

O'REILLY®

FastAPI

веб-разработка на Python



SPRINT
book

Билл Любанович

FastAPI

Modern Python Web Development

Bill Lubanovic

Beijing • Boston • Farnham • Sebastopol • Tokyo

O'REILLY®

FastAPI

веб-разработка на Python

Билл Любанович

ББК 32.988.02-018
УДК 004.738.5
Л93

Любанович Билл

Л93 FastAPI: веб-разработка на Python. — Астана: «Спринт Бук», 2024. — 288 с.: ил.
ISBN 978-601-08-3847-5

FastAPI — относительно новый, но надежный фреймворк с чистым дизайном, использующий преимущества актуальных возможностей Python. Как следует из названия, FastAPI отличается высоким быстродействием и способен конкурировать в этом с аналогичными фреймворками на таких языках, как Golang. Эта практическая книга расскажет разработчикам, знакомым с Python, как FastAPI позволяет достичь большего за меньшее время и с меньшим количеством кода.

Билл Любанович рассказывает о тонкостях разработки с применением FastAPI и предлагает множество рекомендаций по таким темам, как формы, доступ к базам данных, графика, карты и многое другое, что поможет освоить основы и даже пойти дальше. Кроме того, вы познакомитесь с RESTful API, приемами валидации данных, авторизации и повышения производительности. Благодаря сходству с такими фреймворками, как Flask и Django, вы легко начнете работу с FastAPI.

16+ (В соответствии с Федеральным законом от 29 декабря 2010 г. № 436-ФЗ.)

ББК 32.988.02-018
УДК 004.738.5

Права на издание получены по соглашению с O'Reilly. Все права защищены. Никакая часть данной книги не может быть воспроизведена в какой бы то ни было форме без письменного разрешения владельцев авторских прав.

Информация, содержащаяся в данной книге, получена из источников, рассматриваемых издательством как надежные. Тем не менее, имея в виду возможные человеческие или технические ошибки, издательство не может гарантировать абсолютную точность и полноту приводимых сведений и не несет ответственности за возможные ошибки, связанные с использованием книги. Издательство не несет ответственности за доступность материалов, ссылки на которые вы можете найти в этой книге. На момент подготовки книги к изданию все ссылки на интернет-ресурсы были действующими.

Издательство не несет ответственности за доступность материалов, ссылки на которые вы можете найти в этой книге. На момент подготовки книги к изданию все ссылки на интернет-ресурсы были действующими.

ISBN 978-1098135508 англ.

Authorized Russian translation of the English edition FastAPI.

ISBN 978-1098135508 © 2024 Bill Lubanovic.

This translation is published and sold by permission of O'Reilly Media, Inc., which owns or controls all rights to publish and sell the same.

ISBN 978-601-08-3847-5

© Перевод на русский язык ТОО «Спринт Бук», 2024

© Издание на русском языке, оформление ТОО «Спринт Бук», 2024

Оглавление

https://t.me/it_boooks/2

Предисловие	14
Условные обозначения.....	16
Примеры кода	16
Благодарности	17
От издательства	18

ЧАСТЬ I. ЧТО У НАС НОВОГО

Глава 1. Современная Всемирная паутина.....	21
Обзор	21
Сервисы и API.....	22
Конкурентность.....	26
Уровни (слои)	28
Данные	32
Заключение	32
Глава 2. Современный Python	33
Обзор	33
Инструменты	33
Приступим к работе	34
API и сервисы.....	38
Переменные — это имена.....	39
Подсказки типов	40

Структуры данных.....	41
Веб-фреймворки.....	41
Заключение	42

ЧАСТЬ II. ОБЗОР FASTAPI

Глава 3. Обзор FastAPI	44
Обзор	44
Что такое FastAPI	44
Приложение FastAPI	45
HTTP-запросы	49
HTTP-ответы	57
Автоматизированная документация	62
Комплексные данные	65
Заключение	65
Глава 4. Асинхронность, конкурентность и обзор библиотеки Starlette.....	66
Обзор	66
Библиотека Starlette	66
Типы конкурентности.....	67
FastAPI и асинхронность	73
Непосредственное использование Starlette	75
Немного отвлечемся: уборка в доме из игры Clue	76
Заключение	78
Глава 5. Pydantic, подсказки типов и обзор моделей	79
Обзор	79
Подсказки типов данных.....	79
Группировка данных	82
Альтернативы	86

Простой пример	88
Проверка типов.....	91
Проверка значений	92
Заключение	94
Глава 6. Зависимости	95
Обзор	95
Что такое зависимости.....	95
Проблемы с зависимостями	96
Внедрение зависимостей	96
Зависимости FastAPI.....	96
Написание зависимостей	97
Область действия зависимости	98
Заключение	100
Глава 7. Сравнение фреймворков	101
Обзор	101
Flask.....	102
Django	105
Другие функциональные возможности веб-фреймворка.....	105
Базы данных.....	106
Рекомендации	107
Другие веб-фреймворки Python	107
Заключение	108

ЧАСТЬ III. СОЗДАНИЕ ВЕБ-САЙТА

Глава 8. Веб-уровень.....	111
Обзор	111
Немного отвлечемся: сверху вниз, снизу вверх, от центра наружу?.....	112
Проектирование RESTful API	113

Макет сайта с файлами и каталогами	115
Первый код веб-сайта	117
Запросы	119
Несколько маршрутизаторов	121
Создание веб-уровня	122
Определение моделей данных	122
Заглушки и фиктивные данные	123
Создание общих функций с помощью стека	123
Создание фиктивных данных	124
Тестируем!	128
Использование форм автоматизированного тестирования FastAPI	130
Общение с уровнями сервисов и данных	132
Пагинация и сортировка	132
Заключение	134
Глава 9. Сервисный уровень	135
Обзор	135
Определение сервиса	135
Макет	136
Защита	136
Функции	137
Тестируем!	138
Другие нюансы сервисного уровня	140
Заключение	142
Глава 10. Уровень данных	143
Обзор	143
DB-API	143

SQLite.....	145
Макет.....	147
Заставляем все это работать.....	147
Тестируем!.....	152
Заключение.....	164
Глава 11. Аутентификация и авторизация.....	165
Обзор.....	165
Немного отвлечемся. Нужна ли вам аутентификация?.....	166
Методы аутентификации.....	167
Глобальная аутентификация — секретный ключ или общий секрет (Shared Secret).....	167
Простая индивидуальная аутентификация.....	171
Более сложная индивидуальная аутентификация.....	172
Авторизация.....	184
Промежуточное программное обеспечение.....	185
Заключение.....	188
Глава 12. Тестирование.....	189
Обзор.....	189
Тестирование Web API.....	189
Где тестировать.....	190
Что тестировать.....	191
Pytest.....	192
Макет.....	192
Автоматизированные модульные тесты.....	193
Автоматизированные интеграционные тесты.....	204
Паттерн «Репозиторий».....	205
Автоматизированные полные тесты.....	205

Тестирование безопасности.....	208
Нагрузочное тестирование	208
Заключение	209
Глава 13. Запуск в эксплуатацию.....	210
Обзор	210
Развертывание.....	210
Производительность	214
Устранение неполадок.....	216
Заключение	218

ЧАСТЬ IV. ГАЛЕРЕЯ

Глава 14. Базы данных, наука о данных и немного искусственного интеллекта	220
Обзор	220
Альтернативные варианты хранения данных.....	220
Реляционные базы данных и SQL.....	221
Нереляционные (NoSQL) базы данных.....	226
Возможности NoSQL в базах данных SQL.....	227
Нагрузочное тестирование баз данных.....	228
Наука о данных и искусственный интеллект	230
Заключение	233
Глава 15. Файлы	234
Обзор	234
Поддержка Multipart.....	234
Выгрузка файлов.....	234
Загрузка файлов.....	237
Предоставление статических файлов.....	239
Заключение	240

