Linux
Git 2.36.0	Windows, macOS, or Linux

Если вы используете цифровую версию этой книги, мы советуем вам ввести код самостоятельно или получить доступ к коду из репозитория книги на [GitHub](#) (ссылка доступна в следующем разделе). Это поможет вам избежать возможных ошибок, связанных с копированием и вставкой кода.

Загрузите файлы примеров кода

Вы можете загрузить файлы примеров кода для этой книги с GitHub по адресу <https://github.com/PacktPublishing/Building-Python-Web-APIs-with-FastAPI>. Если есть обновление кода, оно будет обновлено в репозитории GitHub.

У нас также есть другие пакеты кода из нашего богатого каталога книг и видео, доступных по адресу <https://github.com/PacktPublishing/>. Проверь их!

Загрузите цветные изображения

Мы также предоставляем PDF-файл с цветными изображениями снимков экрана и диаграмм, использованных в этой книге. Скачать его можно здесь: <https://packt.link/qqhpc>.

Используемые соглашения

В этой книге используется ряд текстовых соглашений.

Код в тексте: указывает кодовые слова в тексте, имена таблиц базы данных, имена папок, имена файлов, расширения файлов, пути, фиктивные URL-адреса, пользовательский ввод и дескрипторы Twitter. Вот пример: «Чтобы вернуться к исходной основной ветке, мы запускаем `git checkout main`». Блок кода устанавливается следующим образом:

```
from fastapi import FastAPI
from routes.user import user_router

import uvicorn
```

Когда мы хотим привлечь ваше внимание к определенной части блока кода, соответствующие строки или элементы выделяются жирным шрифтом:

```
from pydantic import BaseModel
from typing import List

class Event(BaseModel):
    id: int
    title: str
    image: str
    description: str
```

```
tags: List[str]
location: str
```

Любой ввод или вывод командной строки записывается следующим образом:

```
$ git add hello.txt
$ git commit -m "Initial commit"
```

Жирный: Обозначает новый термин, важное слово или слова, которые вы видите на экране. Например, слова в меню или диалоговых окнах выделены жирным шрифтом. Вот пример: «Как показано на предыдущей диаграмме модели, у каждого пользователя будет поле «События», которое представляет собой список событий, на которые он имеет право собственности».

Советы или важные примечания

Появись вот так.

Связаться

Отзывы наших читателей всегда приветствуются.

Общий отзыв: Если у вас есть вопросы по какому-либо аспекту этой книги, напишите нам по адресу customercare@packtpub.com и укажите название книги в теме сообщения.

Исправления: Хотя мы приложили все усилия, чтобы обеспечить точность нашего контента, ошибки случаются. Если вы нашли ошибку в этой книге, мы будем признательны, если вы сообщите нам об этом. Пожалуйста, посетите www.packtpub.com/support/errata и заполните форму.

Пиратство: Если вы столкнетесь с незаконными копиями наших работ в любой форме в Интернете, мы будем признательны, если вы сообщите нам адрес или название веб-сайта. Пожалуйста, свяжитесь с нами по адресу copyright@packt.com со ссылкой на материал.

Если вы заинтересованы в том, чтобы стать автором: если есть тема, в которой у вас есть опыт, и вы заинтересованы в написании книги или участии в ней, пожалуйста, посетите авторов. packtpub.com.

Поделитесь своими мыслями

После того как вы прочитали «*Создание веб-API Python с помощью FastAPI*», нам будет интересно узнать ваше мнение!

Щелкните здесь, чтобы сразу перейти на страницу обзора этой книги на Amazon и поделиться своим мнением.

Ваш отзыв важен для нас и технического сообщества и поможет нам убедиться, что мы предоставляем контент отличного качества.

Часть 1: Введение в FastAPI

По завершении этой части вы будете иметь существенное представление о FastAPI, включая маршрутизацию, обработку файлов, обработку ошибок, построение моделей ответов и шаблоны. Эта часть начинается с знакомства с технологиями, которые будут использоваться на протяжении всей книги, прежде чем перейти к основам структуры FastAPI.

Эта часть состоит из следующих глав:

- Глава 1. Начало работы с FastAPI.
- Глава 2. Маршрутизация в FastAPI.
- Глава 3. Модели ответов и обработка ошибок.
- Глава 4. Шаблоны в FastAPI.

1

Начало работы с FastAPI

FastAPI — это веб-фреймворк Python, который мы собираемся использовать в этой книге. Это быстрый и легкий современный API, который проще в освоении по сравнению с другими веб-фреймворками на основе Python, такими как Flask и Django. FastAPI относительно новый, но его сообщество растет. Он широко используется при создании веб-API и развертывании моделей машинного обучения.

В первой главе вы узнаете, как настроить среду разработки и создать свое первое приложение FastAPI. Вы начнете с изучения основ Git — системы управления версиями — чтобы вооружить вас знаниями о хранении, отслеживании и извлечении изменений файлов при создании приложения. Вы также узнаете, как работать с пакетами в Python с помощью `pip`, как создавать изолированные среды разработки с помощью `Virtualenv` и познакомитесь с основами **Docker**. Наконец, вы познакомитесь с основами FastAPI, создав простое приложение *Hello World*.

Понимание ранее упомянутых технологий требуется для создания полноценного приложения FastAPI. Это также служит дополнением к вашему текущему набору навыков.

По завершении этой главы вы сможете настраивать и использовать Git, устанавливать пакеты и управлять ими с помощью `pip`, создавать изолированную среду разработки с помощью `Virtualenv`, использовать `Docker` и, что наиболее важно, создавать шаблоны для приложений FastAPI.

В этой главе рассматриваются следующие темы:

- Основы Git
- Создание изолированных сред разработки с помощью `Virtualenv`
- Управление пакетами с помощью `pip`
- Настройка и изучение основ `Docker`
- Создание простого приложения FastAPI

Технические требования

Вы можете найти файлы кода для этой главы на GitHub по ссылке <https://github.com/PacktPublishing/Building-Python-Web-APIs-with-FastAPI/tree/main/ch01>

Основы Git

Git — это система контроля версий, которая позволяет разработчикам записывать, отслеживать и возвращаться к более ранним версиям файлов. Это децентрализованный и легкий инструмент, который можно установить в любой операционной системе.

Вы узнаете, как использовать Git для ведения учета. По мере создания каждого уровня приложения будут вноситься изменения, и важно, чтобы эти изменения сохранялись.

Установка Git

Чтобы установить Git, посетите страницу загрузок по ссылке <https://git-scm.com/downloads> и выберите вариант загрузки для вашей текущей операционной системы. Вы будете перенаправлены на страницу с инструкциями по установке Git на свой компьютер.

Также стоит отметить, что Git поставляется как с **CLI**, так и с **GUI** интерфейсом. Таким образом, вы можете скачать тот, который лучше всего подходит для вас.

Git операции

Как упоминалось ранее, Git можно использовать для записи, отслеживания и возврата к более ранним версиям файла. Однако в этой книге будут использоваться только основные операции Git, которые будут представлены в этом разделе.

Чтобы Git работал правильно, папки с файлами должны быть инициализированы. Инициализация папок позволяет Git отслеживать содержимое, за исключением исключений.

Чтобы инициализировать новый репозиторий Git в вашем проекте, вам нужно запустить следующую команду в своем терминале:

```
$ git init
```

Чтобы включить отслеживание файлов, файл необходимо сначала добавить и зафиксировать. Коммит Git позволяет отслеживать изменения файлов между временными рамками; например, коммит, сделанный час назад, и текущая версия файла.

Что такое коммит?

Фиксация — это уникальный захват состояния файла или папки в определенное время, идентифицируемый уникальным кодом.

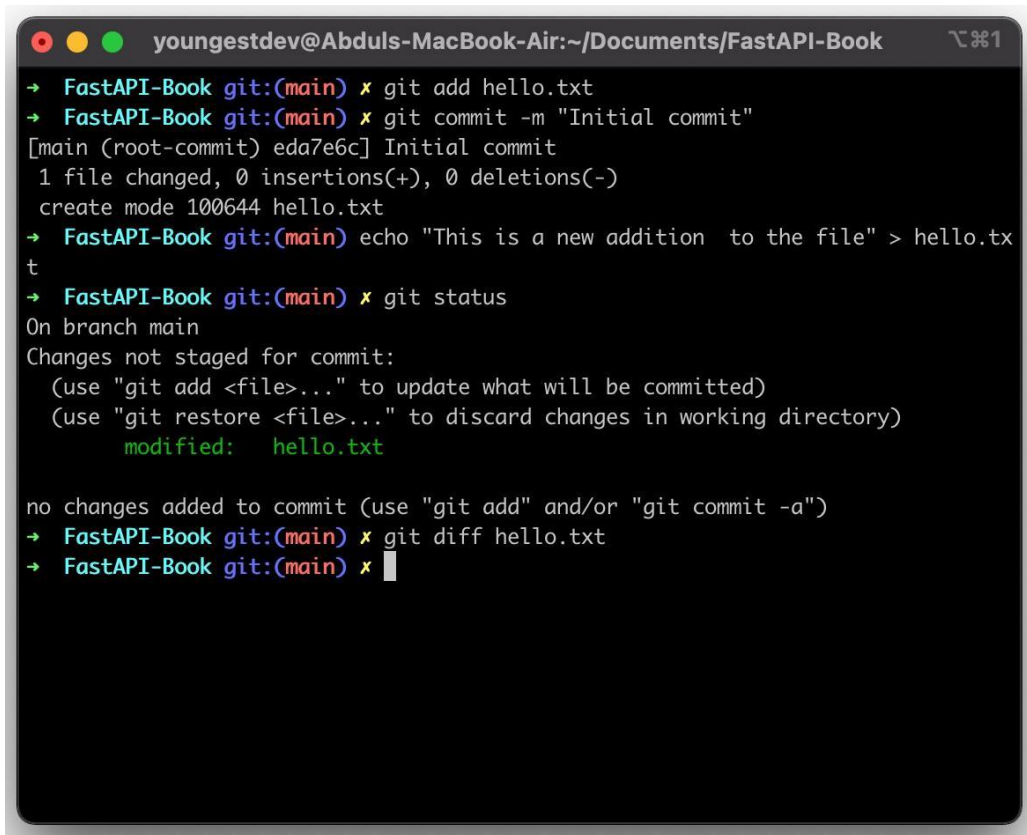
Теперь, когда мы знаем, что такое коммит, мы можем продолжить и зафиксировать файл следующим образом:

```
$ git add hello.txt  
$ git commit -m "Initial commit"
```

Вы можете отслеживать состояние ваших файлов после внесения изменений, выполнив следующую команду:

```
$ git status
```

Ваш терминал должен выглядеть примерно так:

A screenshot of a terminal window on a Mac. The title bar shows the user 'youngestdev' and the path '~/Documents/FastAPI-Book'. The terminal content shows a series of Git commands and their outputs: creating a file 'hello.txt', committing it with the message 'Initial commit', adding a new line to the file, and checking the status. The status shows the file is modified but not staged for commit. The user then runs 'git diff' and the prompt returns.

```
→ FastAPI-Book git:(main) x git add hello.txt
→ FastAPI-Book git:(main) x git commit -m "Initial commit"
[main (root-commit) eda7e6c] Initial commit
1 file changed, 0 insertions(+), 0 deletions(-)
create mode 100644 hello.txt
→ FastAPI-Book git:(main) echo "This is a new addition to the file" > hello.txt
→ FastAPI-Book git:(main) x git status
On branch main
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   hello.txt

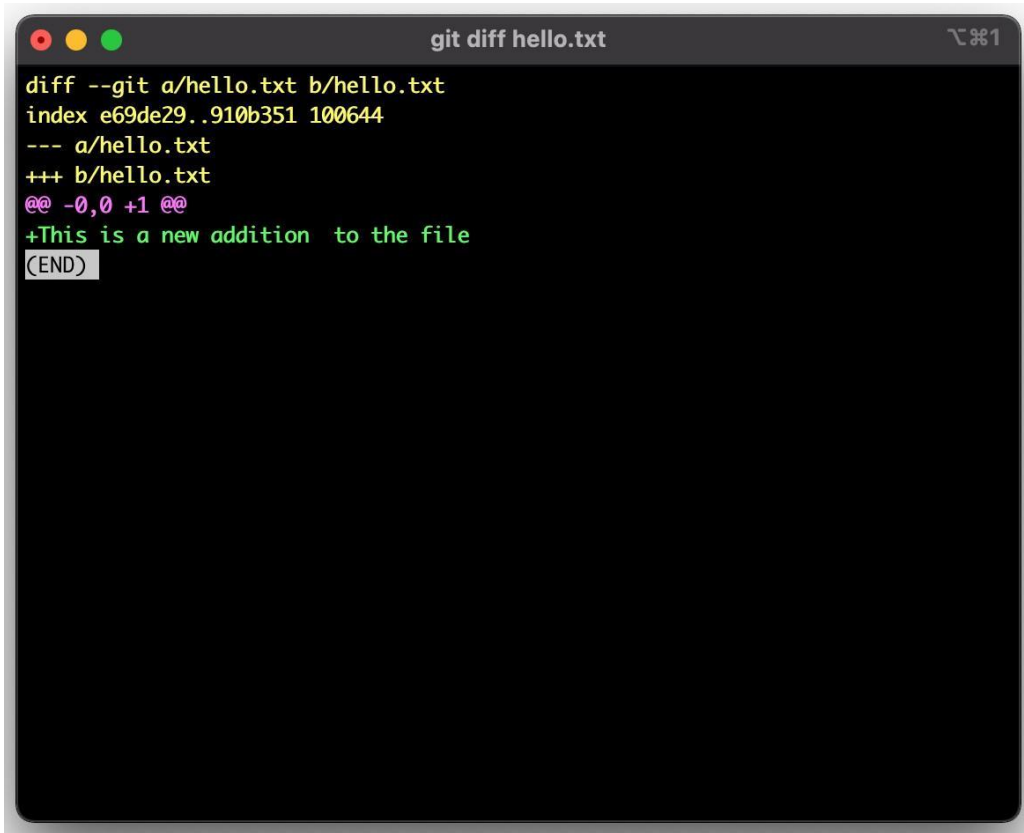
no changes added to commit (use "git add" and/or "git commit -a")
→ FastAPI-Book git:(main) x git diff hello.txt
→ FastAPI-Book git:(main) x
```

Рисунок 1.1 – Git команды

Чтобы просмотреть изменения, внесенные в файл, которые могут быть дополнениями или вычитаниями из содержимого файла, выполните следующую команду:

```
$ git diff
```

Ваш терминал должен выглядеть примерно так:



```
git diff hello.txt
diff --git a/hello.txt b/hello.txt
index e69de29..910b351 100644
--- a/hello.txt
+++ b/hello.txt
@@ -0,0 +1 @@
+This is a new addition to the file
(END)
```

Рисунок 1.2 – Вывод команды git diff

Рекомендуется включать файл `.gitignore` в каждую папку. The Файл `.gitignore` содержит имена файлов и папок, которые Git игнорирует. Таким образом, вы можете добавить и зафиксировать все файлы в вашей папке, не опасаясь зафиксировать такие файлы, как `.env`.

Чтобы включить файл `.gitignore`, выполните в терминале следующую команду:

```
$ touch .gitignore
```

Чтобы освободить файл от отслеживания Git, добавьте его в файл `.gitignore` следующим образом:

```
$ echo ".env" >> .gitignore
```

Общие файлы, содержащиеся в файле `.gitignore`, включают следующее:

- Файлы окружения (`*.env`)
- Виртуальная папка (`env`, `venv`)
- Папки метаданных IDE (такие как `.vscode` и `.idea`)

Git ветки

Ветки — это важная функция, которая позволяет разработчикам легко работать над различными функциями приложения, ошибками и т. д. по отдельности, прежде чем объединиться с основной веткой. Система ветвления используется как в небольших, так и в крупных приложениях и продвигает культуру предварительного просмотра и совместной работы с помощью запросов на включение. Первичная ветвь называется основной ветвью, и это ветвь, из которой создаются другие ветки.

Чтобы создать новую ветку из существующей ветки, мы запускаем команду `git checkout -b newbranch`. Давайте создадим новую ветку, выполнив следующую команду:

```
$ git checkout -b hello-python-branch
```

Предыдущая команда создает новую ветвь из существующей, а затем устанавливает активную ветвь на вновь созданную ветвь. Чтобы вернуться к исходной основной ветке, мы запускаем `git checkout main` следующим образом:

```
$ git checkout main
```

Важная заметка

Запуск `git checkout main` делает `main` активной рабочей веткой, тогда как `git checkout -b newbranch` создает новую ветку из текущей рабочей ветки и устанавливает вновь созданную ветку в качестве активной.

To learn more, refer to the Git documentation: <http://www.git-scm.com/doc>.

Теперь, когда мы изучили основы Git, мы можем приступить к изучению того, как создавать изолированные среды с помощью **virtualenv**.

Создание изолированных сред разработки с помощью Virtualenv

Традиционный подход к разработке приложений на Python заключается в изоляции этих приложений в виртуальной среде. Это сделано для того, чтобы избежать глобальной установки пакетов и уменьшить количество конфликтов во время разработки приложений.

Виртуальная среда — это изолированная среда, в которой установленные зависимости приложений доступны только внутри нее. В результате приложение может получать доступ только к пакетам и взаимодействовать только внутри этой среды.

Создание виртуальной среды

По умолчанию в Python3 установлен модуль `venv` из стандартной библиотеки. Модуль `venv` отвечает за создание виртуальной среды. Давайте создадим папку `todos` и создадим в ней виртуальную среду, выполнив следующие команды:

```
$ mkdir todos && cd todos
$ python3 -m venv venv
```

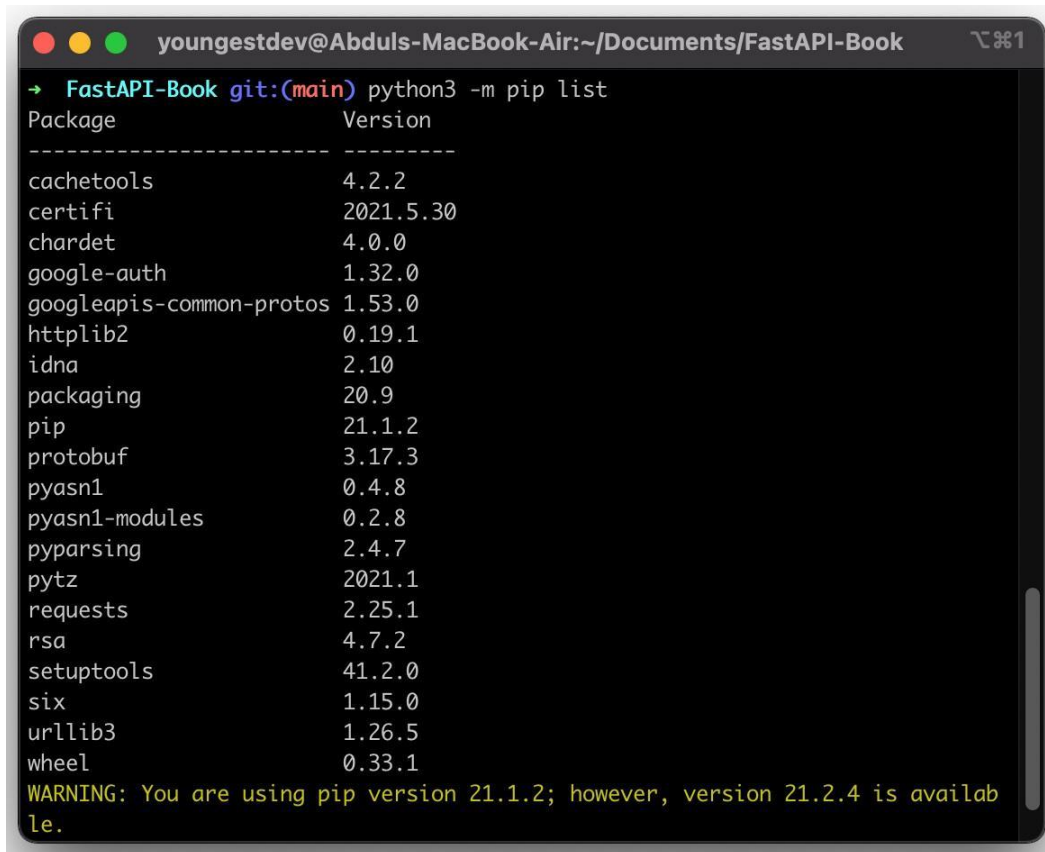
Модуль `venv` принимает в качестве аргумента имя папки, в которую следует установить виртуальную среду. В нашей только что созданной виртуальной среде копия интерпретатора Python установлена в папке `lib` а файлы, обеспечивающие взаимодействие внутри виртуальной среды, хранятся в папке `bin`.

Активация и деактивация виртуальной среды

Чтобы активировать виртуальную среду, мы запускаем следующую команду:

```
$ source venv/bin/activate
```

Предыдущая команда указывает вашей оболочке использовать интерпретатор и пакеты виртуальной среды по умолчанию. После активации виртуальной среды префикс папки виртуальной среды `venv` добавляется перед приглашением следующим образом:



```
→ FastAPI-Book git:(main) python3 -m pip list
Package            Version
-----
cachetools         4.2.2
certifi            2021.5.30
chardet            4.0.0
google-auth       1.32.0
googleapis-common-protos 1.53.0
httplib2          0.19.1
idna              2.10
packaging         20.9
pip              21.1.2
protobuf         3.17.3
pyasn1           0.4.8
pyasn1-modules   0.2.8
pyparsing        2.4.7
pytz             2021.1
requests         2.25.1
rsa              4.7.2
setuptools       41.2.0
six              1.15.0
urllib3          1.26.5
wheel            0.33.1
WARNING: You are using pip version 21.1.2; however, version 21.2.4 is available.
```

Рисунок 1.3 – Подсказка с префиксом

Чтобы деактивировать виртуальную среду, в командной строке запускается команда `deactivate`. При выполнении команды происходит немедленный выход из изолированной среды, а префикс удаляется следующим образом:

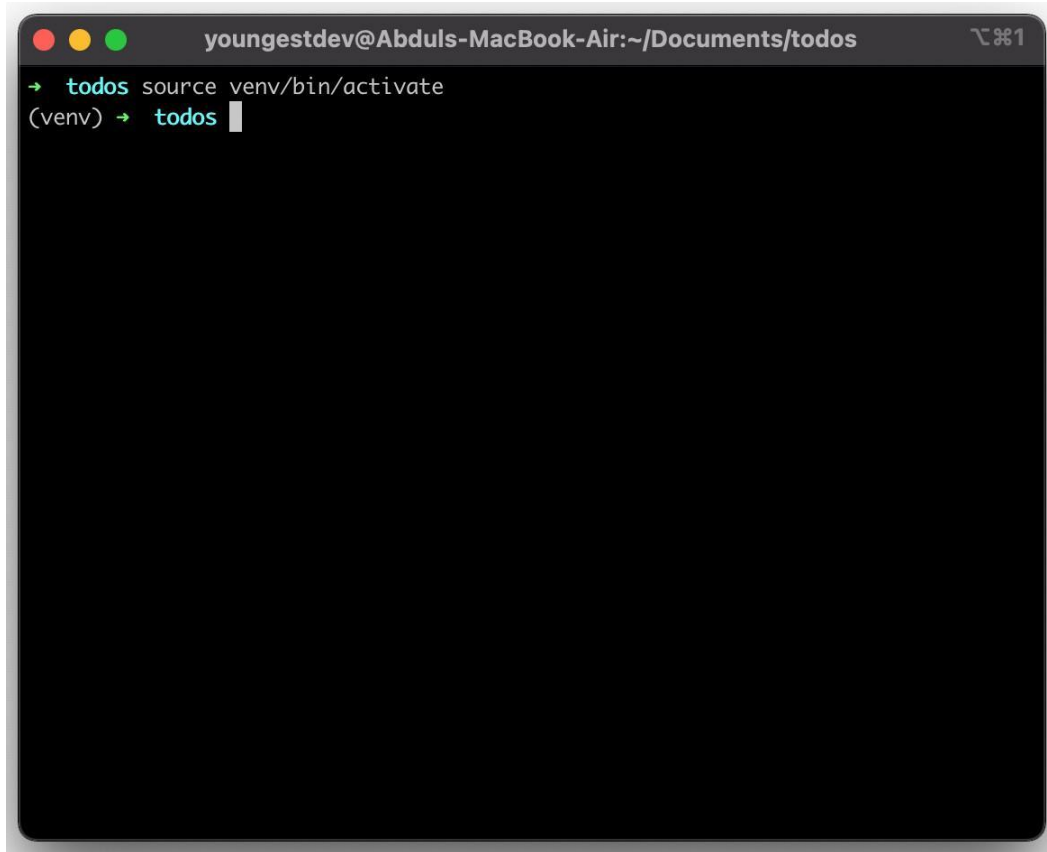


Рисунок 1.4 – Деактивация виртуальной среды

Важная заметка

Вы также можете создать виртуальную среду и управлять зависимостями приложений, используя *Pipenv* и *Poetry*.

Теперь, когда мы создали виртуальную среду, мы можем перейти к пониманию того, как работает управление пакетами с помощью `pip`.

Управление пакетами с помощью `pip`

Приложение FastAPI представляет собой пакеты, поэтому вы познакомитесь с методами управления пакетами, такими как установка пакетов, удаление пакетов и обновление пакетов для вашего приложения.

Установка пакетов из исходного кода может оказаться сложной задачей, поскольку в большинстве случаев она включает в себя загрузку и распаковку файлов `.tar.gz` перед установкой вручную. В сценарии, где необходимо установить сто пакетов, этот метод становится неэффективным. Тогда как автоматизировать этот процесс?

Pip — это менеджер пакетов Python, подобный JavaScript's `yarn`; он позволяет автоматизировать процесс установки пакетов Python как глобально, так и локально..

Установка pip

Pip автоматически устанавливается во время установки Python. Вы можете проверить, установлен ли pip, выполнив следующую команду в своем терминале:

```
$ python3 -m pip list
```

Предыдущая команда должна вернуть список установленных пакетов. Результат должен быть похож на следующий рисунок:

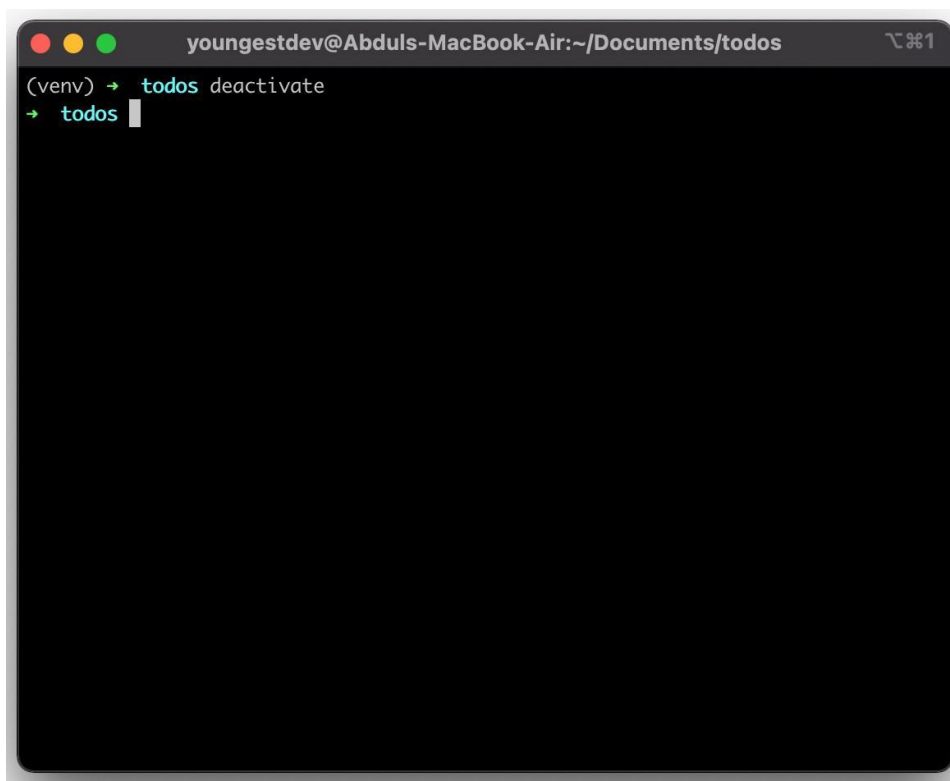


Рисунок 1.5 – Список установленных пакетов Python

Если команда возвращает ошибку, следуйте инструкциям на странице <https://pip.pypa.io/en/stable/installation/>, чтобы установить pip.

Основные команды

Установив pip, давайте изучим его основные команды. Чтобы установить пакет FastAPI с помощью pip, мы запускаем следующую команду:

```
$ pip install fastapi
```

В операционной системе Unix, такой как Mac или Linux, в некоторых случаях ключевое слово `sudo` добавляется перед установкой глобальных пакетов.

Для удаления пакета используется следующая команда:

```
$ pip uninstall fastapi
```

Чтобы собрать текущие пакеты, установленные в проекте, в файл, мы используем следующую команду `freeze`:

```
$ pip freeze > requirements.txt
```

Оператор `>` указывает `bash` сохранить вывод команды в файл `requirements.txt`. Это означает, что запуск `pip freeze` возвращает все установленные в данный момент пакеты.

Чтобы установить пакеты из файла, такого как файл `requirements.txt`, используется следующая команда:

```
$ pip install -r requirements.txt
```

Предыдущая команда в основном используется при развертывании.

Теперь, когда вы изучили основы pip и ознакомились с некоторыми основными командами, давайте изучим основы **Docker**.

Настройка Docker

По мере того, как наше приложение становится многоуровневым, например, база данных, объединение приложения в единый элемент позволяет нам развертывать наше приложение. Мы будем использовать **Docker** для контейнеризации уровней наших приложений в единый образ, который затем можно будет легко развернуть локально или в облаке.

Кроме того, использование **Dockerfile** и файла **docker-compose** избавляет от необходимости загружать образы наших приложений и делиться ими. Новые версии наших приложений можно создавать из файла **Dockerfile** и развертывать с помощью файла **docker-compose**. Образы приложений также можно хранить и извлекать из **Docker Hub**. Это известно, как операция толкания и вытягивания.

Чтобы начать настройку, загрузите и установите Docker с <https://docs.docker.com/install>.

Dockerfile

Dockerfile содержит инструкции о том, как должен быть создан образ нашего приложения. Ниже приведен пример Dockerfile:

```
FROM PYTHON:3.8
# Set working directory to /usr/src/app
WORKDIR /usr/src/app
# Copy the contents of the current local directory into the
container's working directory
ADD . /usr/src/app
# Run a command
CMD ["python", "hello.py"]
```

Далее мы создадим образ контейнера приложения и назовем `getting_started` следующим образом:

```
$ docker build -t getting_started .
```

Если Dockerfile отсутствует в каталоге, где запускается команда, путь к Dockerfile должен быть правильно добавлен следующим образом:

```
$ docker build -t api api/Dockerfile
```

Образ контейнера можно запустить с помощью следующей команды:

```
$ docker run getting-started
```

Docker — эффективный инструмент для контейнеризации. Мы рассмотрели только основные операции, и мы изучим больше практических операций в *Главе 9. Развертывание приложений FastAPI*.

Создание простого приложения FastAPI

Наконец, теперь мы можем перейти к нашему первому проекту FastAPI. Наша цель в этом разделе — представить FastAPI, создав простое приложение. Мы подробно рассмотрим операции в последующих главах.

Мы начнем с установки зависимостей, необходимых для нашего приложения, в папку `todos` которую мы создали ранее. Зависимости следующие:

- `fastapi`: Фреймворк, на котором мы будем строить наше приложение.
- `uvicorn`: Модуль Asynchronous Server Gateway Interface для запуска приложения.

Сначала активируйте среду разработки, выполнив следующую команду в каталоге вашего проекта:

```
$ source venv/bin/activate
```

Затем установите зависимости следующим образом:

```
(venv)$ pip install fastapi uvicorn
```

А пока мы создадим новый файл `api.py` и создадим новый экземпляр FastAPI следующим образом:

```
from fastapi import FastAPI

app = FastAPI()
```

Создав экземпляр FastAPI в переменной приложения, мы можем приступить к созданию маршрутов. Создадим приветственный маршрут.

Маршрут создается, сначала определяя декоратор для указания типа операции, а затем функцию, содержащую операцию, которая будет выполняться при вызове этого маршрута. В следующем примере мы создадим маршрут `"/"`, который принимает только запросы `GET requests` и возвращает приветственное сообщение при посещении:

```
@app.get("/")
async def welcome() -> dict:
    return { "message": "Hello World"}
```

Следующим шагом будет запуск нашего приложения с помощью `uvicorn`. В терминале выполните следующую команду:

```
(venv)$ uvicorn api:app --port 8000 --reload
```

В предыдущей команде, `uvicorn` принимает следующие аргументы:

- `file:instance:` Файл, содержащий экземпляр FastAPI и переменную имени, содержащую экземпляр FastAPI..
- `--port PORT:` Порт, на котором будет обслуживаться приложение.
- `--reload:` Необязательный аргумент, включенный для перезапуска приложения при каждом изменении файла.

Команда возвращает следующий вывод:

```
(venv) → todos uvicorn api:app --port 8080 --reload
INFO:     Will watch for changes in these directories: ['/'
Users/youngestdev/Documents/todos']
INFO:     Uvicorn running on http://0.0.0.0:8080 (Press CTRL+C
to quit)
INFO:     Started reloader process [3982] using statreload
INFO:     Started server process [3984]
INFO:     Waiting for application startup.
INFO:     Application startup complete.
```

Следующий шаг — протестировать приложение, отправив запрос GET в API. В новом терминале отправьте запрос GET с помощью `curl` следующим образом:

```
$ curl http://0.0.0.0:8080/
```

Ответ от приложения в консоле будет следующим:

```
{"message": "Hello World"}
```

Резюме

В этой главе мы узнали, как установить инструменты, необходимые для настройки нашей среды разработки. Мы также создали простой API в качестве введения в FastAPI и научились создавать маршрут в процессе.

В следующей главе вы познакомитесь с маршрутизацией в FastAPI. Во-первых, вы познакомитесь с процессом построения моделей для проверки полезной нагрузки запросов и ответов с помощью Pydantic. Затем вы узнаете о параметрах пути и запроса, а также о теле запроса и, наконец, узнаете, как создать приложение CRUD todo.

Маршрутизация в FastAPI

Маршрутизация является важной частью создания веб-приложения. Маршрутизация в FastAPI гибкая и простая. Маршрутизация — это процесс обработки **HTTP-запросов**, отправляемых клиентом на сервер. HTTP-запросы отправляются по определенным маршрутам, для которых определены обработчики для обработки запросов и ответа. Эти обработчики называются обработчиками маршрутов.

К концу этой главы вы будете знать, как создавать маршруты с использованием экземпляра **APIRouter** и подключаться к основному приложению **FastAPI**. Вы также узнаете, что такое модели и как их использовать для проверки тела запроса. Вы также узнаете, что такое параметры пути и запроса и как их использовать в своем приложении FastAPI. Знание маршрутизации в FastAPI необходимо при создании малых и больших приложений.

В этой главе мы рассмотрим следующие темы:

- Маршрутизация в FastAPI
- Класс `APIRouter`
- Валидация с использованием моделей Pydantic
- Путь и параметры запроса
- Тело запроса
- Создание простого CRUD приложения

Технические требования

Код, использованный в этой главе, можно найти на сайте <https://github.com/PacktPublishing/Building-Python-Web-APIs-with-FastAPI/tree/main/ch02/todos>.

Понимание маршрутизации в FastAPI

Маршрут определяется для приема запросов от метода HTTP-запроса и, при необходимости, для получения параметров. Когда запрос отправляется на маршрут, приложение проверяет, определен ли маршрут перед обработкой запроса в обработчике маршрута. С другой стороны, обработчик маршрута — это функция, которая обрабатывает запрос, отправленный на сервер. Примером обработчика маршрута является функция, извлекающая записи из базы данных при отправке запроса на маршрутизатор через маршрут.

Что такое методы HTTP-запроса?

HTTP-методы — это идентификаторы для указания типа выполняемого действия. Стандартные методы включают GET, POST, PUT, PATCH, and DELETE. Вы можете узнать больше о методах HTTP на <https://developer.mozilla.org/en-US/docs/Web/HTTP/Methods>.

Пример маршрутизации

В разделе *Скаффолдинг проекта* в предыдущей главе мы создали приложение с одним маршрутом. Маршрутизация была обработана экземпляром `FastAPI()` инициализированным в переменной приложения:

```
from fastapi import FastAPI

app = FastAPI()

@app.get("/")
async def welcome() -> dict:
    return { "message": "Hello World"}
```

Инструмент **uvicorn** был направлен на экземпляр `FastAPI` для обслуживания приложения:

```
(venv)$ uvicorn api:app --port 8080 --reload
```


Традиционно экземпляр `FastAPI()` может использоваться для операций маршрутизации, как показано ранее. Однако этот метод обычно используется в приложениях, которым требуется один путь во время маршрутизации. В ситуации, когда с помощью экземпляра `FastAPI` создается отдельный маршрут, выполняющий уникальную функцию, приложение не сможет запустить оба маршрута, так как `uvicorn` может запустить только одну точку входа.

Как же тогда вы справляетесь с обширными приложениями, которые требуют серии маршрутов, выполняющих различные функции? В следующем разделе мы рассмотрим, как класс `APIRouter` помогает с множественной маршрутизацией.