Глава 16. Формы и шаблоны	241
Обзор	241
Формы	241
Шаблоны.....	244
Заключение	246
 Глава 17. Обнаружение и визуализация данных	 247
Обзор	247
Python и данные.....	247
Текстовый вывод с помощью PSV	248
Источник данных SQLite и веб-вывод	251
Заключение	259
 Глава 18. Игры.....	 260
Обзор	260
Игровые пакеты в Python	260
Разделение игровой логики	261
Гейм-дизайн	261
Первая веб-часть — инициализация игры.....	263
Вторая веб-часть — этапы игры	264
Первая сервисная часть — инициализация	266
Вторая сервисная часть — определение результатов	266
Тестируем!.....	267
Данные — инициализация.....	268
Давайте поиграем в «Криптономикон».....	268
Заключение	270
 Приложение А. Дополнительная литература	 271
Python	271
FastAPI	272
Starlette.....	273
Pydantic	273

Приложение Б. Существа и люди	274
Существа.....	275
Исследователи.....	278
Публикации исследователей	279
Другие источники.....	279
 Об авторе	 280
Иллюстрация на обложке.....	281
Алфавитный указатель	282

*В память о моих жене Мэри, родителях,
Билле и Тилли, и друге Риче. Мне вас не хватает.*

Предисловие

Перед вами прагматичное введение в FastAPI — современный веб-фреймворк на Python. Это история о том, как время от времени встречающиеся нам яркие и блестящие предметы могут оказаться очень полезными. Серебряная пуля не мешает, если вы столкнетесь с оборотнем. (А в этой книге вы еще встретитесь с оборотнями.)

Я начал программировать научные приложения в середине 1970-х годов. И после первого знакомства с Unix и C на компьютере PDP-11 в 1977 году у меня появилось чувство, что Unix может укорениться.

В 1980-х и начале 1990-х годов Интернет был еще некоммерческим, но уже служил хорошим источником свободного программного обеспечения и технической информации. И когда в 1993 году в зарождающемся открытом Интернете появился браузер Mosaic, у меня возникло ощущение, что эта веб-возможность может прижиться.

Когда через несколько лет я открыл собственную компанию по веб-разработке, моими инструментами были самые обычные на тот момент PHP, HTML и Perl. Несколько лет спустя, работая по контракту, я наконец-то поэкспериментировал с Python и был удивлен тем, как быстро смог получить доступ к данным, манипулировать ими и отображать их. В свободное время за две недели я смог воспроизвести большую часть приложения на языке C, на написание которого у четырех разработчиков ушел год. Теперь у меня было ощущение, что история с Python — это хорошо.

После этого большая часть моей работы была связана с Python и его веб-фреймворками, в основном Flask и Django. Мне особенно понравилась простота Flask, и я предпочитаю использовать этот фреймворк для многих задач. А несколько лет назад я заметил нечто мелькнувшее на периферии моего зрения — новый веб-фреймворк на Python под названием FastAPI, написанный Себастьяном Рамиресом.

Я был впечатлен его продуманностью в процессе чтения превосходной документации (<https://fastapi.tiangolo.com>). В частности, страница описания — history (<https://oreil.ly/Ds-xM>) — показывает, насколько тщательно автор оценивал альтернативные варианты. Это был не эго-проект или забавный эксперимент, а серьезная основа для разработки в реальном мире. У меня было ощущение, что FastAPI может прижиться.

С помощью FastAPI я написал биомедицинский API-сайт, и все прошло настолько хорошо, что в течение следующего года наша команда переписала старый основной API на базе FastAPI. Он до сих пор находится в эксплуатации и отлично себя зарекомендовал. Наша группа изучила основы, приведенные в этой книге, и все заметили, что мы стали писать более качественный код, быстрее и с меньшим количеством ошибок. И кстати, некоторые из нас раньше не писали на Python — только я использовал FastAPI. Поэтому, когда появилась возможность предложить издательству O'Reilly продолжение моей книги под названием «Простой Python», FastAPI был на первом месте в списке тем. На мой взгляд, FastAPI окажет как минимум такое же влияние, как Flask и Django, а может, и большее.

Как я уже упоминал, на самом сайте FastAPI представлена документация мирового уровня, включающая множество подробностей по обычным веб-темам: базы данных, аутентификация, развертывание и т. д. Так зачем же писать что-то еще?

Эта книга не претендует на то, чтобы стать исчерпывающим материалом, потому что такой объем будет утомителен для читателя. Ее предназначение — быть полезной, помочь вам быстро понять основные идеи FastAPI и применить их на практике. Я укажу на различные приемы, потребовавшие определенных изысканий, и дам советы по использованию лучших практик в повседневной жизни.

В начале каждой главы я рассказываю о предстоящем материале. Далее стараюсь не забыть о том, что только что обещал, предлагая детали и отступления от темы. И наконец, делаю краткий обзор.

Как говорится, «это мнения, на которых основаны мои факты». Ваш опыт будет уникальным, но я надеюсь, что вы найдете здесь достаточно полезного, чтобы стать более продуктивным веб-разработчиком.

Условные обозначения

В этой книге используются следующие шрифтовые обозначения.

Курсив

Отмечает новые термины.

Рубленый шрифт

Им обозначены URL-адреса, адреса электронной почты и элементы интерфейса.

Моноширинный шрифт

Используется для примеров программного кода, а также внутри абзацев для ссылки на элементы программы, такие как имена переменных или функций, базы данных, типы данных, переменные окружения, операторы, ключевые слова, а также имена и расширения файлов.

Моноширинный полужирный шрифт

Показывает команды или другой текст, который должен быть набран пользователем буквально.

Моноширинный курсивный шрифт

Показывает текст, требующий замены значением, заданным пользователем или определяемым контекстом.



Этот элемент означает общее замечание.



Этот значок означает совет.

Примеры кода

Дополнительные материалы (примеры кода, упражнения и т. д.) доступны для скачивания на странице <https://github.com/madscheme/fastapi>.

Если у вас возникли технические вопросы или проблемы с использованием примеров кода, отправьте электронное письмо на наш почтовый ящик support@oreilly.com.

В общем случае все примеры кода из книги вы можете использовать в своих программах и в документации. Вам не нужно обращаться в издательство за разрешением, если вы не собираетесь воспроизводить существенные части программного кода. Если вы разрабатываете программу и используете в ней несколько фрагментов кода из книги, вам не нужно обращаться за разрешением. Но для продажи или распространения примеров из книги вам потребуется разрешение от издательства O'Reilly. Вы можете отвечать на вопросы, цитируя данную книгу или примеры из нее, но для включения существенных объемов программного кода из книги в документацию вашего продукта потребуется разрешение.

Мы рекомендуем, но не требуем добавлять ссылку на первоисточник при цитировании. Под ссылкой на первоисточник мы подразумеваем указание авторов, издательства и ISBN.

За получением разрешения на использование значительных объемов программного кода из книги обращайтесь по адресу permissions@oreilly.com.

Благодарности

Хотелось бы поблагодарить множество людей из различных организаций: школы Serra High School, Питтсбургского университета, лаборатории хронобиологии Миннесотского университета, авиакомпании Northwest Airlines, компаний Crosfield-Dicomed, Tela, WAM!NET, Mad Scheme, SESCO, Intradyn, Keep, Cray, Penguin Computing, Flywheel, медиакомпаний Thomson Reuters, организаций Intran и «Архив Интернета», стартапа CrowdStrike. Я многому у вас научился.

От издательства

Ваши замечания, предложения, вопросы отправляйте по адресу comp@sprintbook.kz (издательство «SprintBook», компьютерная редакция).

Мы будем рады узнать ваше мнение!

ЧАСТЬ I

Что у нас нового

https://t.me/it_boooks/2

Мир получил огромную пользу от изобретения Всемирной паутины (или просто «веб» как транслитерации названия Web) сэром Тимом Бернерсом-Ли¹ и языка программирования Python Гвидо ван Россумом.

Единственная небольшая проблема заключается в том, что безымянное издательство компьютерных книг часто помещает пауков и змей на свои обложки книг по тематике Web и Python соответственно. Если бы только сеть называлась Всемирный Гав, а язык Python был бы (Винни) Пухом, эта книга могла бы получить обложку как на рис. I.1.



Рис. I.1. FastAPI: современная разработка Гав-Пух

¹ Однажды я пожал ему руку. Я не мыл свою в течение месяца, но могу поспорить, что он сделал это сразу же.

Но я отвлекся¹. Эта книга посвящена таким темам, как:

- *Всемирная паутина* — особенно эффективная технология, как она изменилась и как теперь разрабатывать для нее программное обеспечение;
- *Python* — очень продуктивный язык для веб-разработки;
- *FastAPI* — особенно производительный веб-фреймворк для Python.

В двух главах части I книги обсуждаются новые темы в веб-разработке и языке Python — сервисы и API, конкурентность, многоуровневые архитектуры и большие-большие данные.

Часть II — это обзор FastAPI, свежего веб-фреймворка на Python. В этой части содержатся ответы на заданные в части I вопросы.

В части III мы углубляемся в инструментарий FastAPI, включая советы, полученные в процессе разработки.

Наконец, в части IV представлена галерея веб-примеров FastAPI. Для них использовался общий источник данных — список воображаемых существ, что может быть немного интереснее и целостнее, чем обычные случайные представления данных. Это должно дать вам отправную точку для конкретного применения этого веб-фреймворка.

¹ И точно не в последний раз.

Современная Всемирная паутина

Всемирную паутину, какой я ее себе представлял, мы еще не видели. Будущее все еще намного больше, чем прошлое.

Тим Бернерс-Ли

Обзор

Когда-то Всемирная паутина была маленькой и простой. Разработчикам было так весело отправлять вызовы PHP, HTML и MySQL в отдельные файлы и с гордостью говорить всем, что они могут заглянуть на свой веб-сайт. Но со временем Сеть разрослась до невообразимого количества страниц и развивающаяся игровая площадка превратилась в метавселенную тематических парков.

В этой главе отмечены некоторые все более актуальные для современной Всемирной паутины области:

- сервисы и API;
- конкурентность;
- уровни (слои);
- данные.

В следующей главе я расскажу о том, какие возможности предоставляет Python для работы в этих областях. После этого погрузимся в веб-фреймворк FastAPI и посмотрим, что он может предложить.

Сервисы и API

Паутина — это отличная соединительная ткань. Несмотря на то что большая часть деятельности по-прежнему происходит на стороне *контента* — HTML, JavaScript, изображений и т. д., все большее внимание уделяется интерфейсам прикладного программирования (API), соединяющим различные элементы программ.

Обычно *веб-сервис* управляет низкоуровневым доступом к базе данных и бизнес-логикой среднего уровня (часто их объединяют в *бэкенд*), а JavaScript или мобильные приложения обеспечивают богатый *фронтенд* верхнего уровня (интерактивный пользовательский интерфейс). Эти миры становятся все более сложными и разнообразными, что обычно требует от разработчиков специализации в том или ином направлении.

Быть разработчиком *полного стека* (фулстека) сейчас сложнее, чем раньше¹.

Эти два мира общаются друг с другом с помощью API. В современном Интернете дизайн API так же важен, как и дизайн самих сайтов. API — это контракт, подобный схеме базы данных. Определение и модификация API — это уже серьезная работа.

Виды API

Каждый API определяет следующее:

- *протокол* — структуру управления;
- *формат* — структуру содержимого.

Многочисленные методы API развивались по мере эволюции технологий от изолированных машин до многозадачных систем и сетевых серверов. Вероятно, в какой-то момент вы столкнетесь с одним или несколькими из них, поэтому далее приведу краткое описание, прежде чем перейти к языку *HTTP* и его друзьям, описанным в этой книге.

- До появления сетей API обычно означал очень тесную связь, например вызов функции из *библиотеки* на том же языке, что и ваше приложение, — скажем, вычисление квадратного корня в математической библиотеке.

¹ Я бросил попытки несколько лет назад.

- *Удаленные вызовы процедур* (remote procedure call, RPC) были придуманы для вызова функций в других процессах на той же или другой машине, как если бы они находились в вызывающем приложении. Популярным примером в настоящее время служит система gRPC (<https://grpc.io>).
- С помощью *системы передачи сообщений* отправляются небольшие фрагменты данных по конвейеру между процессами. Сообщения могут быть глагольными командами или просто обозначать интересующие вас *события*, похожие на существительные. В настоящее время популярными решениями для обмена сообщениями, которые варьируются от наборов инструментов до полноценных серверов, являются брокеры Apache Kafka (<https://kafka.apache.org>), RabbitMQ (<https://www.rabbitmq.com>), NATS (<https://nats.io>) и ZeroMQ (<https://zeromq.org>). Взаимодействие может строиться по разным схемам:
 - *запрос — ответ*. Точно так, как браузер вызывает веб-сервер;
 - *издатель — подписчик*, или *pub-sub*. Издатель (publisher, или pub) рассылает сообщения, а подписчики (subscribers, или sub) обрабатывают каждое из них в соответствии с некоторыми данными, содержащимися в сообщении, например с темой;
 - *очереди*. Работают как подход pub-sub, но только один подписчик из пула получает сообщение и действует в соответствии с ним.

Любой из этих подходов может использоваться вместе с веб-сервисом, например, для выполнения медленной задачи бэкенда, такой как отправка электронной почты или создание уменьшенного изображения.

HTTP

Бернерс-Ли предложил для своей Всемирной паутины три компонента:

- *HTML* — язык для отображения данных;
- *HTTP* — протокол «клиент — сервер»;
- *URL* — схему адресации для веб-ресурсов.

Хотя в ретроспективе все кажется очевидным, на деле это оказалось до смешного полезной комбинацией. По мере развития Интернета люди экспериментировали, и некоторые идеи, такие как тег `img`, выжили в борьбе по Дарвину. По мере того как потребности пользователей прояснялись, люди всерьез занялись определением стандартов.

REST(ful)

В одной из глав докторской диссертации (<https://oreil.ly/TwGmX>) Роя Филдинга есть определение *передачи репрезентативного состояния* (Representational State Transfer, REST) — *архитектурного стиля* для использования HTTP¹. Несмотря на то что на эту работу часто ссылаются, ее по большей части неправильно понимают (<https://oreil.ly/bsSry>).

В современной Паутине развилась и доминирует примерно одинаковая адаптация. Она называется *RESTful* и обладает следующими характеристиками:

- использует HTTP и протокол «клиент — сервер»;
- не имеет состояния (каждое соединение независимо);
- кэшируема;
- основана на ресурсах.

Ресурс — это данные, которые можно определять и с которыми можно выполнять операции. Веб-сервис предоставляет *конечную точку* — отдельный URL и HTTP-*глагол* (действие) — для каждой функции. Конечную точку называют также *маршрутом*, поскольку она направляет URL к функции.