Маршрутизация с помощью класса APIRouter

Класс `APIRouter` принадлежит пакету `FastAPI` и создает операции пути для нескольких маршрутов. Класс `APIRouter` поощряет модульность и организацию маршрутизации и логики приложений.

Класс `APIRouter` импортируется из пакета `fastapi` и создается экземпляр. Методы маршрута создаются и распространяются из созданного экземпляра, например:

```
from fastapi import APIRouter

router = APIRouter()

@router.get("/hello")
async def say_hello() -> dict:
    return {"message": "Hello!"}
```

Давайте создадим новую операцию пути с классом `APIRouter` для создания и получения задач. В папке `todos` из предыдущей главы создайте новый файл `todo.py`:

```
(venv)$ touch todo.py
```

Начнем с импорта класса `APIRouter` из пакета `fastapi` и создания экземпляра:

```
from fastapi import APIRouter

todo_router = APIRouter().
```

Далее мы создадим временную базу данных в приложении, а также два маршрута для добавления и извлечения задач:

```
todo_list = []

@todo_router.post("/todo")
async def add_todo(todo: dict) -> dict:
    todo_list.append(todo)
    return {"message": "Todo added successfully"}

@todo_router.get("/todo")
async def retrieve_todos() -> dict:
    return {"todos": todo_list}
```

В предыдущем блоке кода мы создали два маршрута для наших операций с задачами. Первый маршрут добавляет задачу в список задач с помощью метода POST, а второй маршрут извлекает все элементы задачи из списка задач с помощью метода GET.

Мы завершили операции пути для маршрута todo. Следующим шагом является передача приложения в производство, чтобы мы могли протестировать определенные операции пути.

Класс `APIRouter` работает так же, как и класс `FastAPI`. Однако `uvicorn` не может использовать экземпляр `APIRouter` для обслуживания приложения, в отличие от `FastAPIs`.

Маршруты, определенные с помощью класса `APIRouter`, добавляются в экземпляр `fastapi` для обеспечения их видимости.

Чтобы обеспечить видимость маршрутов todo, мы включим обработчик операций пути `todo_router` в основной экземпляр `FastAPI` с помощью метода `include_router()`.

include_router()

Метод `include_router(router, ...)` отвечает за добавление маршрутов, определенных с помощью класса `APIRouter`, в экземпляр основного приложения, чтобы сделать маршруты видимыми.

В `api.py`, `import todo_router` из `todo.py`:

```
from todo import todo_router
```

Включите `todo_router` в приложение FastAPI, используя метод `include_router` из экземпляра **FastAPI**:

```
from fastapi import FastAPI
from todo import todo_router

app = FastAPI()

@app.get("/")
async def welcome() -> dict:
    return {
        "message": "Hello World"
    }

app.include_router(todo_router)
```

Когда все на месте, запустите приложение с вашего терминала:

```
(venv)$ uvicorn api:app --port 8000 --reload
```

Предыдущая команда запускает наше приложение и дает нам журнал процессов нашего приложения в реальном времени:

```
(venv) → todos git:(main) X uvicorn api:app --port 8000
--reload
INFO: Will watch for changes in these directories: ['/Users/
youngestdev/Work/Building-Web-APIs-with-FastAPI-and-Python/
ch02/todos']
INFO:      uvicorn running on http://127.0.0.1:8000 (Press
CTRL+C to quit)
INFO:      Started reloader process [4732] using statreload
INFO:      Started server process [4734]
INFO:      Waiting for application startup.
INFO:      Application startup complete.
```

Следующий шаг — протестировать приложение, отправив запрос GET с помощью curl:

```
(venv)$ curl http://0.0.0.0:8080/
```

Ответ от приложения, в консоли:

```
{"message": "Hello World"}
```

Далее мы проверяем работоспособность todo-маршрутов:

```
(venv)$ curl -X 'GET' \
'http://127.0.0.1:8000/todo' \
-H 'accept: application/json'
```

Ответ от приложения в консоли должна быть следующим:

```
{
  "todos": []
}
```

Маршрут todo сработал! Давайте проверим операцию POST, отправив запрос на добавление элемента в наш список задач:

```
(venv)$ curl -X 'POST' \
'http://127.0.0.1:8000/todo' \
-H 'accept: application/json' \
-H 'Content-Type: application/json' \
-d '{
  "id": 1,
  "item": "First Todo is to finish this book!"
}'
```

Имеем следующий ответ:

```
{
  "message": "Todo added successfully."
}
```

Мы узнали, как работает класс `APIRouter` и как включить его в основной экземпляр приложения, чтобы разрешить использование определенных операций пути. В маршрутах todo, построенных в этом разделе, отсутствовали модели, также известные как схемы. В следующем разделе мы рассмотрим модели **Pydantic** и варианты их использования.

Валидация тела запроса с использованием моделей Pydantic

В FastAPI тела запросов могут быть проверены, чтобы гарантировать отправку только определенных данных. Это крайне важно, поскольку служит для очистки данных запросов и снижения рисков вредоносных атак. Этот процесс известен как валидация.

Модель в FastAPI — это структурированный класс, который определяет, как следует получать или анализировать данные. Модели создаются путем определения подкласса класса `BaseModel` Pydantic.

Что такое Pydantic?

Pydantic — это библиотека Python, которая выполняет проверку данных с помощью аннотаций типа Python.

Модели, когда они определены, используются в качестве подсказок типа для объектов тела запроса и объектов запроса-ответа. В этой главе мы рассмотрим только использование моделей Pydantic для тел запросов.

Примерная модель выглядит следующим образом:

```
from pydantic import BaseModel

class PacktBook(BaseModel):
    id: int
    Name: str
    Publishers: str
    Isbn: str
```

В предыдущем блоке кода выше мы определили модель `PacktBook` как подкласс класса `BaseModel` Pydantic. Тип переменной, подсказанный классу `PacktBook`, может принимать только четыре поля, как определено ранее. В следующих нескольких примерах мы видим, как Pydantic помогает в проверке входных данных.

В нашем приложении `todo` ранее мы определили маршрут для добавления элемента в список `todo`. В определении маршрута мы устанавливаем тело запроса в словарь:

```
async def add_todo(todo: dict) -> dict:
    ...
```

В примере запроса `POST` отправленные данные были в следующем формате:

```
{
    "id": id,
    "item": item
}
```

Однако пустой словарь также мог быть отправлен без возврата какой-либо ошибки. Пользователь может отправить запрос с телом, отличным от показанного ранее. Создание модели с требуемой структурой тела запроса и присвоение ее в качестве типа телу запроса гарантирует, что будут переданы только те поля данных, которые присутствуют в модели.

Например, чтобы в предыдущем примере поля содержались только в теле запроса, создайте новый файл `model.py` и добавьте в него приведенный ниже код:

```
from Pydantic import BaseModel

class Todo(BaseModel):
    id: int
    item: str
```

В предыдущем блоке кода мы создали модель `Pydantic`, которая принимает только два поля:

- `id` – целочисленное число
- `item` – строка

Давайте продолжим и используем модель в маршруте `POST`. В `api.py`, импортируйте модель:

```
from model import Todo
```

Затем замените тип переменной тела запроса с `dict` на `Todo`:

```
todo_list = []

@todo_router.post("/todo")
async def add_todo(todo: Todo) -> dict:
    todo_list.append(todo)
    return {"message": "Todo added successfully"}
```

```
@todo_router.get("/todo")
async def retrieve_todos() -> dict:
    return {"todos": todo_list}
```

Давайте проверим новый валидатор тела запроса, отправив пустой словарь в качестве тела запроса:

```
(venv)$ curl -X 'POST' \
  'http://127.0.0.1:8000/todo' \
  -H 'accept: application/json' \
  -H 'Content-Type: application/json' \
  -d '{
}'
```

Получаем ответ, указывающий на отсутствие поля `id` и `item` в теле запроса:

```
{
  "detail": [
    {
      "loc": [
        "body",
        "id"
      ],
      "msg": "field required",
      "type": "value_error.missing"
    },
    {
      "loc": [
        "body",
        "item"
      ],
      "msg": "field required",
      "type": "value_error.missing"
    }
  ]
}
```

Отправка запроса с правильными данными возвращает успешный ответ:

```
(venv)$ curl -X 'POST' \  
  'http://127.0.0.1:8000/todo' \  
  -H 'accept: application/json' \  
  -H 'Content-Type: application/json' \  
  -d '{  
    "id": 2,  
    "item": "Validation models help with input types"  
  }'
```

Вот ответ:

```
{  
  "message": "Todo added successfully."  
}
```

Вложенные модели

Модели Pydantic также могут быть вложенными, например, следующие:

```
class Item(BaseModel)  
    item: str  
    status: str  
  
class Todo(BaseModel)  
    id: int  
    item: Item
```

В результате задача типа `Todo` будет представлена следующим образом:

```
{  
  "id": 1,  
  "item": {  
    "item": "Nested models",  
    "Status": "completed"  
  }  
}
```


Мы узнали, что такое модели, как их создавать и как их использовать. Мы будем использовать его впоследствии в оставшихся частях этой книги. В следующем разделе рассмотрим параметры пути и запроса.

Путь и параметры запроса

В предыдущем разделе мы узнали, что такое модели и как они используются для проверки тела запроса. В этом разделе вы узнаете, что такое параметры пути и запроса, какую роль они играют в маршрутизации и как их использовать.

Параметры пути

Параметры пути — это параметры, включенные в маршрут API для идентификации ресурсов. Эти параметры служат идентификатором, а иногда и связующим звеном, позволяющим выполнять дальнейшие операции в веб-приложении.

В настоящее время у нас есть маршруты для добавления задачи и получения всех задач в нашем приложении задач. Давайте создадим новый маршрут для получения одной задачи, добавив идентификатор задачи в качестве параметра пути.

В `todo.py`, добавьте новый маршрут:

```
from fastapi import APIRouter, Path
from model import Todo

todo_router = APIRouter()

todo_list = []

@todo_router.post("/todo")
async def add_todo(todo: Todo) -> dict:
    todo_list.append(todo)
    return {
        "message": "Todo added successfully."
    }
```

```
@todo_router.get("/todo")
async def retrieve_todo() -> dict:
    return {
        "todos": todo_list
    }

@todo_router.get("/todo/{todo_id}")
async def get_single_todo(todo_id: int = Path(..., title="The
ID of the todo to retrieve. ")) -> dict:
    for todo in todo_list:
        if todo.id == todo_id:
            return {
                "todo": todo
            }
    return {
        "message": "Todo with supplied ID doesn't exist."
    }
```

В предыдущем блоке кода, {todo_id} является параметром пути. Этот параметр позволяет приложению возвращать совпадающую задачу с переданным идентификатором.

Проверим маршрут:

```
(venv)$ curl -X 'GET' \
    'http://127.0.0.1:8000/todo/1' \
    -H 'accept: application/json'
```

В предыдущем запросе GET request, 1 — это параметр пути. Здесь мы говорим нашему приложению todo вернуть элемент с идентификатором 1.

Выполнение предыдущего запроса приводит к следующему ответу:

```
{
  "todo": {
    "id": 1,
    "item": "First Todo is to finish this book!"
  }
}
```

FastAPI также предоставляет класс `Path`, который отличает параметры пути от других аргументов, присутствующих в функции маршрута. Класс `Path` также помогает дать параметрам маршрута больше контекста во время документации, автоматически предоставляемой OpenAPI через **Swagger** и **ReDoc**, и действует как валидатор.

Давайте изменим определение маршрута:

```
from fastAPI import APIRouter, Path

from model import Todo

todo_router = APIRouter()

todo_list = []

@todo_router.post("/todo")
async def add_todo(todo: Todo) -> dict:
    todo_list.append(todo)
    return {
        "message": "Todo added successfully."
    }

@todo_router.get("/todo")
async def retrieve_todo() -> dict:
    return {
        "todos": todo_list
    }

@todo_router.get("/todo/{todo_id}")
async def get_single_todo(todo_id: int = Path(..., title="The ID of the todo to retrieve")) -> dict:
    for todo in todo_list:
        if todo.id == todo_id:
            return {
                "todo": todo
            }
```

```

    }
    return {
        "message": "Todo with supplied ID doesn't exist."
    }

```

Подсказка – Путь(..., kwargs)

Класс `Path` принимает первый позиционный аргумент, равный `None` или многоточие (...). Если в качестве первого аргумента задано многоточие (...), параметр пути становится обязательным. Класс `Path` также содержит аргументы, используемые для числовой проверки, если параметр пути является числом. Определения включают `gt` и `le` – `gt` означает больше, а `le` означает меньше. При использовании маршрут будет проверять параметр пути на соответствие этим аргументам.

Параметры запроса

Параметр запроса — это необязательный параметр, который обычно появляется после вопросительного знака в URL-адресе. Он используется для фильтрации запросов и возврата определенных данных на основе предоставленных запросов.

В функции обработчика маршрута аргумент, не совпадающий с параметром пути, является запросом. Вы также можете определить запрос, создав экземпляр класса `FastAPI Query()` в аргументе функции, например:

```

async query_route(query: str = Query(None)):
    return query

```

Мы рассмотрим варианты использования параметров запроса позже в книге, когда будем обсуждать, как создавать более продвинутые приложения, чем приложение `todo`.

Теперь, когда вы узнали, как создавать маршруты, проверять тело запроса и использовать параметры пути и запроса в своем приложении FastAPI, в следующем разделе вы узнаете, как эти компоненты работают рука об руку для формирования тела запроса.

Тело запроса

В предыдущих разделах мы узнали, как использовать класс `APIRouter` и модели `Pydantic` для проверки тела запроса, а также обсудили пути и параметры запроса.

Тело запроса — это данные, которые вы отправляете в свой API, используя метод маршрутизации, такой как `POST` и `UPDATE`.

POST и UPDATE

Метод POST используется, когда необходимо выполнить вставку на сервер, а метод UPDATE используется, когда необходимо обновить существующие данные на сервере.

Давайте взглянем на запрос POST ранее в этой главе:

```
(venv)$ curl -X 'POST' \
  'http://127.0.0.1:8000/todo' \
  -H 'accept: application/json' \
  -H 'Content-Type: application/json' \
  -d '{
    "id": 2,
    "item": "Validation models help with input types"
  }'
```

В предыдущем запросе тело запроса выглядит следующим образом:

```
{
  "id": 2,
  "item": "Validation models help with input types.."
}
```

Совет

FastAPI также предоставляет класс `Body()` для дополнительной проверки.

Мы узнали о моделях в FastAPI. Они также служат дополнительной цели в документировании наших конечных точек API и типов тела запроса. В следующем подразделе мы узнаем о страницах документации, генерируемых по умолчанию в приложениях FastAPI.

Автоматические документы FastAPI

FastAPI генерирует определения схемы JSON для наших моделей и автоматически документирует наши маршруты, включая их тип тела запроса, параметры пути и запроса, а также модели ответов. Эта документация бывает двух типов:

- **Swagger**
- **ReDoc**

Swagger

Документация, размещенная на swagger, предоставляет интерактивную среду для тестирования нашего API. Вы можете получить к нему доступ, добавив `/docs` к адресу приложения. В веб-браузере перейдите по адресу `http://127.0.0.1:8000/docs`:

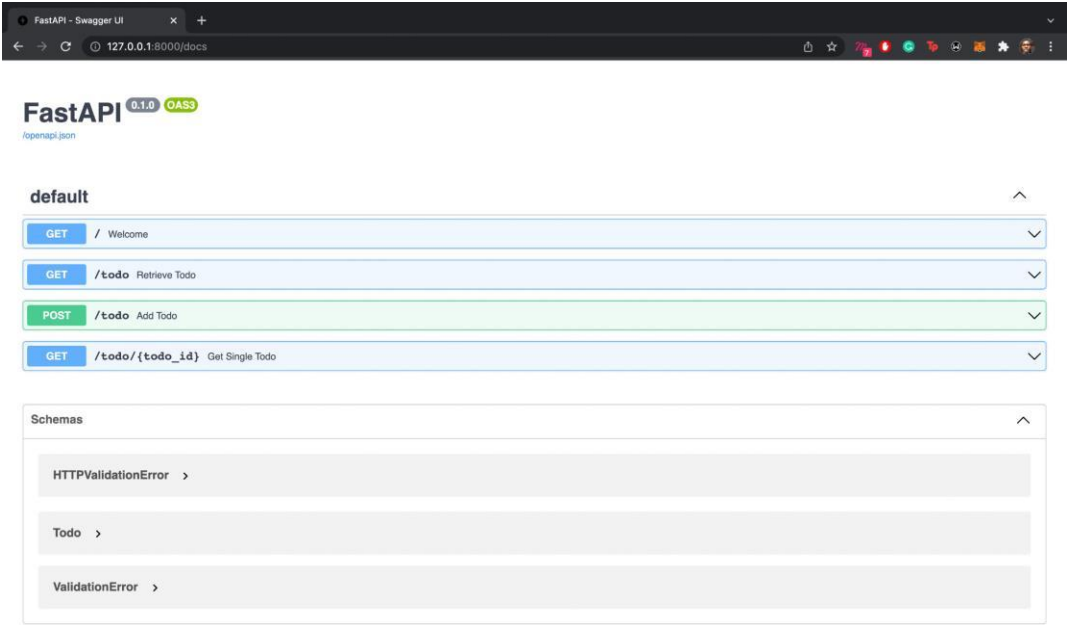


Рисунок 2.1 – Интерактивная документация FastAPI

Интерактивная документация позволяет нам тестировать наши методы. Добавим задачу из интерактивной документации:

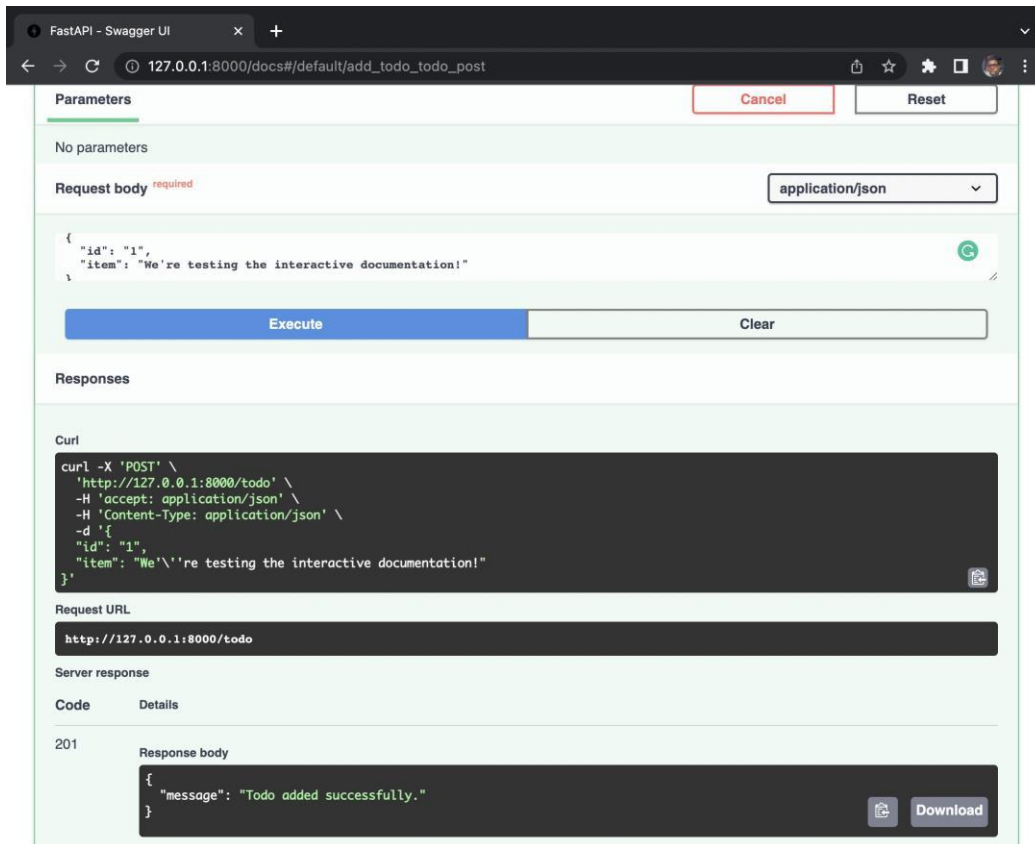


Рисунок 2.2 – Тестирование маршрутов из интерактивной документации

Теперь, когда мы знаем, как выглядит интерактивная документация, давайте проверим документацию, сгенерированную ReDoc.

ReDoc

Документация ReDoc дает более подробное и прямое представление о моделях, маршрутах и API. Вы можете получить к нему доступ, добавив `/redoc` к адресу приложения. В веб-браузере перейдите по адресу `http://127.0.0.1:8000/redoc`:

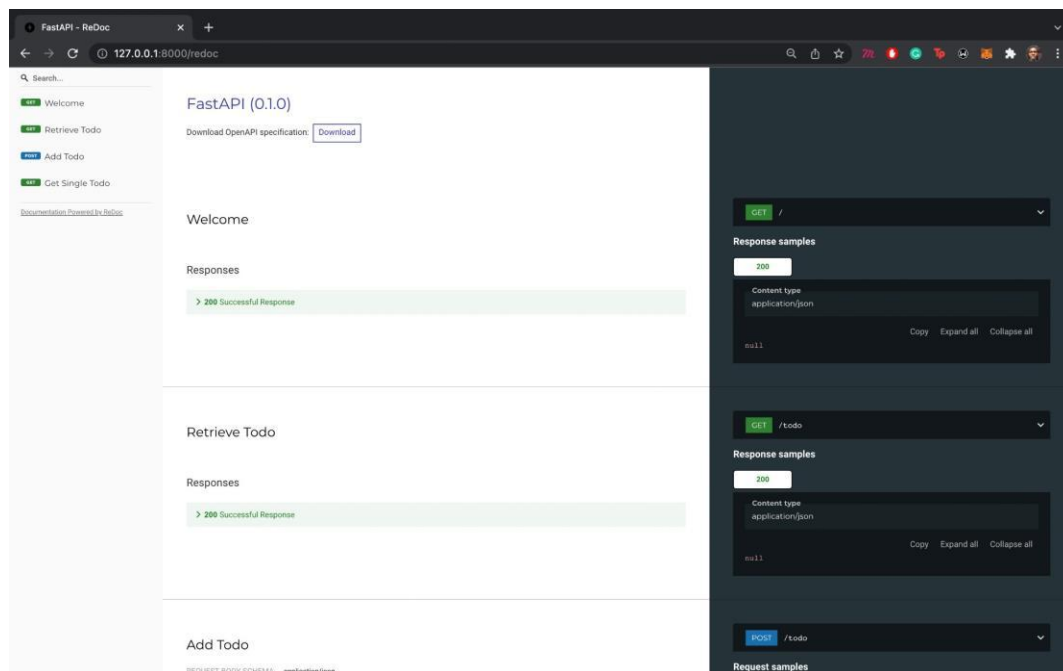


Рисунок 2.3 – Портал документации на базе ReDoc

Чтобы правильно сгенерировать **JSON** схему, вы можете указать примеры того, как пользователь будет заполнять данные в модели. Пример задается путем встраивания класса `Config` в класс модели. Давайте добавим пример схемы в нашу модель `Todo`:

```
class Todo(BaseModel):
    id: int
    item: str

    class Config:
        Schema_extra = {
            "Example": {
                "id": 1,
```



```
"item": "Example schema!"
}
```

Обновите страницу документации для ReDoc и нажмите **Добавить задачу** на левой панели. Пример показан на правой панели:

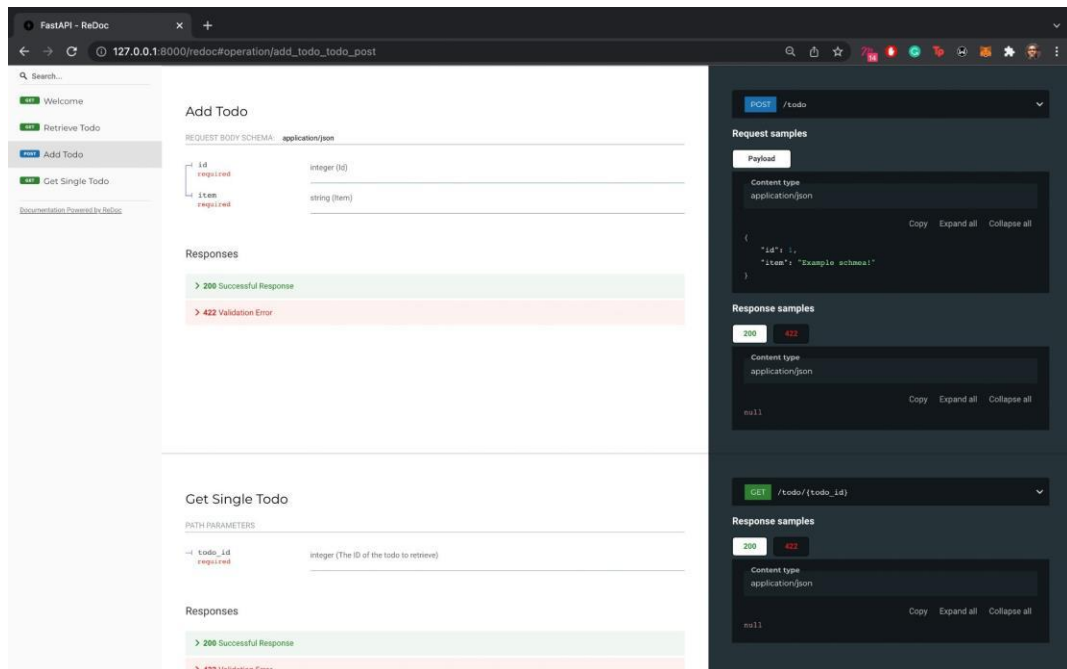


Рисунок 2.4 – Портал документации показывает примерную схему

Также в интерактивной документации пример схемы можно увидеть:

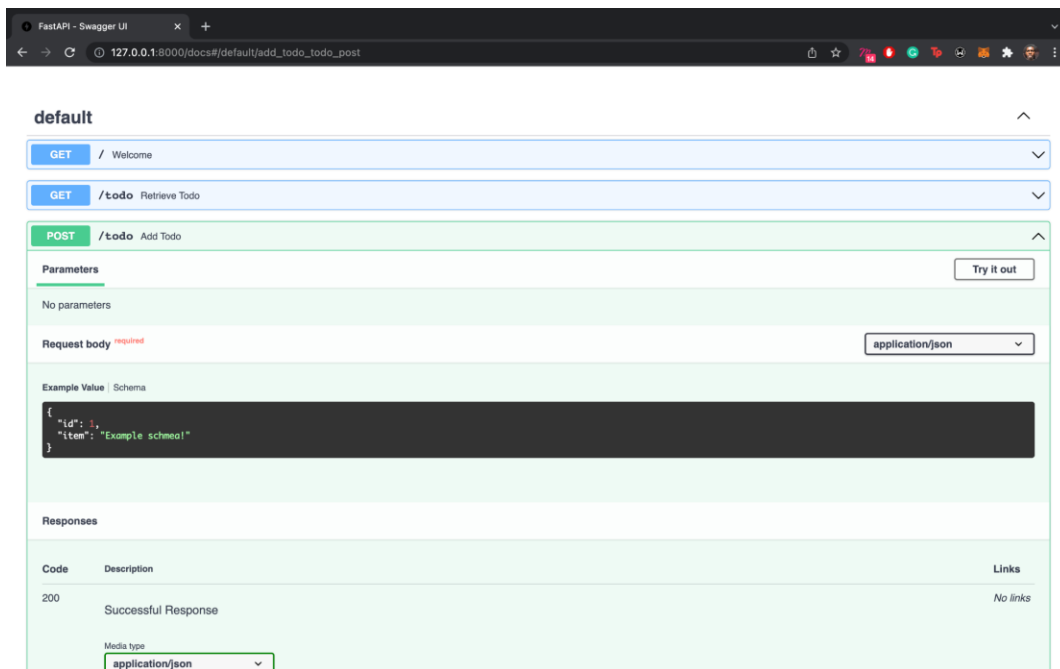


Рисунок 2.5 – Портал документации показывает примерную схему

Мы узнали, как добавить примеры данных схемы, чтобы помочь пользователям отправлять запросы к API и тестировать приложение из интерактивной документации Swagger. Документация, предоставляемая ReDoc, не осталась без внимания, поскольку она служит базой знаний о том, как использовать API.

Теперь, когда мы узнали, что такое класс `APIRouter` и как его использовать, тело запроса, параметры пути и запроса, а также проверку тела запроса с помощью моделей Pydantic, давайте обновим наше приложение `todo`, включив в него маршруты для обновления и удаления элемента `todo`.

Создание простого CRUD-приложения

Мы создали маршруты для **создания** и **получения** задач. Построим маршруты для **обновления** и **удаления** добавленных задач. Начнем с создания модели тела запроса для маршрута UPDATE в `model.py`:

```
class TodoItem(BaseModel):
    item: str

    class Config:
        schema_extra = {
            "example": {
                "item": "Read the next chapter of the book"
            }
        }
```

Далее давайте напишем маршрут для обновления задачи в `todo.py`:

```
from fastapi import APIRouter, Path
from model import Todo, TodoItem

todo_router = APIRouter()

todo_list = []

@todo_router.post("/todo")
async def add_todo(todo: Todo) -> dict:
    todo_list.append(todo)
    return {
        "message": "Todo added successfully."
    }

@todo_router.get("/todo")
async def retrieve_todo() -> dict:
    return {
        "todos": todo_list
```

```
}

@todo_router.get("/todo/{todo_id}")
async def get_single_todo(todo_id: int = Path(..., title="The
ID of the todo to retrieve")) -> dict:
    for todo in todo_list:
        if todo.id == todo_id:
            return {
                "todo": todo
            }
    return {
        "message": "Todo with supplied ID doesn't exist."
    }

@todo_router.put("/todo/{todo_id}")
async def update_todo(todo_data: TodoItem, todo_id: int =
Path(..., title="The ID of the todo to be updated")) -> dict:
    for todo in todo_list:
        if todo.id == todo_id:
            todo.item = todo_data.item
            return {
                "message": "Todo updated successfully."
            }
    return {
        "message": "Todo with supplied ID doesn't exist."
    }
```

Протестируем новый маршрут. Во-первых, давайте добавим задачу:

```
(venv)$ curl -X 'POST' \
  'http://127.0.0.1:8000/todo' \
  -H 'accept: application/json' \
  -H 'Content-Type: application/json' \
  -d '{
    "id": 1,
```

```
"item": "Example Schema!"
}'
```

Вот ответ:

```
(venv)$ {
  "message": "Todo added successfully."
}
```

Далее давайте обновим задачу, отправив запрос PUT:

```
(venv)$ curl -X 'PUT' \
'http://127.0.0.1:8000/todo/1' \
-H 'accept: application/json' \
-H 'Content-Type: application/json' \
-d '{
  "item": "Read the next chapter of the book."
}'
```

Вот ответ:

```
(venv)$ {
  "message": "Todo updated successfully."
}
```

Давайте проверим, что наша задача действительно была обновлена:

```
(venv)$ curl -X 'GET' \
'http://127.0.0.1:8000/todo/1' \
-H 'accept: application/json'
```

Вот ответ:

```
(venv)$ {
  "todo": {
    "id": 1,
    "item": "Read the next chapter of the book"
  }
}
```

Из возвращенного ответа мы видим, что задача успешно обновлена. Теперь давайте создадим маршрут для удаления задачи и всех задач.

В `todo.py`, обновите маршруты:

```
@todo_router.delete("/todo/{todo_id}")
async def delete_single_todo(todo_id: int) -> dict:
    for index in range(len(todo_list)):
        todo = todo_list[index]
        if todo.id == todo_id:
            todo_list.pop(index)
            return {
                "message": "Todo deleted successfully."
            }
    return {
        "message": "Todo with supplied ID doesn't exist."
    }

@todo_router.delete("/todo")
async def delete_all_todo() -> dict:
    todo_list.clear()
    return {
        "message": "Todos deleted successfully."
    }
```

Давайте протестируем маршрут удаления. Сначала мы добавляем задачу:

```
(venv)$ curl -X 'POST' \
  'http://127.0.0.1:8000/todo' \
  -H 'accept: application/json' \
  -H 'Content-Type: application/json' \
  -d '{
    "id": 1,
    "item": "Example Schema!"
  }'
```

Вот ответ:

```
(venv)$ {  
  "message": "Todo added successfully."  
}
```

Затем удалите задачу:

```
(venv)$ curl -X 'DELETE' \  
  'http://127.0.0.1:8000/todo/1' \  
  -H 'accept: application/json'
```

Вот ответ:

```
(venv)$ {  
  "message": "Todo deleted successfully."  
}
```

Давайте проверим, что задача была удалена, отправив запрос GET для получения задачи:

```
(venv)$ curl -X 'GET' \  
  'http://127.0.0.1:8000/todo/1' \  
  -H 'accept: application/json'
```

Вот ответ:

```
(venv)$ {  
  "message": "Todo with supplied ID doesn't exist."  
}
```

В этом разделе мы создали приложение с CRUD операциями, объединив уроки, извлеченные из предыдущих разделов. Подтвердив тело запроса, мы смогли убедиться, что в API отправляются правильные данные. Включение параметров пути в наши маршруты также позволило нам получить и удалить одну задачу из нашего списка задач.

Резюме

В этой главе мы узнали, как использовать класс `APIRouter` и подключать маршруты, определенные с его помощью, к основному экземпляру FastAPI. Мы также узнали, как создавать модели для наших тел запросов и добавлять параметры пути и запроса к нашим операциям пути. Эти модели служат дополнительной проверкой неправильных типов данных, предоставляемых для полей тела запроса. Мы также создали приложение с CRUD операциями, чтобы применить на практике все, что мы узнали в этой главе.

В следующей главе вы познакомитесь с ответами, моделированием ответов и обработкой ошибок в FastAPI. Сначала вы познакомитесь с концепцией ответов и тем, как знания о моделях Pydantic, полученные в этой главе, помогают создавать модели для ответов API. Затем вы узнаете о кодах состояния и о том, как их использовать в объектах ответа, а также о правильной обработке ошибок.

Модели ответов и обработка ошибок

Модели ответов служат шаблонами для возврата данных из пути маршрута API. Они построены на **Pydantic** для правильной обработки ответов на запросы, отправленные на сервер.

Обработка ошибок включает методы и действия, связанные с обработкой ошибок в приложении. Эти методы включают возврат адекватных кодов состояния ошибки и сообщений об ошибках.

К концу этой главы вы будете знать, что такое ответ и из чего он состоит, а также будете знать об обработке ошибок и о том, как обрабатывать ошибки в вашем приложении FastAPI. Вы также узнаете, как создавать модели ответов на запросы с помощью Pydantic.

В этой главе мы рассмотрим следующие темы:

- Ответы в FastAPI
- Построение модели ответа
- Обработка ошибок

Технические требования

Код, использованный в этой главе, можно найти по адресу <https://github.com/PacktPublishing/Building-Python-Web-APIs-with-FastAPI/tree/main/ch03/todos>.

Понимание ответов в FastAPI

Ответы являются неотъемлемой частью жизненного цикла API. Ответы — это отзывы, полученные при взаимодействии с маршрутом API с помощью любого из стандартных методов HTTP. Ответ API обычно предоставляется в формате JSON или XML, но также может быть в форме документа. Ответ состоит из заголовка и тела.

Что такое заголовок ответа?

Заголовок ответа состоит из статуса запроса и дополнительной информации, необходимой для доставки тела ответа. Примером информации, содержащейся в заголовке ответа, является `Content-Type`, который сообщает клиенту возвращаемый тип содержимого.

Что такое тело ответа?

Тело ответа, с другой стороны, представляет собой данные, запрошенные клиентом с сервера. Тело ответа определяется из переменной заголовка `Content-Type` наиболее часто используемой является `application/json`. В предыдущей главе возвращаемый список задач был телом ответа.

Теперь, когда вы узнали, что такое ответы и из чего они состоят, давайте взглянем на коды состояния HTTP, включенные в ответы в следующем разделе.

Коды состояния

Коды состояния — это уникальные короткие коды, выдаваемые сервером в ответ на запрос клиента. Коды состояния ответа сгруппированы в пять категорий, каждая из которых обозначает отдельный ответ:

- 1XX: Запрос получен.
- 2XX: Запрос выполнен успешно.
- 3XX: Запрос перенаправлен.
- 4XX: Ошибка клиента.
- 5XX: Ошибка сервера.

Полный список кодов состояния HTTP можно найти по адресу <https://httpstatuses.com/>.

Первая цифра кода состояния определяет его категорию. Общие коды состояния включают 200 для успешного запроса, 404 для запроса, не найденного и 500 указывающего на внутреннюю ошибку сервера.

Стандартная практика создания веб-приложений, независимо от платформы, заключается в возврате соответствующих кодов состояния для отдельных событий. Код состояния 400 не должен возвращаться в случае ошибки сервера. Точно так же код состояния 200 не должен возвращаться для неудачной операции запроса.

Теперь, когда вы узнали, что такое коды состояния, давайте научимся строить модели ответов в следующем разделе.

Построение моделей ответа

В начале этой главы мы установили назначение моделей отклика. В предыдущей главе вы также узнали, как создавать модели с помощью Pydantic. Модели реагирования также построены на Pydantic, но служат другой цели.