Пользователи баз данных знакомы с акронимом *CRUD* для процедур: создание (create), чтение (read), модификация (update), удаление (delete). HTTP-глаголы довольно хорошо вписываются в понятие CRUD:

- POST — создание (запись);
- PUT — полная модификация (замена);
- PATCH — частичная модификация (обновление);
- GET — получение (считывание, извлечение);
- DELETE — удаление.

Клиент отправляет запрос на конечную точку RESTful с данными в одной из таких областей HTTP-сообщения, как:

¹ Под стилем понимается шаблон более высокого уровня, например «клиент — сервер», а не конкретная конструкция.

- заголовки;
- строка URL;
- параметры запроса;
- значения в теле сообщения.

В свою очередь, HTTP-ответ возвращает:

- целочисленное значение *кода состояния* (<https://oreil.ly/oBena>), определяющее такие состояния, как:
 - группа кодов 100 — информация, продолжение выполнения;
 - группа кодов 200 — успешное выполнение;
 - группа кодов 300 — перенаправление;
 - группа кодов 400 — ошибка на стороне клиента;
 - группа кодов 500 — ошибка на стороне сервера;
- различные заголовки;
- тело сообщения, которое может быть пустым, единым или разделенным на *части* (последовательные фрагменты).

По крайней мере один код состояния можно считать пасхалкой — 418 (I'm a teapot, <https://www.google.com/teapot>). На странице должен появиться подключенный к сети чайник. Если попросить, он нальет вам чашечку чая.



В широком доступе существует множество сайтов и книг о проектировании RESTful API, и все они содержат полезные практические указания. Эта книга станет вашим помощником в пути.

Форматы данных JSON и API

Фронтенд-приложения могут обмениваться обычным текстом на основе стандарта ASCII с веб-сервисами бэкенда, но как выразить структуры данных, такие как списки элементов?

Как раз в момент острой нужды появился формат «*обозначения объектов JavaScript*» (JavaScript Object Notation, JSON) — еще одна простая идея, решающая важную проблему и кажущаяся очевидной в ретроспективе. Хотя *J* означает *JavaScript*, синтаксис очень похож на Python.

JSON в значительной степени заменил такие более ранние попытки реализации этой идеи, как XML и SOAP. В оставшейся части этой книги вы увидите, что JSON — это формат для ввода и вывода у веб-сервисов по умолчанию.

JSON:API

Сочетание RESTful-дизайна и форматов данных JSON уже стало привычным. Но некоторые возможности для двусмысленности и занудства все же остаются. Недавнее предложение JSON:API (<https://jsonapi.org>) направлено на то, чтобы немного ужесточить спецификации. В этой книге используется свободный подход RESTful, но JSON:API или что-то подобное может оказаться полезным, если у вас возникнут серьезные затруднения.

GraphQL

Для некоторых целей RESTful-интерфейсы могут быть громоздкими. Facebook (сейчас Meta¹) разработала язык под названием Graph Query Language (GraphQL) (<https://graphql.org>). Он позволяет определить более гибкие запросы. В этой книге GraphQL не рассматривается, но, возможно, стоит обратить на него внимание, если вы считаете, что RESTful-дизайн не подходит для вашего приложения.

Конкурентность

Наряду с ростом ориентированности на сервисы стремительное увеличение количества подключений к веб-сервисам требует все большей эффективности и масштабирования. Нужно снизить следующие показатели:

¹ Деятельность запрещена в РФ.

- *время ожидания* — предварительное время ожидания;
- *пропускную способность* — количество байтов в секунду между сервисом и его абонентами.

В давние времена, работая во Всемирной паутине¹, люди мечтали о поддержке сотен одновременных подключений, затем беспокоились о «проблеме 10 000», а теперь обсуждают миллионные значения.

Термин «конкурентность» не означает полный параллелизм. Множественная обработка не происходит в одну и ту же наносекунду в одном процессоре. Конкурентность в основном позволяет избежать напряженного ожидания (простаивания центрального процессора (ЦП) до получения ответа). ЦП работают быстро, а сети и диски — в тысячи и миллионы раз медленнее. Поэтому при обращении к сети или диску никто не хочет просто сидеть с пустым взглядом, пока не поступит ответ.

Обычное выполнение Python *синхронизировано* — выполняется одно действие за раз в порядке, указанном кодом. Иногда требуется работа в *асинхронном* режиме — выполнить немного одного, потом немного другого, вернуться к первому и т. д. Если весь код использует центральный процессор для вычислений (*CPU bound*), то у нас нет свободного времени на асинхронность. Но если выполняется процесс, указывающий процессору ждать завершения операции от внешнего источника (*I/O bound*), можно организовать асинхронность.

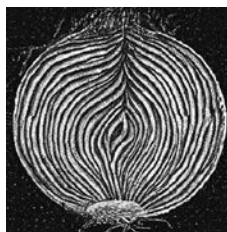
Асинхронные системы обеспечивают *цикл событий*: запросы на медленные операции отправляются и отмечаются, но в ожидании их ответов не происходит задержка работы ЦП. Вместо этого при каждом проходе через цикл выполняется немедленная обработка, а все ответы, поступившие за это время, обрабатываются при следующем проходе.

Эффект от этих действий может быть драматическим. Далее в книге вы увидите, как поддержка асинхронной обработки в FastAPI делает его намного быстрее, чем типичные веб-фреймворки. Асинхронная обработка — это не волшебство. Вам все еще нужно проявлять осторожность, чтобы не выполнять слишком много работы, требующей больших затрат ресурсов ЦП, во время цикла событий, потому что это замедлит весь процесс выполнения. Далее в книге вы увидите, как используются ключевые слова *async* и *await* языка Python и как FastAPI позволяет сочетать синхронную и асинхронную обработку.

¹ Примерно тогда, когда пещерные люди играли в футбол с гигантскими наземными ленивцами.

Уровни (слои)

Поклонники Шрека, возможно, помнят, как он отметил слои своей личности, на что Осел уточнил: «Как луковица?»



Ну, если у людоедов и вызывающих слезотечение овощей есть слои, то и у программного обеспечения тоже. Чтобы управлять размером и сложностью, многие приложения уже давно используют так называемую *трехуровневую модель*¹. Это не так уж и ново. Термины могут быть различными², но в этой книге я подразумеваю следующее простое разделение понятий (рис. 1.1):

- *веб-уровень* — уровень ввода/вывода поверх HTTP. Он собирает клиентские запросы, вызывает сервисный уровень и возвращает ответы;
- *сервис* — бизнес-логика, при необходимости выполняющая обращения к уровню данных;
- *данные* — доступ к хранилищам данных и другим сервисам;
- *модель* — определения данных, общие для всех уровней;
- *веб-клиент* — веб-браузер или другое программное обеспечение на стороне клиента HTTP;
- *база данных* — хранилище данных, часто SQL- или NoSQL-сервер.

Эти компоненты помогут вам масштабировать сайт, не начиная с нуля. Их нельзя сравнивать с законами квантовой механики, поэтому считайте их руководством к изложению материала в этой книге.

¹ Выберите свой вариант: уровень/слой, помидор/томат.

² Часто можно встретить термин «модель — представление — контроллер» (Model — View — Controller, MVC) и похожие варианты. Обычно это сопровождается религиозными войнами, но здесь я агностик.

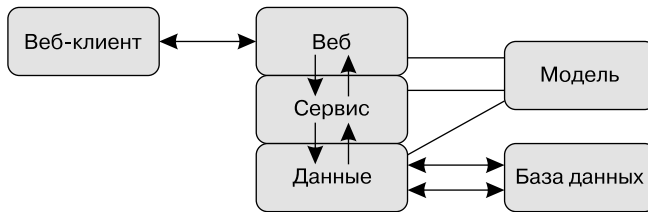


Рис. 1.1. Вертикальные уровни

Уровни взаимодействуют друг с другом через API. Это могут быть простые вызовы функций к отдельным модулям Python, но можно обращаться к внешнему коду через любой метод. Как я показывал ранее, это могут быть RPC, сообщения и т. д. В этой книге я предполагаю наличие одного веб-сервера с кодом Python, импортирующим другие модули Python. Разделение и сокрытие информации выполняется модулями.

Пользователи видят *веб-уровень*, задействуя клиентские приложения и API. Обычно мы говорим о RESTful-веб-интерфейсе с URL-адресами, запросами и закодированными в формате JSON ответами. Но наряду с веб-слоем могут быть созданы альтернативные текстовые клиенты или интерфейс командной строки (Command-Line Interface, CLI). Веб-код Python может импортировать модули сервисного уровня, но не должен импортировать модули уровня данных.

Сервисный уровень содержит фактические данные о том, что предоставляет этот веб-сайт. По сути, этот уровень похож на *библиотеку*. Он импортирует модули уровня данных для доступа к базам данных и внешним сервисам, но не должен получать от них детальную информацию.

Уровень данных предоставляет уровню сервисов доступ к данным через файлы или клиентские вызовы других сервисов. Могут существовать и альтернативные уровни данных, взаимодействующие с одним сервисным уровнем.

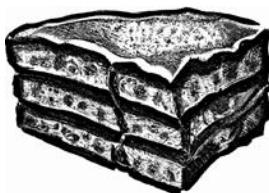
Блочная модель (*model box*) — это не настоящий уровень, а источник определений данных, общих для всех уровней. Он не требуется, если вы передаете между уровнями встроенные структуры данных Python. Позже вы увидите, что включение библиотеки Pydantic в FastAPI позволяет определять структуры данных с множеством полезных функциональных возможностей.

Зачем проводить такие разделения? По многим причинам каждый уровень может быть:

- написан специалистами;
- изолированно протестирован;
- заменен или дополнен — вы можете добавить второй веб-уровень, использующий другой API, например gRPC, наряду с веб-уровнем.

Следуйте одному правилу из фильма «Охотники за привидениями» — не скрещивайте лучи. То есть не позволяйте частям веб-сайтов просачиваться за пределы веб-уровня, а деталям баз данных — за пределы уровня данных.

Вы можете представить себе уровни в виде вертикальной стопки, как торт в телепередаче «Лучший пекарь Британии»¹.



Вот несколько причин для разделения уровней.

- Если вы не разделите уровни, то ожидайте, что станете широко известным веб-мемом: *«Теперь у вас две проблемы»*.
- Разделить уровни, если они смешаются, будет очень сложно.
- Вам потребуется знание двух или более специальностей, чтобы понять и написать тесты, если логика кода запуталась.

Кстати, хотя я и называю их *уровнями*, не нужно считать, что один из них находится выше или ниже другого и что команды перемещаются с помощью гравитации. Это было бы проявлением вертикального шовинизма! Можете рассматривать уровни как блоки, стоящие бок о бок друг с другом (рис. 1.2).

Как бы вы их ни представляли, *единственными* путями связи между блоками/уровнями могут служить стрелки (API). Это важно для тестирования и от-

¹ Как известно, если слои вашего торта сложены небрежно, вы можете не вернуться в шатер на следующей неделе.

ладки. Если на фабрике есть неизвестные двери, ночной сторож неизбежно будет удивлен.

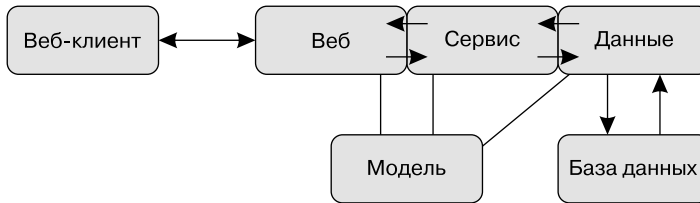


Рис. 1.2. Блоки, стоящие рядом

Стрелки между веб-клиентом и веб-уровнем задействуют протокол HTTP или HTTPS для передачи текста, преимущественно в формате JSON. Стрелки между уровнями данных и баз данных используют особый протокол для баз данных и передают текст в формате SQL или в другом. Стрелки между уровнями — это вызовы функций, переносящие модели данных.

Кроме того, рекомендуемые форматы данных, проходящих через стрелки, таковы:

- *клиент* \Leftrightarrow *веб-уровень* — RESTful HTTP с помощью JSON;
- *веб-уровень* \Leftrightarrow *сервис* — модели;
- *сервис* \Leftrightarrow *данные* — модели;
- *данные* \Leftrightarrow *базы данных* и *сервис* — специализированные API.

Основываясь на собственном опыте, я выбрал именно такую структуру тем для этой книги. Она вполне работоспособная и подходит для довольно сложных сайтов, но не единственно верная. Вы можете создать лучший дизайн! Как бы вы ни поступили, обратите внимание на основные ключевые точки.

- Отделите свойственные домену детали.
- Определите стандартные API между уровнями.
- Не обманывайте, не допускайте утечек.

Иногда решить, какой уровень лучше всего подходит для кода, бывает непросто. Например, в главе 11 рассматриваются требования к аутентификации и авторизации и способы их реализации — в качестве дополнительного уровня между веб- и сервисным уровнем или внутри одного из них. Разработка программного обеспечения — это порой не только искусство, но и наука.

Данные

Веб-уровень часто использовался как фронтенд для реляционных баз данных, хотя в настоящее время появилось множество других способов хранения данных и доступа к ним, например базы данных типа NoSQL или NewSQL.

Помимо баз данных, кардинально меняет технологический ландшафт *машинное обучение* (Machine Learning, ML), или глубокое обучение, или просто искусственный интеллект (ИИ). Разработка больших моделей требует значительной работы с данными, традиционно называемой извлечением, преобразованием, загрузкой (Extract, Transform, Load, ETL).

Будучи универсальной сервисной архитектурой, веб может помочь в решении многих сложных задач, связанных с системами ML.

Заключение

Во Всемирной паутине используется много API, но особенно много взаимодействий на основе RESTful. Асинхронные вызовы обеспечивают лучшую конкурентность, что ускоряет общий процесс выполнения. Приложения веб-сервисов часто бывают достаточно большими для того, чтобы разделить их на уровни. Данные стали самостоятельной важной областью. Все эти понятия рассматриваются в языке программирования Python. О нем и пойдет речь в следующей главе.

Современный Python

Это обычный рабочий день для
«Удиви-кота».

Монти Пайтон

https://t.me/it_boooks/2

Обзор

Python развивается, чтобы идти в ногу с меняющимся техническим миром. В этой главе рассматриваются специфические возможности этого языка, относящиеся к описанным в предыдущей главе вопросам, а также некоторые дополнительные:

- инструменты;
- API и сервисы;
- переменные и подсказки типов;
- структуры данных;
- веб-фреймворки.

Инструменты

В каждом языке программирования есть следующие элементы:

- основной язык и встроенные стандартные пакеты;
- способы добавления сторонних пакетов;
- рекомендуемые сторонние пакеты;
- среда инструментов разработки.

В следующих разделах перечислены инструменты Python, необходимые или рекомендуемые для работы с книгой.

Со временем они могут измениться! Средства упаковки и разработки Python — это движущиеся цели, и время от времени появляются более совершенные решения.

Приступим к работе

Вы должны уметь написать и запустить программу на Python, подобную приведенной в примере 2.1.