В определении путей маршрута мы имеем, например, следующее:

```
@app.get("/todo")
async def retrieve_todo() -> dict:
    return {
        "todos": todo_list
    }
```

Маршрут возвращает список задач, присутствующих в базе данных. Вот пример вывода:

```
{
  "todos": [
    {
      "id": 1,
      "item": "Example schema 1!"
    },
    {
      "id": 2,
      "item": "Example schema 2!"
    },
    {
      "id": 3,
```

```

        "item": "Example schema 5!"
    }
]
}

```

Маршрут возвращает весь контент, хранящийся в массиве `todos`. Чтобы указать возвращаемую информацию, нам пришлось бы либо отделить отображаемые данные, либо ввести дополнительную логику. К счастью, мы можем создать модель, содержащую поля, которые мы хотим вернуть, и добавить ее в определение нашего маршрута, используя аргумент `response_model`.

Давайте обновим маршрут, который извлекает все задачи, чтобы он возвращал массив только элементов задач, а не идентификаторов. Начнем с определения нового класса модели для возврата списка дел в `model.py`:

```

from typing import List

class TodoItem(BaseModel):
    item: str

    class Config:
        schema_extra = {
            "example": {
                "item": "Read the next chapter of the book"
            }
        }

class TodoItems(BaseModel):
    todos: List[TodoItem]

    class Config:
        schema_extra = {
            "example": {
                "todos": [
                    {
                        "item": "Example schema 1!"
                    },
                    {
                        "item": "Example schema 2!"
                    }
                ]
            }
        }

```

```
        }  
    ]  
}  
}
```

В предыдущем блоке кода мы определили новую модель, `TodoItems`, которая возвращает список переменных, содержащихся в модели `TodoItem`. Давайте обновим наш маршрут в `todo.py` добавив в него модель ответа:

```
from model import Todo, TodoItem, TodoItems  
...  
@todo_router.get("/todo", response_model=TodoItems)  
async def retrieve_todo() -> dict:  
    return {  
        "todos": todo_list  
    }
```

Активируйте виртуальную среду и запустите приложение:

```
$ source venv/bin/activate  
(venv)$ uvicorn api:app --host=0.0.0.0 --port 8000 --reload
```

Затем добавьте новую задачу:

```
(venv)$ curl -X 'POST' \  
  'http://127.0.0.1:8000/todo' \  
  -H 'accept: application/json' \  
  -H 'Content-Type: application/json' \  
  -d '{  
    "id": 1,  
    "item": "This todo will be retrieved without exposing my  
      ID!"  
  }'
```

Получить список дел:

```
(venv)$ curl -X 'GET' \
  'http://127.0.0.1:8000/todo' \
  -H 'accept: application/json'
```

Полученный ответ выглядит следующим образом:

```
{
  "todos": [
    {
      "item": " This todo will be retrieved without
        exposing my ID!"
    }
  ]
}
```

Теперь, когда мы узнали, что такое модели реагирования и как их использовать, мы продолжим использовать их там, где они подходят, в последующих главах. Давайте рассмотрим ответы на ошибки и способы обработки ошибок в следующем разделе.

Обработка ошибок

Ранее в этой главе мы узнали, что такое коды состояния и как они полезны для информирования клиента о состоянии запроса. Запросы могут возвращать ошибочные ответы, и эти ответы могут быть некрасивыми или содержать недостаточную информацию о причине сбоя.

Ошибки запросов могут возникать из-за попыток доступа к несуществующим ресурсам, защищенным страницам без достаточных разрешений и даже из-за ошибок сервера. Ошибки в FastAPI обрабатываются путем создания исключения с использованием класса FastAPI's `HTTPException`.

Что такое исключение HTTP?

Исключение HTTP — это событие, которое используется для указания на ошибку или проблему в потоке запросов.

Класс `HTTPException` принимает три аргумента:

- `status_code`: Код состояния, который будет возвращен для этого сбоя
- `detail`: Сопроводительное сообщение для отправки клиенту
- `headers`: Необязательный параметр для ответов, требующих заголовков

В наших определениях пути маршрута задачи мы возвращаем сообщение, когда задача не может быть найдена. Мы будем обновлять его, чтобы вызывать `HTTPException`. `HTTPException` позволяет нам вернуть адекватный код ответа на ошибку.

В нашем текущем приложении получение несуществующей задачи возвращает код статуса ответа 200 вместо кода статуса ответа 404 на `http://127.0.0.1:8000/docs`:

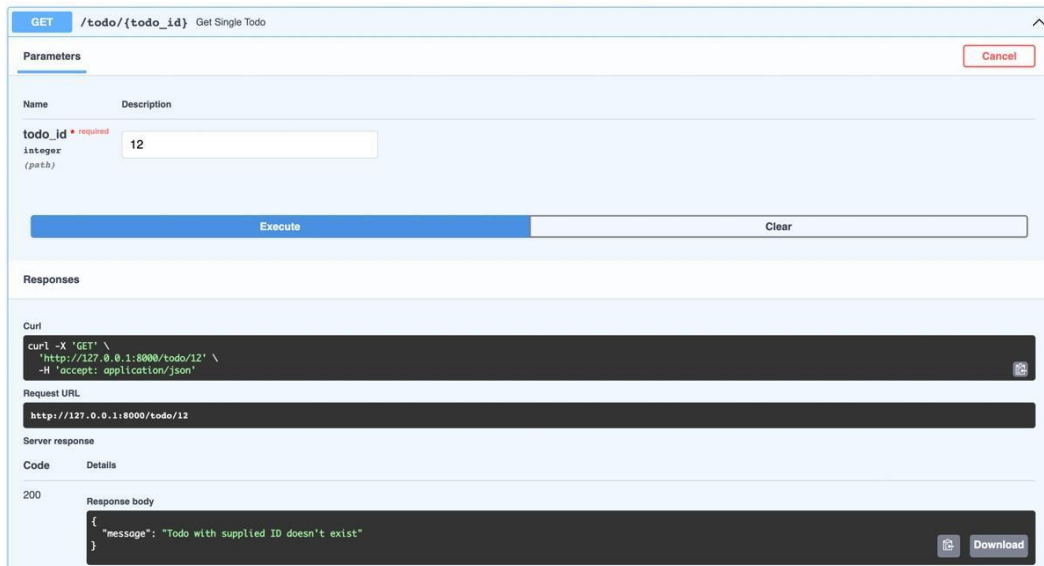


Рисунок 3.1 – Запрос возвращает ответ 200 вместо ответа 404

Обновляя маршруты для использования класса `HTTPException`, мы можем возвращать соответствующие детали в нашем ответе. В `todo.py`, обновите маршруты для получения, обновления и удаления списка дел:

```
from fastapi import APIRouter, Path, HTTPException, status
...
@todo_router.get("/todo/{todo_id}")
async def get_single_todo(todo_id: int = Path(..., title="The
ID of the todo to retrieve.")-> dict:
    for todo in todo_list:
```

```
        if todo.id == todo_id:
            return {
                "todo": todo
            }
        raise HTTPException(
            status_code=status.HTTP_404_NOT_FOUND,
            detail="Todo with supplied ID doesn't exist",
        )

@todo_router.put("/todo/{todo_id}")
async def update_todo(todo_data: TodoItem, todo_id: int =
    Path(..., title="The ID of the todo to be updated.)) -> dict:
    for todo in todo_list:
        if todo.id == todo_id:
            todo.item = todo_data.item
            return {
                "message": "Todo updated successfully."
            }

        raise HTTPException(
            status_code=status.HTTP_404_NOT_FOUND,
            detail="Todo with supplied ID doesn't exist",
        )

@todo_router.delete("/todo/{todo_id}")
async def delete_single_todo(todo_id: int) -> dict:
    for index in range(len(todo_list)):
        todo = todo_list[index]
        if todo.id == todo_id:
            todo_list.pop(index)
            return {
                "message": "Todo deleted successfully."
            }

        raise HTTPException(
            status_code=status.HTTP_404_NOT_FOUND,
```



```

    detail="Todo with supplied ID doesn't exist",
)

```

Теперь давайте повторим попытку получения несуществующей задачи, чтобы убедиться, что возвращается правильный код ответа:

Responses

Curl

```
curl -X 'GET' \
'http://127.0.0.1:8000/todo/12' \
-H 'accept: application/json'
```

Request URL

```
http://127.0.0.1:8000/todo/12
```

Server response

Code	Details
404	<p><i>Undocumented</i> Error: Not Found</p> <p>Response body</p> <pre>{ "detail": "Todo with supplied ID doesn't exist" }</pre> <p>Response headers</p>

Рисунок 3.2 – Отображается правильный код ответа 404

Наконец, мы можем объявить код состояния HTTP для переопределения кода состояния по умолчанию для успешных операций, добавив аргумент `status_code` в функцию декоратора:

```

@todo_router.post("/todo", status_code=201)
async def add_todo(todo: Todo) -> dict:
    todo_list.append(todo)
    return {
        "message": "Todo added successfully."
    }

```

В этом разделе мы узнали, как возвращать правильные коды ответов клиентам, а также переопределять код состояния по умолчанию. Также важно отметить, что код состояния успешный по умолчанию — 200.

Резюме

В этой главе мы узнали, что такое ответы и модели ответов и что подразумевается под обработкой ошибок. Мы также узнали о кодах состояния HTTP и о том, почему важно их использовать.

Мы также создали модели ответа на основе знаний, которые мы получили о создании моделей из предыдущей главы, и создали модель ответа, чтобы возвращать только элементы в списке дел без их идентификаторов. Наконец, мы узнали об ошибках и обработке ошибок. Мы обновили наши существующие маршруты, чтобы они возвращали правильный код ответа вместо стандартного кода состояния 200.

В следующей главе вы познакомитесь с созданием шаблонов приложений FastAPI с помощью Jinja. Сначала вы познакомитесь с основами, необходимыми для запуска и работы с шаблонами Jinja, после чего вы создадите пользовательский интерфейс, используя свои знания в области шаблонов для нашего простого приложения.

4

Шаблоны в FastAPI

Теперь, когда мы узнали, как обрабатывать ответы на запросы, включая ошибки в предыдущей главе, мы можем приступить к отображению ответов на запросы на веб-странице. В этой главе мы узнаем, как отображать ответы от нашего API на веб-странице, используя шаблоны на основе **Jinja**, который представляет собой язык шаблонов, написанный на Python, предназначенный для облегчения процесса визуализации ответов API.

Шаблонирование — это процесс отображения данных, полученных от API, в различных форматах. Шаблоны действуют как компонент интерфейса в веб-приложениях.

К концу этой главы вы будете владеть знаниями о том, что такое шаблоны и как использовать шаблоны для рендеринга информации из вашего API. В этой главе мы рассмотрим следующие темы:

- Понимание Jinja
- Использование шаблонов Jinja2 в FastAPI

Технические требования

Код, использованный в этой главе, можно найти по адресу <https://github.com/PacktPublishing/Building-Python-Web-APIs-with-FastAPI/tree/main/ch04/todos>.

Понимание Jinja

Jinja — это механизм шаблонов, написанный на Python, предназначенный для облегчения процесса рендеринга ответов API. В каждом языке шаблонов есть переменные, которые заменяются фактическими значениями, переданными им при отображении шаблона, и есть теги, управляющие логикой шаблона.

Механизм шаблонов Jinja использует фигурные скобки `{ }`, чтобы отличить свои выражения и синтаксис от обычного HTML, текста и любой другой переменной в файле шаблона.

Синтаксис `{{ }}` называется **блоком переменных**. Синтаксис `{% %}` содержит управляющие структуры, такие как **if/else**, **циклы** и **макросы**.

Три общих синтаксических блока, используемых в языке шаблонов Jinja, включают следующее:

- `{% ... %}` – Этот синтаксис используется для операторов, таких как управляющие структуры.
- `{{ todo.item }}` – Этот синтаксис используется для вывода значений переданных ему выражений.
- `{# This is a great API book! #}` – Этот синтаксис используется при написании комментариев и не отображается на веб-странице.

Переменные шаблона Jinja могут относиться к любому типу или объекту Python, если их можно преобразовать в строки. Тип модели, списка или словаря можно передать шаблону и отобразить его атрибуты, поместив эти атрибуты во второй блок, указанный ранее.

В следующем разделе мы рассмотрим фильтры. Фильтры являются важной частью каждого механизма шаблонов, и в Jinja фильтры позволяют нам выполнять определенные функции, такие как объединение значений из списка и получение длины объекта, среди прочего.

В следующих подразделах мы рассмотрим некоторые общие функции, используемые в Jinja: фильтры, операторы if, циклы, макросы и наследование шаблонов.

Фильтры

Несмотря на сходство синтаксиса Python и Jinja, такие модификации, как объединение строк, установка первого символа строки в верхний регистр и т. д., не могут быть выполнены с использованием синтаксиса Python в Jinja. Поэтому для выполнения таких модификаций у нас в Jinja есть фильтры.

Фильтр отделяется от переменной вертикальной чертой (`|`) и может содержать необязательные аргументы в круглых скобках. Фильтр определяется в этом

формате:

```
{{ variable | filter_name(*args) }}
```

Если нет аргументов, определение становится следующим:

```
{{ variable | filter_name }}
```

Давайте рассмотрим некоторые распространенные фильтры в следующих подразделах.

Фильтр по умолчанию

Переменная фильтра по умолчанию используется для замены вывода переданного значения, если оно оказывается None:

```
{{ todo.item | default('This is a default todo item') }}  
This is a default todo item
```

Эвакуационный фильтр

Этот фильтр используется для отображения необработанного вывода HTML:

```
{{ "<title>Todo Application</title>" | escape }}  
<title>Todo Application</title>
```

Фильтры преобразования

Эти фильтры включают фильтры `int` и `float`, используемые для преобразования из одного типа данных в другой:

```
{{ 3.142 | int }}  
3  
{{ 31 | float }}  
31.0
```

Фильтр объединения

Этот фильтр используется для объединения элементов списка в строку, как в Python:

```
{{ ['Packt', 'produces', 'great', 'books!'] | join(' ') }}  
Packt produces great books!
```

Фильтр длины

Этот фильтр используется для возврата длины переданного объекта. Он выполняет ту же роль, что и `len()` в Python:

```
Todo count: {{ todos | length }}
Todo count: 4
```

Примечание

Полный список фильтров и дополнительные сведения о фильтрах в Jinja см. на странице <https://jinja.palletsprojects.com/en/3.0.x/templates/#builtin-filters>.

Использование операторов if

Использование операторов if в Jinja аналогично их использованию в Python. if операторы используются в блоках управления {% %}. Давайте посмотрим на пример:

```
{% if todo | length < 5 %}
    You don't have much items on your todo list!
{% else %}
    You have a busy day it seems!
{% endif %}
```

Циклы

Мы также можем перебирать переменные в Jinja. Это может быть список или общая функция, например, следующая:

```
{% for todo in todos %}
    <b> {{ todo.item }} </b>
{% endfor %}
```

Вы можете получить доступ к специальным переменным внутри цикла for, таким как loop.index, который дает индекс текущей итерации. Ниже приведен список специальных переменных и их описания:

Variable	Description
loop.index	The current iteration of the loop (1 indexed)
loop.index0	The current iteration of the loop (0 indexed)
loop.revindex	The number of iterations from the end of the loop (1 indexed)
loop.revindex0	The number of iterations from the end of the loop (0 indexed)
loop.first	True if first iteration

Variable	Description
<code>loop.last</code>	True if last iteration
<code>loop.length</code>	The number of items in the sequence
<code>loop.cycle</code>	A helper function to cycle between a list of sequences
<code>loop.depth</code>	Indicates how deep in a recursive loop the rendering currently is; starts at level 1
<code>loop.depth0</code>	Indicates how deep in a recursive loop the rendering currently is; starts at level 0
<code>loop.previtem</code>	The item from the previous iteration of the loop; undefined during the first iteration
<code>loop.nextitem</code>	The item from the following iteration of the loop; undefined during the last iteration
<code>loop.changed(*val)</code>	True if previously called with a different value (or not called at all)

Макросы

Макрос в Jinja — это функция, которая возвращает строку HTML. Основной вариант использования макросов — избежать повторения кода и вместо этого использовать один вызов функции. Например, макрос ввода определен для сокращения непрерывного определения тегов ввода в HTML-форме:

```
{% macro input(name, value='', type='text', size=20 %)
    <div class="form">
        <input type="{{ type }}" name="{{ name }}"
            value="{{ value|escape }}" size="{{ size }}">
    </div>
{% endmacro %}
```

Теперь, чтобы быстро создать ввод в вашей форме, вызывается макрос:

```
{{ input('item') }}
```

Это вернет следующее:

```
<div class="form">
    <input type="text" name="item" value="" size="20">
</div>
```

Теперь, когда мы узнали, что такое макросы, мы приступим к изучению того, что такое наследование шаблонов и как оно работает в FastAPI.

Наследование шаблонов

Самая мощная функция Jinja — наследование шаблонов. Эта функция продвигает принцип «не повторяйся» (DRY) и удобна в больших веб-приложениях. Наследование шаблона — это ситуация, когда базовый шаблон определен, а дочерние шаблоны могут взаимодействовать, наследовать и заменять определенные разделы базового шаблона.

Примечание

Вы можете узнать больше о наследовании шаблонов Jinja на <https://jinja.palletsprojects.com/en/3.0.x/templates/#template-inheritance>.

Теперь, когда вы изучили основы синтаксиса Jinja, давайте научимся использовать шаблоны в FastAPI в следующем разделе.

Использование шаблонов Jinja в FastAPI

Для начала нам нужно установить пакет Jinja и создать новую папку, `templates`, в каталоге нашего проекта. В этой папке будут храниться все наши файлы Jinja, которые представляют собой файлы HTML, смешанные с синтаксисом Jinja. Поскольку эта книга не посвящена дизайну пользовательского интерфейса, мы будем использовать библиотеку CSS Bootstrap и не будем писать собственные стили.

Библиотека Bootstrap будет загружена из CDN при загрузке страницы. Однако дополнительные активы можно хранить в другой папке. Мы рассмотрим обслуживание статических файлов в следующей главе.

Мы начнем с создания шаблона домашней страницы, на котором будет размещен раздел для создания новых задач. Ниже приведен макет того, как мы хотим, чтобы наш шаблон домашней страницы выглядел:

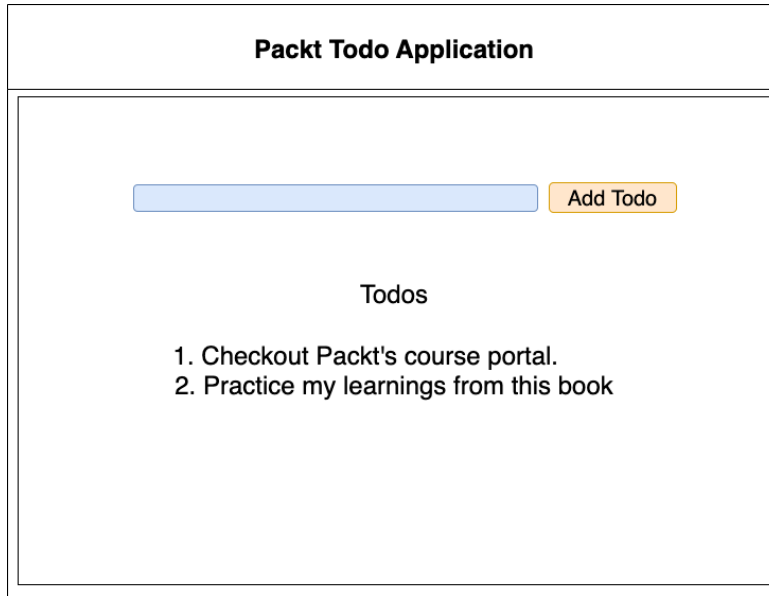


Рисунок 4.1 – Макет нашего шаблона домашней страницы

1. Во-первых, давайте установим пакет Jinja и создадим папку `templates`:

```
(venv)$ pip install jinja2
(venv)$ mkdir templates
```

2. Во вновь созданной папке создайте два новых файла, `home.html` и `todo.html`:

```
(venv)$ cd templates
(venv)$ touch {home,todo}.html
```

В предыдущем командном блоке мы создали два файла шаблона:

- `home.html` для главной страницы приложения
- `todo.html` для страницы задач

В макете на *Рисунке 4.1*, внутреннее поле обозначает шаблон `todo`, а большее поле — шаблон домашней страницы.

Прежде чем перейти к созданию наших шаблонов, давайте настроим Jinja в нашем приложении FastAPI:

1. Давайте изменим POST маршрут компонента API задач, `todo.py`:

```
from fastapi import APIRouter, Path, HTTPException,
status, Request, Depends
from fastapi.templating import Jinja2Templates
from model import Todo, TodoItem, TodoItems

todo_router = APIRouter()

todo_list = []

templates = Jinja2Templates(directory="templates/")

@todo_router.post("/todo")
async def add_todo(request: Request, todo: Todo =
Depends(Todo.as_form)):
    todo.id = len(todo_list) + 1
    todo_list.append(todo)
    return templates.TemplateResponse("todo.html",
    {
        "request": request,
        "todos": todo_list
    })
```

2. Затем обновите маршруты GET:

```
@todo_router.get("/todo", response_model=TodoItems)
async def retrieve_todo(request: Request):
    return templates.TemplateResponse("todo.html", {
        "request": request,
        "todos": todo_list
    })

@todo_router.get("/todo/{todo_id}")
async def get_single_todo(request: Request, todo_id: int
= Path(..., title="The ID of the todo to retrieve.")):
```

```

for todo in todo_list:
    if todo.id == todo_id:
        return templates.TemplateResponse(
            "todo.html", {
                "request": request,
                "todo": todo
            })
raise HTTPException(
    status_code=status.HTTP_404_NOT_FOUND,
    detail="Todo with supplied ID doesn't exist",
)

```

В предыдущем блоке кода мы настроили Jinja для просмотра каталога `templates` для обслуживания шаблонов, переданных в `templates`.

Метод `TemplateResponse()`.

Метод для добавления задачи также был обновлен, чтобы включить зависимость от переданного ввода. Зависимости будут подробно обсуждаться в *Главе 6 «Подключение к базе данных»*.

3. В `model.py`, добавьте выделенный код перед классом `Config`:

```

from typing import List, Optional

```

```

class Todo(BaseModel):
    id: Optional[int]
    item: str

    @classmethod
    def as_form(
        cls,
        item: str = Form(...)
    ):
        return cls(item=item)

```

Теперь, когда мы обновили наш код API, давайте напомним наши шаблоны. Мы начнем с написания базового шаблона `home.html` на следующем шаге.

4. В `home.html`, мы начнем с объявления типа документа:

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8">
    <meta http-equiv="X-UA-Compatible"
      content="IE=edge">
    <meta name="viewport" content="width=device-
      width, initial-scale=1.0">
    <title>Packt Todo Application</title>
    <link rel="stylesheet" href=
      "https://stackpath.bootstrapcdn.com/
      bootstrap/4.1.0/css/bootstrap.min.css"
      integrity="sha384-9gVQ4dYFwwWSjIDZ
      nLEWnxCjeSWEFphJiwGPXrljddIhOegi
      u1FwO5qRGvFXOdJZ4" crossorigin="anonymous">
    <link rel="stylesheet" href=
      "https://use.fontawesome.com/releases
      /v5.0.10/css/all.css" integrity="sha384-
      +d0P83n9kaQMCwj8F4RJB66tzIwOKmrdb46+porD/
      OvrJ+37WqIM7UoBtwHO6Nlg" crossorigin=
      "anonymous">
  </head>
```

5. Следующим шагом является написание содержимого для тела шаблона. В тело шаблона мы включим имя приложения под тегом `<header></header>` и ссылку на `todo_container` дочернего шаблона, заключенную в тег блока. Дочерний шаблон будет написан на *шаге 8*.

Включите следующий код сразу после тега `</head>` в файл шаблона `home.html`:

```
<body>
  <header>
    <nav class="navar">
      <div class="container-fluid">
        <center>
```

```

        <h1>Packt Todo
        Application</h1>
    </center>
</div>
</nav>
</header>
<div class="container-fluid">
    {% block todo_container %}{% endblock %}
</div>
</body>
</html>
</html>

```

Выделенный код сообщает родительскому шаблону, что блок `todo_container` будет определяться дочерним шаблоном. Содержимое дочернего шаблона, содержащего блок `todo_container` и расширяющего родительский шаблон, будет отображаться там.

6. Чтобы увидеть изменения, активируйте виртуальную среду и запустите приложение:

```

$ source venv/bin/activate
(venv)$ uvicorn api:app --host=0.0.0.0 --port 8000
--reload

```

7. Откройте <http://127.0.0.1:8000/todo>, чтобы просмотреть изменения:

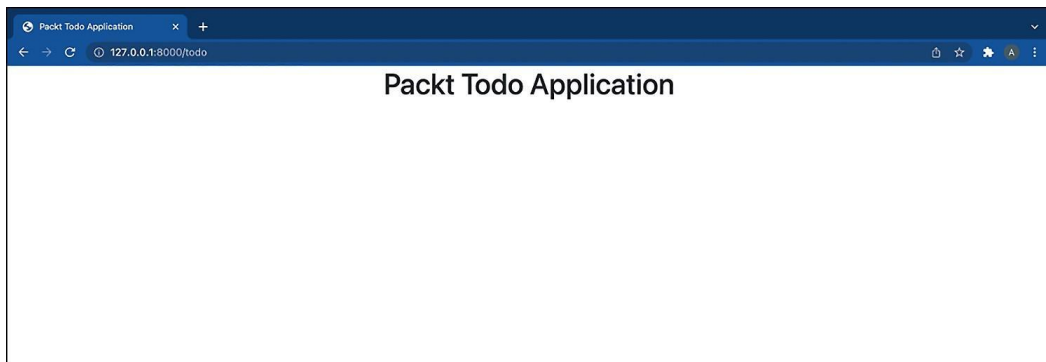


Рисунок 4.2 – Домашняя страница приложения Todo

8. Далее давайте напишем шаблон todo в todo.html:

```
{% extends "home.html" %}

{% block todo_container %}
<main class="container">
    <hr>
    <section class="container-fluid">
        <form method="post">
            <div class="col-auto">
                <div class="input-group mb-3">
                    <input type="text" name="item"
                        value="{{ item }}" class="form-
                        control" placeholder="Purchase
                        Packt's Python workshop course"
                        aria-label="Add a todo"
                        aria-describedby="button-addon2" />
                    <button class="btn btn-outline-
                        primary" type="submit" id=
                        "button-addon2" data-mdb-ripple-
                        color="dark">
                        Add Todo
                    </button>
                </div>
            </div>
        </form>
    </section>
    {% if todo %}
        <article class="card container-fluid">
            <br/>
            <h4>Todo ID: {{ todo.id }} </h4>
            <p>
                <strong>
                    Item: {{ todo.item }}
                </strong>
            </p>
        </article>
    {% endif %}
</main>
{% endblock %}
```

```

        </p>
    </article>
{% else %}
    <section class="container-fluid">
        <h2 align="center">Todos</h2>
        <br>
        <div class="card">
            <ul class="list-group list-group-
            flush">
                {% for todo in todos %}
                    <li class="list-group-item">
                        {{ loop.index }}. <a href=
                        "/todo/{{loop.index}}"> {{
                        todo.item }} </a>
                    </li>
                {% endfor %}
            </ul>
        </div>
    {% endif %}
</section>
</main>

{% endblock %}

```

В предыдущем блоке кода шаблон `todo` наследует шаблон домашней страницы. Мы также определили блок `todo_container`, содержимое которого будет отображаться в родительском шаблоне.

Шаблон задачи используется как для получения всех задач, так и для одной задачи. В результате шаблон отображает различный контент в зависимости от используемого маршрута.

В шаблоне Jinja проверяет, передается ли переменная `todo` с помощью блока `{% if todo %}`. Подробная информация о задаче отображается, если передается переменная задачи, в противном случае она отображает содержимое в блоке `{% else %}`, который является списком.

9. Обновите веб-браузер, чтобы просмотреть последние изменения:

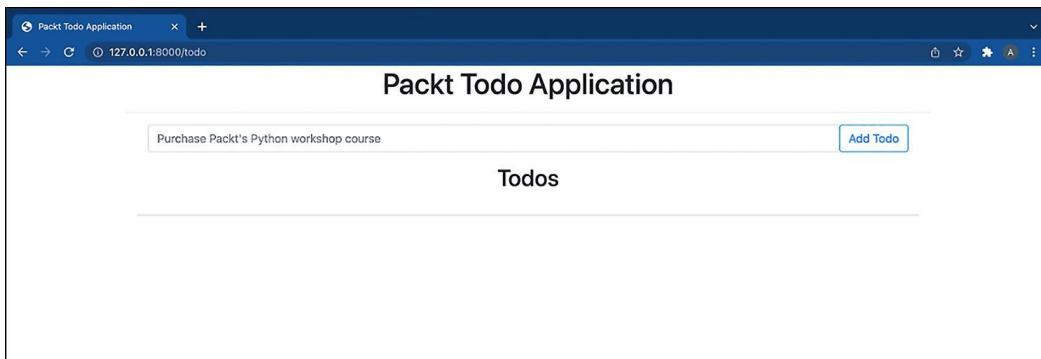


Рисунок 4.3 – Обновите домашнюю страницу todo

10. Давайте добавим задачу, чтобы убедиться, что домашняя страница работает должным образом:

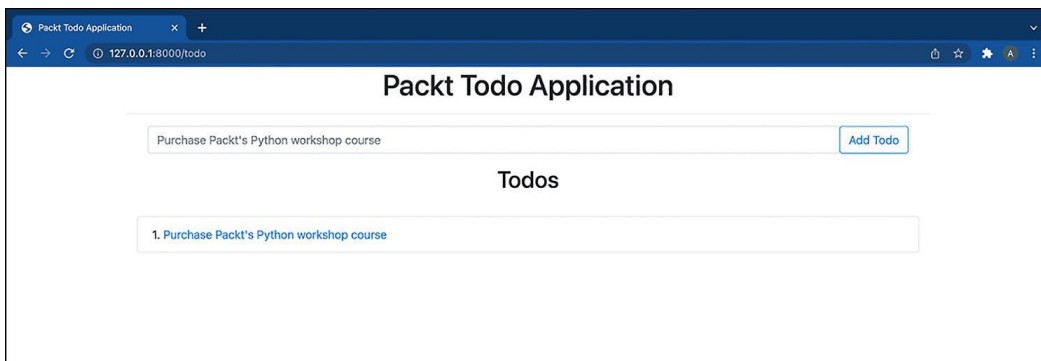


Рисунок 4.4 – Отображаемый список задач

11. Задача кликабельна. Нажмите на `todo`, и вы должны увидеть следующую страницу:

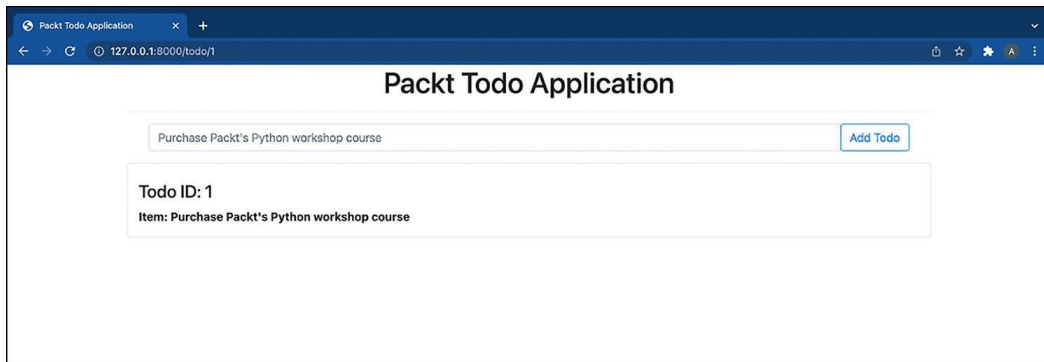


Рисунок 4.5 – Одна страница задач

Мы успешно добавили шаблон в наше приложение FastAPI.

Резюме

В этой главе мы узнали, что такое шаблоны, основы системы шаблонов Jinja и как использовать ее в FastAPI. Мы использовали основы, изученные в первом разделе этой главы, чтобы решить, какой контент отображать. Мы также узнали, что такое наследование шаблонов и как оно работает, на примере шаблонов главной страницы и задач.

В следующей главе вы познакомитесь со структурированием приложений в FastAPI. В этой главе вы будете создавать приложение планировщика, используя знания из этой и предыдущих глав. Сначала вы познакомитесь со структурой приложений, прежде чем приступить к созданию приложения-планировщика.

Часть 2:

Создание и Защита приложений FastAPI

По завершении этой части вы сможете создать полнофункциональное и безопасное приложение с помощью FastAPI. В этой части используются знания, полученные в предыдущей части, и они подкрепляются созданием более функционального приложения с более высокой сложностью, чем приложение, созданное во второй главе. Вы также сможете интегрировать и подключаться к базе данных SQL и NoSQL (MongoDB), а также сможете защитить приложение FastAPI к концу этой части.

Эта часть состоит из следующих глав:

- Глава 5. Структурирование приложений FastAPI
- Глава 6. Подключение к базе данных
- Глава 7. Защита приложений FastAPI

5

Структурирование приложений FastAPI

В последних четырех главах мы рассмотрели основные шаги, связанные с пониманием FastAPI и созданием приложения FastAPI. Приложение, которое мы создали до сих пор, однофайловое приложение `todo`, демонстрирующее гибкость и мощь FastAPI. Ключевым выводом из предыдущих глав является простота создания приложения с использованием FastAPI. Однако необходимо правильно структурировать приложение с повышенной сложностью и функциональностью.

Структурирование относится к размещению компонентов приложения в организованном формате, который может быть модульным для улучшения читаемости кода и содержимого приложения. Приложение с правильной структурой обеспечивает более быструю разработку, более быструю отладку и общее повышение производительности.

К концу этой главы вы будете владеть знаниями о том, что такое структурирование и как структурировать свой API. В этой главе вы затронете следующие темы:

- Структурирование маршрутов и моделей приложений
- Реализация моделей для API планировщика

Технические требования

Код, использованный в этой главе, можно найти по адресу <https://github.com/PacktPublishing/Building-Python-Web-APIs-with-FastAPI/tree/main/ch05/planner>.

Структурирование в приложениях FastAPI

В этой главе мы создадим планировщик событий. Давайте разработаем структуру приложения, чтобы она выглядела так:

```
planner/  
  main.py  
  database/  
    __init__.py  
    connection.py  
  routes/  
    __init__.py  
    events.py  
    users.py  
  models/  
    __init__.py  
    events.py  
    users.py
```

Первый шаг — создать новую папку для приложения. Он будет называться планировщик:

```
$ mkdir planner && cd planner
```

Во вновь созданной папке планировщика, создайте файл ввода, `main.py`, и три подпапки

– `database`, `routes`, and `models`:

```
$ touch main.py  
$ mkdir database routes models
```

Затем создайте `__init__.py` в каждой папке:

```
$ touch {database,routes,models}/__init__.py
```

В папке `database`, давайте создадим пустой файл, `database.py`, который будет обрабатывать абстракции и конфигурации базы данных, которые мы будем использовать в следующей главе:

```
$ touch database/connection.py
```

В папках `routes` and `models`, мы создадим два файла: `events.py` и `users.py`:

```
$ touch {routes,models}/{events,users}.py
```

Каждый файл имеет свою функцию, как указано здесь:

- Модули в пакете `route`:
 - `events.py`: Этот модуль будет обрабатывать операции маршрутизации, такие как создание, обновление и удаление событий.
 - `users.py`: Этот модуль будет обрабатывать операции маршрутизации, такие как регистрация и вход пользователей.
- Модули в пакете `models`:
 - `events.py`: Этот модуль будет содержать определение модели для операций с событиями.
 - `users.py`: Этот модуль будет содержать определение модели для пользовательских операций.

Теперь, когда мы успешно структурировали наш API и сгруппировали похожие по функциям файлы в компоненты, давайте приступим к реализации приложения в следующем разделе.

Создание приложения для планирования мероприятий

В этом разделе мы будем создавать приложение планировщика событий. В этом приложении зарегистрированные пользователи смогут создавать, обновлять и удалять события. Созданные события можно просмотреть, перейдя на страницу события, автоматически созданную приложением.

Каждый зарегистрированный пользователь и событие будут иметь уникальный идентификатор. Это сделано для предотвращения конфликтов при управлении пользователями и событиями с одним и тем же идентификатором. В этом разделе мы не будем отдавать приоритет аутентификации или управлению базой данных, так как это будет подробно обсуждаться в *Главе 6 «Подключение к базе данных»*, и *Главе 7 «Защита приложений FastAPI»*.

Чтобы начать разработку, давайте создадим виртуальную среду и активируем ее в каталоге нашего проекта:

```
$ python3 -m venv venv
$ source venv/bin/activate
```

Далее давайте установим зависимости приложения:

```
(venv)$ pip install fastapi uvicorn "pydantic[email]"
```

Наконец, сохраните требования в `requirements.txt`:

```
(venv)$ pip freeze > requirements.txt
```

Теперь, когда мы успешно установили наши зависимости и настроили среду разработки, давайте реализуем следующие модели приложения.

Реализация моделей

Давайте посмотрим на шаги для реализации нашей модели:

1. Первым шагом в создании нашего приложения является определение моделей для события и пользователя. Модели описывают, как данные будут храниться, вводиться и представляться в нашем приложении. На следующей диаграмме показано моделирование пользователя и события, а также их отношения:

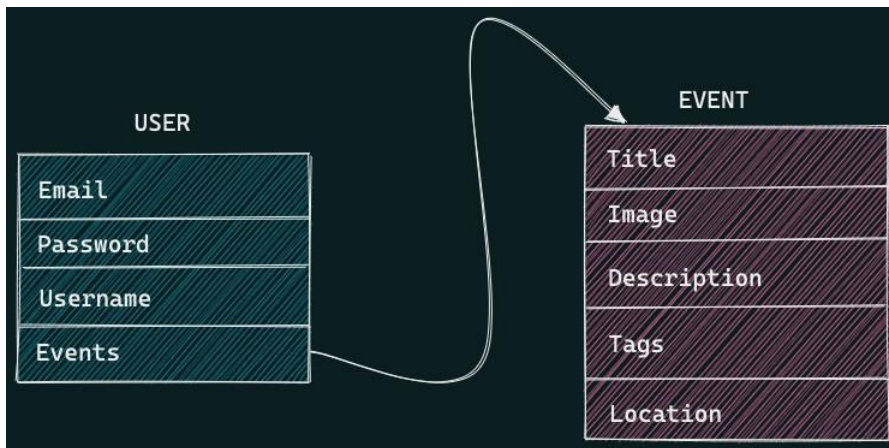


Рисунок 5.1 – Отношение моделей для пользователя и события

Как показано на предыдущей диаграмме модели, у каждого пользователя будет поле **Events**, представляющее собой список событий, на которые он имеет право собственности.

2. Определим модель Event в `models/events.py`:

```
from pydantic import BaseModel
from typing import List

class Event(BaseModel):
    id: int
    title: str
    image: str
    description: str
    tags: List[str]
    location: str
```

3. Давайте определим подкласс Config в классе Event, чтобы показать пример того, как будут выглядеть данные модели, когда мы посетим документацию:

```
class Config:
    schema_extra = {
        "example": {
            "title": "FastAPI Book Launch",
            "image": "https://linktomyimage.com/image.png",
            "description": "We will be discussing the contents of the FastAPI book in this event. Ensure to come with your own copy to win gifts!",
            "tags": ["python", "fastapi", "book", "launch"],
            "location": "Google Meet"
        }
    }
```

Наша модель событий в первом блоке кода содержит пять полей:

- Название события
- Ссылка на баннер изображения события
- Описание события
- Теги событий для группировки
- Место проведения

Во втором блоке кода мы определяем пример данных события. Это направлено на то, чтобы направлять нас при создании нового события из нашего API.

4. Теперь, когда мы определили нашу модель событий, давайте определим модель User:

```
from pydantic import BaseModel, EmailStr
from typing import Optional, List
from models.events import Event

class User(BaseModel):
    email: EmailStr
    password: str
    events: Optional[List[Event]]
```

Наша модель пользователя, определенная ранее, содержит следующие поля:

- Электронная почта пользователя
- Пароль пользователя
- Список событий, созданный пользователем, который по умолчанию пуст

5. Теперь, когда мы определили нашу модель User, давайте создадим пример, показывающий, как хранятся и устанавливаются пользовательские данные:

```
class Config:
    schema_extra = {
        "example": {
            "email": fastapi@packt.com,
            "username": "strong!!!",
            "events": [],
        }
    }
```


6. Далее мы создадим новую модель, `NewUser`, которая наследуется от модели `User`; эта новая модель будет использоваться в качестве типа данных при регистрации нового пользователя. Модель `User` будет использоваться в качестве модели ответа, когда мы не хотим взаимодействовать с паролем, уменьшая объем работы, которую необходимо выполнить.
7. Наконец, давайте реализуем модель для входа пользователей в:

```
class UserSignIn(BaseModel):
    email: EmailStr
    password: str

class Config:
    schema_extra = {
        "example": {
            "email": fastapi@packt.com,
            "password": "strong!!!",
            "events": [],
        }
    }
```

Теперь, когда мы успешно реализовали наши модели, давайте реализуем маршруты в следующем разделе.

Реализация маршрутов

Следующим шагом в создании нашего приложения является настройка системы маршрутизации нашего API. Мы будем разрабатывать систему маршрутизации для событий и пользователей. Маршрут пользователя будет состоять из маршрутов входа, выхода и регистрации. Аутентифицированный пользователь будет иметь доступ к маршрутам для создания, обновления и удаления события, в то время как публика сможет просматривать событие после его создания. На следующей диаграмме показана взаимосвязь между обоими маршрутами:

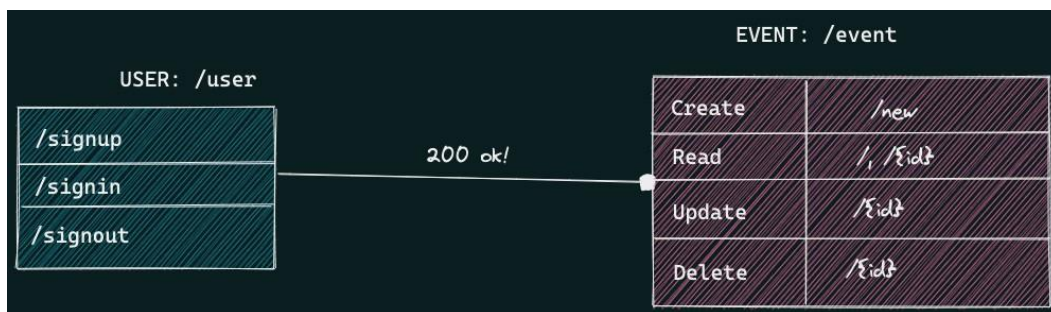


Рисунок 5.2 – Маршруты для операций пользователей и событий

Далее рассмотрим оба маршрута подробнее.

Маршруты пользователей

Теперь, когда у нас есть четкое представление о том, какие маршруты реализовать из *Рисунка 5.2*, мы начнем с определения пользовательских маршрутов в `users.py`. Давайте посмотрим на шаги:

1. Начните с определения основного маршрута регистрации:

```
from fastapi import APIRouter, HTTPException, status
from models.user import User, UserSignIn

user_router = APIRouter(
    tags=["User"]
)

users = {}

@user_router.post("/signup")
async def sign_new_user(data: NewUser) -> dict:
    if data.email in users:
        raise HTTPException(
```

```
        status_code=status.HTTP_409_CONFLICT,
        detail="User with supplied username
        exists"
    )
    users[data.email] = data
    return {
        "message": "User successfully registered!"
    }
```

В маршруте регистрации, определенном ранее, мы используем базу данных в приложении. (Мы познакомимся с базой данных в *Главе 6 «Подключение к базе данных»*.)

Маршрут проверяет, существует ли пользователь с похожим адресом электронной почты в базе данных перед добавлением нового.

2. Давайте реализуем маршрут входа:

```
@user_router.post("/signin")
async def sign_user_in(user: UserSignIn) -> dict:
    if users[user.email] not in users:
        raise HTTPException(
            status_code=status.HTTP_404_NOT_FOUND
            detail="User does not exist"
        )
    if users[user.email].password != user.password:
        raise HTTPException(
            status_code=status.HTTP_403_FORBIDDEN
            detail="Wrong credentials passed"
        )

    return {
        "message": "User signed in successfully"
    }
```

В этом маршруте первым делом проверяется, существует ли такой пользователь в базе данных, и, если такой пользователь не существует, возникает исключение. Если пользователь существует, приложение проверяет, совпадают ли пароли, прежде чем вернуть успешное сообщение или исключение.

В наших маршрутах мы храним пароли в чистом виде без какого-либо шифрования. Это используется только в демонстрационных целях и является неправильной практикой в разработке программного обеспечения в целом. Надлежащие механизмы хранения, такие как шифрование, будут обсуждаться в *Главе 6 Подключение к базе данных*, где наше приложение переместится из базы данных в приложении в реальную базу данных.

3. Теперь, когда мы определили маршруты для пользовательских операций, давайте зарегистрируем их в `main.py` и запустим наше приложение. Давайте начнем с импорта наших библиотек и определения пользовательских маршрутов:

```
from fastapi import FastAPI
from routes.user import user_router

import uvicorn
```

4. Далее создадим экземпляр FastAPI и зарегистрируем маршрут и приложение:

```
app = FastAPI()

# Register routes

app.include_router(user_router, prefix="/user")

if __name__ == "__main__":
    uvicorn.run("main:app", host="0.0.0.0", port=8080,
                reload=True)
```

В этом блоке кода мы создали экземпляр FastAPI и зарегистрировали маршрут.

5. Затем мы используем метод `uvicorn.run()` для запуска нашего приложения на порту 8080 и устанавливаем для перезагрузки значение `True`. В терминале запустите приложение:

```
(venv)$ python main.py
INFO:      Will watch for changes in these directories:
['/Users/youngestdev/Work/Building-Web-APIs-with-FastAPI-
and-Python/ch05/planner']
INFO:      Uvicorn running on http://0.0.0.0:8080 (Press
CTRL+C to quit)
INFO:      Started reloader process [6547] using
statreload
```

```
INFO:      Started server process [6550]
INFO:      Waiting for application startup.
INFO:      Application startup complete.
```

6. Теперь, когда наше приложение успешно запустилось, давайте проверим реализованные нами пользовательские маршруты. Начнем с регистрации пользователя:

```
(venv)$ curl -X 'POST' \
  'http://0.0.0.0:8080/user/signup' \
  -H 'accept: application/json' \
  -H 'Content-Type: application/json' \
  -d '{
    "email": "fastapi@packt.com",
    "password": "Stro0ng!",
    "username": "FastPackt"
  }'
```

Предыдущий запрос возвращает ответ:

```
{
  "message": "User successfully registered!"
}
```

7. Предыдущий ответ указывает на успешность выполненной операции. Давайте проверим маршрут входа:

```
(venv)$ curl -X 'POST' \
  'http://0.0.0.0:8080/user/signin' \
  -H 'accept: application/json' \
  -H 'Content-Type: application/json' \
  -d '{
    "email": "fastapi@packt.com",
    "password": "Stro0ng!"
  }'
```

Предыдущий ответ на запрос выглядит следующим образом:

```
{
  "message": "User signed in successfully"
}
```

8. Если мы передадим неверный пароль, наше приложение должно вернуть другое сообщение:

```
(venv)$ curl -X 'POST' \  
  'http://0.0.0.0:8080/user/signin' \  
  -H 'accept: application/json' \  
  -H 'Content-Type: application/json' \  
  -d '{  
    "email": "fastapi@packt.com",  
    "password": "password!"  
  }'
```

Ответ на предыдущий запрос выглядит следующим образом:

```
{  
  "detail": "Wrong credential passed"  
}
```

Мы также можем посмотреть наши маршруты из интерактивной документации, предоставленной FastAPI, работающей на Swagger. Давайте посетим <http://0.0.0.0:8080/docs> в нашем браузере, чтобы получить доступ к интерактивной документации:

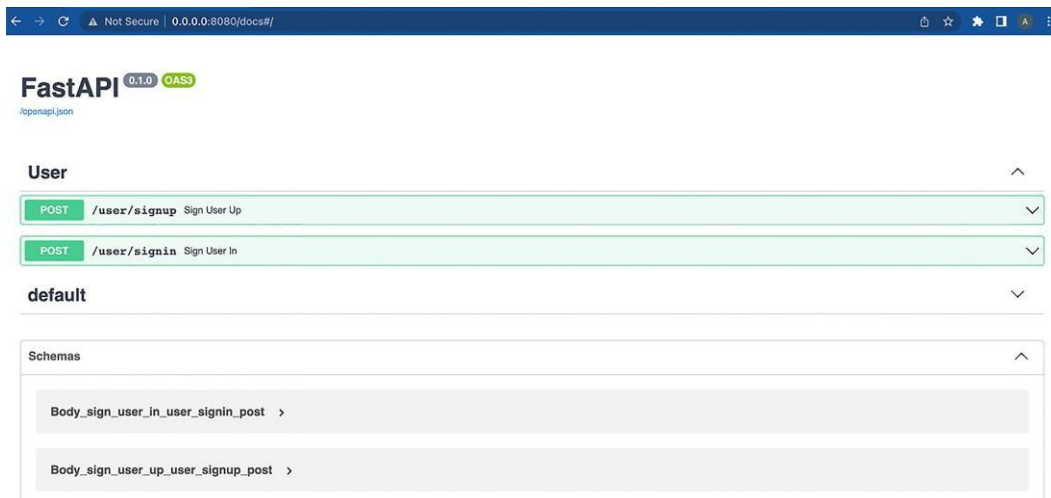


Рисунок 5.3 – Приложение планировщика, просмотренное на странице интерактивной документации

Теперь, когда мы успешно реализовали пользовательские маршруты, давайте реализуем маршруты для событийных операций в следующем разделе.

Маршруты событий

После создания пользовательских маршрутов следующим шагом будет реализация маршрутов для событийных операций. Давайте посмотрим на шаги:

1. Начните с импорта зависимостей и определения маршрутизатора событий:

```
from fastapi import APIRouter, Body, HTTPException, status
from models.events import Event
from typing import List

event_router = APIRouter(
    tags=["Events"]
)

events = []
```

2. Следующим шагом является определение маршрута для получения всех событий и события, соответствующего предоставленному идентификатору в базе данных:

```
@event_router.get("/", response_model=List[Event])
async def retrieve_all_events() -> List[Event]:
    return events

@event_router.get("/{id}", response_model=Event)
async def retrieve_event(id: int) -> Event:
    for event in events:
        if event.id == id:
            return event
    raise HTTPException(
        status_code=status.HTTP_404_NOT_FOUND,
        detail="Event with supplied ID does not exist"
    )
```

Во втором маршруте мы вызываем исключение `HTTP_404_NOT_FOUND`, когда событие с предоставленным идентификатором не существует.

3. Давайте реализуем маршруты для создания события, удаления одного события и удаления всех событий, содержащихся в базе данных:

```
@event_router.post("/new")
async def create_event(body: Event = Body(...)) -> dict:
    events.append(body)
    return {
        "message": "Event created successfully"
    }

@event_router.delete("/{id}")
async def delete_event(id: int) -> dict:
    for event in events:
        if event.id == id:
            events.remove(event)
            return { "message": "Event deleted
                successfully" }
    raise HTTPException(
        status_code=status.HTTP_404_NOT_FOUND,
        detail="Event with supplied ID does not exist"
    )

@event_router.delete("/")
async def delete_all_events() -> dict:
    events.clear()
    return {
        "message": "Events deleted successfully"
    }
```

Мы успешно реализовали маршруты для событий. Маршрут UPDATE будет реализован в главе 6 «Подключение к базе данных», где мы перенесем наше приложение для использования реальной базы данных.

4. Теперь, когда мы реализовали маршруты, давайте обновим нашу конфигурацию маршрута, чтобы включить маршрут события в `main.py`:

```
from fastapi import FastAPI
from routes.user import user_router
from routes.events import event_router
```



```
import uvicorn
app = FastAPI()

# Register routes

app.include_router(user_router, prefix="/user")
app.include_router(event_router, prefix="/event")

if __name__ == "__main__":
    uvicorn.run("main:app", host="0.0.0.0", port=8080,
                reload=True)
```

Приложение автоматически перезагружается при каждом изменении. Проверим маршруты:

- Маршрут GET — следующая операция возвращает пустой массив, сообщая нам об отсутствии данных:

```
(venv)$ curl -X 'GET' \
  'http://0.0.0.0:8080/event/' \
  -H 'accept: application/json'
[]
```

Далее добавим данные в наш массив.

- Маршрут POST – в терминале выполните следующую команду:

```
(venv)$ curl -X 'POST' \
  'http://0.0.0.0:8080/event/new' \
  -H 'accept: application/json' \
  -H 'Content-Type: application/json' \
  -d '{
    "id": 1,
    "title": "FastAPI Book Launch",
    "image": "https://linktomyimage.com/image.png",
    "description": "We will be discussing the contents
of the FastAPI book in this event.Ensure to come
with your own copy to win gifts!",
    "tags": [
      "python",
```

```
    "fastapi",
    "book",
    "launch"
],
"location": "Google Meet"
}'
```

Вот ответ:

```
{
  "message": "Event created successfully"
}
```

Эта операция прошла успешно, судя по полученному ответу. Теперь давайте попробуем получить конкретное событие, которое мы только что создали:

- Маршрут GET:

```
(venv)$ curl -X 'GET' \
  'http://0.0.0.0:8080/event/1' \
  -H 'accept: application/json'
```

Вот ответ:

```
{
  "id": 1,
  "title": "FastAPI BookLaunch",
  "image": "https://linktomyimage.com/image.png",
  "description": "We will be discussing the contents
of the FastAPI book in this event.Ensure to come
with your own copy to win gifts!",
  "tags": [
    "python",
    "fastapi",
    "book",
    "launch"
  ],
  "location": "Google Meet"
}
```

Наконец, давайте удалим событие, чтобы убедиться, что маршрут события работает:

- Маршрут DELETE – в терминале выполните следующую команду:

```
(venv)$ curl -X 'DELETE' \
    'http://0.0.0.0:8080/event/1' \
    -H 'accept: application/json'
```

Вот ответ:

```
{
  "message": "Event deleted successfully"
}
```

Если я повторю ту же команду, я получу следующий ответ:

```
(venv)$ {
  "detail": "Event with supplied ID does not exist"
}
```

Мы успешно реализовали маршруты и модели для нашего приложения-планировщика. Мы также протестировали их, чтобы оценить их рабочее состояние.

Резюме

В этой главе мы узнали, как структурировать приложение FastAPI и реализовать маршруты и модели для приложения планирования событий. Мы использовали основы маршрутизации и знания о маршрутизации и моделировании, полученные в предыдущей главе.

В следующей главе вы познакомитесь с подключением вашего приложения к базам данных SQL и NoSQL. Вы продолжите создание приложения для планирования мероприятий, улучшая существующее приложение и добавляя дополнительные функции. Перед этим вас познакомят что такое базы данных, разные типы и как использовать обе (SQL и NoSQL) в приложении FastAPI.

6

Подключение к базе данных

В предыдущей главе мы рассмотрели, как структурировать приложение FastAPI. Мы успешно реализовали некоторые маршруты и модели для нашего приложения и протестировали конечные точки.

Однако приложение по-прежнему использует внутреннюю базу данных для хранения событий. В этой главе мы перенесем приложение для использования правильной базы данных.

Базу данных можно просто назвать хранилищем данных. В этом контексте база данных позволяет нам хранить данные постоянно, в отличие от встроенной в приложение базы данных, которая стирается при любом перезапуске или сбое приложения. **База данных — это таблица, содержащая столбцы, называемые полями, и строки, называемые записями.**

К концу этой главы вы будете владеть знаниями о том, как подключить приложение FastAPI к базе данных. В этой главе объясняется, как подключиться к базе данных SQL с использованием **SQLModel** и база данных **MongoDB** с помощью **Beanie**. (Однако в последующих главах приложение будет использовать MongoDB в качестве основной базы данных.) В этой главе вы затронете следующие темы.):

- Настройка SQLModel
- CRUD операции в базе данных SQL с использованием SQLModel
- Настройка MongoDB
- CRUD операции в MongoDB с помощью Beanie

Технические требования

Чтобы продолжить, требуется компонент базы данных MongoDB. Процедуры установки для вашей операционной системы можно найти в их официальной документации. Код, использованный в этой главе, можно найти по адресу <https://github.com/PacktPublishing/Building-Python-Web-APIs-with-FastAPI/tree/main/ch06/planner>.

Настройка SQLAlchemy

Первым шагом для интеграции базы данных SQL в наше приложение планировщика является установка библиотеки SQLAlchemy. Библиотека SQLAlchemy была создана создателем FastAPI и поддерживается Pydantic и SQLAlchemy. Поддержка Pydantic упростит нам определение моделей, как мы узнали из главы 3 «Модели ответов и обработка ошибок».

Поскольку мы будем реализовывать базы данных SQL и NoSQL, мы создадим новую ветку GitHub для этого раздела. В своем терминале перейдите в каталог проекта, инициализируйте репозиторий GitHub и зафиксируйте существующие файлы:

```
$ git init
$ git add database models routes main.py
$ git commit -m "Committing bare application without a
database"
```

Далее создайте новую ветку:

```
$ git checkout -b planner-sql
```

Теперь мы готовы настроить SQLAlchemy в нашем приложении. В терминале активируйте виртуальную среду и установите библиотеку SQLAlchemy:

```
$ source venv/bin/activate
(venv)$ pip install sqlalchemy
```

Прежде чем углубиться в добавление базы данных в наше приложение планировщика, давайте рассмотрим некоторые из методов, содержащихся в SQLAlchemy, которые мы будем использовать в этой главе.

Таблицы

Таблица — это, по сути, объект, который содержит данные, хранящиеся в базе данных — например, данные о событиях будут храниться в таблице событий. Таблица будет состоять из столбцов и строк, где в конечном итоге будут храниться данные.

Переменные, определенные в классе, будут представлять столбцы по умолчанию, если они не обозначены как поля. Давайте посмотрим, как будет определена таблица событий:

```
class Event(SQLModel, table=True):
    id: Optional[int] = Field(default=None,
                              primary_key=True)
    title: str
    image: str
    description: str
    location: str
    tags: List[str]
```

В этом классе таблицы все определенные переменные являются столбцами, кроме `id`, который был определен как поле. Поля обозначаются с помощью объекта `Field` из библиотеки `SQLModel`. Поле `id` также является первичным ключом в таблице базы данных.

Первичный ключ — это уникальный идентификатор записи, содержащейся в таблице базы данных.

Теперь, когда мы узнали, что такое таблицы и как их создавать, давайте рассмотрим строки в следующем разделе.

Данные, отправляемые в таблицу базы данных, хранятся в строках под указанными столбцами. Чтобы вставить данные в строки и сохранить их, создается экземпляр таблицы, а переменные заполняются нужными входными данными. Например, чтобы вставить данные о событии в таблицу событий, мы сначала создадим экземпляр модели:

```
new_event = Event(title="Book Launch",  
                  image="src/fastapi.png",  
                  description="The book launch event will  
be held at Packt HQ, Packt city",
```

```
location="Google Meet",  
tags=["packt", "book"])
```

Далее мы создаем транзакцию базы данных, используя класс `Session`:

```
with Session(engine) as session:  
    session.add(new_event)  
    session.commit()
```

Предыдущая операция может показаться вам чуждой. Давайте посмотрим, что такое класс **Session** и что он делает.

Сессии

Объект сеанса обрабатывает взаимодействие кода с базой данных. Он в первую очередь выступает в качестве посредника при выполнении операций. Класс `Session` принимает аргумент, который является экземпляром механизма SQL.

Теперь, когда мы узнали, как создаются таблицы и строки, мы рассмотрим, как создается база данных. Некоторые из методов класса `session`, которые мы будем использовать в этой главе, включают следующие:

- `add()`: Этот метод отвечает за добавление объекта базы данных в память в ожидании дальнейших операций. В предыдущем блоке кода объект `new_event` добавляется в память сеанса, ожидая фиксации в базе данных методом `commit()`.
- `commit()`: Этот метод отвечает за сброс транзакций, присутствующих в сеансе.
- `get()`: Этот метод принимает два параметра — модель и идентификатор запрошенного документа. Этот метод используется для извлечения одной строки из базы данных.

Теперь, когда мы знаем, как создавать таблицы, строки и столбцы, а также вставлять данные с помощью класса `Session`, давайте перейдем к созданию базы данных и выполнению операций CRUD в следующем разделе.

Создание базы данных

В `SQLModel` подключение к базе данных осуществляется с помощью механизма `SQLAlchemy`. Движок создается методом `create_engine()`, импортированным из библиотеки `SQLModel`.

Метод `create_engine()` принимает в качестве аргумента URL-адрес базы данных. URL-адрес базы данных имеет вид `sqlite:///database.db` или `sqlite:///database.sqlite`. Он также принимает необязательный аргумент `echo`, который, если установлено значение `True` распечатывает команды SQL, выполняемые при выполнении операции.

Однако одного метода `create_engine()` одного метода для создания файла базы данных. Чтобы создать файл базы данных, вызывается метод, `SQLModel.metadata.create_all(engine)`, аргументом которого является экземпляр метода `create_engine()`, например:

```
database_file = "database.db"
engine = create_engine(database_file, echo=True)
SQLModel.metadata.create_all(engine)
```

Метод `create_all()` создает базу данных, а также определенные таблицы. Важно отметить, что файл, содержащий таблицы, импортируется в файл, в котором происходит подключение к базе данных.

В нашем приложении-планировщике мы выполняем CRUD операции для событий. В папке базы данных создайте следующий модуль:

`connection.py`

В этом файле мы настроим необходимые данные для базы данных:

```
(venv)$ touch database/connection.py
```

Теперь, когда мы создали файл подключения к базе данных, давайте создадим функции, необходимые для подключения нашего приложения к базе данных:

1. Мы начнем с обновления класса модели событий, определенного в `models/events.py`, до класса модели таблицы `SQLModel`:

```
from sqlmodel import JSON, SQLModel, Field, Column
from typing import Optional, List

class Event(SQLModel, table=True):
    id: int = Field(default=None, primary_key=True)
    title: str
    image: str
    description: str
    tags: List[str] = Field(sa_column=Column(JSON))
```



```
location: str

class Config:
    arbitrary_types_allowed = True
    schema_extra = {
        "example": {
            "title": "FastAPI Book Launch",
            "image": "https://linktomyimage.com/image.png",
            "description": "We will be discussing the contents of the FastAPI book in this event. Ensure to come with your own copy to win gifts!",
            "tags": ["python", "fastapi", "book", "launch"],
            "location": "Google Meet"
        }
    }
```

В этом блоке кода мы изменили исходный класс модели, чтобы он стал классом таблицы SQL.

2. Добавим еще один класс `SQLModel`, который будет использоваться в качестве типа тела во время операций `UPDATE`:

```
class EventUpdate(SQLModel):
    title: Optional[str]
    image: Optional[str]
    description: Optional[str]
    tags: Optional[List[str]]
    location: Optional[str]

class Config:
    schema_extra = {
        "example": {
            "title": "FastAPI Book Launch",
            "image": "https://linktomyimage.com/image.png",
```

```

        //linktomyimage.com/image.png",
        "description": "We will be discussing
        the contents of the FastAPI book in
        this event. Ensure to come with your
        own copy to win gifts!",
        "tags": ["python", "fastapi", "book",
        "launch"],
        "location": "Google Meet"
    }
}

```

3. Далее давайте определим конфигурацию, необходимую для создания нашей базы данных и таблицы в `connection.py`:

```

from sqlmodel import SQLModel, Session, create_engine
from models.events import Event

database_file = "planner.db"
database_connection_string = f"sqlite:/// {database_file}"
connect_args = {"check_same_thread": False}
engine_url = create_engine(database_connection_string,
echo=True, connect_args=connect_args)

def conn():
    SQLModel.metadata.create_all(engine_url)

def get_session():
    with Session(engine_url) as session:
        yield session

```

В этом блоке кода мы начинаем с определения зависимостей, а также импортируем класс модели таблицы. Затем мы создаем переменную, содержащую расположение файла базы данных (который будет создан, если он не существует), строку подключения и экземпляр созданной базы данных SQL. В функции `conn()` мы инструктируем `SQLModel` создать базу данных, а также таблицу, представленную в файле, `Events`, и сохранить сеанс в нашем приложении, определено, `get_session()`.

4. Далее давайте проинструктируем наше приложение создавать базу данных при запуске. Обновите `main.py` следующим кодом:

```
from fastapi import FastAPI
from fastapi.responses import RedirectResponse
from database.connection import conn

from routes.users import user_router
from routes.events import event_router

import uvicorn

app = FastAPI()

# Register routes

app.include_router(user_router, prefix="/user")
app.include_router(event_router, prefix="/event")

@app.on_event("startup")
def on_startup():
    conn()

@app.get("/")
async def home():
    return RedirectResponse(url="/event/")

if __name__ == '__main__':
    uvicorn.run("main:app", host="0.0.0.0", port=8080,
                reload=True)
```

База данных будет создана после запуска приложения. В событии запуска мы вызвали функцию `conn()`, отвечающую за создание базы данных. Запустите приложение в своем терминале, и вы должны увидеть вывод в своей консоли, указывающий, что база данных была создана, а также таблица:

```
python main.py
INFO: Will watch for changes in these directories: ['/Users/youngestdev/Work/Building-Web-APIs-with-FastAPI-and-Python/ch06/planner']
INFO: Uvicorn running on http://0.0.0.0:8080 (Press CTRL+C to quit)
INFO: Started reloader process [5668] using statreload
INFO: Started server process [5671]
INFO: Waiting for application startup.
2022-04-06 16:25:36,230 INFO sqlalchemy.engine.Engine BEGIN (implicit)
2022-04-06 16:25:36,230 INFO sqlalchemy.engine.Engine PRAGMA main.table_info("event")
2022-04-06 16:25:36,230 INFO sqlalchemy.engine.Engine [raw sql] ()
2022-04-06 16:25:36,230 INFO sqlalchemy.engine.Engine PRAGMA temp.table_info("event")
2022-04-06 16:25:36,231 INFO sqlalchemy.engine.Engine [raw sql] ()
2022-04-06 16:25:36,231 INFO sqlalchemy.engine.Engine
CREATE TABLE event (
  tags JSON,
  id INTEGER,
  title VARCHAR NOT NULL,
  image VARCHAR NOT NULL,
  description VARCHAR NOT NULL,
  location VARCHAR NOT NULL,
  PRIMARY KEY (id)
)
2022-04-06 16:25:36,231 INFO sqlalchemy.engine.Engine [no key 0.00003s] ()
2022-04-06 16:25:36,232 INFO sqlalchemy.engine.Engine COMMIT
```

Рисунок 6.1 – База данных планировщика и таблица событий успешно созданы

Команды SQL, отображаемые в терминале, появляются из-за того, что при создании ядра базы данных для `echo` установлено значение `True`. Теперь, когда мы успешно создали базу данных, давайте обновим маршруты CRUD операций наших событий, чтобы использовать базу данных.

Создание событий

Давайте посмотрим на шаги:

1. В файле `routes/events.py`, обновите импорт, включив в него класс модели таблицы событий, а также функцию `get_session()`. Функция `get_session()` импортируется, чтобы маршруты могли получить доступ к созданному объекту сеанса:

```
from fastapi import APIRouter, Depends, HTTPException, Request, status

from database.connection import get_session

from models.events import Event, EventUpdate
```

Что такое Depends?

Класс `Depends` отвечает за выполнение внедрения зависимостей в приложениях FastAPI. Класс `Depends` принимает источник истинности, такой как функция, в качестве аргумента и передается в качестве аргумента функции в маршруте, требуя, чтобы условие зависимости было выполнено до того, как любая операция может быть выполнена.

2. Далее давайте обновим функцию маршрута `POST`, отвечающую за создание нового события, `create_event()`:

```
@event_router.post("/new")
async def create_event(new_event: Event,
                       session=Depends(get_session)) -> dict:
    session.add(new_event)
    session.commit()
    session.refresh(new_event)

    return {
        "message": "Event created successfully"
    }
```

В этом блоке кода мы указали, что объект сеанса, необходимый для выполнения транзакций базы данных, зависит от функции `get_session()`, которую мы создали ранее.

В теле функции данные добавляются в сессию, а затем фиксируются в базе данных, после чего база данных обновляется.

3. Давайте протестируем маршруты для предварительного просмотра изменений:

```
(venv)$ curl -X 'POST' \
    'http://0.0.0.0:8080/event/new' \
    -H 'accept: application/json' \
    -H 'Content-Type: application/json' \
    -d '{
        "title": "FastAPI Book Launch",
        "image": "fastapi-book.jpeg",
        "description": "We will be discussing the contents
        of the FastAPI book in this event. Ensure to come
        with your own copy to win gifts!",
```

```

    "tags": [
        "python",
        "fastapi",
        "book",
        "launch"
    ],
    "location": "Google Meet"
}'

```

Возвращается успешный ответ:

```

{
    "message": "Event created successfully"
}

```

Если операцию выполнить не удалось, библиотека выдаст исключение.

Чтение событий

Давайте обновим маршрут GET, который извлекает список событий для извлечения данных из базы данных:

```

@event_router.get("/", response_model=List[Event])
async def retrieve_all_events(session=Depends(get_session)) ->
List[Event]:
    statement = select(Event)
    events = session.exec(statement).all()
    return events

```

Аналогичным образом, маршрут для отображения данных события при получении по его идентификатору также обновляется:

```

@event_router.get("/{id}", response_model=Event)
async def retrieve_event(id: int, session=Depends(get_session))
-> Event:
    event = session.get(Event, id)
    if event:
        return event
    raise HTTPException(
        status_code=status.HTTP_404_NOT_FOUND,
        detail="Event with supplied ID does not exist"
    )

```

```
)

raise HTTPException(
    status_code=status.HTTP_404_NOT_FOUND,
    detail="Event with supplied ID does not exist"
)
```

Модель ответа для обоих маршрутов была установлена в классе модели. Давайте протестируем оба маршрута, сначала отправив запрос GET для получения списка событий:

```
(venv)$ curl -X 'GET' \
'http://0.0.0.0:8080/event/' \
-H 'accept: application/json'
```

Мы получаем ответ:

```
[
  {
    "id": 1,
    "title": "FastAPI Book Launch",
    "image": "fastapi-book.jpeg",
    "description": "We will be discussing the contents of
the FastAPI book in this event.Ensure to come with your
own copy to win gifts!",
    "tags": [
      "python",
      "fastapi",
      "book",
      "launch"
    ],
    "location": "Google Meet"
  }
]
```

Далее давайте извлечем событие по его ID:

```
(venv)$ curl -X 'GET' \
'http://0.0.0.0:8080/event/1' \
-H 'accept: application/json'
```

```

}
{
    "id": 1,
    "title": "FastAPI Book Launch",
    "image": "fastapi-book.jpeg",
    "description": "The launch of the FastAPI book will hold
on xyz.",
    "tags": [
        "python",
        " fastapi"
    ],
    "location": "virtual"
}

```

После успешной реализации операций READ давайте добавим функцию редактирования для нашего приложения.

Обновление событий

Давайте добавим маршрут UPDATE в `routes/events.py`:

```

@event_router.put("/edit/{id}", response_model=Event)
async def update_event(id: int, new_data: EventUpdate,
session=Depends(get_session)) -> Event:

```

В тело функции добавьте следующий блок кода, чтобы получить существующее событие и обработать изменения события:

```

event = session.get(Event, id)
if event:
    event_data = new_data.dict(exclude_unset=True)
    for key, value in event_data.items():
        setattr(event, key, value)
    session.add(event)
    session.commit()
    session.refresh(event)

    return event
raise HTTPException(

```



```
        status_code=status.HTTP_404_NOT_FOUND,  
        detail="Event with supplied ID does not exist"  
    )
```

В предыдущем блоке кода мы проверяем наличие события, прежде чем приступить к обновлению данных события. После обновления события возвращаются обновленные данные. Обновим существующий заголовок статьи:

```
(venv)$ curl -X 'PUT' \  
  'http://0.0.0.0:8080/event/edit/1' \  
  -H 'accept: application/json' \  
  -H 'Content-Type: application/json' \  
  -d '{  
    "title": "Packt\'\'s FastAPI book launch II"  
  }'  
  
{  
  "id": 1,  
  "title": "Packt's FastAPI book launch II",  
  "image": "fastapi-book.jpeg",  
  "description": "The launch of the FastAPI book will hold  
on xyz.",  
  "tags": ["python", "fastapi"],  
  "location": "virtual" }
```

Теперь, когда мы добавили функцию обновления, давайте быстро добавим операцию удаления в следующем разделе.

Удаление событий

В `events.py`, обновите маршрут `delete` определенный ранее:

```
@event_router.delete("/delete/{id}")  
async def delete_event(id: int, session=Depends(get_session))  
-> dict:  
    event = session.get(Events, id)  
    if event:  
        session.delete(event)
```

```

        session.commit()

    return {
        "message": "Event deleted successfully"
    }

    raise HTTPException(
        status_code=status.HTTP_404_NOT_FOUND,
        detail="Event with supplied ID does not exist"
    )

```

В этом блоке кода функция проверяет, существует ли событие, идентификатор которого был предоставлен, а затем удаляет его из базы данных. После выполнения операции возвращается сообщение об успешном выполнении и выдается исключение, если событие не существует. Удалим событие из базы:

```

(venv)$ curl -X 'DELETE' \
  'http://0.0.0.0:8080/event/delete/1' \
  -H 'accept: application/json'

```

Запрос возвращает успешный ответ:

```

{
  "message": "Event deleted successfully"
}

```

Теперь, если мы получим список событий, мы получим пустой массив для ответа:

```

(venv)$ curl -X 'GET' \
  'http://0.0.0.0:8080/event/' \
  -H 'accept: application/json'
[]

```

Мы успешно внедрили базу данных SQL в наше приложение с помощью SQLAlchemy, а также реализовали CRUD операции. Давайте зафиксируем изменения в приложении, прежде чем научиться реализовывать CRUD операции в MongoDB:

```

(venv)$ git add .
(venv)$ git commit -m "[Feature] Incorporate a SQL database and
implement CRUD operations "

```

Вернуться к main ветке:

```
(venv)$ git checkout main
```

Теперь, когда вы вернулись к исходной версии приложения, давайте включим MongoDB в качестве платформы базы данных и реализуем CRUD операции в следующем разделе.

Настройка MongoDB

Существует ряд библиотек, которые позволяют нам интегрировать MongoDB в наше приложение FastAPI. Однако мы будем использовать **Beanie**, асинхронную библиотеку **Object Document Mapper (ODM)**, для выполнения операций с базой данных из нашего приложения.

Давайте установим библиотеку `beanie`, выполнив следующую команду:

```
(venv)$ pip install beanie
```

Прежде чем погрузиться в интеграцию, давайте рассмотрим некоторые методы из библиотеки `Beanie`, а также то, как создаются таблицы базы данных в этом разделе.

Документ

В SQL данные, хранящиеся в строках и столбцах, содержатся в таблице. В базе данных NoSQL это называется документом. Документ представляет, как данные будут храниться в коллекции базы данных. Документы определяются так же, как и модель `Pydantic`, за исключением того, что вместо этого наследуется класс `Document` из библиотеки `Beanie`.

Пример документа определяется следующим образом:

```
from beanie import Document

class Event(Document):
    name: str
    location: str

    class Settings:
        name = "events"
```

Подкласс `Settings` определен, чтобы указать библиотеке создать имя коллекции, переданное в базе данных MongoDB.

Теперь, когда мы знаем, как создать документ, давайте рассмотрим методы, используемые для выполнения CRUD операций:

- `.insert()` и `.create()`: Методы `.insert()` и `.create()` вызываются экземпляром документа для создания новой записи в базе данных. Вы также можете использовать метод `.insert_one()` для добавления отдельной записи в базу данных.

Чтобы вставить много записей в базу данных, вызывается метод `.insert_many()`, который принимает список экземпляров документа, например:

```
event = Event(name="Packt office launch",
              location="Hybrid")
await event.create()
await Event.insert_one(event)
```

- `.find()` и `.get()`: Метод `.find()` используется для поиска списка документов, соответствующих критериям поиска, переданным в качестве аргумента метода. Метод `.get()` используется для получения одного документа, соответствующего предоставленному идентификатору. Отдельный документ, соответствующий критерию поиска, можно найти с помощью метода `.find_one()`, например следующего:

```
event = await Event.get("74478287284ff")
event = await Event.find(Event.location == "Hybrid").to_
list() # Returns a list of matching items
event = await Event.find_one(Event.location == "Hybrid") #
Returns a single event
```

- `.save()`, `.update()`, и `.upsert()`: Для обновления документа можно использовать любой из этих методов. Метод `.update()` принимает запрос на обновление, а метод `.upsert()` используется, когда документ не соответствует критериям поиска. В этой главе мы будем использовать метод `.update()`. Запрос на обновление — это инструкция, за которой следует база данных MongoDB, например, следующая:

```
event = await Event.get("74478287284ff")
update_query = {"$set": {"location": "virtual"}}
await event.update(update_query)
```

В этом блоке кода мы сначала извлекаем событие, а затем создаем запрос на обновление, чтобы установить для поля `location` в коллекции событий значение `virtual`.

- `.delete()`: Этот метод отвечает за удаление записи документа из базы данных, например:

```
event = await Event.get("74478287284ff")
await event.delete()
```

Теперь, когда мы узнали, как работают методы, содержащиеся в библиотеке Beanie, давайте инициализируем базу данных в нашем приложении планировщика событий, определим наши документы и реализуем CRUD операции.

Инициализация базы данных

Давайте посмотрим на шаги, чтобы сделать это:

1. В папке базы данных создайте модуль `connection.py`:

```
(venv)$ touch connection.py
```

Pydantic позволяет нам читать переменные среды, создавая дочерний класс родительского класса `BaseSettings`. При создании веб-API стандартной практикой является хранение переменных конфигурации в файле среды.

2. В `connection.py`, добавьте следующее:

```
from beanie import init_beanie
from motor.motor_asyncio import AsyncIOMotorClient
from typing import Optional
from pydantic import BaseSettings

class Settings(BaseSettings):
    DATABASE_URL: Optional[str] = None

    async def initialize_database(self):
        client = AsyncIOMotorClient(self.DATABASE_URL)
        await init_beanie(
            database=client.get_default_database(),
            document_models=[]
        )

class Config:
    env_file = ".env"
```

В этом блоке кода мы начинаем с импорта зависимостей, необходимых для инициализации базы данных. Затем мы определяем класс `Settings`, который имеет значение `DATABASE_URL`, которое считывается из среды `env`, определенной в подклассе `Config`. Мы также определяем метод `initialize_database` для инициализации базы данных.

Метод `init_beanie` принимает клиент базы данных, который представляет собой версию движка `mongo`, созданную в разделе `SQLModel`, и список документов.

3. Давайте обновим файлы модели в каталоге моделей, чтобы включить документы MongoDB. В `models/events.py`, замените содержимое следующим:

```
from beanie import Document
from typing import Optional, List

class Event(Document):
    title: str
    image: str
    description: str
    tags: List[str]
    location: str

class Config:
    schema_extra = {
        "example": {
            "title": "FastAPI Book Launch",
            "image": "https://linktomyimage.com/image.png",
            "description": "We will be discussing the contents of the FastAPI book in this event. Ensure to come with your own copy to win gifts!",
            "tags": ["python", "fastapi", "book", "launch"],
            "location": "Google Meet"
        }
    }
```

```
class Settings:
    name = "events"
```

4. Давайте создадим модель Pydantic для операций UPDATE:

```
class EventUpdate(BaseModel):
    title: Optional[str]
    image: Optional[str]
    description: Optional[str]
    tags: Optional[List[str]]
    location: Optional[str]

class Config:
    schema_extra = {
        "example": {
            "title": "FastAPI Book Launch",
            "image": "https://linktomyimage.com/image.png",
            "description": "We will be discussing the contents of the FastAPI book in this event. Ensure to come with your own copy to win gifts!",
            "tags": ["python", "fastapi", "book", "launch"],
            "location": "Google Meet"
        }
    }
```

5. В `model/users.py`, замените содержимое модуля следующим:

```
from typing import Optional, List
from beanie import Document, Link

from pydantic import BaseModel, EmailStr

from models.events import Event
```

```

class User(Document):
    email: EmailStr
    password: str
    events: Optional[List[Link[Event]]]

class Settings:
    name = "users"

class Config:
    schema_extra = {
        "example": {
            "email": "fastapi@packt.com",
            "password": "strong!!!",
            "events": [],
        }
    }

class UserSignIn(BaseModel):
    email: EmailStr
    password: str

```

6. Теперь, когда мы определили документы, давайте обновим поле `document_models` в `connection.py`:

```

from models.users import User
from models.events import Event

async def initialize_database(self):
    client = AsyncIOMotorClient(self.DATABASE_URL)
    await init_beanie(
        database=client.get_default_database(),
        document_models=[Event, User])

```

7. Наконец, давайте создадим файл среды, `.env`, и добавим URL-адрес базы данных, чтобы завершить этап инициализации базы данных:

```

(venv)$ touch .env
(venv)$ echo DATABASE_URL=mongodb://localhost:27017/
planner >> .env

```


Теперь, когда мы успешно добавили блоки кода для инициализации базы данных, давайте приступим к реализации методов для CRUD операций.

CRUD операции

В `connection.py`, создайте новый класс `Database`, который принимает модель в качестве аргумента во время инициализации:

```
from pydantic import BaseSettings, BaseModel
from typing import Any, List, Optional

class Database:
    def __init__(self, model):
        self.model = model
```

Модель, передаваемая во время инициализации, представляет собой класс модели документа `Event` или `User`.

Создать

Давайте создадим метод в классе `Database`, чтобы добавить запись в коллекцию базы данных:

```
async def save(self, document) -> None:
    await document.create()
    return
```

В этом блоке кода мы определили метод `save` для получения документа, который будет экземпляром документа, переданного в экземпляр `Database` в момент создания экземпляра.

Читать

Давайте создадим методы для извлечения записи базы данных или всех записей, присутствующих в коллекции базы данных.:

```
async def get(self, id: PydanticObjectId) -> Any:
    doc = await self.model.get(id)
    if doc:
        return doc
```

```

        return False

    async def get_all(self) -> List[Any]:
        docs = await self.model.find_all().to_list()
        return docs

```

Первый метод, `get()`, принимает идентификатор в качестве аргумента метода и возвращает соответствующую запись из базы данных, в то время как метод `get_all()` не принимает аргументов и возвращает список всех записей, имеющих в базе данных.

Обновить

Давайте создадим метод для обработки процесса обновления существующей записи:

```

    async def update(self, id: PydanticObjectId, body:
        BaseModel) -> Any:
        doc_id = id
        des_body = body.dict()
        des_body = {k:v for k,v in des_body.items() if v is
            not None}
        update_query = {"$set": {
            field: value for field, value in
                des_body.items()
        }}

        doc = await self.get(doc_id)
        if not doc:
            return False
        await doc.update(update_query)
        return doc

```

В этом блоке кода метод `update` принимает ID и ответственную схему Pydantic, которая будет содержать поля, обновленные из запроса PUT отправленного клиентом. Обновленное тело запроса сначала анализируется в словаре, а затем фильтруется для удаления значений `None`.

Как только это будет сделано, он вставляется в запрос на обновление, который, наконец, выполняется методом `update()` Beanie.

Удалить

Наконец, давайте создадим метод для удаления записи из базы данных:

```
async def delete(self, id: PydanticObjectId) -> bool:
    doc = await self.get(id)
    if not doc:
        return False
    await doc.delete()
    return True
```

В этом блоке кода метод проверяет, существует ли такая запись, прежде чем приступить к ее удалению из базы данных.

Теперь, когда мы заполнили наш файл базы данных нужными методами, необходимыми для выполнения CRUD операций, давайте также обновим маршруты.

routes/events.py

Начнем с обновления импорта и создания экземпляра database:

```
from beanie import PydanticObjectId
from fastapi import APIRouter, HTTPException, status
from database.connection import Database

from models.events import Event
from typing import List
event_database = Database(Event)
```

Имея импорт и экземпляр базы данных, давайте обновим все маршруты. Начните с обновления маршрутов GET:

```
@event_router.get("/", response_model=List[Event])
async def retrieve_all_events() -> List[Event]:
    events = await event_database.get_all()
    return events

@event_router.get("/{id}", response_model=Event)
async def retrieve_event(id: PydanticObjectId) -> Event:
    event = await event_database.get(id)
    if not event:
```

```

        raise HTTPException(
            status_code=status.HTTP_404_NOT_FOUND,
            detail="Event with supplied ID does not exist"
        )
    return event

```

В маршрутах GET мы вызываем методы, которые мы определили ранее в модуле базы данных. Давайте обновим POST маршруты:

```

@event_router.post("/new")
async def create_event(body: Event) -> dict:
    await event_database.save(body)
    return {
        "message": "Event created successfully"
    }

```

Давайте создадим маршрут UPDATE:

```

@event_router.put("/{id}", response_model=Event)
async def update_event(id: PydanticObjectId, body: EventUpdate)
-> Event:
    updated_event = await event_database.update(id, body)
    if not updated_event:
        raise HTTPException(
            status_code=status.HTTP_404_NOT_FOUND,
            detail="Event with supplied ID does not exist"
        )
    return updated_event

```

Наконец, давайте обновим маршрут DELETE:

```

@event_router.delete("/{id}")
async def delete_event(id: PydanticObjectId) -> dict:
    event = await event_database.delete(id)
    if not event:
        raise HTTPException(
            status_code=status.HTTP_404_NOT_FOUND,
            detail="Event with supplied ID does not exist"
        )

```

```
    return {  
        "message": "Event deleted successfully."  
    }
```

Теперь, когда мы реализовали CRUD операции для наших маршрутов событий, давайте реализуем маршруты для регистрации пользователя и входа пользователя.

routes/users.py

Начнем с обновления импорта и создания экземпляра базы данных:

```
from fastapi import APIRouter, HTTPException, status  
from database.connection import Database  
  
from models.users import User, UserSignIn  
  
user_router = APIRouter(  
    tags=["User"],  
)  
  
user_database = Database(User)
```

Затем обновите маршрут POST для подписи новых пользователей:

```
@user_router.post("/signup")  
async def sign_user_up(user: User) -> dict:  
    user_exist = await User.find_one(User.email ==  
    user.email)  
    if user_exist:  
        raise HTTPException(  
            status_code=status.HTTP_409_CONFLICT,  
            detail="User with email provided exists  
            already."  
        )  
    await user_database.save(user)  
    return {  
        "message": "User created successfully"  
    }
```

В этом блоке кода мы проверяем, существует ли такой пользователь с переданным адресом электронной почты, прежде чем добавлять его в базу данных. Давайте добавим маршрут для входа пользователей:

```
@user_router.post("/signin")
async def sign_user_in(user: UserSignIn) -> dict:
    user_exist = await User.find_one(User.email ==
    user.email)
    if not user_exist:
        raise HTTPException(
            status_code=status.HTTP_404_NOT_FOUND,
            detail="User with email does not exist."
        )
    if user_exist.password == user.password:
        return {
            "message": "User signed in successfully."
        }
    raise HTTPException(
        status_code=status.HTTP_401_UNAUTHORIZED,
        detail="Invalid details passed."
    )
```

В этом определенном маршруте мы сначала проверяем, существует ли пользователь, прежде чем проверять действительность его учетных данных. Используемый здесь метод аутентификации является базовым и *не рекомендуется* в производственной среде. Мы рассмотрим правильные процедуры аутентификации в следующей главе.

Теперь, когда мы реализовали маршруты, давайте запустим экземпляр MongoDB, а также наше приложение. Создайте папку для размещения нашей базы данных MongoDB и запустите экземпляр MongoDB:

```
(venv)$ mkdir store
(venv)$ mongod --dbpath store
```

Далее в другом окне запускаем приложение:

```
(venv)$ python main.py
INFO:      Uvicorn running on http://0.0.0.0:8080 (Press CTRL+C
to quit)
INFO:      Started reloader process [3744] using statreload
INFO:      Started server process [3747]
INFO:      Waiting for application startup.
INFO:      Application startup complete.
```

Давайте протестируем маршруты событий:

1. Создайте событие:

```
(venv)$ curl -X 'POST' \
  'http://0.0.0.0:8080/event/new' \
  -H 'accept: application/json' \
  -H 'Content-Type: application/json' \
  -d '{
    "title": "FastAPI Book Launch",
    "image": "https://linktomyimage.com/image.png",
    "description": "We will be discussing the contents
of the FastAPI book in this event. Ensure to come
with your own copy to win gifts!",
    "tags": [
      "python",
      "fastapi",
      "book",
      "launch"
    ],
    "location": "Google Meet"
  }'
```

Вот ответ от предыдущей операции:

```
{
  "message": "Event created successfully"
}
```

2. Получить все события:

```
(venv)$ curl -X 'GET' \  
  'http://0.0.0.0:8080/event/' \  
  -H 'accept: application/json'
```

Предыдущий запрос возвращает список событий:

```
[  
  {  
    "_id": "624daab1585059e8a3fa77ac",  
    "title": "FastAPI Book Launch",  
    "image": "https://linktomyimage.com/image.png",  
    "description": "We will be discussing the contents  
of the FastAPI book in this event. Ensure to come  
with your own copy to win gifts!",  
    "tags": [  
      "python",  
      "fastapi",  
      "book",  
      "launch"  
    ],  
    "location": "Google Meet"  
  }  
]
```

3. Получить событие:

```
(venv)$ curl -X 'GET' \  
  'http://0.0.0.0:8080/event/624daab1585059e8a3fa77ac' \  
  -H 'accept: application/json'
```

Эта операция возвращает событие, соответствующее предоставленному ID:

```
{  
  "_id": "624daab1585059e8a3fa77ac",  
  "title": "FastAPI Book Launch",  
  "image": "https://linktomyimage.com/image.png",  
  "description": "We will be discussing the contents  
of the FastAPI book in this event. Ensure to come  
with your own copy to win gifts!",  
}
```



```
"tags": [  
    "python",  
    "fastapi",  
    "book",  
    "launch"  
],  
"location": "Google Meet"  
}
```

4. Обновим локацию события на Hybrid:

```
(venv)$ curl -X 'PUT' \  
    'http://0.0.0.0:8080/event/624daab1585059e8a3fa77ac'  
 \  
-H 'accept: application/json' \  
-H 'Content-Type: application/json' \  
-d '{  
    "location": "Hybrid"  
}'  
  
{  
    "_id": "624daab1585059e8a3fa77ac",  
    "title": "FastAPI Book Launch",  
    "image": "https://linktomyimage.com/image.png",  
    "description": "We will be discussing the contents  
of the FastAPI book in this event. Ensure to come  
with your own copy to win gifts!",  
    "tags": [  
        "python", "fastapi",  
        "book",  
        "launch"  
    ],  
    "location": "Hybrid"  
}
```

5. Наконец, давайте удалим событие:

```
(venv)$ curl -X 'DELETE' \  
  'http://0.0.0.0:8080/event/624daab1585059e8a3fa77ac' \  
  \  
  -H 'accept: application/json'
```

Вот ответ, полученный на запрос:

```
{  
  "message": "Event deleted successfully."  
}
```

6. Теперь, когда мы протестировали маршруты для событий, давайте создадим нового пользователя, а затем войдем в систему:

```
(venv)$ curl -X 'POST' \  
  'http://0.0.0.0:8080/user/signup' \  
  -H 'accept: application/json' \  
  -H 'Content-Type: application/json' \  
  -d '{  
    "email": "fastapi@packt.com",  
    "password": "strong!!!",  
    "events": []  
  }'
```

Запрос возвращает ответ:

```
{  
  "message": "User created successfully"  
}
```

Повторный запуск запроса возвращает ошибку HTTP 409, указывающую на конфликт:

```
{  
  "detail": "User with email provided exists already."  
}
```

Изначально мы разработали маршрут для проверки существующих пользователей, чтобы избежать дублирования.

7. Теперь давайте отправим POST запрос для входа только что созданному пользователю:

```
(venv)$ curl -X 'POST' \  
  'http://0.0.0.0:8080/user/signin' \  
  -H 'accept: application/json' \  
  -H 'Content-Type: application/json' \  
  -d '{  
    "email": "fastapi@packt.com",  
    "password": "strong!!!"  
  }'
```

Запрос возвращает сообщение об успешном завершении HTTP 200:

```
{  
  "message": "User signed in successfully."  
}
```

Мы успешно реализовали CRUD-операции с помощью библиотеки Beanie.

Резюме

В этой главе мы узнали, как добавлять базы данных SQL и NoSQL с помощью SQLAlchemy и Beanie соответственно. Мы использовали все наши знания из предыдущих глав. Мы также проверили маршруты, чтобы убедиться, что они работают по плану.

В следующей главе вы познакомитесь с защитой вашего приложения. Сначала вас научат основам аутентификации, а также различным методам аутентификации, доступным разработчикам FastAPI. Затем вы внедрите систему аутентификации, основанную на **JSON веб-токенах (JWT)**, и защитите маршруты для создания, обновления и удаления событий. Наконец, вы измените маршрут, чтобы создать события, позволяющие связать события с пользователем.

7

Защита приложений FastAPI

В предыдущей главе мы рассмотрели, как подключить приложение FastAPI к базе данных SQL и NoSQL. Мы успешно реализовали методы базы данных и обновили существующие маршруты, чтобы обеспечить взаимодействие между приложением и базой данных. Однако приложение планировщика по-прежнему позволяет любому пользователю добавлять событие, а не только пользователям, прошедшим проверку подлинности. В этой главе мы защитим приложение с помощью **JSON веб-токенах (JWT)** и ограничим некоторые операции с событиями только для аутентифицированных пользователей.

Защита приложения включает в себя дополнительные меры безопасности для ограничения доступа к функциям приложения от неавторизованных лиц для предотвращения взлома или незаконных модификаций приложения. Аутентификация — это процесс проверки учетных данных, переданных объектом, а авторизация просто означает предоставление объекту разрешения на выполнение определенных действий. После проверки учетных данных объект получает право выполнять различные действия.

К концу этой главы вы сможете добавить уровень аутентификации в приложение FastAPI. В этой главе объясняются процессы защиты паролей путем их хеширования, добавления уровня аутентификации и защиты маршрутов от неавторизованных пользователей. В этой главе мы рассмотрим следующие темы:

- Методы аутентификации в FastAPI
- Защита приложения с помощью OAuth2 и JWT
- Защита маршрутов с помощью внедрения зависимостей
- Настройка CORS

Технические требования

Чтобы продолжить, требуется компонент базы данных MongoDB. Процедуры установки для вашей операционной системы можно найти в их официальной документации. Код, использованный в этой главе, можно найти по адресу <https://github.com/PacktPublishing/Building-Python-Web-APIs-with-FastAPI/tree/main/ch07/planner>.

Методы аутентификации в FastAPI

В FastAPI доступно несколько методов аутентификации. FastAPI поддерживает распространенные методы аутентификации: базовую HTTP-аутентификацию, файлы cookie и аутентификацию на основе токенов. Давайте кратко рассмотрим, что влечет за собой каждый метод:

- **Базовая HTTP-аутентификация:** В этом методе аутентификации учетные данные пользователя, которые обычно представляют собой имя пользователя и пароль, отправляются через HTTP-заголовок авторизации. Запрос, в свою очередь, возвращает заголовок `WWW-Authenticate` содержащий, базовое значение и необязательный параметр области, который указывает ресурс, к которому выполняется запрос аутентификации.
- **Cookies:** Файлы cookie используются, когда данные должны храниться на стороне клиента, например, в веб-браузерах. Приложения FastAPI также могут использовать файлы cookie для хранения пользовательских данных, которые могут быть получены сервером в целях аутентификации.
- **Аутентификацию на основе токенов:** Этот метод аутентификации включает использование токенов безопасности, называемых токенами носителя. Эти токены отправляются вместе с ключевым словом `Bearer` в запросе заголовка авторизации. Наиболее часто используемым токеном является JWT, который обычно представляет собой словарь, содержащий идентификатор пользователя и срок действия токена.

Каждый из перечисленных здесь методов аутентификации имеет свои конкретные варианты использования, а также свои плюсы и минусы. Однако в этой главе мы будем использовать аутентификацию по токену.

Методы аутентификации внедряются в приложения FastAPI в виде зависимостей, которые вызываются во время выполнения. Это просто означает, что, когда методы аутентификации определены, они бездействуют до тех пор, пока не будут внедрены в место их использования. Это действие называется **внедрением зависимостей**.

Внедрение зависимостей

Внедрение зависимостей — это шаблон, в котором объект — в данном случае функция — получает переменную экземпляра, необходимую для дальнейшего выполнения функции.

В FastAPI зависимости внедряются путем объявления их в аргументах функции операции пути. Мы использовали внедрение зависимостей в предыдущих главах. Вот пример из предыдущей главы, где мы извлекаем поле электронной почты из пользовательской модели, переданной в функцию:

```
@user_router.post("/signup")
async def sign_user_up(user: User) -> dict:
    user_exist = await User.find_one(User.email == user.email)
```

В этом блоке кода определенная зависимость — это класс модели User, который вводится в функцию `sign_user_up()`. Внедрив модель User в аргумент пользовательской функции, мы можем легко получить атрибуты объекта.

Создание и использование зависимости

В FastAPI зависимость может быть определена либо как функция, либо как класс. Созданная зависимость дает нам доступ к ее базовым значениям или методам, избавляя от необходимости создавать эти объекты в наследующих их функциях. Внедрение зависимостей помогает уменьшить повторение кода в некоторых случаях, например, при принудительной проверке подлинности и авторизации.

Пример зависимости определяется следующим образом:

```
async def get_user(token: str):
    user = decode_token(token)
    return user
```

Эта зависимость представляет собой функцию, которая принимает `token` в качестве аргумента и возвращает `user` параметр из внешней функции, `decode_token`. Чтобы использовать эту зависимость, объявленный аргумент зависимой функции должен иметь параметр `Depends`, например:

```
from fastapi import Depends

@router.get("/user/me")
```

```
async def get_user_details(user: User = Depends(get_user)):
    return user
```

Функция маршрута здесь зависит от функции `get_user`, которая служит ее зависимостью. Это означает, что для доступа к предыдущему маршруту должна быть удовлетворена зависимость `get_user`.

Класс `Depends`, который импортируется из библиотеки `FastAPI`, отвечает за прием функции, переданной в качестве аргумента, и выполнение ее при вызове конечной точки, автоматически делая доступными для конечной точки, они возвращают значение переданной ей функции.

Теперь, когда у вас есть представление о том, как создается зависимость и как она используется, давайте создадим зависимость аутентификации для приложения планировщика событий.

Защита приложения с помощью OAuth2 и JWT

В этом разделе мы создадим систему аутентификации для приложения планировщика событий. Мы будем использовать поток паролей OAuth2, который требует от клиента отправки имени пользователя и пароля в качестве данных формы. Имя пользователя в нашем случае — это электронная почта, используемая при создании учетной записи.

Когда данные формы отправляются на сервер от клиента, в качестве ответа отправляется **токен доступа**, который является подписанным JWT. Обычно выполняется фоновая проверка для проверки учетных данных, отправленных на сервер, перед созданием токена для дальнейшей авторизации. Чтобы авторизовать аутентифицированного пользователя, JWT имеет префикс `Bearer` при отправке через заголовок для авторизации действия на сервере.

Что такое JWT и почему он подписан?

JWT — это закодированная строка, обычно содержащая словарь, содержащий полезную нагрузку, подпись и алгоритм. JWT подписываются с использованием уникального ключа, известного только серверу и клиенту, чтобы избежать подделки закодированной строки внешним органом.



Рисунок 7.1 – Схема аутентификации

Теперь, когда у нас есть представление о том, как работает процесс аутентификации, давайте создадим необходимую папку и файлы, необходимые для настройки системы аутентификации в нашем приложении:

1. В папке проекта сначала создайте папку `auth`:

```
(venv)$ mkdir auth
```

2. Далее создайте следующие файлы в папке `auth`:

```
(venv)$ cd auth && touch {__init__,jwt_handler,authenticate,hash_password}.py
```

Предыдущая команда создает четыре файла:

- `jwt_handler.py`: Этот модуль будет содержать функции, необходимые для кодирования и декодирования строк JWT.
- `authenticate.py`: Этот модуль будет содержать зависимость аутентификации, которая будет внедрена в наши маршруты для принудительной аутентификации и авторизации.
- `hash_password.py`: Этот модуль будет содержать функции, которые будут использоваться для шифрования пароля пользователя при регистрации и сравнения паролей при входе.
- `__init__.py`: Этот файл указывает содержимое папки как пакет.

Теперь, когда файлы созданы, давайте создадим отдельные компоненты. Начнем с создания компонентов для хеширования паролей пользователей.

Хэширование паролей

В предыдущей главе мы хранили пароли пользователей в виде обычного текста. Это очень небезопасная и запрещенная практика при создании API. Пароли должны быть зашифрованы или хешированы с использованием соответствующих библиотек. Мы будем шифровать пароли пользователей с помощью `bcrypt`.

Давайте установим библиотеку `passlib`. В этой библиотеке находится алгоритм хеширования `bcrypt`, который мы будем использовать для хеширования паролей пользователей:

```
(venv)$ pip install passlib[bcrypt]
```

Теперь, когда мы установили библиотеку, давайте создадим функции для хеширования паролей в `hash_password.py`:

```
from passlib.context import CryptContext

pwd_context = CryptContext(schemes=["bcrypt"],
```



```
deprecated="auto")

class HashPassword:
    def create_hash(self, password: str):
        return pwd_context.hash(password)

    def verify_hash(self, plain_password: str,
hashed_password: str):
        return pwd_context.verify(plain_password,
hashed_password)
```

В предыдущем блоке кода мы начинаем с импорта `CryptContext`, который использует схему `bcrypt` для хеширования переданных ему строк. Контекст хранится в переменной контекста `pwd_context`, что дает нам доступ к методам, необходимым для выполнения нашей задачи.

Затем определяется класс `HashPassword`, который содержит два метода: `create_hash` и `verify_hash`:

- Метод `create_hash` принимает строку и возвращает хешированное значение.
- `verify_hash` берет простой пароль и хешированный пароль и сравнивает их. Функция возвращает логическое значение, указывающее, совпадают ли переданные значения или нет.

Теперь, когда мы создали класс для обработки хеширования паролей, давайте обновим маршрут регистрации, чтобы хешировать пароль пользователя перед его сохранением в базе данных:

`routes/users.py`

```
from auth.hash_password import HashPassword
from database.connection import Database

user_database = Database(User)
hash_password = HashPassword()

@user_router.post("/signup")
async def sign_user_up(user: User) -> dict:
    user_exist = await User.find_one(User.email ==
```

```

user.email)

if user_exist:
    raise HTTPException(
        status_code=status.HTTP_409_CONFLICT,
        detail="User with email provided exists
        already."
    )
    hashed_password = hash_password.create_hash(
    user.password)
    user.password = hashed_password
    await user_database.save(user)
    return {
        "message": "User created successfully"
    }

```

Теперь, когда мы обновили маршрут регистрации пользователя, чтобы хешировать пароль перед сохранением, давайте создадим нового пользователя для подтверждения. В окне терминала запустите приложение:

```

(venv)$ python main.py
INFO:      Uvicorn running on http://0.0.0.0:8080 (Press CTRL+C
to quit)
INFO:      Started reloader process [8144] using statreload
INFO:      Started server process [8147]
INFO:      Waiting for application startup.
INFO:      Application startup complete.

```

В другом окне терминала запустите экземпляр MongoDB:

```
$ mongod --dbpath database --port 27017
```

Далее создадим нового пользователя:

```

$ curl -X 'POST' \
'http://0.0.0.0:8080/user/signup' \
-H 'accept: application/json' \
-H 'Content-Type: application/json' \
-d '{
    "email": "reader@packt.com",

```

```
"password": "exemplary"
}'
```

Мы получаем успешный ответ на запрос выше:

```
{
  "message": "User created successfully"
}
```

Теперь, когда мы создали пользователя, давайте проверим, что пароль, отправленный в базу данных, был хеширован. Для этого мы создадим интерактивный сеанс MongoDB, который позволит нам запускать команды из базы данных.

В новом окне терминала выполните следующие команды:

```
$ mongo --port 27017
```

Запускается интерактивный сеанс MongoDB:

```

an upcoming release.
We recommend you begin using "mongosh".
For installation instructions, see
https://docs.mongodb.com/mongodb-shell/install/
=====
---
The server generated these startup warnings when booting:
  2022-04-24T14:45:42.676+01:00: Access control is not enabled for the database. Read and write access to data and configuration is unrestricted
  2022-04-24T14:45:42.677+01:00: This server is bound to localhost. Remote systems will be unable to connect to this server. Start the server with --bind_ip <address> to specify which IP addresses it should serve responses from, or with --bind_ip_all to bind to all interfaces. If this behavior is desired, start the server with --bind_ip 127.0.0.1 to disable this warning
  2022-04-24T14:45:42.677+01:00: Soft rlimits for open file descriptors too low
  2022-04-24T14:45:42.677+01:00:           currentValue: 10240
  2022-04-24T14:45:42.677+01:00:           recommendedMinimum: 64000
---
---
  Enable MongoDB's free cloud-based monitoring service, which will then receive and display metrics about your deployment (disk utilization, CPU, operation statistics, etc).

  The monitoring data will be available on a MongoDB website with a unique URL accessible to you and anyone you share the URL with. MongoDB may use this information to make product improvements and to suggest MongoDB products and deployment options to you.

  To enable free monitoring, run the following command: db.enableFreeMonitoring()
  To permanently disable this reminder, run the following command: db.disableFreeMonitoring()
---
>

```

Рисунок 7.2 – Интерактивный сеанс MongoDB

При работающем интерактивном сеансе выполните серию команд, чтобы переключиться на базу данных планировщика и получить все пользовательские записи:

```
> use planner
> db.users.find({})
```



```

mongo --port 27017
> use planner
switched to db planner
> db.users.find({})
{ "_id" : ObjectId("62655d4b52b6386b8b11b5fb"), "email" : "reader@packt.com", "password" : "$2b$12$Jcc5VXty397UDGeg3bdq0encodqNvi
f8npVj06P1IU1NFIjONGP/m", "events" : [ ] }
>

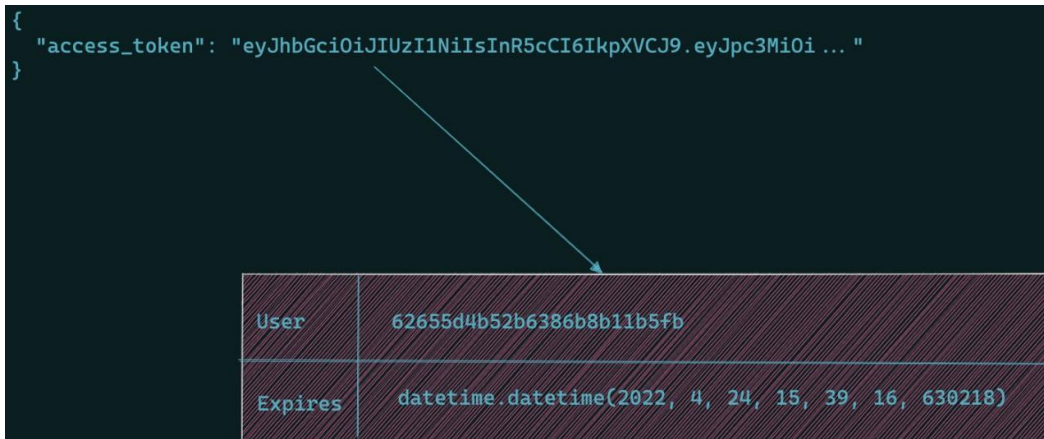
```

Рисунок 7.3 – Результат запроса найти всех пользователей

Предыдущая команда возвращает список пользователей, и теперь мы можем подтвердить, что пароль пользователя был хеширован до того, как он был сохранен в базе данных. Теперь, когда мы успешно создали компоненты для безопасного хранения паролей пользователей, давайте создадим компоненты для создания и проверки JWTs.

Создание и проверка токенов доступа

Создание JWT делает нас на шаг ближе к защите нашего приложения. Полезная нагрузка токена будет содержать идентификатор пользователя и время истечения срока действия перед кодированием в длинную строку, как показано здесь:



```

{
  "access_token": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJpc3MiOi... "
}

```

User	62655d4b52b6386b8b11b5fb
Expires	datetime.datetime(2022, 4, 24, 15, 39, 16, 630218)

Рисунок 7.4 – Анатомия JWT

Ранее мы узнали, почему JWT подписаны. JWT подписываются секретным ключом, известным только отправителю и получателю. Давайте обновим класс `Settings` в модуле `database/database.py`, а также файл среды, `.env`, включив в него переменную `SECRET_KEY`, которая будет использоваться для подписи JWTs:

database/database.py

```

class Settings(BaseSettings):
    SECRET_KEY: Optional[str] = None

```

.env

```
SECRET_KEY=HI5HL3V3L$3CR3T
```

После этого добавьте следующий импорт в `jwt_handler.py`:

```
import time
from datetime import datetime

from fastapi import HTTPException, status
from jose import jwt, JWTError
from database.database import Settings
```

В предыдущем блоке кода мы импортировали модули `time`, класс `HTTPException`, а также статус из `FastAPI`. Мы также импортировали библиотеку `jose` отвечающую за кодирование и декодирование JWT, и класс `Settings`.

Далее мы создадим экземпляр класса `Settings`, чтобы мы могли получить переменную `SECRET_KEY` и создать функцию, отвечающую за создание токена:

```
settings = Settings()

def create_access_token(user: str) -> str:
    payload = {
        "user": user,
        "expires": time.time() + 3600
    }

    token = jwt.encode(payload, settings.SECRET_KEY,
                       algorithm="HS256")
    return token
```

В предыдущем блоке кода функция принимает строковый аргумент, который передается в словарь полезной нагрузки. Словарь полезной нагрузки содержит пользователя и время истечения срока действия, которое возвращается при декодировании JWT.

Срок действия устанавливается равным часу с момента создания. Затем полезная нагрузка передается методу `encode()`, который принимает три параметра:

- **Payload:** Словарь, содержащий значения для кодирования.
- **Key:** Ключ, используемый для подписи полезной нагрузки.
- **Algorithm:** Алгоритм, используемый для подписи полезной нагрузки. По умолчанию и наиболее распространенным является алгоритм **HS256**.

Далее давайте создадим функцию для проверки подлинности токена, отправленного в наше приложение:

```
def verify_access_token(token: str) -> dict:
    try:
        data = jwt.decode(token, settings.SECRET_KEY,
                           algorithms=["HS256"])

        expire = data.get("expires")

        if expire is None:
            raise HTTPException(
                status_code=status.HTTP_400_BAD_REQUEST,
                detail="No access token supplied"
            )

        if datetime.utcnow() >
            datetime.utcfromtimestamp(expire):
            raise HTTPException(
                status_code=status.HTTP_403_FORBIDDEN,
                detail="Token expired!"
            )

        return data

    except JWTError:
        raise HTTPException(
            status_code=status.HTTP_400_BAD_REQUEST,
            detail="Invalid token"
        )
```

В предыдущем блоке кода функция принимает в качестве аргумента строку токена и выполняет несколько проверок в блоке `try`. Сначала функция проверяет срок действия токена. Если срок действия не указан, значит, токен не был предоставлен. Вторая проверка — валидность токена — генерируется исключение, информирующее пользователя об истечении срока действия токена. Если токен действителен, возвращается декодированная полезная нагрузка.

В блоке `except` для любой ошибки JWT выдается исключение неверного запроса.

Теперь, когда мы реализовали функции для создания и проверки токенов, управляемых в приложении, давайте создадим функцию, которая проверяет аутентификацию пользователя и служит зависимостью.

Обработка аутентификации пользователя

Мы успешно внедрили компоненты для хеширования и сравнения паролей, а также компоненты для создания и декодирования JWT. Давайте реализуем функцию зависимости, которая будет внедрена в маршруты событий. Эта функция будет служить единственным источником правды для извлечения пользователя для активного сеанса.

В `auth/authenticate.py` добавьте следующее:

```
from fastapi import Depends, HTTPException, status
from fastapi.security import OAuth2PasswordBearer

from auth.jwt_handler import verify_access_token

oauth2_scheme = OAuth2PasswordBearer(tokenUrl="/user/signin")

async def authenticate(token: str = Depends(oauth2_scheme)) -> str:
    if not token:
        raise HTTPException(
            status_code=status.HTTP_403_FORBIDDEN,
            detail="Sign in for access"
        )

    decoded_token = verify_access_token(token)
    return decoded_token["user"]
```

В предыдущем блоке кода мы начинаем с импорта необходимых зависимостей:

- `Depends`: Это вводит `oauth2_scheme` в функцию в качестве зависимости.
- `OAuth2PasswordBearer`: Этот класс сообщает приложению, что схема безопасности присутствует.
- `verify_access_token`: Эта функция, определенная в разделе создания и проверки токена доступа, будет использоваться для проверки действительности токена.

Затем мы определяем URL токена для схемы OAuth2 и функцию аутентификации. Функция аутентификации принимает токен в качестве аргумента. В функцию в качестве зависимости внедрена схема OAuth. Токен декодируется, и пользовательское поле полезной нагрузки возвращается, если токен действителен, в противном случае возвращаются адекватные ответы об ошибках, как определено в функции `verify_access_token`.

Теперь, когда мы успешно создали зависимость для защиты маршрутов, давайте обновим поток аутентификации в маршрутах, а также добавим функцию аутентификации в маршруты событий.

Обновление приложения

В этом разделе мы обновим маршруты для использования новой модели аутентификации. Наконец, мы обновим маршрут POST для добавления события, чтобы заполнить поле событий в записи пользователя.

Обновление маршрута входа пользователя

В `routes/users.py` обновите импорт:

```
from fastapi import APIRouter, Depends, HTTPException, status
from fastapi.security import OAuth2PasswordRequestForm
from auth.jwt_handler import create_access_token

from models.users import User
```

Мы импортировали класс `OAuth2PasswordRequestForm` из модуля безопасности FastAPI. Это будет введено в маршрут входа для получения отправленных учетных данных: имя пользователя и пароль. Давайте обновим функцию маршрута `sign_user_in()`:

```
async def sign_user_in(user: OAuth2PasswordRequestForm =
    Depends()) -> dict:
    user_exist = await User.find_one(User.email ==
```



```
user.username)
..
if hash_password.verify_hash(user.password,
user_exist.password):
    access_token = create_access_token(
        user_exist.email)
    return {
        "access_token": access_token,
        "token_type": "Bearer"
    }
```

В предыдущем блоке кода мы внедрили класс OAuth2PasswordRequestForm в качестве зависимости для этой функции, гарантируя строгое соблюдение спецификации OAuth. В теле функции мы сравниваем пароль и возвращаем токен доступа и тип токена. Прежде чем протестировать обновленный маршрут, давайте создадим модель ответа для маршрута входа в `models/users.py`, чтобы заменить класс модели `UserSignIn`, который больше не используется:

```
class TokenResponse(BaseModel):
    access_token: str
    token_type: str
```

Обновите импорт и модель ответа для маршрута входа:

```
from models.users import User, TokenResponse

@user_router.post("/signin", response_model=TokenResponse)
```

Давайте посетим интерактивные документы, чтобы убедиться, что тело запроса соответствует спецификациям OAuth2 по адресу <http://0.0.0.0:8080/docs>:

POST /user/signin Sign User In

Parameters Try it out

No parameters

Request body required application/x-www-form-urlencoded

```

grant_type
string
pattern: password

username * required
string

password * required
string

scope
string

client_id
string

client_secret
string

```

Responses

Рисунок 7.5 – Текст запроса для обновленного маршрута входа

Давайте войдем в систему, чтобы убедиться, что маршрут работает правильно

```

$ curl -X 'POST' \
  'http://0.0.0.0:8080/user/signin' \
  -H 'accept: application/json' \
  -H 'Content-Type: application/x-www-form-urlencoded' \
  -d 'grant_type=&username=reader%40packt.
com&password=exemplary&scope=&client_id=&client_secret='

```

Возвращаемый ответ представляет собой токен доступа и тип токена:

```
{
  "access_token": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1c2VyIjoicmVhZGVyQHhY2t0LmNvbSIsImV4cGlyZXMiOjE2NTA4Mjc0MjQuMDg2NDAxZQ.LY4i5EjIzlsKdfMyWKi7XH7lLeDuVt3832hNfkQx8C8",
  "token_type": "Bearer"
}
```

Теперь, когда мы убедились, что маршрут работает должным образом, давайте обновим маршруты событий, чтобы разрешать только авторизованным пользователям события **CREATE**, **UPDATE** и **DELETE**.

Обновление маршрутов событий

Теперь, когда у нас есть наша аутентификация, давайте добавим зависимость аутентификации в функции маршрута **POST**, **PUT** и **DELETE**:

```
from auth.authenticate import authenticate

async def create_event(body: Event, user: str =
    Depends(authenticate)) -> dict:
    ..

async def update_event(id: PydanticObjectId, body: EventUpdate,
    user: str = Depends(authenticate)) -> Event:
    ..

async def delete_event(id: PydanticObjectId, user: str =
    Depends(authenticate)) -> dict:
    ..
```

После внедрения зависимостей веб-сайт интерактивной документации автоматически обновляется для отображения защищенных маршрутов. Если мы войдем в <http://0.0.0.0:8080/docs>, мы увидим кнопку **авторизации** в правом верхнем углу и замки на маршрутах событий:

FastAPI 0.1.0 OAS3
/openapi.json

Authorize

User

Events

GET /event/ Retrieve All Events

GET /event/{id} Retrieve Event

PUT /event/{id} Update Event

DELETE /event/{id} Delete Event

POST /event/new Create Event

default

Schemas

Рисунок 7.6 – Обновленная страница документации

Если мы нажмем кнопку «Авторизовать», отобразится модальное окно входа. Ввод наших учетных данных и пароля возвращает следующий экран:

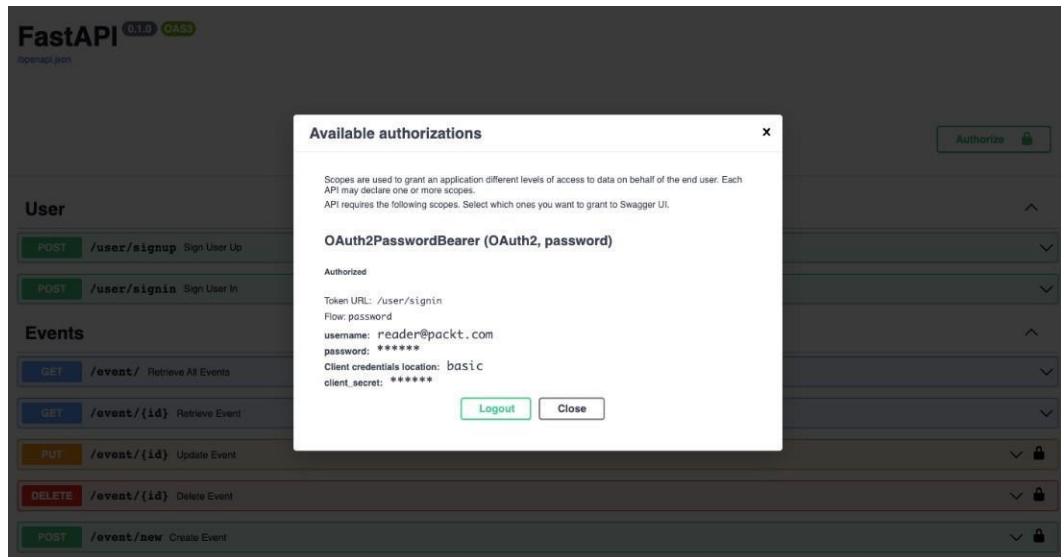


Рисунок 7.7 – Авторизованный пользователь

Теперь, когда мы успешно вошли в систему, мы можем создать событие:

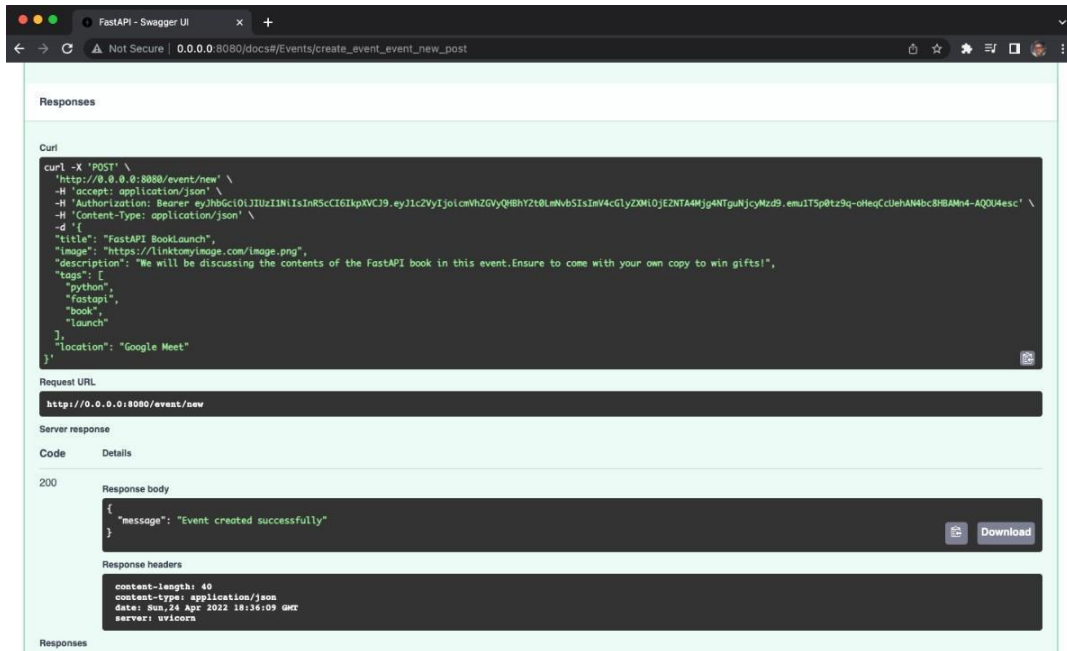


Рисунок 7.8 – Создать новое событие

Те же операции можно выполнить из командной строки. Во-первых, давайте получим наш токен доступа:

```
$ curl -X 'POST' \
  'http://0.0.0.0:8080/user/signin' \
  -H 'accept: application/json' \
  -H 'Content-Type: application/x-www-form-urlencoded' \
  -d 'grant_type=&username=reader%40packt.com&password=exemplary&scope=&client_id=&client_secret='
```

Отправленный запрос возвращает токен доступа, который представляет собой строку JWT, и тип токена, который имеет тип Bearer:

```
{
  "access_token": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1c2VyIjoicmVhZGVyQHhY2t0LmNvbSIsImV4cGlyZXMiOjE2NTA4MjkxODMuNTg3NjAyfQ.MOXjI5GXnyzGNftdlxDGyM119_L11uPq8yCxBHepf04",
  "token_type": "Bearer"
}
```

Теперь давайте создадим новое событие из командной строки:

```
$ curl -X 'POST' \
  'http://0.0.0.0:8080/event/new' \
  -H 'accept: application/json' \
  -H 'Authorization: Bearer eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXV
CJ9.eyJ1c2VyIjoicmVhZGVyQHBhY2t0LmNvbSIsImV4cGlyZXMiOjE2NTA4Mjk
xODMuNTg3NjAyfQ.MOXjI5GXnyzGNftdlxDGyM119_L1luPq8yCxBHepf04' \
  -H 'Content-Type: application/json' \
  -d '{
    "title": "FastAPI Book Launch CLI",
    "image": "https://linktomyimage.com/image.png",
    "description": "We will be discussing the contents of the
FastAPI book in this event.Ensure to come with your own
copy to win gifts!",
    "tags": [
      "python",
      "fastapi",
      "book",
      "launch"
    ],
    "location": "Google Meet"
  }'
```

В отправленном здесь запросе также отправляется заголовок `Authorization: Bearer`, чтобы сообщить приложению, что мы уполномочены выполнять это действие. Полученный ответ следующий:

```
{ "message": "Event created successfully" }
```

Если мы попытаемся создать событие без передачи заголовка авторизации с действительным токеном, будет возвращена ошибка HTTP 401 Unauthorized:

```
$ curl -X 'POST' \
  'http://0.0.0.0:8080/event/new' \
  -H 'accept: application/json' \
  -H 'Content-Type: application/json' \
  -d '{
    "title": "FastAPI BookLaunch",
```

```

"image": "https://linktomyimage.com/image.png",
"description": "We will be discussing the contents of the
FastAPI book in this event.Ensure to come with your own
copy to win gifts!",
"tags": [
    "python",
    "fastapi",
    "book",
    "launch"
],
"location": "Google Meet"
}'

```

Вот ответ:

```

$ {
  "detail": "Not authenticated"
}

```

Теперь, когда мы успешно защитили маршруты, давайте обновим защищенные маршруты следующим образом:

- Маршрут POST: добавление созданного события в список событий, принадлежащих пользователю.
- Маршрут UPDATE: измените маршрут, чтобы можно было обновить только событие, созданное пользователем.
- Маршрут DELETE: измените маршрут, чтобы удалить можно было только событие, созданное пользователем.

В предыдущем разделе мы успешно внедрили зависимости аутентификации в наши операции маршрутизации. Чтобы легко идентифицировать события и предотвратить удаление пользователем события другого пользователя, мы обновим класс документа события, а также маршруты.

Обновление класса документа события и маршрутов

Добавьте поле `creator` в класс документа `Event` в `models/events.py`:

```

class Event(Document):
    creator: Optional[str]

```

Это поле позволит нам ограничить операции, выполняемые с событием, только пользователем.

Далее давайте изменим маршрут POST, чтобы обновить поле `creator` при создании нового события в `routes/events.py`:

```
@event_router.post("/new")
async def create_event(body: Event, user: str =
Depends(authenticate)) -> dict:
    body.creator = user
    await event_database.save(body)
    return {
        "message": "Event created successfully"
    }
```

В предыдущем блоке кода мы обновили маршрут POST, чтобы добавить адрес электронной почты текущего пользователя в качестве создателя события. Если вы создаете новое мероприятие, оно сохраняется вместе с адресом электронной почты создателя:

```
$ curl -X 'POST' \
  'http://0.0.0.0:8080/event/new' \
  -H 'accept: application/json' \
  -H 'Authorization: Bearer eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXV
CJ9.eyJlc2VyIjoicmVhZGVyQHBhY2t0LmNvbSIsImV4cGlyZXMiOjE2NTA4MzI
5NjQumTU3MjQ4fQ.RxR1TYMx91JtVMNzYcT7718xXWX7skTCfWbnJxyf6fU' \
  -H 'Content-Type: application/json' \
  -d '{
    "title": "FastAPI Book Launch",
    "image": "https://linktomyimage.com/image.png",
    "description": "We will be discussing the contents of the
FastAPI book in this event.Ensure to come with your own
copy to win gifts!",
    "tags": [
      "python",
      "fastapi",
      "book",
      "launch"
    ],
    "location": "Google Meet"
```



```
}'
```

Ответ, возвращенный из запроса выше:

```
{  
  "message": "Event created successfully"  
}
```

Далее давайте получим список событий, хранящихся в базе данных:

```
$ curl -X 'GET' \  
  'http://0.0.0.0:8080/event/' \  
  -H 'accept: application/json'
```

Ответ на запрос выше:

```
[  
  {  
    "_id": "6265a807e0c8daefb72261ea",  
    "creator": "reader@packt.com",  
    "title": "FastAPI BookLaunch",  
    "image": "https://linktomyimage.com/image.png",  
    "description": "We will be discussing the contents of the  
FastAPI book in this event.Ensure to come with your own  
copy to win gifts!",  
    "tags": [  
      "python",  
      "fastapi",  
      "book",  
      "launch"  
    ],  
    "location": "Google Meet"  
  },  
]
```

Далее давайте обновим маршрут UPDATE:

```
@event_router.put("/{id}", response_model=Event)  
async def update_event(id: PydanticObjectId, body: EventUpdate,
```

```

user: str = Depends(authenticate)) -> Event:
    event = await event_database.get(id)
    if event.creator != user:
        raise HTTPException(
            status_code=status.HTTP_400_BAD_REQUEST,
            detail="Operation not allowed"
        )

```

В предыдущем блоке кода функция маршрута проверяет, может ли текущий пользователь редактировать событие, прежде чем продолжить, в противном случае она вызывает исключение неверного запроса HTTP 400.

Вот пример использования другого пользователя:

```

$ curl -X 'PUT' \
  'http://0.0.0.0:8080/event/6265a83fc823a3c912830074' \
  -H 'accept: application/json' \
  -H 'Authorization: Bearer eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1c2VyIjoiaZmFzdGFwaUBwYWNrdC5jb20iLCJleHBpcmVzIjoxNjUwODMzOTc2LjI2Nzg5MX0.MMRT6pwEDBVHTU5C1a6MV8j9wCfWhqbza9NBpZz08xE' \
  -H 'Content-Type: application/json' \
  -d '{
    "title": "FastAPI Book Launch"
  }'

```

Вот ответ:

```

{
  "detail": "Operation not allowed"
}

```

Наконец, давайте обновим маршрут DELETE:

```

@event_router.delete("/{id}")
async def delete_event(id: PydanticObjectId, user: str =
Depends(authenticate)):
    event = await event_database.get(id)
    if event.creator != user:
        raise HTTPException(

```

```
status_code=status.HTTP_404_NOT_FOUND,
detail="Event not found"
)
```

В предыдущем блоке кода мы указываем функции маршрута сначала проверить, является ли текущий пользователь создателем, в противном случае вызвать исключение. Давайте рассмотрим пример, когда другой пользователь пытается удалить событие другого пользователя:

```
$ curl -X 'DELETE' \
'http://0.0.0.0:8080/event/6265a83fc823a3c912830074' \
-H 'accept: application/json' \
-H 'Authorization: Bearer eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXV
Cj9.eyJ1c2VyIjoizmFzdGFwaUBwYWNRdC5jb20iLCJleHBpcnVzIjoxNjUwOD
MzOTc2LjI2NzgzMk0.MMRT6pwEDBVHTU5C1a6MV8j9wCfWhqbza9NBpZz08xE'
```

Ненайденное событие возвращается в качестве ответа:

```
{
  "detail": "Event not found"
}
```

Однако владелец может удалить событие:

```
$ curl -X 'DELETE' \
'http://0.0.0.0:8080/event/6265a83fc823a3c912830074' \
-H 'accept: application/json' \
-H 'Authorization: Bearer eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1c2VyIjoicmVhZGVyQHBhY2t0LmNvbSIsImV4cGlyZXMiOiJlE2NTA4MzQzOTUuMDkzMDI3fQ.IKYHWQ2YO3rQc-KR8kyfoY 54MsEVE75WbRqoVbdoW0'
```

Вот ответ:

```
{
  "message": "Event deleted successfully."
}
```

Мы успешно защитили наше приложение и его маршруты. Давайте завершим эту главу, настроив промежуточное ПО Cross-Origin Resource Sharing (CORS) в следующем разделе.

Настройка CORS

Совместное использование ресурсов между источниками (CORS) служит правилом, которое предотвращает доступ незарегистрированных клиентов к ресурсу.

Когда наш веб-API используется внешним приложением, браузер не разрешает HTTP-запросы из разных источников. Это означает, что доступ к ресурсам возможен только из точного источника, как API или источников, разрешенных API.

FastAPI предоставляет **CORS middleware**, `CORSMiddleware`, которое позволяет нам регистрировать домены, которые могут получить доступ к нашему API. Middleware принимает массив источников, которым будет разрешен доступ к ресурсам на сервере.

Что такое middleware?

Middleware — это функция, выступающая посредником между операциями. В веб-API middleware служит посредником в операции запрос-ответ.

Например, чтобы разрешить доступ к нашему API только Packt, мы определяем URL-адреса в исходном массиве:

```
origins = [
    "http://packtpub.com",
    "https://packtpub.com"
]
```

Чтобы разрешить запросы от любого клиента, массив `origins` будет содержать только одно значение — звездочку (*). Звездочка — это подстановочный знак, который указывает нашему API разрешать запросы из любого места.

В `main.py` настроим приложение так, чтобы оно принимало запросы отовсюду:

```
from fastapi.middleware.cors import CORSMiddleware

# register origins

origins = ["*"]

app.add_middleware(
```

```
CORSMiddleware,  
    allow_origins=origins,  
    allow_credentials=True,  
    allow_methods=["*"],  
    allow_headers=["*"],  
)
```

В приведенном выше блоке кода мы начали с импорта класса `CORSMiddleware` из `FastAPI`. Мы зарегистрировали массив `origins` и, наконец, зарегистрировали промежуточное ПО в приложении с помощью метода `add_middleware`.

Дополнительная информация

Документация FastAPI содержит более подробную информацию о CORS - <https://fastapi.tiangolo.com/tutorial/cors/>

Мы успешно настроили наше приложение, чтобы разрешить запросы из любого источника во всемирной паутине.

Резюме

В этой главе мы узнали, как защитить приложение FastAPI с помощью OAuth и JWT. Мы также узнали, что такое внедрение зависимостей, как оно используется в приложениях FastAPI и как защитить маршруты от неавторизованных пользователей. Мы также добавили промежуточное ПО CORS, чтобы разрешить доступ к нашему API из любого клиента. Мы использовали знания из предыдущих глав.

В следующей главе вы познакомитесь с тестированием вашего приложения FastAPI. Вы узнаете, что такое тестирование приложения, почему вы должны тестировать приложения и как тестировать приложение FastAPI.

Часть 3: Тестирование и развертывание приложений FastAPI

По завершении этой части вы сможете писать и выполнять тесты и развертывать приложения FastAPI, используя знания, полученные из включенных глав.

Эта часть состоит из следующих глав:

- Глава 8. Тестирование приложений FastAPI
- Глава 9. Развертывание приложений FastAPI

8

Тестирование приложений FastAPI

В последней главе мы узнали, как защитить приложение FastAPI с помощью OAuth и **JSON Web Token (JWT)**. Мы успешно внедрили систему аутентификации и узнали, что такое внедрение зависимостей. Мы также узнали, как внедрять зависимости в наши маршруты для ограничения несанкционированного доступа и операций. Мы успешно создали безопасный веб-API, который поддерживает базу данных и может легко выполнять CRUD операции. В этой главе мы узнаем, что такое тестирование и как писать тесты, чтобы гарантировать, что наше приложение ведет себя так, как ожидается.

Тестирование является неотъемлемой частью цикла разработки приложения. Тестирование приложений выполняется для обеспечения правильного функционирования приложения и легкого обнаружения аномалий в приложении перед развертыванием в рабочей среде. Хотя мы вручную тестировали конечную точку нашего приложения в последних нескольких главах, мы научимся автоматизировать эти тесты.

К концу этой главы вы сможете писать тесты для маршрутов вашего приложения FastAPI. В этой главе объясняется, что такое модульное тестирование и как выполнять модульное тестирование на маршрутах приложений. В этой главе вы затронете следующие темы:

- Модульное тестирование с помощью `pytest`
- Настройка нашей тестовой среды

- Написание тестов для конечных точек REST API
- Тестовое покрытие

Технические требования

Для этой главы вам понадобится работающий сервер MongoDB на вашем локальном компьютере. Шаги, описанные в Главе 6 «Подключение к базе данных», необходимо выполнить, чтобы настроить и запустить сервер базы данных.

Код, использованный в этой главе, можно найти по адресу <https://github.com/PacktPublishing/Building-Python-Web-APIs-with-FastAPI/tree/main/ch08/planner>.

Модульное тестирование с помощью pytest

Модульное тестирование — это процедура тестирования, при которой тестируются отдельные компоненты приложения. Эта форма тестирования позволяет нам проверить работоспособность отдельных компонентов. Например, модульные тесты используются для тестирования отдельных маршрутов в приложении, чтобы убедиться, что возвращаются правильные ответы.

В этой главе мы будем использовать `pytest`, библиотеку тестирования Python, для проведения наших операций модульного тестирования. Хотя Python поставляется со встроенной библиотекой модульного тестирования под названием `unittest` библиотека, `pytest` имеет более короткий синтаксис и более предпочтительна для тестирования приложений. Давайте установим `pytest` и напишем наш первый образец теста.

Давайте установим библиотеку `pytest`:

```
(venv)$ pip install pytest
```

Затем создайте папку с именем `tests`, в которой будут храниться тестовые файлы для нашего приложения:

```
(venv)$ mkdir tests && cd
(venv)$ touch __init__.py
```

Имена отдельных тестовых модулей во время создания будут иметь префикс `test_`. Это позволит библиотеке `pytest` распознать и запустить тестовый модуль. Создадим в только что созданном пакете `tests` тестовый модуль, проверяющий правильность выполнения арифметических операций сложения, вычитания, умножения и деления:

```
(venv)$ touch test_arithmetic_operations.py
```


Сначала определим функцию, которая выполняет арифметические операции. В модуле тестов добавьте следующее:

```
def add(a: int , b: int) -> int:
    return a + b

def subtract(a: int, b: int) -> int:
    return b - a

def multiply(a: int, b: int) -> int:
    return a * b

def divide(a: int, b: int) -> int:
    return b // a
```

Теперь, когда мы определили операции для тестирования, мы создадим функции, которые будут обрабатывать эти тесты. В тестовых функциях определяется операция, которая должна быть выполнена. Ключевое слово `assert` используется для проверки того, что вывод в левой части соответствует результат операции в правой части. В нашем случае мы будем проверять, равны ли арифметические операции их соответствующим результатам.

Добавьте следующее в модуль `tests`:

```
def test_add() -> None:
    assert add(1, 1) == 2

def test_subtract() -> None:
    assert subtract(2, 5) == 3

def test_multiply() -> None:
    assert multiply(10, 10) == 100

def test_divide() -> None:
    assert divide(25, 100) == 4
```

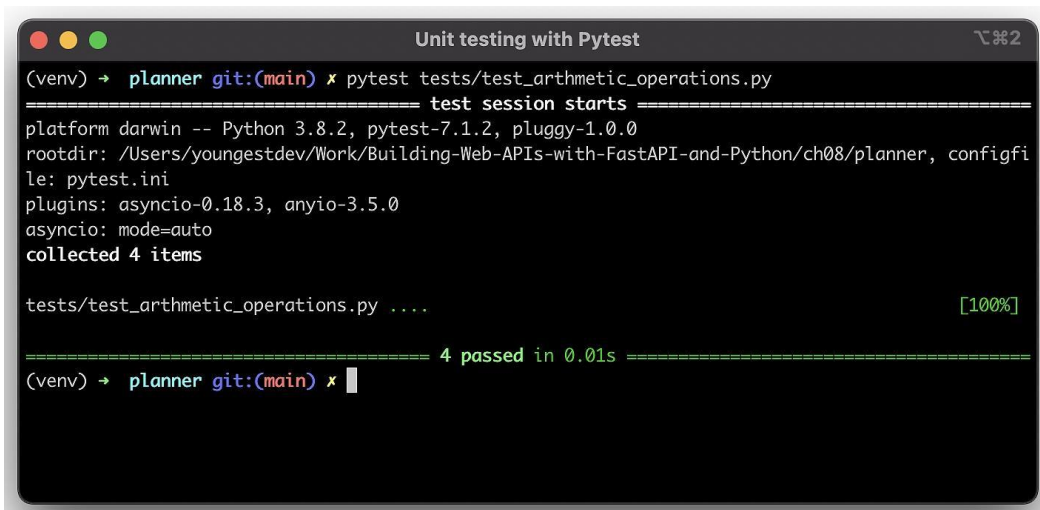
Совет

Стандартной практикой является определение функций, которые будут тестироваться во внешнем расположении (в нашем случае `add()`, `subtract()` и т. д.). Затем этот модуль импортируется, и тестируемые функции вызываются в тестовых функциях.

Имея тестовые функции, мы готовы запустить тестовый модуль. Тесты можно выполнить, выполнив команду `pytest`. Однако эта команда запускает все тестовые модули, содержащиеся в пакете. Для выполнения одного теста в качестве аргумента передается имя тестового модуля. Запустим тестовый модуль:

```
(venv)$ pytest test_arithmetic_operations.py
```

Тесты определили все пройдено. Об этом свидетельствует ответ, выделенный зеленым цветом:



```

Unit testing with Pytest
(venv) -> planner git:(main) x pytest tests/test_arithmetic_operations.py
===== test session starts =====
platform darwin -- Python 3.8.2, pytest-7.1.2, pluggy-1.0.0
rootdir: /Users/youngestdev/Work/Building-Web-APIs-with-FastAPI-and-Python/ch08/planner, configfile: pytest.ini
plugins: asyncio-0.18.3, anyio-3.5.0
asyncio: mode=auto
collected 4 items

tests/test_arithmetic_operations.py .... [100%]

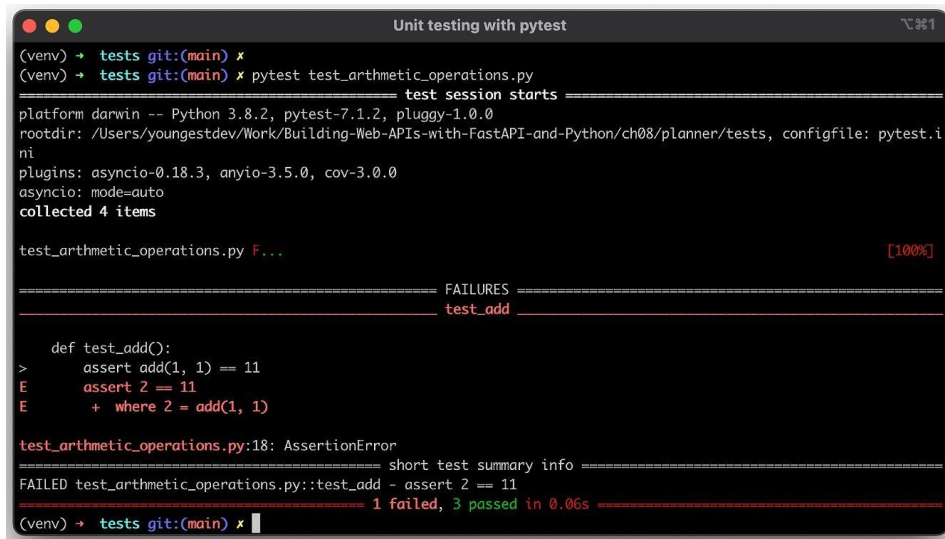
===== 4 passed in 0.01s =====
(venv) -> planner git:(main) x

```

Рисунок 8.1 – Результат модульного тестирования, выполненного на арифметических операциях. Провалившиеся тесты, а также точка, сбоя выделены красным цветом. Например, скажем, мы модифицируем функцию `test_add()` как таковую:

```
def test_add() -> None:
    assert add(1, 1) == 11
```

На следующем рисунке неудачный тест, а также точка отказа выделены красным цветом.



```
(venv) → tests git:(main) x
(venv) → tests git:(main) x pytest test_arithmetic_operations.py

===== test session starts =====
platform darwin -- Python 3.8.2, pytest-7.1.2, pluggy-1.0.0
rootdir: /Users/youngestdev/Work/Building-Web-APIs-with-FastAPI-and-Python/ch08/planner/tests, configfile: pytest.ini
plugins: asyncio-0.18.3, anyio-3.5.0, cov-3.0.0
asyncio: mode=auto
collected 4 items

test_arithmetic_operations.py F... [100%]

===== FAILURES =====
test_add

def test_add():
> assert add(1, 1) == 11
E       assert 2 == 11
E       + where 2 = add(1, 1)

test_arithmetic_operations.py:18: AssertionError

===== short test summary info =====
FAILED test_arithmetic_operations.py::test_add - assert 2 == 11
===== 1 failed, 3 passed in 0.06s =====
(venv) → tests git:(main) x
```

Рисунок 8.2 – Неудачный тест

Тест не пройден в операторе `assert`, где отображается правильный результат `2`.

Сбой резюмируется как `AssertionError`, что говорит нам о том, что тест не пройден из-за неверного утверждения (`2 == 1`).

Теперь, когда у нас есть представление о том, как работает `pytest`, давайте взглянем на фикстуры в `pytest`.

Устранение повторений с помощью фикстур `pytest`

Фикстуры — это повторно используемые функции, определенные для возврата данных, необходимых в тестовых функциях. Фикстуры оборачиваются декоратором `pytest.fixture`. Пример использования фикстуры — возврат экземпляра приложения для выполнения тестов для конечных точек API. Фикстура может использоваться для определения клиента приложения, который возвращается и используется в тестовых функциях, что устраняет необходимость переопределять экземпляр приложения в каждом тесте. Мы увидим, как это используется в разделе «Написание тестов для конечных точек REST API».

Давайте посмотрим на пример:

```
import pytest

from models.events import EventUpdate
```

```
# Fixture is defined.
@pytest.fixture
def event() -> EventUpdate:
    return EventUpdate(
        title="FastAPI Book Launch",
        image="https://packt.com/fastapi.png",
        description="We will be discussing the contents of
        the FastAPI book in this event.Ensure to come with
        your own copy to win gifts!",
        tags=["python", "fastapi", "book", "launch"],
        location="Google Meet"
    )
def test_event_name(event: EventUpdate) -> None:
    assert event.title == "FastAPI Book Launch"
```

В предыдущем блоке кода мы определили фикстуру, которая возвращает экземпляр `pydantic` модели `EventUpdate`. Это приспособление передается в качестве аргумента в функцию `test_event_name`, что позволяет сделать свойства доступными.

Декоратор фикстуры может дополнительно принимать аргументы. Одним из этих аргументов является область действия — область действия фикстуры сообщает `pytest` какова продолжительность функции фикстуры.

В этой главе мы будем использовать две области видимости:

- `session`: Эта область сообщает `pytest`, что нужно создать экземпляр функции один раз для всего сеанса тестирования.
- `module`: Эта область инструктирует `pytest` выполнять добавленную функцию только после выполнения тестового модуля.

Теперь, когда мы знаем, что такое фикстура, давайте настроим нашу тестовую среду в следующем разделе.

Настройка тестовой среды

В предыдущем разделе мы узнали об основах тестирования, а также о том, что такое фикстуры. Теперь мы проверим конечные точки для CRUD операций, а также аутентификацию пользователей. Чтобы протестировать наши асинхронные API, мы будем использовать `httpx` и установим библиотеку `pytest-asyncio`, чтобы мы могли протестировать наш асинхронный API.

Установите дополнительные библиотеки:

```
(venv)$ pip install httpx pytest-asyncio
```

Далее мы создадим файл конфигурации с именем `pytest.ini`. Добавьте в него следующий код:

```
[pytest]
asyncio_mode = True
```

Файл конфигурации читается при запуске `pytest`. Это автоматически заставляет `pytest` запускать все тесты в асинхронном режиме.

Имея файл конфигурации, давайте создадим тестовый модуль `conftest.py`, который будет отвечать за создание экземпляра нашего приложения, необходимого для тестовых файлов. В папке с тестами создайте модуль `conftest`:

```
(venv)$ touch tests/conftest.py
```

Мы начнем с импорта необходимых зависимостей в `conftest.py`:

```
import asyncio
import httpx
import pytest

from main import app
from database.connection import Settings
from models.events import Event
from models.users import User
```

В предыдущем блоке кода мы импортировали модули `asyncio`, `httpx` и `pytest`. Модуль `asyncio` будет использоваться для создания активного сеанса цикла, чтобы тесты выполнялись в одном потоке, чтобы избежать конфликтов. Тест `httpx` будет действовать как асинхронный клиент для выполнения CRUD операций HTTP. Библиотека `pytest` необходима для определения фикстур.

Мы также импортировали приложение-экземпляр нашего приложения, а также модели и класс `Settings`. Давайте определим фикстуру сеанса цикла:

```
@pytest.fixture(scope="session")
def event_loop():
    loop = asyncio.get_event_loop()
    yield loop
    loop.close()
```

Имея это в виду, давайте создадим новый экземпляр базы данных из класса `Settings`:

```
async def init_db():
    test_settings = Settings()
    test_settings.DATABASE_URL =
        "mongodb://localhost:27017/testdb"

    await test_settings.initialize_database()
```

В предыдущем блоке кода мы определили новый `DATABASE_URL`, а также вызвали функцию инициализации, определенную в главе 6 «Подключение к базе данных». Сейчас мы используем новую базу данных `testdb`.

Наконец, давайте определим клиентскую фикстуру по умолчанию, которая возвращает экземпляр нашего приложения, работающего асинхронно через `httpx`:

```
@pytest.fixture(scope="session")
async def default_client():
    await init_db()
    async with httpx.AsyncClient(app=app,
        base_url="http://app") as client:
        yield client
    # Clean up resources
    await Event.find_all().delete()
    await User.find_all().delete()
```

В предыдущем блоке кода сначала инициализируется база данных, а приложение запускается как `AsyncClient`, который остается активным до конца тестового сеанса. В конце сеанса тестирования коллекция событий и пользователей стирается, чтобы убедиться, что база данных пуста перед каждым запуском теста.

В этом разделе вы познакомились с шагами, связанными с настройкой вашей тестовой среды. В следующем разделе вы познакомитесь с процессом написания тестов для каждой конечной точки, созданной в приложении.

Написание тестов для конечных точек REST API

Когда все готово, давайте создадим модуль `test_login.py`, в котором мы будем тестировать маршруты аутентификации:

```
(venv)$ touch tests/test_login.py
```

В тестовом модуле мы начнем с импорта зависимостей:

```
import httpx
import pytest
```

Тестирование маршрута регистрации

Первая конечная точка, которую мы будем тестировать, — это конечная точка регистрации. Мы добавим декоратор `pytest.mark.asyncio`, который сообщает `pytest` что нужно рассматривать это как асинхронный тест. Давайте определим функцию и полезную нагрузку запроса:

```
@pytest.mark.asyncio
async def test_sign_new_user(default_client: httpx.AsyncClient)
-> None:
    payload = {
        "email": "testuser@packt.com",
        "password": "testpassword",
    }
```

Определим заголовок запроса и ожидаемый ответ:

```
headers = {
    "accept": "application/json",
    "Content-Type": "application/json"
}

test_response = {
    "message": "User created successfully"
}
```

Теперь, когда мы определили ожидаемый ответ на этот запрос, давайте иницилируем запрос:

```
response = await default_client.post("/user/signup",
    json=payload, headers=headers)
```

Далее мы проверим, был ли запрос успешным, сравнив ответы:

```
assert response.status_code == 200
assert response.json() == test_response
```

Перед запуском этого теста давайте кратко прокомментируем строку, которая стирает пользовательские данные в `conf/test.py`, поскольку это приведет к сбою аутентифицированных тестов:

```
# await User.find_all().delete()
```

Со своего терминала запустите сервер MongoDB и запустите тест:

```
(venv)$ pytest tests/test_login.py
```

Маршрут регистрации успешно протестирован:

```

pytest
(venv) → planner git:(main) x pytest tests/test_login.py
test session starts
platform darwin -- Python 3.8.2, pytest-7.1.2, pluggy-1.0.0
rootdir: /Users/youngestdev/Work/Building-Web-APIs-with-FastAPI-and-Python/ch08/planner, configfile: pytest.ini
plugins: asyncio-0.18.3, anyio-3.5.0
asyncio: mode=auto
collected 1 item

tests/test_login.py . [100%]

===== 1 passed in 0.28s =====
(venv) → planner git:(main) x

```

Рисунок 8.3 – Успешный тестовый прогон на маршруте регистрации

Приступим к написанию теста для маршрута входа. Тем временем вы можете быстро настроить ответ теста, чтобы увидеть, не прошел ли ваш тест или нет!

Тестирование маршрута входа

Ниже теста для маршрута регистрации давайте определим тест для маршрута входа. Мы начнем с определения полезной нагрузки запроса и заголовков, прежде чем инициировать запрос, как в первом тесте:

```

@pytest.mark.asyncio
async def test_sign_user_in(default_client: httpx.AsyncClient)
-> None:
    payload = {
        "username": "testuser@packt.com",
        "password": "testpassword"

```



```

    }

    headers = {
        "accept": "application/json",
        "Content-Type": "application/x-www-form-urlencoded"
    }

```

Далее мы иницилируем запрос и тестируем ответы:

```

response = await default_client.post("/user/signin",
data=payload, headers=headers)

assert response.status_code == 200
assert response.json()["token_type"] == "Bearer"

```

Повторим тест:

```
(venv)$ pytest tests/test_login.py
```

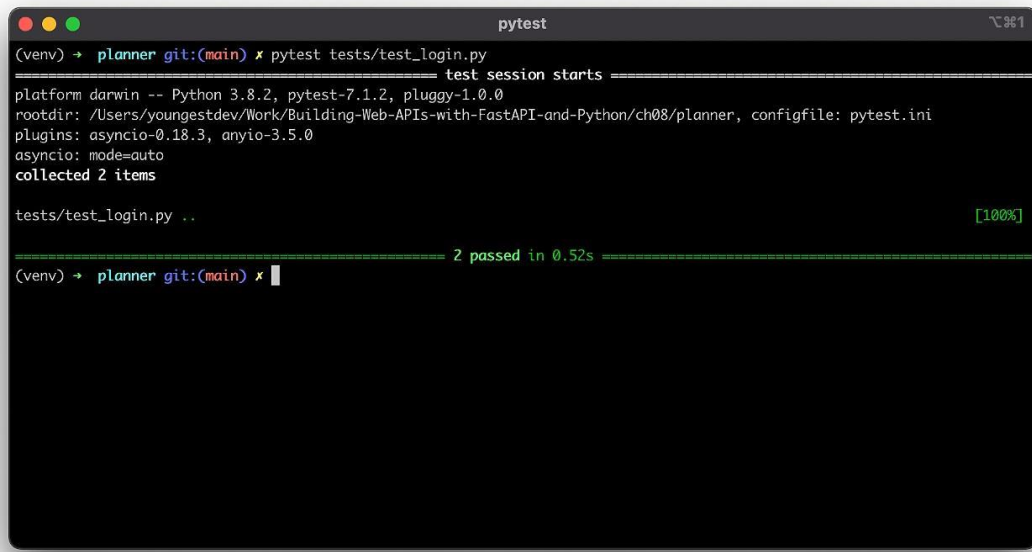
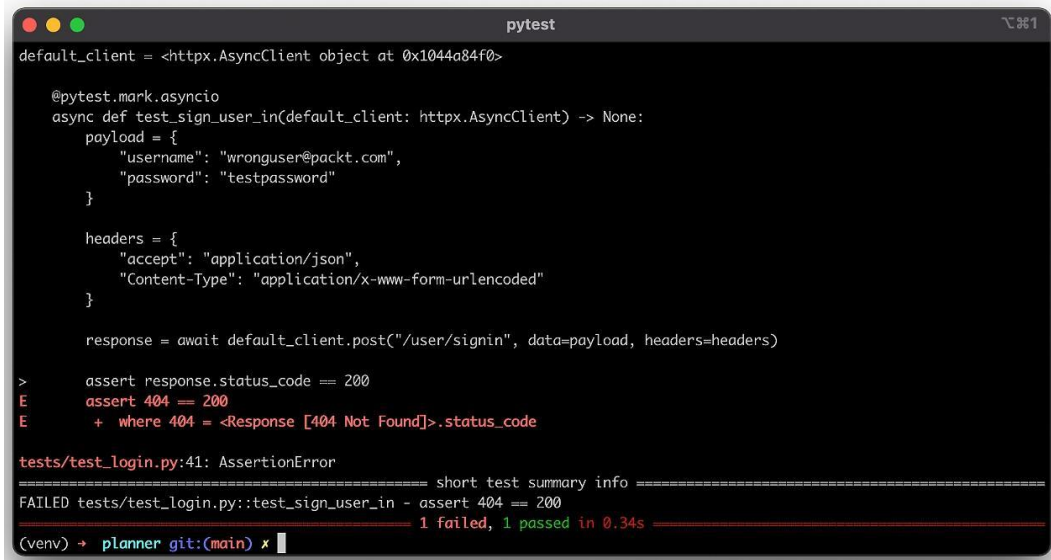


Рисунок 8.4 – Успешный тестовый прогон для обоих маршрутов

Давайте изменим имя пользователя для входа на неправильное, чтобы подтвердить, что тест не пройден:

```
payload = {
    "username": "wronguser@packt.com",
    "password": "testpassword"
}
```



```
pytest
default_client = <httpx.AsyncClient object at 0x1044a84f0>

@pytest.mark.asyncio
async def test_sign_user_in(default_client: httpx.AsyncClient) -> None:
    payload = {
        "username": "wronguser@packt.com",
        "password": "testpassword"
    }

    headers = {
        "accept": "application/json",
        "Content-Type": "application/x-www-form-urlencoded"
    }

    response = await default_client.post("/user/signin", data=payload, headers=headers)

> assert response.status_code == 200
E   assert 404 == 200
E   + where 404 = <Response [404 Not Found]>.status_code

tests/test_login.py:41: AssertionError
===== short test summary info =====
FAILED tests/test_login.py::test_sign_user_in - assert 404 == 200
===== 1 failed, 1 passed in 0.34s =====
(venv) + planner git:(main) x
```

Рисунок 8.5 – Неудачный тест из-за неправильной полезной нагрузки запроса

Мы успешно написали тесты для маршрутов регистрации и входа. Перейдем к тестированию CRUD-маршрутов для API планировщика событий.

Тестирование конечных точек CRUD

Мы начнем с создания нового модуля с именем `test_routes.py`:

```
(venv) $ touch test_routes.py
```

Во вновь созданный модуль добавьте следующий код:

```
import httpx
import pytest

from auth.jwt_handler import create_access_token
from models.events import Event
```

В предыдущем блоке кода мы импортировали обычные зависимости. Мы также импортировали функцию `create_access_token (user)` и модель `Event`. Поскольку некоторые из маршрутов защищены, мы будем генерировать токен доступа самостоятельно. Давайте создадим новую фикстуру, которая при вызове возвращает токен доступа. Приспособление имеет область модуля, что означает, что оно запускается только один раз — при выполнении тестового модуля — и не вызывается при каждом вызове функции. Добавьте следующий код:

```
@pytest.fixture(scope="module")
async def access_token() -> str:
    return create_access_token("testuser@packt.com")
```

Давайте создадим новый прибор, который добавляет событие в базу данных. Это действие выполняется для запуска предварительных тестов перед тестированием конечных точек CRUD. Добавьте следующий код:

```
@pytest.fixture(scope="module")
async def mock_event() -> Event:
    new_event = Event(
        creator="testuser@packt.com",
        title="FastAPI Book Launch",
        image="https://linktomyimage.com/image.png",
        description="We will be discussing the contents of
        the FastAPI book in this event.Ensure to come with
        your own copy to win gifts!",
        tags=["python", "fastapi", "book", "launch"],
        location="Google Meet"
    )

    await Event.insert_one(new_event)

    yield new_event
```

Тестирование конечных точек READ

Далее давайте напишем тестовую функцию, которая проверяет **GET-метод HTTP** на маршруте `/event`:

```
@pytest.mark.asyncio
async def test_get_events(default_client: httpx.AsyncClient,
    mock_event: Event) -> None:
    response = await default_client.get("/event/")
```

```
assert response.status_code == 200
assert response.json()[0]["_id"] == str(mock_event.id)
```

В предыдущем блоке кода мы тестируем путь маршрута события, чтобы проверить, присутствует ли событие, добавленное в базу данных в фикстуре `mock_event`. Давайте запустим тест:

```
(venv)$ pytest tests/test_routes.py
```

Вот результат:



Рисунок 8.6 – Успешный тестовый запуск

Далее давайте напишем тестовую функцию для конечной точки `/event/{id}`:

```
@pytest.mark.asyncio
async def test_get_event(default_client: httpx.AsyncClient,
mock_event: Event) -> None:
    url = f"/event/{str(mock_event.id)}"
    response = await default_client.get(url)

    assert response.status_code == 200
    assert response.json()["creator"] == mock_event.creator
    assert response.json()["_id"] == str(mock_event.id)
```

В предыдущем блоке кода мы тестируем конечную точку, которая извлекает одно событие. Переданный идентификатор события извлекается из фикстуры `mock_event`, а результат запроса сравнивается с данными, хранящимися в фикстуре `mock_event`. Давайте запустим тест:

```
(venv)$ pytest tests/test_routes.py
```

Вот результат:

```

(pyenv) → planner git:(main) x pytest tests/test_routes.py
===== test session starts =====
platform darwin -- Python 3.8.2, pytest-7.1.2, pluggy-1.0.0
rootdir: /Users/youngestdev/Work/Building-Web-APIs-with-FastAPI-and-Python/ch08/planner, configfile: pytest.ini
plugins: asyncio-0.18.3, anyio-3.5.0
asyncio: mode=auto
collected 2 items

tests/test_routes.py .. [100%]

===== 2 passed in 0.03s =====
(pyenv) → planner git:(main) x

```

Рисунок 8.7 – Успешный тестовый прогон для извлечения одного события

Далее напомним тестовую функцию для создания нового события.

Тестирование конечной точки CREATE

Мы начнем с определения функции и получения токена доступа из ранее созданного прибора. Мы создадим полезную нагрузку запроса, которая будет отправлена на сервер, заголовки запроса, которые будут содержать тип контента, а также значение заголовка авторизации. Также будет определен тестовый ответ, после чего инициируется запрос и сравниваются ответы. Добавьте следующий код:

```

@pytest.mark.asyncio
async def test_post_event(default_client: httpx.AsyncClient,
                          access_token: str) -> None:
    payload = {
        "title": "FastAPI Book Launch",
        "image": "https://linktomyimage.com/image.png",
        "description": "We will be discussing the contents
of the FastAPI book in this event.Ensure to come
with your own copy to win gifts!",
        "tags": [

```

```
        "python",
        "fastapi",
        "book",
        "launch"
    ],
    "location": "Google Meet",
}

headers = {
    "Content-Type": "application/json",
    "Authorization": f"Bearer {access_token}"
}

test_response = {
    "message": "Event created successfully"
}

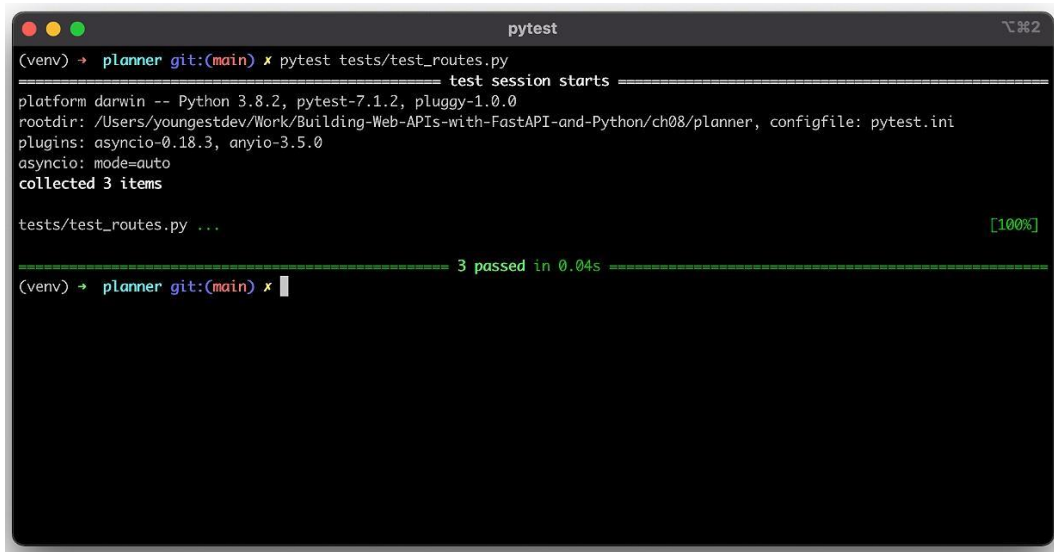
response = await default_client.post("/event/new",
    json=payload, headers=headers)

assert response.status_code == 200
assert response.json() == test_response
```

Давайте перезапустим тестовый модуль:

```
(venv)$ pytest tests/test_routes.py
```

Результат выглядит так:



```

(pyenv) → planner git:(main) ✗ pytest tests/test_routes.py
===== test session starts =====
platform darwin -- Python 3.8.2, pytest-7.1.2, pluggy-1.0.0
rootdir: /Users/youngestdev/Work/Building-Web-APIs-with-FastAPI-and-Python/ch08/planner, configfile: pytest.ini
plugins: asyncio-0.18.3, anyio-3.5.0
asyncio: mode=auto
collected 3 items

tests/test_routes.py ... [100%]

===== 3 passed in 0.04s =====
(pyenv) → planner git:(main) ✗

```

Рисунок 8.8 – Успешный тестовый запуск запроса POST

Давайте напишем тест для проверки количества событий, хранящихся в базе данных (в нашем случае 2). Добавьте следующее:

```

@pytest.mark.asyncio
async def test_get_events_count(default_client: httpx.AsyncClient) -> None:
    response = await default_client.get("/event/")

    events = response.json()

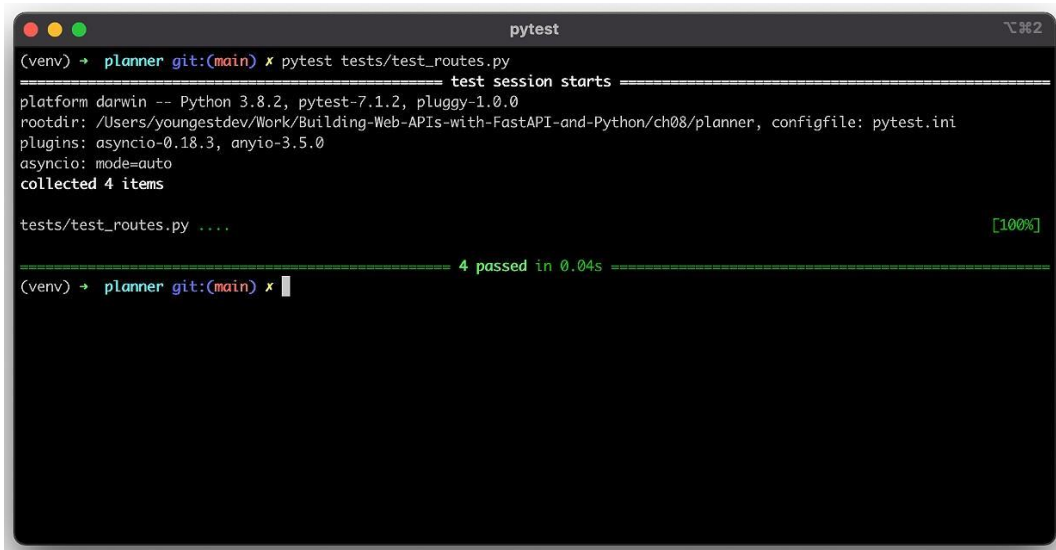
    assert response.status_code == 200
    assert len(events) == 2

```

В предыдущем блоке кода мы сохранили ответ JSON в переменной `events`, длина которой используется для нашего тестового сравнения. Давайте перезапустим тестовый модуль:

```
(venv)$ pytest tests/test_routes.py
```

Вот результат:



```

(pyenv) → planner git:(main) x pytest tests/test_routes.py
===== test session starts =====
platform darwin -- Python 3.8.2, pytest-7.1.2, pluggy-1.0.0
rootdir: /Users/youngestdev/Work/Building-Web-APIs-with-FastAPI-and-Python/ch08/planner, configfile: pytest.ini
plugins: asyncio-0.18.3, anyio-3.5.0
asyncio: mode=auto
collected 4 items

tests/test_routes.py .... [100%]

===== 4 passed in 0.04s =====
(pyenv) → planner git:(main) x

```

Рисунок 8.9 – Успешный тестовый прогон для подтверждения количества событий

Мы успешно протестировали конечные точки GET /event и /event/{id} и конечную точку POST /event/new, соответственно. Давайте проверим конечные точки UPDATE и DELETE для /event/new дальше.

Тестирование конечной точки UPDATE

Начнем с конечной точки UPDATE:

```

@pytest.mark.asyncio
async def test_update_event(default_client: httpx.AsyncClient,
                             mock_event: Event, access_token: str) -> None:
    test_payload = {
        "title": "Updated FastAPI event"
    }

    headers = {
        "Content-Type": "application/json",
        "Authorization": f"Bearer {access_token}"
    }

```



```
url = f"/event/{str(mock_event.id)}"

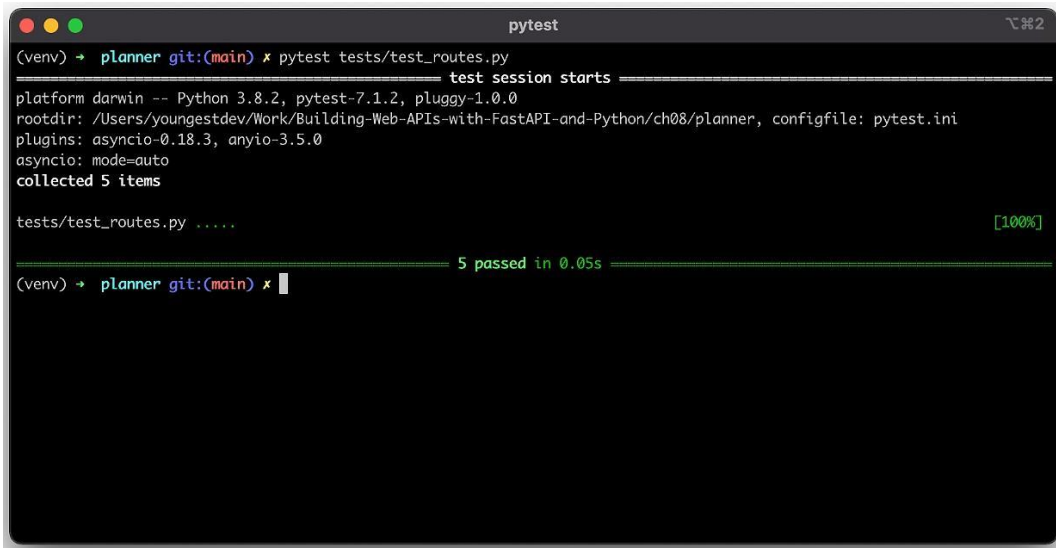
response = await default_client.put(url,
    json=test_payload, headers=headers)

assert response.status_code == 200
assert response.json()["title"] ==
    test_payload["title"]
```

В предыдущем блоке кода мы изменяем событие, хранящееся в базе данных, извлекая ID из фикстуры `mock_event`. Затем мы определяем полезную нагрузку запроса и заголовки. В переменной `response` инициируется запрос и сравнивается полученный ответ. Давайте подтвердим, что тест работает правильно:

```
(venv)$ pytest tests/test_routes.py
```

Вот результат:



```
pytest
(venv) → planner git:(main) x pytest tests/test_routes.py
test session starts
platform darwin -- Python 3.8.2, pytest-7.1.2, pluggy-1.0.0
rootdir: /Users/youngestdev/Work/Building-Web-APIs-with-FastAPI-and-Python/ch08/planner, configfile: pytest.ini
plugins: asyncio-0.18.3, anyio-3.5.0
asyncio: mode=auto
collected 5 items

tests/test_routes.py ..... [100%]

===== 5 passed in 0.05s =====
(venv) → planner git:(main) x
```

Рисунок 8.10 – Успешный запуск запроса UPDATE

Совет

Приспособление `mock_event` пригодится, поскольку ID документов MongoDB уникально генерируется каждый раз, когда документ добавляется в базу данных.

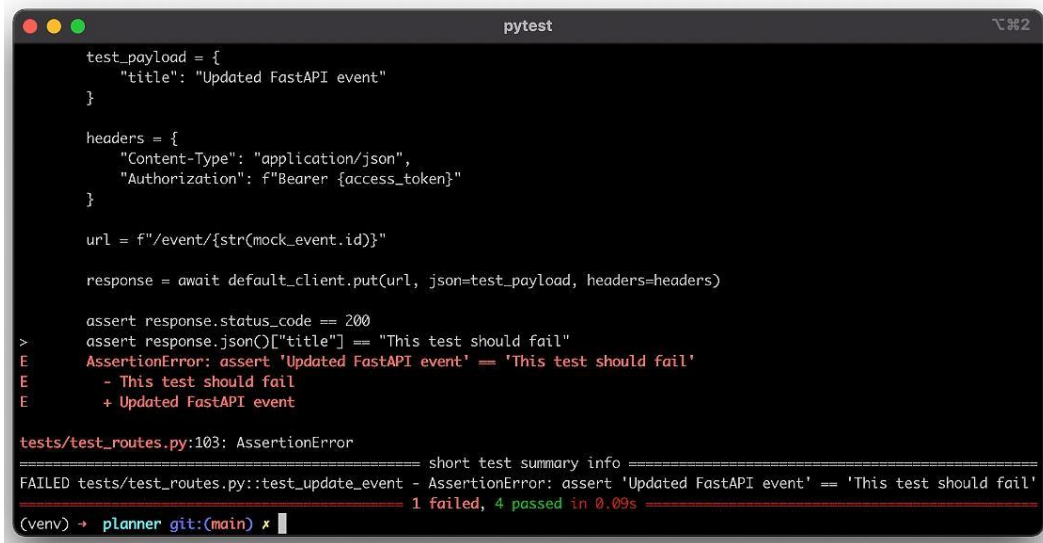
Давайте изменим ожидаемый ответ, чтобы подтвердить достоверность нашего теста:

```
assert response.json()["title"] == "This test should fail"
```

Повторите тест:

```
(venv) $ pytest tests/test_routes.py
```

Вот результат:



```

test_payload = {
    "title": "Updated FastAPI event"
}

headers = {
    "Content-Type": "application/json",
    "Authorization": f"Bearer {access_token}"
}

url = f"/event/{str(mock_event.id)}"

response = await default_client.put(url, json=test_payload, headers=headers)

assert response.status_code == 200
> assert response.json()["title"] == "This test should fail"
E   AssertionError: assert 'Updated FastAPI event' == 'This test should fail'
E   - This test should fail
E   + Updated FastAPI event

tests/test_routes.py:103: AssertionError
===== short test summary info =====
FAILED tests/test_routes.py::test_update_event - AssertionError: assert 'Updated FastAPI event' == 'This test should fail'
===== 1 failed, 4 passed in 0.09s =====
(venv) → planner git:(main) x

```

Рисунок 8.11 – Неудачный тест из-за разницы в объектах ответа

Тестирование конечной точки DELETE

Наконец, давайте напишем тестовую функцию для конечной точки DELETE:

```

@pytest.mark.asyncio
async def test_delete_event(default_client: httpx.AsyncClient,
                           mock_event: Event, access_token: str) -> None:
    test_response = {
        "message": "Event deleted successfully."
    }

    headers = {
        "Content-Type": "application/json",

```

```

    "Authorization": f"Bearer {access_token}"
}

url = f"/event/{mock_event.id}"

response = await default_client.delete(url,
headers=headers)

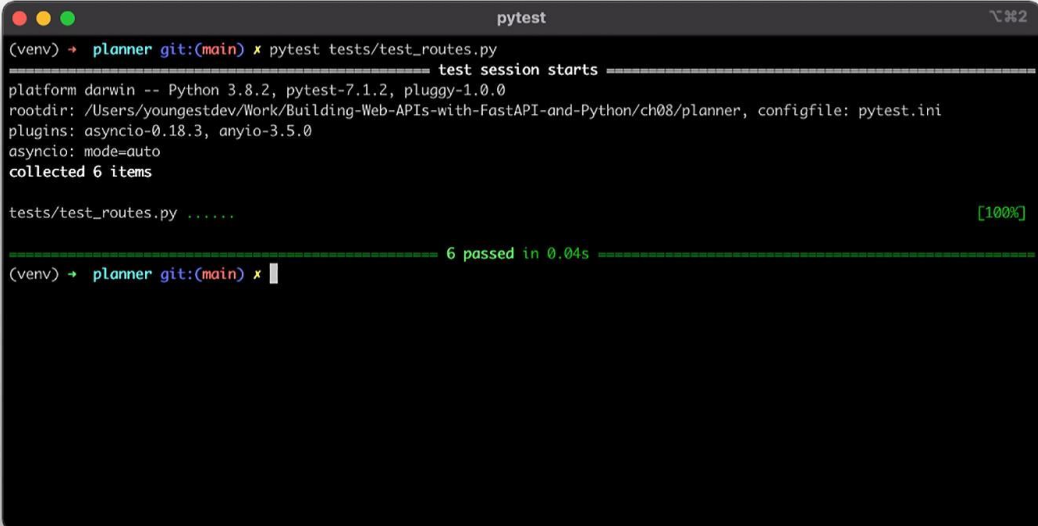
assert response.status_code == 200
assert response.json() == test_response

```

Как и в предыдущих тестах, определяется ожидаемый ответ теста, а также заголовки. Маршрут DELETE задействован, и ответ сравнивается. Давайте запустим тест:

```
(venv)$ pytest tests/test_routes.py
```

Вот результат:



```

pytest
(venv) + planner git:(main) x pytest tests/test_routes.py
===== test session starts =====
platform darwin -- Python 3.8.2, pytest-7.1.2, pluggy-1.0.0
rootdir: /Users/youngestdev/Work/Building-Web-APIs-with-FastAPI-and-Python/ch08/planner, configfile: pytest.ini
plugins: asyncio-0.18.3, anyio-3.5.0
asyncio: mode=auto
collected 6 items

tests/test_routes.py ..... [100%]

===== 6 passed in 0.04s =====
(venv) + planner git:(main) x

```

Рисунок 8.12 – Успешный тест УДАЛИТЬ

Чтобы убедиться, что документ действительно был удален, добавим финальную проверку:

```

@pytest.mark.asyncio
async def test_get_event_again(default_client: httpx.
AsyncClient, mock_event: Event) -> None:

```

```
url = f"/event/{str(mock_event.id)}"
response = await default_client.get(url)

assert response.status_code == 200
assert response.json()["creator"] == mock_event.creator
assert response.json()["_id"] == str(mock_event.id)
```

Ожидаемый ответ — отказ. Давайте попробуем:

```
(venv) $ pytest tests/test_routes.py
```

Вот результат:

```
pytest
tests/test_routes.py .....F [100%]

===== FAILURES =====
test_get_event_again

default_client = <httpx.AsyncClient object at 0x107fb4310>
mock_event = Event(id=ObjectId('627efe676d3e951f9d01578e'), revision_id=None, creator='testuser@packt.com', title='FastAPI
Book Lau...Ensure to come with your own copy to win gifts!', tags=['python', 'fastapi', 'book', 'launch'], location='Goog
le Meet')

@pytest.mark.asyncio
async def test_get_event_again(default_client: httpx.AsyncClient, mock_event: Event) -> None:
    url = f"/event/{str(mock_event.id)}"
    response = await default_client.get(url)

>     assert response.status_code == 200
E     assert 404 == 200
E     + where 404 = <Response [404 Not Found]>.status_code

tests/test_routes.py:130: AssertionError
===== short test summary info =====
FAILED tests/test_routes.py::test_get_event_again - assert 404 == 200
===== 1 failed, 6 passed in 0.09s =====
(venv) + planner git:(main) x
```

Рисунок 8.13 – Неудачный тестовый ответ

Как видно из предыдущего снимка экрана, элемент больше не может быть найден в базе данных. Теперь, когда вы успешно реализовали тесты для аутентификации и маршрутов событий, раскомментируйте код, отвечающий за очистку пользовательских данных из базы данных.:

```
await User.find_all().delete()
```

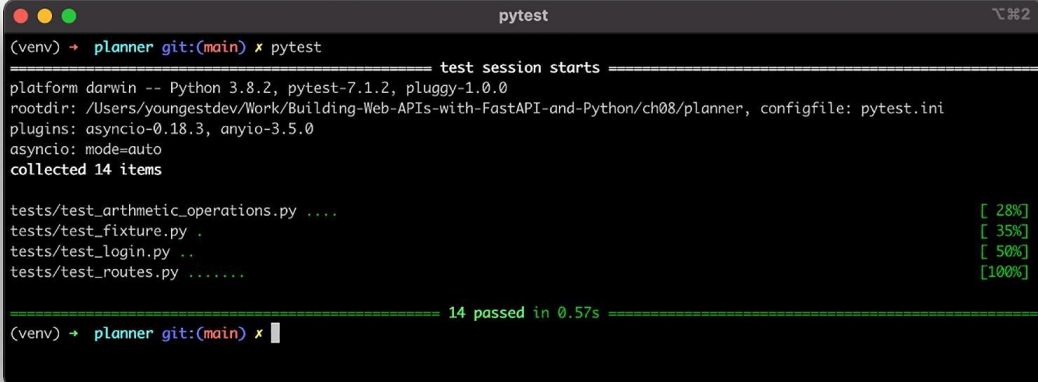
Обновить последний тест:

```
assert response.status_code == 404
```

Наконец, давайте запустим все тесты, присутствующие в нашем приложении.:

```
(venv) $ pytest
```

Вот результат:



```
pytest
(venv) → planner git:(main) ✖ pytest
===== test session starts =====
platform darwin -- Python 3.8.2, pytest-7.1.2, pluggy-1.0.0
rootdir: /Users/youngestdev/Work/Building-Web-APIs-with-FastAPI-and-Python/ch08/planner, configfile: pytest.ini
plugins: asyncio-0.18.3, anyio-3.5.0
asyncio: mode=auto
collected 14 items

tests/test_arithmetic_operations.py .... [ 28%]
tests/test_fixture.py . [ 35%]
tests/test_login.py .. [ 50%]
tests/test_routes.py ..... [100%]

===== 14 passed in 0.57s =====
(venv) → planner git:(main) ✖
```

Рисунок 8.14 – Полные тесты выполнялись за 0,57 секунды

Теперь, когда мы успешно протестировали конечные точки, содержащиеся в API планировщика событий, давайте запустим тест покрытия, чтобы определить процент нашего кода, задействованного в тестовой операции.

Покрытие тестами

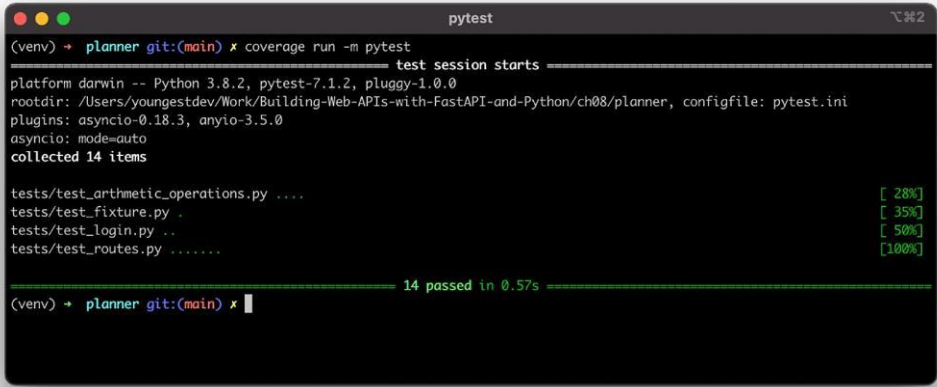
Отчет о покрытии тестами полезен для определения процента нашего кода, который был выполнен в ходе тестирования. Давайте установим модуль `coverage`, чтобы мы могли измерить, был ли наш API адекватно протестирован:

```
(venv) $ pip install coverage
```

Далее давайте создадим отчет о покрытии, выполнив эту команду:

```
(venv) $ coverage run -m pytest
```

Вот результат:



```

(pyenv) + planner git:(main) x coverage run -m pytest
test session starts
platform darwin -- Python 3.8.2, pytest-7.1.2, pluggy-1.0.0
rootdir: /Users/youngestdev/Work/Building-Web-APIs-with-FastAPI-and-Python/ch08/planner, configfile: pytest.ini
plugins: asyncio-0.18.3, anyio-3.5.0
asyncio: mode=auto
collected 14 items

tests/test_arithmetic_operations.py .... [ 28%]
tests/test_fixture.py . [ 35%]
tests/test_login.py .. [ 50%]
tests/test_routes.py ..... [100%]

===== 14 passed in 0.57s =====
(pyenv) + planner git:(main) x

```

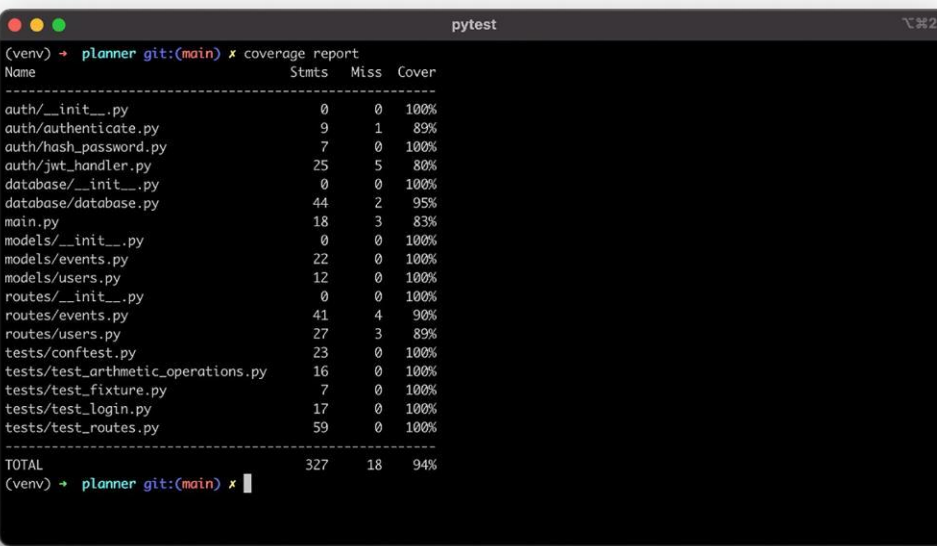
Рисунок 8.15 – Создан отчет о покрытии

Далее давайте посмотрим отчет, сгенерированный командой `coverage run -m pytest`. Мы можем выбрать просмотр отчета на терминале или на веб-странице, создав отчет в формате HTML. Мы сделаем оба.

Давайте рассмотрим отчет с терминала:

```
(venv) $ coverage report
```

Вот результат:



```

(pyenv) + planner git:(main) x coverage report
Name                               Stmts  Miss  Cover
-----
auth/__init__.py                    0      0  100%
auth/authenticate.py                 9      1   89%
auth/hash_password.py                 7      0  100%
auth/jwt_handler.py                 25      5   80%
database/__init__.py                 0      0  100%
database/database.py                 44      2   95%
main.py                             18      3   83%
models/__init__.py                   0      0  100%
models/events.py                     22      0  100%
models/users.py                      12      0  100%
routes/__init__.py                   0      0  100%
routes/events.py                     41      4   90%
routes/users.py                      27      3   89%
tests/conftest.py                    23      0  100%
tests/test_arithmetic_operations.py  16      0  100%
tests/test_fixture.py                 7      0  100%
tests/test_login.py                  17      0  100%
tests/test_routes.py                 59      0  100%
TOTAL                               327     18   94%
(pyenv) + planner git:(main) x

```

Рисунок 8.16 – Отчет о покрытии с терминала

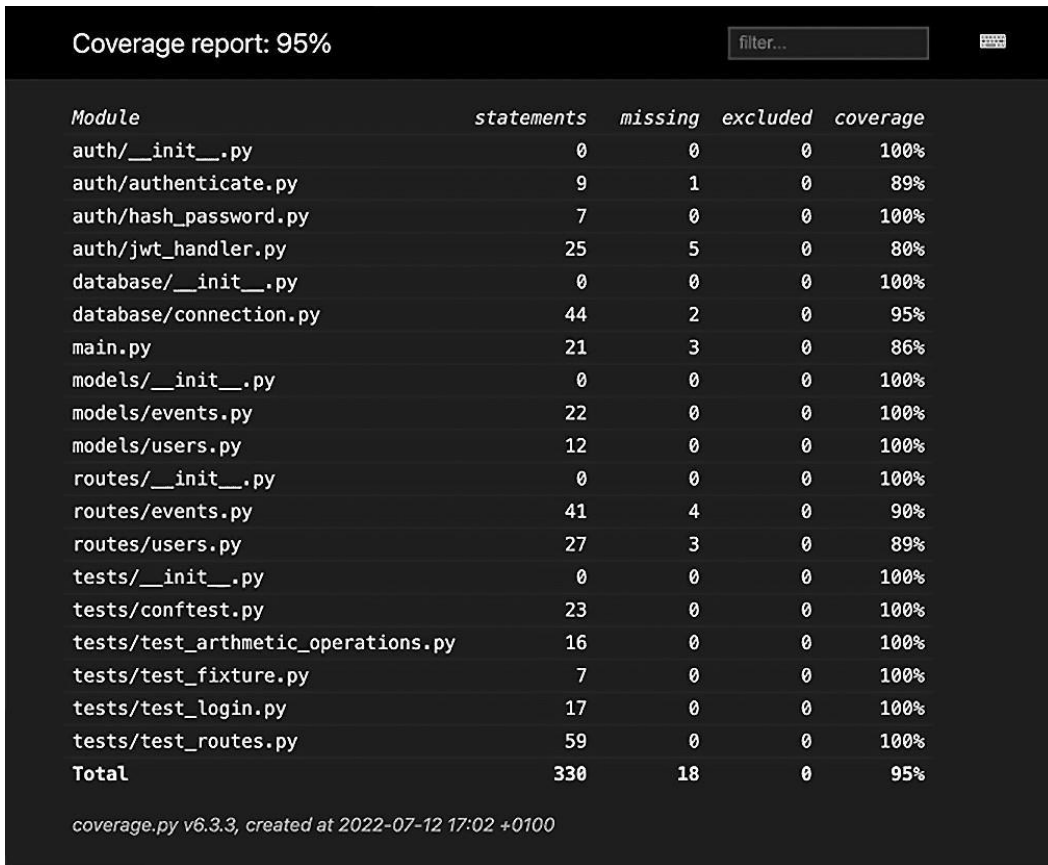
Из предыдущего отчета проценты означают количество кода, выполненного и с которым взаимодействовали. Давайте создадим HTML-отчет, чтобы мы могли проверить блоки кода, с которыми мы взаимодействовали.