Пример 2.1. Программа на языке Python: `this.py`

```
def paid_promotion():
    print("(that calls this function!)")

print("This is the program")
paid_promotion()
print("that goes like this.")
```

Чтобы запустить эту программу из командной строки в текстовом окне или терминале, я буду использовать *подсказку \$* (ваша система умоляет вас набрать что-нибудь поскорее). То, что вы вводите после подсказки, отображается **полужирным шрифтом**. Если вы сохранили пример 2.1 в файл `this.py`, его можно запустить, как показано в примере 2.2.

Пример 2.2. Тестирование файла `this.py`

```
$ python this.py
This is the program
(that calls this function!)
that goes like this.
```

В некоторых примерах кода используется интерактивный интерпретатор Python. Доступ к нему можно получить, просто набрав слово `python`:

```
$ python
Python 3.9.1 (v3.9.1:1e5d33e9b9, Dec 7 2020, 12:10:52)
[Clang 6.0 (clang-600.0.57)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Первые несколько строк зависят от вашей операционной системы и версии Python. Символы `>>>` означают предложение ввода команд. Удобная дополнительная функция интерактивного интерпретатора заключается в том, что он выводит значение переменной, если вы введете ее имя:

```
>>> wrong_answer = 43
>>> wrong_answer
43
```

Это работает и для выражений:

```
>>> wrong_answer = 43
>>> wrong_answer - 3
40
```

Если вы совсем недавно познакомились с Python или хотите получить краткий обзор, прочтите следующие несколько подразделов.

Непосредственно Python

Вам понадобится как минимум Python 3.7. У него имеются такие возможности, как подсказки типов и модуль `asyncio`, которые являются основными требованиями FastAPI. Я же рекомендую использовать по крайней мере Python 3.9 с более длительным сроком поддержки. Стандартный источник для получения Python — Python Software Foundation (<https://www.python.org>).

Управление пакетами

Вам нужно будет загрузить сторонние пакеты Python и безопасно установить их на свой компьютер. Классическим инструментом для этого служит система `pip` (<https://pip.pypa.io>).

Но как загрузить этот загрузчик? Если вы установили Python от Python Software Foundation, у вас уже должен быть `pip`. Если нет, следуйте инструкциям на сайте `pip`, чтобы получить его. На протяжении всей этой книги, когда я буду представлять новый пакет Python, стану указывать и команду `pip` для его загрузки.

Вы можете многое сделать с помощью старого доброго `pip`, но, скорее всего, вам захочется также использовать виртуальные среды и рассмотреть альтернативные инструменты, такие как Poetry.

Виртуальные среды

Pip загрузит и установит пакеты, но куда он должен их поместить? Хотя стандартный Python и входящие в него библиотеки обычно устанавливаются в стандартное место в вашей операционной системе, вы не сможете (и, скорее всего, не должны) ничего там изменить. Pip использует каталог по умолчанию, отличный от системного, поэтому установка не попадет на стандартные файлы Python в вашей системе. Это можно изменить. Подробности для своей операционной системы смотрите на сайте `pip`.

Часто приходится работать с несколькими версиями Python или устанавливать версии для конкретного проекта, поэтому вам точно будет известно, какие пакеты находятся в системе. Для этого Python поддерживает *виртуальные среды*. Это просто каталоги (*папки* в не-Unix-мире), в которые pip записывает загруженные пакеты. Когда вы *активируете* виртуальную среду, ваша оболочка (интерпретатор основных системных команд) при загрузке модулей Python в первую очередь обращается к ней. Для этого используется программа `venv` (<https://oreil.ly/9kv5T>), которая входит в стандартный пакет установки Python начиная с версии 3.4.

Создадим виртуальную среду под названием `venv1`. Вы можете запустить модуль `venv` как отдельную программу:

```
$ venv venv1
```

или в качестве модуля Python:

```
$ python -m venv venv1
```

Чтобы сделать его своей текущей средой Python, выполните эту команду оболочки (в Linux или Mac, для Windows и других ОС смотрите документацию `venv`):