```
pytest
(venv) → planner git:(main) * coverage html
Wrote HTML report to htmlcov/index.html
(venv) → planner git:(main) *
```

Рисунок 8.17 – Создание отчета о покрытии в формате HTML

Затем откройте `htmlcov/index.html` в своем браузере.

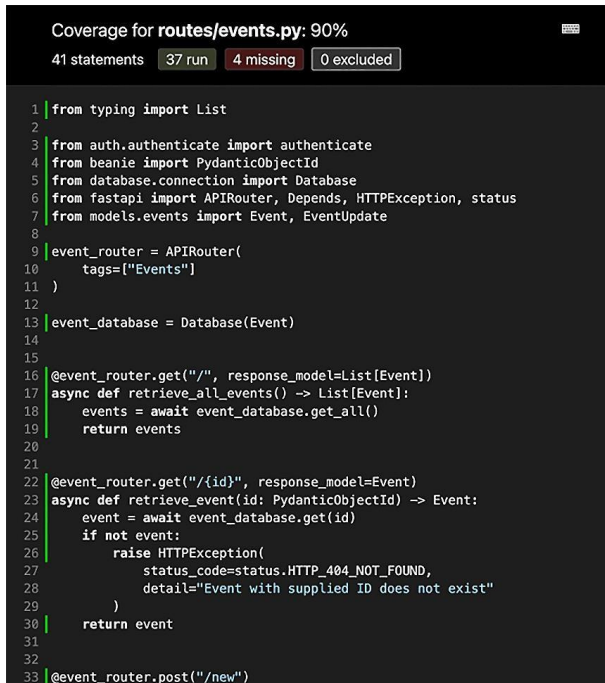


Module	statements	missing	excluded	coverage
auth/__init__.py	0	0	0	100%
auth/authenticate.py	9	1	0	89%
auth/hash_password.py	7	0	0	100%
auth/jwt_handler.py	25	5	0	80%
database/__init__.py	0	0	0	100%
database/connection.py	44	2	0	95%
main.py	21	3	0	86%
models/__init__.py	0	0	0	100%
models/events.py	22	0	0	100%
models/users.py	12	0	0	100%
routes/__init__.py	0	0	0	100%
routes/events.py	41	4	0	90%
routes/users.py	27	3	0	89%
tests/__init__.py	0	0	0	100%
tests/conftest.py	23	0	0	100%
tests/test_arithmetic_operations.py	16	0	0	100%
tests/test_fixture.py	7	0	0	100%
tests/test_login.py	17	0	0	100%
tests/test_routes.py	59	0	0	100%
Total	330	18	0	95%

coverage.py v6.3.3, created at 2022-07-12 17:02 +0100

Рисунок 8.18 – Отчет о покрытии из веб-браузера

Давайте проверим отчет о покрытии для `routes/events.py`. Нажмите на него, чтобы отобразить его.



```

Coverage for routes/events.py: 90%
41 statements 37 run 4 missing 0 excluded

1 | from typing import List
2 |
3 | from auth.authenticate import authenticate
4 | from beanie import PydanticObjectId
5 | from database.connection import Database
6 | from fastapi import APIRouter, Depends, HTTPException, status
7 | from models.events import Event, EventUpdate
8 |
9 | event_router = APIRouter(
10 |     tags=["Events"]
11 | )
12 |
13 | event_database = Database(Event)
14 |
15 |
16 | @event_router.get("/", response_model=List[Event])
17 | async def retrieve_all_events() -> List[Event]:
18 |     events = await event_database.get_all()
19 |     return events
20 |
21 |
22 | @event_router.get("/{id}", response_model=Event)
23 | async def retrieve_event(id: PydanticObjectId) -> Event:
24 |     event = await event_database.get(id)
25 |     if not event:
26 |         raise HTTPException(
27 |             status_code=status.HTTP_404_NOT_FOUND,
28 |             detail="Event with supplied ID does not exist"
29 |         )
30 |     return event
31 |
32 |
33 | @event_router.post("/new")

```

Рисунок 8.19 – Отчет о покрытии, показывающий выполненный код зеленым цветом и нетронутый код красным

Резюме

В этой главе вы успешно протестировали API, написав тесты для маршрутов аутентификации и маршрута CRUD. Вы узнали, что такое тестирование и как писать тесты `pytest`, библиотекой быстрого тестирования, созданной для приложений Python. Вы также узнали, что такое фикстуры `pytest`, и использовали их для создания многократно используемых токенов доступа и объектов базы данных, а также для сохранения экземпляра приложения на протяжении всего сеанса тестирования. Вы смогли утвердить ответы на ваши HTTP-запросы API и проверить поведение вашего API. Наконец, вы научились генерировать отчет о покрытии для своих тестов и различать блоки кода, выполняемые во время сеанса тестирования.

Теперь, когда вы получили знания о тестировании веб-API, вы готовы опубликовать свое приложение во всемирной паутине через канал развертывания. В следующей и последней главе вы узнаете, как контейнеризовать приложение и развернуть его локально с помощью `Docker` и `docker-compose`.

9

Развертывание приложений FastAPI

В последней главе вы узнали, как писать тесты для конечных точек API, созданных в приложении FastAPI. Мы начали с изучения того, что означает тестирование, и рассмотрели основы модульного тестирования с использованием библиотеки `pytest`. Мы также рассмотрели, как исключить повторение и повторное использование тестовых компонентов с фикстурами, а затем приступили к настройке нашей тестовой среды. Мы завершили последнюю главу, написав тесты для каждой конечной точки, а затем протестировали их вместе с проверкой отчетов о покрытии тестами после тестирования.

В этой главе вы узнаете, как развернуть приложение FastAPI локально с помощью **Docker** и **docker-compose**. Также добавлен краткий раздел с внешними ресурсами для развертывания вашего приложения на бессерверных платформах по вашему выбору.

В этой главе мы рассмотрим следующие темы:

- Подготовка к развертыванию
- Развертывание с помощью Docker
- Развертывание образов Docker

Технические требования

Код, использованный в этой главе, можно найти по адресу <https://github.com/PacktPublishing/Building-Python-Web-APIs-with-FastAPI/tree/main/ch09/planner>.

Подготовка к развертыванию

Развертывание обычно знаменует собой конец жизненного цикла приложения. Перед развертыванием наших приложений мы должны убедиться, что установлены правильные параметры, необходимые для плавного развертывания. Эти параметры включают обеспечение актуальности зависимостей приложения в файле `requirements.txt`, настройку переменных среды и т. д.

Управление зависимостями

В нескольких предыдущих главах мы установили такие пакеты, как `beanie` и `pytest`. Эти пакеты отсутствуют в файле `requirements`, который служит менеджером зависимостей для нашего приложения. Важно, чтобы файл `requirements.txt` обновлялся.

В Python список пакетов, используемых в среде разработки, можно получить с помощью команды `pip freeze`. Команда `pip freeze` возвращает список всех пакетов, установленных напрямую, и зависимостей для каждого установленного пакета. К счастью, файл `requirements.txt` можно поддерживать вручную, что позволяет нам перечислять только основные пакеты, тем самым упрощая управление зависимостями.

Давайте перечислим зависимости, используемые приложением, прежде чем перезаписывать файл `requirements.txt`:

```
(venv)$ pip freeze
anyio==3.5.0
asgi-lifespan==1.0.1
asgiref==3.5.0
attrs==21.4.0
bcrypt==3.2.2
cffi==1.15.0
python-multipart==0.0.5
...
```

Команда возвращает несколько зависимостей, некоторые из которых мы не используем напрямую в приложении. Давайте вручную заполним файл `requirements.txt` пакетами, которые мы будем использовать:

requirements.txt

```
fastapi==0.78.0
bcrypt==3.2.2
beanie==1.11.1
email-validator==1.2.1
httpx==0.22.0
Jinja2==3.0.3
motor==2.5.1
passlib==1.7.4
pytest==7.1.2
python-multipart==.0.0.5
python-dotenv==0.20.0
python-jose==3.3.0
sqlmodel==0.0.6
uvicorn==0.17.6
```

В этом блоке кода мы заполнили файл `requirements.txt` зависимостями, используемыми непосредственно в нашем приложении.

Настройка переменных среды

Мы использовали переменные среды в главе 6 «Подключение к базе данных». Переменные среды могут быть введены во время развертывания, как мы увидим в следующем разделе.

Примечание

Важно отметить, что переменные среды должны правильно обрабатываться и храниться вне систем контроля версий, таких как GitHub.

Теперь, когда мы выполнили необходимые шаги по подготовке к развертыванию, давайте перейдем к локальному развертыванию нашего приложения с помощью Docker в следующем разделе.

Развертывание с помощью Docker

В главе 1 «Начало работы с FastAPI» вы познакомились с основами Docker и Dockerfile. В этом разделе вы будете писать Dockerfile для API планировщика событий.

Docker — самая популярная технология, используемая для контейнеризации. Контейнеры — это автономные системы, состоящие из пакетов, кода и зависимостей, которые позволяют им работать в разных средах практически без зависимости от среды их выполнения. Docker использует Dockerfiles для процесса контейнеризации.

Docker можно использовать как для локальной разработки, так и для развертывания приложений в рабочей среде. В этой главе мы рассмотрим только локальное развертывание, а также будут включены ссылки на официальные руководства по развертыванию в облачных службах.

Для управления приложениями с несколькими контейнерами, такими как контейнер приложения и контейнер базы данных, используется инструмент компоновки. Compose — это инструмент, используемый для управления многоконтейнерными приложениями Docker, определенными в файле конфигурации, обычно `docker-compose.yaml`. Инструмент компоновки, `docker-compose`, устанавливается вместе с движком Docker.

Написание Dockerfile

Dockerfile содержит набор инструкций, используемых для создания образа Docker. Созданный образ Docker затем можно распространять в реестры (частные и общедоступные), развертывать на облачных серверах, таких как AWS и Google Cloud, и использовать в разных операционных системах путем создания контейнера.

Теперь, когда мы знаем, что делает Dockerfile, давайте создадим Dockerfile для сборки образа приложения. В каталоге проекта создайте файл Dockerfile:

```
(venv)$ touch Dockerfile
```

Dockerfile

```
FROM python:3.10

WORKDIR /app

COPY requirements.txt /app

RUN pip install --upgrade pip && pip install -r /app/requirements.txt
```

```
EXPOSE 8080
```

```
COPY ./ /app
```

```
CMD ["python", "main.py"]
```

Давайте пройдемся по инструкциям, содержащимся в предыдущем файле Dockerfile, одну за другой:

- Первая инструкция, которую выполняет Dockerfile, — установить базовый образ для нашего собственного образа с помощью ключевого слова `FROM`. Другие варианты этого образа можно найти по адресу https://hub.docker.com/_/python.
- В следующей строке ключевое слово `WORKDIR` используется для задания рабочего каталога `/app`. Рабочий каталог помогает организовать структуру проекта, построенного на образе.
- Затем мы копируем файл `requirements.txt` из локального каталога в рабочий каталог контейнера Docker, используя ключевое слово `COPY`.
- Следующая инструкция — это команда `RUN`, которая используется для обновления пакета `pip` и последующей установки зависимостей из файла `requirements.txt`.
- Следующая команда предоставляет `ПОРТ`, через который к нашему приложению можно получить доступ из локальной сети.
- Следующая команда копирует остальные файлы и папки в рабочий каталог контейнера Docker.
- Наконец, последняя команда запускает приложение с помощью команды `CMD`.

Каждый набор инструкций, перечисленных в Dockerfile, создается как отдельный уровень. Docker выполняет интеллектуальную работу по кэшированию каждого слоя во время сборки, чтобы сократить время сборки и исключить повторение. Если слой, который по сути является инструкцией, остается нетронутым, этот слой пропускается и используется ранее созданный. То есть Docker использует систему кэширования при сборке образов.

Давайте создадим файл `.dockerignore`, прежде чем приступить к сборке образа:

```
(venv)$ touch .dockerignore
```

.dockerignore

```
Venv  
.env  
.git
```

Что такое .dockerignore?

Файл .dockerignore содержит файлы и папки, которые должны быть исключены из инструкций, определенных в файле Dockerfile.

Создание Docker образа

Чтобы создать образ приложения, выполните следующую команду в базовом каталоге:

```
(venv)$ docker build -t event-planner-api .
```

Эта команда просто указывает Docker создать образ с тегом event-planner-api из инструкций, определенных в текущем каталоге, который представлен точкой в конце команды. Процесс сборки начинается после запуска команды и выполнения инструкций:

```

→ planner git:(main) x docker build -t event-planner-api .
[+] Building 107.0s (10/10) FINISHED
=> [internal] load build definition from Dockerfile                                0.0s
=> => transferring dockerfile: 244B                                              0.0s
=> [internal] load .dockerignore                                                  0.0s
=> => transferring context: 34B                                                  0.0s
=> [internal] load metadata for docker.io/library/python:3.10                  3.0s
=> [1/5] FROM docker.io/library/python:3.10@sha256:bef6fd726fb8825d5cf26933d8477505b 72.8s
=> => resolve docker.io/library/python:3.10@sha256:bef6fd726fb8825d5cf26933d8477505b1 0.0s
=> => sha256:bef6fd726fb8825d5cf26933d8477505b14505be1a98394405c8840c 2.35kB / 2.35kB 0.0s
=> => sha256:204633a9721c179b4ee2870a56bf3d50be77c83a6fd4cda99fd6b0cb 8.54kB / 8.54kB 0.0s
=> => sha256:29e37b4c58dd1db7ead6f3c2cdf757f490b4e29c958d2a70559c31 10.66MB / 10.66MB 5.8s
=> => sha256:de898bf7d6164091d354ef4d27a3e175a9aabd907317019c7e186fc5 2.22kB / 2.22kB 0.0s
=> => sha256:d794814721d57f8aaec06ab3652e90212cc3beccf5ff5c87f6ecf 53.70MB / 53.70MB 22.5s
=> => sha256:bf62ee63325dbbad699d6845f68c2391db3bf158f60373849c2d1cb6 4.94MB / 4.94MB 4.1s
=> => sha256:9cb366fec153b3461ef6bedb0d03ddf52da9b314025abfccb50a 54.67MB / 54.67MB 38.4s
=> => sha256:59fa95e4c98bdbabeee8d446eb3369c6717c3f04eb0a3bee45 189.65MB / 189.65MB 66.9s
=> => sha256:cd9a5a779b84da710e04b3ac413a7e6b4e243a9bc5510aaecc67811 6.16MB / 6.16MB 28.4s
=> => extracting sha256:d794814721d57f8aaec06ab3652e90212cc3beccf5ff5c87f6ecf8375784b 1.3s
=> => extracting sha256:bf62ee63325dbbad699d6845f68c2391db3bf158f60373849c2d1cb6bb479 0.1s
=> => extracting sha256:29e37b4c58dd1db7ead6f3c2cdf757f490b4e29c958d2a70559c313e9a03a 0.2s
=> => sha256:1270f467b9d2bf240cca84722b20333ba60872159d16b4cf26cde 19.30MB / 19.30MB 40.4s
=> => extracting sha256:9cb366fec153b3461ef6bedb0d03ddf52da9b314025abfccb50a3e8e8669 1.5s
=> => sha256:3d670a2dea2b75a4315a86d3cd54803d766e63c9a51a2f18d42e13b7e7 233B / 233B 40.2s
=> => sha256:9f0a233bbdcca5f5faea823f944fcc148a4de3cb02c5e1280e8b7e 2.87MB / 2.87MB 42.0s

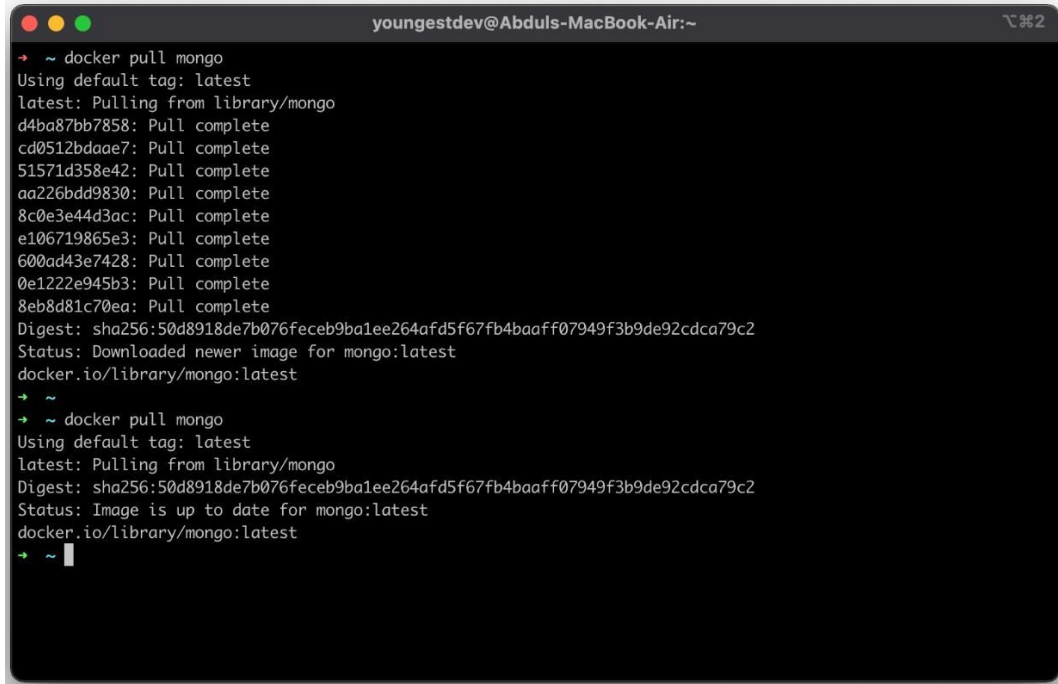
```

Рисунок 9.1 – Процесс сборки Docker

Теперь, когда мы успешно создали образ нашего приложения, давайте вытащим образ MongoDB:

```
(venv)$ docker pull mongo
```

Мы извлекаем образ MongoDB, чтобы создать автономный контейнер базы данных, доступный из контейнера API при создании. По умолчанию контейнер Docker имеет отдельную сетевую конфигурацию, и подключение к локальному адресу хостов машины запрещено.



```

youngestdev@Abduls-MacBook-Air:~
→ ~ docker pull mongo
Using default tag: latest
latest: Pulling from library/mongo
d4ba87bb7858: Pull complete
cd0512bdaae7: Pull complete
51571d358e42: Pull complete
aa226bdd9830: Pull complete
8c0e3e44d3ac: Pull complete
e106719865e3: Pull complete
600ad43e7428: Pull complete
0e1222e945b3: Pull complete
8eb8d81c70ea: Pull complete
Digest: sha256:50d8918de7b076feceb9ba1ee264afd5f67fb4baaff07949f3b9de92cdca79c2
Status: Downloaded newer image for mongo:latest
docker.io/library/mongo:latest
→ ~
→ ~ docker pull mongo
Using default tag: latest
latest: Pulling from library/mongo
Digest: sha256:50d8918de7b076feceb9ba1ee264afd5f67fb4baaff07949f3b9de92cdca79c2
Status: Image is up to date for mongo:latest
docker.io/library/mongo:latest
→ ~

```

Рисунок 9.2 – Получение образа MongoDB

Что такое docker pull?

Команда `docker pull` отвечает за загрузку образов из реестра. Если не указано иное, эти образы загружаются из общедоступного реестра Docker Hub.

Развертывание приложения локально

Теперь, когда мы создали образы для API и загрузили образ для базы данных MongoDB, давайте приступим к написанию манифеста для обработки развертывания нашего приложения. Манифест `docker-compose` будет состоять из службы API и службы базы данных MongoDB. В корневом каталоге создайте файл манифеста:

```
(venv)$ touch docker-compose.yml
```

Содержимое файла манифеста docker-compose будет следующим:

docker-compose.yml

```
version: "3"

services:
  api:
    build: .
    image: event-planner-api:latest
    ports:
      - "8080:8080"
    env_file:
      - .env.prod

  database:
    image: mongo
    ports:
      - "27017"
    volumes:
      - data:/data/db

volumes:
  data:
```

В разделе `services` у нас есть служба `api` и служба `database`. В службе `api` применяется следующий набор инструкций:

- Поле `build` указывает Docker на создание образа `event-planner-api:latest` для службы `api` из файла `Dockerfile`, расположенного в текущем каталоге, обозначенном `.`
- Порт 8080 открыт из контейнера, чтобы мы могли получить доступ к службе через HTTP.

- Файл среды имеет значение `.env.prod`. Кроме того, переменные среды могут быть установлены в этом формате:

```
environment:  
- DATABASE_URL=mongodb://database:27017/planner  
- SECRET_KEY=secretkey
```

Этот формат в основном используется, когда переменные среды должны вводиться из службы развертывания. Рекомендуется использовать файл окружения.

В службе `database` применяется следующий набор инструкций:

- Служба `database` использует образ `mongo`, который мы получили ранее.
- Порт `27017` определен, но не доступен извне. Порт доступен только внутри службы `api`.
- К службе подключен постоянный том для хранения наших данных. Для этого выделена папка `/data/db`.
- Наконец, том для этого развертывания создается с именем `data`.

Теперь, когда мы поняли содержимое манифеста компоновки, давайте создадим файл среды, `.env.prod`:

.env.prod

```
DATABASE_URL=mongodb://database:27017/planner  
SECRET_KEY=NOTSTRONGENOUGH!
```

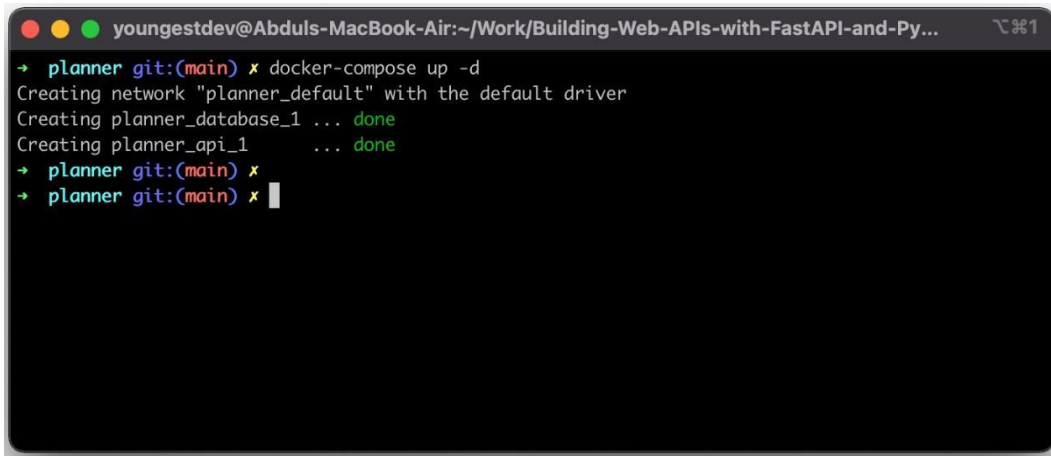
В файле среды `DATABASE_URL` задается именем службы MongoDB, созданной манифестом компоновки.

Запускаем приложение

Мы настроены на развертывание и запуск приложения из манифеста `docker-compose`. Давайте запустим сервисы с помощью инструмента компоновки:

```
(venv)$ docker-compose up -d
```

Эта команда запускает службы в автономном режиме:



```

youngestdev@Abduls-MacBook-Air:~/Work/Building-Web-APIs-with-FastAPI-and-Py...
→ planner git:(main) ✗ docker-compose up -d
Creating network "planner_default" with the default driver
Creating planner_database_1 ... done
Creating planner_api_1      ... done
→ planner git:(main) ✗
→ planner git:(main) ✗

```

Рисунок 9.3 – Запускаем приложение с помощью инструмента docker-compose

Службы приложений созданы и развернуты. Давайте проверим, проверив список запущенных контейнеров:

```
(venv) $ docker ps
```

Результат выглядит следующим образом:



```

youngestdev@Abduls-MacBook-Air:~/Work/Building-Web-APIs-with-FastAPI-and-Py...
→ planner git:(main) ✗ docker ps
CONTAINER ID   IMAGE                                COMMAND                  CREATED        STATUS
PORTS         NAMES
84423100df3e   event-planner-api:latest            "python main.py"        2 minutes ago Up 2 minutes
0.0.0.0:8080->8080/tcp, :::8080->8080/tcp   planner_api_1
01646e18fbb2   mongo                               "docker-entrypoint.s..." 2 minutes ago Up 2 minutes
0.0.0.0:50879->27017/tcp                    planner_database_1
→ planner git:(main) ✗

```

Рисунок 9.4 – Список запущенных контейнеров

Команда возвращает список контейнеров, работающих вместе с портами, через которые к ним можно получить доступ. Давайте проверим рабочее состояние, отправив запрос GET развернутому приложению:

```
(venv)$ curl -X 'GET' \  
  'http://localhost:8080/event/' \  
  -H 'accept: application/json'
```

Получаем следующий ответ:

```
[ ]
```

Замечательно! Развернутое приложение работает корректно. Давайте проверим, что база данных также работает, создав пользователя:

```
(venv)$ curl -X 'POST' \  
  'http://localhost:8080/user/signup' \  
  -H 'accept: application/json' \  
  -H 'Content-Type: application/json' \  
  -d '{  
    "email": "fastapi@packt.com",  
    "password": "strong!!!"  
  }'
```

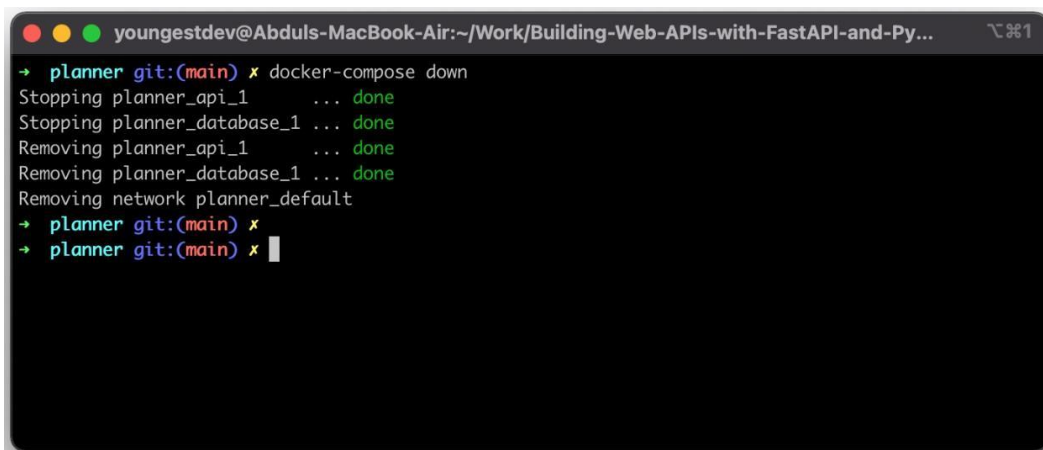
Мы также получаем положительный ответ:

```
{  
  "message": "User created successfully"  
}
```

Теперь, когда мы протестировали оба маршрута, вы можете приступить к тестированию других маршрутов. Чтобы остановить сервер развертывания после проверки, из корневого каталога запускается следующая команда:

```
(venv)$ docker-compose down
```

Результат выглядит следующим образом:



```
youngestdev@Abduls-MacBook-Air:~/Work/Building-Web-APIs-with-FastAPI-and-Py...
→ planner git:(main) x docker-compose down
Stopping planner_api_1 ... done
Stopping planner_database_1 ... done
Removing planner_api_1 ... done
Removing planner_database_1 ... done
Removing network planner_default
→ planner git:(main) x
→ planner git:(main) x
```

Рисунок 9.5 – Остановка экземпляров приложения

Развертывание Docker образов

В последнем разделе мы узнали, как создавать и развертывать образы Docker локально. Эти образы можно развернуть на любой виртуальной машине и на бессерверных платформах, таких как Google Cloud и AWS.

Обычный режим работы включает в себя отправку ваших образов Docker в частный реестр на бессерверной платформе. Процесс, связанный с развертыванием образов Docker на бессерверных платформах, варьируется от поставщика к поставщику, поэтому здесь приведены ссылки на избранных поставщиков бессерверных услуг:

- Google Cloud Run: <https://cloud.google.com/run/docs/quickstarts/build-and-deploy/python>
- Amazon EC2: <https://docs.aws.amazon.com/AmazonECS/latest/developerguide/getting-started-ecs-ec2.html>
- Развертывание в Microsoft Azure: <https://docs.microsoft.com/en-us/azure/container-instances/container-instances-tutorial-deploy-app>

Шаги, описанные в предыдущем разделе, можно выполнить, если образы Docker установлены на компьютере или традиционном сервере.

Развертывание баз данных

Платформы, такие как Google Cloud, AWS, предоставляют возможность размещения контейнеров вашей базы данных. Однако это может быть дорого с точки зрения эксплуатационных расходов и общей управляемости.

Для платформ, не поддерживающих развертывание манифестов docker-compose, базу данных MongoDB можно разместить на MongoDB Atlas (<https://www.mongodb.com/atlas/database>), а переменную среды DATABASE_URL перезаписать строкой подключения. Подробное руководство по настройке базы данных на атласе MongoDB можно найти по адресу <https://www.mongodb.com/docs/atlas/getting-started/>.

Резюме

В этой главе вы узнали, как подготовить приложение к развертыванию. Вы начали с обновления зависимостей, используемых для приложения, и сохранения их в Файл `requirements.txt`, прежде чем переходить к управлению переменными среды, используемыми для API.

Мы также рассмотрели шаги, связанные с развертыванием приложения в рабочей среде: создание образа Docker из Dockerfile, настройка манифеста компоновки для API и служб базы данных, а затем развертывание приложения. Вы также узнали новые команды для проверки списка запущенных контейнеров, а также для запуска и остановки контейнеров Docker. Наконец, вы протестировали приложение, чтобы убедиться, что развертывание прошло успешно.

На этом книга заканчивается, и теперь вы должны быть готовы к созданию, тестированию и развертыванию приложения FastAPI в Интернете. Мы рассмотрели различные концепции и убедились, что каждая концепция должным образом обсуждается с адекватными примерами: маршрутизация, шаблоны, аутентификация, подключение к базе данных и развертывание приложения. Обязательно ознакомьтесь с внешними ресурсами, время от времени упоминаемыми в книге, чтобы получить больше знаний!



Packt.com

Подпишитесь на нашу цифровую онлайн-библиотеку, чтобы получить полный доступ к более чем 7,000 книг и видео, а также к лучшим в отрасли инструментам, которые помогут вам спланировать свое личное развитие и продвинуться по карьерной лестнице. Для получения более подробной информации, пожалуйста, посетите наш веб-сайт.

Зачем подписываться?

- Тратьте меньше времени на обучение и больше времени на программирование с практическими электронными книгами и видео от более чем 4000 профессионалов отрасли
- Улучшите свое обучение с помощью планов навыков, созданных специально для вас
- Получайте бесплатную электронную книгу или видео каждый месяц
- Полная возможность поиска для легкого доступа к важной информации
- Копировать и вставлять, печатать и добавлять в закладки содержимое
- Знаете ли вы, что Packt предлагает электронные версии каждой опубликованной книги с доступными файлами PDF и ePub? Вы можете перейти на версию электронной книги на сайте packt.com, и, как покупатель печатной книги, вы имеете право на скидку на копию электронной книги. Свяжитесь с нами по адресу customercare@packtpub.com для получения более подробной информации.

На сайте www.packt.com, вы также можете прочитать подборку бесплатных технических статей, подписаться на ряд бесплатных информационных бюллетеней и получать эксклюзивные скидки и предложения на книги и электронные книги Packt.

Другие книги, которые могут вам понравиться

Если вам понравилась эта книга, возможно, вас заинтересуют и другие книги Packt:



Python Web Development with Sanic

Adam Hopkins

ISBN: 978-1-80181-441-6

- Понимать разницу между серверами WSGI, Async и ASGI
- Узнайте, как Sanic упорядочивает входящие данные, почему он это делает и как извлечь из этого максимальную пользу
- Внедрение передовых методов создания надежных, производительных и безопасных веб-приложений
- Изучите полезные методы для успешного тестирования и развертывания веб-приложения Sanic



Becoming an Enterprise Django Developer

Michael Dinder

ISBN: 978-1-80107-363-9

- Создайте и настройте экспериментальный проект Django на локальном компьютере
- Понимать шаги и инструменты, используемые для масштабирования проекта проверки концепции до производства, не углубляясь в конкретные технологии
- Узнайте об основных компонентах Django и о том, как использовать их по-разному в соответствии с потребностями вашего приложения
- Узнайте, как Django позволяет создавать RESTful API.
- Пишите и запускайте тестовые примеры, используя встроенные инструменты тестирования в Django

Рекст ищет таких авторов, как вы

Если вы хотите стать автором Packt, посетите страницу authors.packtpub.com и подайте заявку сегодня. Мы работали с тысячами разработчиков и технических специалистов, как и вы, чтобы помочь им поделиться своими знаниями с мировым техническим сообществом. Вы можете подать общую заявку, подать заявку на конкретную горячую тему, для которой мы набираем автора, или представить свою собственную идею.

Поделитесь своими мыслями

Теперь, когда вы закончили *создание веб-API Python с помощью FastAPI*, мы будем рады узнать ваше мнение! Если вы приобрели книгу на Amazon, нажмите здесь, чтобы перейти прямо на страницу обзора этой книги на Amazon и поделиться своим мнением или оставить отзыв на сайте, на котором вы ее приобрели.

Ваш отзыв важен для нас и технического сообщества и поможет нам убедиться, что мы предоставляем контент отличного качества.