```
$ source venv1/bin/activate
```

Теперь каждый раз, когда вы запускаете `pip install`, пакеты будут устанавливаться в среду `venv1`. Когда запускаете программы Python, именно там находятся ваш интерпретатор Python и модули.

Чтобы отключить виртуальную среду, нажмите сочетание клавиш **Control+D** (для Linux или Mac) или введите команду `deactivate` (для Windows).

Можно создать альтернативные среды разработки, например `venv2`, и деактивировать/активировать их, чтобы переходить от одной к другой (хотя я надеюсь, что у вас лучше фантазия в плане названий, чем у меня).

Инструмент Poetry

Сочетание `pip` и `venv` настолько распространено, что люди начали комбинировать их, чтобы сократить этапы работы и избежать `source`-премудростей оболочки. Одним из таких пакетов стал `Pipenv` (<https://pipenv.pypa.io>), но более новый конкурент под названием `Poetry` (<https://python-poetry.org>) становится все популярнее.

Я использовал `pip`, `Pipenv` и `Poetry`, но теперь предпочитаю `Poetry`. Установить его можно с помощью команды `pip install poetry`. В `Poetry` есть множество дополнительных команд, таких как `poetry add` для добавления пакета в виртуальную среду, `poetry install`, чтобы загрузить и установить инструмент, и т. д. Просмотрите веб-сайт `Poetry` или запустите команду `poetry`, чтобы открыть раздел помощи.

Помимо загрузки отдельных пакетов, `pip` и `Poetry` управляют несколькими пакетами в файлах конфигурации — `requirements.txt` для `pip` и `pyproject.toml` для `Poetry`. `Poetry` и `pip` не просто загружают пакеты, но и управляют сложными зависимостями, которые могут существовать между пакетами. Вы можете задать желаемые варианты пакетов в виде минимумов, максимумов, диапазонов или точных значений, известных также как *pinning* или *привязка*. Этот вопрос может стать важным по мере роста проекта и изменения пакетов, от которых он зависит. Может понадобиться минимальная версия пакета, если используемая вами функция появилась в нем впервые, или максимальная, если функция была отменена.

Форматирование исходного кода

Форматирование исходного кода менее важно, чем темы предыдущих разделов, но все же полезно. Избегайте споров о форматировании кода (*bikeshedding*) с помощью инструмента, приводящего исходный текст к стандартному, не странному формату. Хороший вариант выбора — пакет `Black` (<https://black.readthedocs.io>). Установить его можно с помощью команды `pip install black`.

Тестирование

Тестирование подробно рассматривается в главе 12. Хотя стандартным тестовым пакетом Python является `unittest`, промышленный тестовый пакет Python, используемый большинством разработчиков Python, — это `pytest` (<https://docs.pytest.org>). Установить его можно с помощью команды `pip install pytest`.

Контроль исходного кода и непрерывная интеграция

Почти универсальным решением для контроля исходного кода сейчас является *Git* с хранилищами (репозиториями) на таких сайтах, как GitHub и GitLab. Использование *Git* нельзя считать чем-то специфическим для Python или FastAPI, но, скорее всего, вы будете проводить с ним большую часть своего времени в процессе разработки. Инструмент *pre-commit* (<https://pre-commit.com>) запускает на вашей локальной машине различные тесты, такие как *black* и *pytest*, перед тем как выполнить коммит в *Git*. После размещения в удаленном репозитории *Git* можно запустить там больше тестов непрерывной интеграции (Continuous Integration, CI).

Более подробная информация содержится в главе 12 и в разделе «Устранение неполадок» главы 13.

Веб-инструменты

В главе 3 показано, как установить и применять основные веб-инструменты Python, используемые в этой книге:

- *FastAPI* — сам веб-фреймворк;
- *Uvicorn* — асинхронный веб-сервер;
- *HTTPIe* — текстовый веб-клиент, похожий на *curl*;
- *Requests* — пакет синхронного веб-клиента;
- *HTTPX* — пакет синхронного/асинхронного веб-клиента.

API и сервисы

Модули и пакеты Python необходимы для создания больших приложений, которые не превращаются в «большие комки грязи» (<https://oreil.ly/zzX5T>). Даже в однопроцессном веб-сервисе можно сохранить описанное в главе 1 разделение с помощью тщательного проектирования модулей и импортов.

Встроенные структуры данных Python очень гибкие, и их очень заманчиво использовать повсюду. Но в следующих главах вы увидите, что можно определять модели более высокого уровня, чтобы сделать межуровневое взаимодействие более чистым. Эти модели опираются на относительно недавнее дополнение к Python, называемое *подсказкой типов* (type hinting). Давайте разберемся в этом вопросе, но сначала коротко о том, как Python работает с *переменными*. Это не мешает.

Переменные — это имена

Термин «объект» получил множество определений в мире программного обеспечения, возможно, даже слишком много. В Python *объект* — это структура данных для упаковки каждого отдельного фрагмента данных в программе, от целого числа 5 до функции и всего, что вы можете определить. В нем, помимо прочей отчетной информации, указываются:

- уникальное *идентификационное* значение;
- низкоуровневый *тип*, соответствующий аппаратному обеспечению;
- конкретное *значение* (физические биты);
- *подсчет* количества переменных, ссылающихся на него.

Python *строго типизирован* на уровне объектов (тип объекта не меняется, хотя его значение может меняться). Объект называется *изменяемым*, если его значение может быть изменено, и *неизменяемым*, если нет.

На уровне *переменных* Python отличается от многих других вычислительных языков, и это может сбивать с толку. Во многих других языках *переменная* — это, по сути, прямой указатель на область памяти, содержащую необработанное *значение*, хранящееся в битах, которые соответствуют аппаратному дизайну компьютера. Если вы присваиваете этой переменной новое значение, язык перезаписывает предыдущее значение в памяти новым.

Это прямое и быстрое решение. Компилятор следит за тем, что куда записывается. Это одна из причин того, почему такие языки, как С, быстрее Python. Как разработчику, вам необходимо убедиться, что вы присваиваете каждой переменной только значения правильного типа.

И здесь Python имеет существенное отличие — переменная в нем представляет собой просто *имя*, временно ассоциируемое с *объектом* более высокого уровня в памяти. Если вы присваиваете новое значение переменной, ссылающейся на неизменяемый объект, то фактически создаете новый объект, содержащий это значение, а затем получаете имя для ссылки на данный объект. Старый объект (на который раньше ссылалось имя) освобождается, и его память может быть восстановлена, если на него не ссылаются другие имена, то есть счетчик ссылок равен 0.

В книге «Простой Python»¹ я сравниваю объекты с пластиковыми коробками, стоящими на полках памяти, а имена/переменные — со стикерами на этих

¹ Любанович Б. Простой Python. Современный стиль программирования. 2-е изд. — СПб.: Питер, 2021.

коробках. Или вы можете представить имена как бирки, прикрепленные ниточками к этим коробкам.

Обычно при использовании имени вы присваиваете его одному объекту, и оно сохраняется за ним. Такая простая последовательность помогает понять ваш код. *Область видимости* переменной — это область кода, в которой имя ссылается на один и тот же объект, например, внутри функции. Вы можете применять одно и то же имя в разных областях видимости, но каждое из них будет ссылаться на разные объекты.

Можно сделать так, чтобы переменная ссылалась на разные объекты в программе Python, однако это не всегда хорошо. Не посмотрев, вы не узнаете, находится ли имя `x` в строке 100 в той же области видимости, что и имя `x` в строке 20. (Кстати, `x` — ужасный выбор имени переменной. Следует выбирать действительно значимые имена.)

Подсказки типов

Вся эта предыстория имеет определенный смысл.

В Python 3.6 добавлены *подсказки типов* (type hints) для объявления типа объекта, на который ссылается переменная. Они не выполняются интерпретатором Python во время его работы! Вместо этого их могут задействовать различные инструменты для обеспечения последовательного использования переменной. Стандартная программа проверки типов называется *туру*, и позже я покажу вам, как она работает.

Подсказка типа может показаться просто приятной вещью, как и многие инструменты *lint*, помогающие программистам избежать ошибок. Например, она может напомнить, что ваша переменная `count` ссылается на объект Python типа `int`. Но подсказки, хотя они и представляют собой дополнительную возможность и являются необязательными примечаниями (буквально намеками), оказываются неожиданно полезными. Далее в этой книге вы увидите, как FastAPI адаптировал пакет Pydantic, чтобы грамотно использовать подсказки типов.

Добавление объявлений типов может стать тенденцией в других языках, ранее не имевших типов. Например, многие разработчики JavaScript перешли на TypeScript (<https://www.typescriptlang.org>).

Структуры данных

Подробнее о Python и структурах данных вы узнаете в главе 5.

Веб-фреймворки

Помимо прочего, веб-фреймворк осуществляет перевод между байтами HTTP кода и структурами данных Python. Это поможет сэкономить много сил. В то же время если часть его работает не так, как вам нужно, то может понадобиться взломать решение. Как говорится, не надо изобретать колесо — разве что вы не можете найти круглое.

Web Server Gateway Interface (WSGI) (<https://wsgi.readthedocs.io>) — это спецификация стандарта синхронизации (<https://peps.python.org/pep-3333>) Python для подключения кода приложения к веб-серверам. Все традиционные веб-фреймворки Python построены на WSGI. Но синхронное взаимодействие может означать, что вы заняты ожиданием чего-то, что работает гораздо медленнее процессора, например запросов к дискам или сети. Тогда вы будете искать лучшую *конкурентность*. В последние годы она приобретает все большее значение. В результате была разработана спецификация Asynchronous Server Gateway Interface (ASGI) (<https://asgi.readthedocs.io>) для Python. Об этом рассказывается в главе 4.

Django

Django (<https://www.djangoproject.com>) — это полнофункциональный веб-фреймворк, обозначающий себя как веб-фреймворк для перфекционистов с жесткими сроками выполнения работы. Он был анонсирован Адрианом Холовати и Саймоном Уиллисоном в 2003 году и назван в честь Джанго Рейнхардта — бельгийского джазового гитариста XX века. Django часто используется для корпоративных сайтов с базами данных. Более подробное описание приводится в главе 7.

Flask

Flask (<https://flask.palletsprojects.com>), представленный Армином Ронахером в 2010 году, является *микрофреймворком*. В главе 7 вы найдете более подробную информацию о нем и его сравнение с Django и FastAPI.

FastAPI

Встретив на балу других ухажеров, мы наконец-то сталкиваемся с интригующим FastAPI, о котором и пойдет речь в этой книге. Хотя FastAPI был опубликован Себастьяном Рамиресом в 2018 году, он уже поднялся на третье место среди веб-фреймворков Python, уступая лишь Flask и Django, и развивается все быстрее. Выполненное в 2022 году сравнение (<https://oreil.ly/36WTQ>) показывает, что в какой-то момент он может обойти конкурентов.



Перед вами данные о количестве звезд на GitHub по состоянию на конец октября 2023 года:

- Django — 73,8 тыс.;
- Flask — 64,8 тыс.;
- FastAPI — 64 тыс.

После тщательного изучения альтернативных вариантов (<https://oreil.ly/JDDOm>) Рамирес разработал дизайн (<https://oreil.ly/zJFTX>), по большей части основанный на двух сторонних пакетах Python:

- *Starlette* — для получения подробной информации о веб-странице;
- *Pydantic* — для получения подробной информации о данных.

А в готовый продукт он добавлял собственные ингредиенты и особые соусы. В следующей главе вам станет ясно, о чем идет речь.

Заключение

В этой главе было рассмотрено множество вопросов, связанных с современным Python:

- полезные инструменты для веб-разработчика на Python;
- значимость API и сервисов;
- подсказки типов, объекты и переменные в Python;
- структуры данных для веб-сервисов;
- веб-фреймворки.

ЧАСТЬ II

Обзор FastAPI

Главы этой части дают представление о FastAPI с высоты птичьего полета, но скорее с высоты полета дрона, чем как взгляд со спутника-шпиона. В них кратко рассказывается об основах, но при этом описание не уходит ниже ватерлинии, чтобы не утопить вас в деталях. Эти главы относительно коротки и призваны обеспечить контекст для глубокого погружения в материал в части III книги.

После того как вы освоитесь с идеями, изложенными в этой части, в части III вы сможете более детально изучить их. Именно здесь вы можете принести серьезную пользу или вред своим познаниям. Не осуждайте, все зависит от вас.

ГЛАВА 3

Обзор FastAPI

FastAPI — это современный быстрый (высокопроизводительный) веб-фреймворк для создания API на Python 3.6+, основанный на стандартных подсказках типов Python.

Себастьян Рамирес, создатель FastAPI

Обзор

FastAPI (<https://fastapi.tiangolo.com>) был представлен в 2018 году Себастьяном Рамиресом (<https://tiangolo.com>). Во многих смыслах это более современный, чем большинство веб-фреймворков Python, и он использует добавленный в Python 3 за последние несколько лет функционал. Эта глава представляет собой краткий обзор основных возможностей FastAPI с акцентом на первом из интересующих вас вопросов: как обрабатывать веб-запросы и ответы?

Что такое FastAPI

Как и любой другой веб-фреймворк, FastAPI помогает создавать веб-приложения. Каждый фреймворк призван облегчить выполнение некоторых операций за счет особенностей, допущений и настроек по умолчанию. Как следует из названия, FastAPI предназначен для разработки веб-интерфейсов API, хотя можно использовать его и для традиционных приложений с веб-контентом.

На сайте FastAPI заявлены такие его преимущества:

- *высокая производительность* — в некоторых случаях он работает так же быстро, как Node.js и Go, что необычно для фреймворков Python;
- *ускоренный процесс разработки* — никаких острых углов или странностей;
- *повышение качества кода* — подсказки типов и модели помогают уменьшить количество ошибок;
- *автоматически генерируемая документация и тестовые страницы* — это гораздо проще, чем вручную редактировать описания OpenAPI.

В FastAPI используются:

- подсказки типов Python;
- пакет Starlette для веб-машин, включая поддержку асинхронности;
- пакет Pydantic для определения и проверки данных;
- специальная интеграция, позволяющая использовать и расширять возможности других фреймворков.

Такое сочетание создает приятную среду для разработки веб-приложений, особенно RESTful-веб-сервисов.

Приложение FastAPI

Напишем маленькое приложение FastAPI — веб-сервис с одной конечной точкой. Пока что будем работать на так называемом веб-уровне, где обрабатываются только веб-запросы и ответы. Сначала установите основные необходимые пакеты Python:

- фреймворк FastAPI (<https://fastapi.tiangolo.com>) — `pip install fastapi`;
- веб-сервер Uvicorn (<https://www.uvicorn.org>) — `pip install uvicorn`;
- текстовый веб-клиент HTTPie (<https://httpie.io>) — `pip install httpie`;
- пакет синхронного веб-клиента Requests (<https://requests.readthedocs.io>) — `pip install requests`;
- пакет синхронного/асинхронного веб-клиента HTTPX (<https://www.python-httpx.org>) — `pip install httpx`.

Хотя curl (<https://curl.se>) — это самый известный текстовый веб-клиент, я считаю, что HTTPie проще в использовании. Кроме того, по умолчанию он задействует кодирование и декодирование JSON, что лучше подходит для FastAPI. Далее в этой главе вы увидите снимок экрана, содержащий синтаксис командной строки curl, необходимой для доступа к определенной конечной точке.

Станем тенью веб-разработчика-интроверта в примере 3.1 и сохраним этот код в файле `hello.py`.

Пример 3.1. Робкая конечная точка (`hello.py`)

```
from fastapi import FastAPI

app = FastAPI()

@app.get("/hi")
def greet():
    return "Hello? World?"
```

Нужно обратить внимание на следующие моменты.

- `app` — это объект FastAPI верхнего уровня, представляющий все веб-приложение.
- `@app.get("/hi")` — это *декоратор пути*. Он сообщает FastAPI следующее:
 - запрос к URL-адресу `/hi` на этом сервере должен быть направлен на следующую функцию;
 - этот декоратор применяется только к HTTP-глаголу `GET`. Также можно ответить на URL-запросы `/hi`, отправленные другими HTTP-глаголами (`PUT`, `POST` и т. д.), каждый с отдельной функцией.
- `def greet()` представляет собой *функцию пути* — основную точку контакта с HTTP-запросами и ответами. В этом примере у нее нет аргументов, но следующие разделы показывают, что в недрах FastAPI скрывается гораздо больше.

Следующим шагом будет запуск этого веб-приложения на веб-сервере. Сам FastAPI не включает в себя веб-сервер, но рекомендует использовать Uvicorn. Запустить Uvicorn и веб-приложение FastAPI можно двумя способами — извне или изнутри.

Чтобы запустить Uvicorn извне, через командную строку, смотрите пример 3.2.

Пример 3.2. Запуск Uvicorn с помощью командной строки

```
$ uvicorn hello:app --reload
```

Слово `hello` дает ссылку на файл `hello.py`, а слово `app` — это имя переменной FastAPI в этом файле.

Кроме того, вы можете запустить Uvicorn внутри самого приложения, как показано в примере 3.3.

Пример 3.3. Запуск Uvicorn внутри приложения

```
from fastapi import FastAPI

app = FastAPI()

@app.get("/hi")
def greet():
    return "Hello? World?"

if __name__ == "__main__":
    import uvicorn
    uvicorn.run("hello:app", reload=True)
```

В любом случае параметр `reload` указывает Uvicorn перезапустить веб-сервер, если содержимое файла `hello.py` изменится. В этой главе мы будем часто использовать автоматическую перезагрузку.

По умолчанию будет задействоваться порт 8000 вашей машины под названием `localhost`. И у внешнего, и у внутреннего методов есть аргументы `host` и `port`, но, возможно, вы предпочитаете что-то другое.

Теперь у сервера есть единственная конечная точка (`/hi`), и он готов к приему запросов. Протестируем его с помощью нескольких веб-клиентов.

- В браузере введите URL-адрес в строку сверху окна.
- Для текстового веб-клиента HTTPie введите показанную ниже, в примере 3.7, команду (символ `$` означает командную строку, используемую в вашей системной оболочке).
- Для запросов или HTTPX применяйте Python в интерактивном режиме и набирайте текст после `>>>`.

Как написано в предисловии, то, что вы вводите, выделено **полужирным моноширинным шрифтом**, а вывод представлен в формате обычного моноширинного шрифта.

В примерах 3.4–3.7 показаны различные способы тестирования новой конечной точки /hi веб-сервера.

Пример 3.4. Проверка /hi в браузере

```
http://localhost:8000/hi
```

Пример 3.5. Проверка /hi с помощью Requests

```
>>> import requests
>>> r = requests.get("http://localhost:8000/hi")
>>> r.json()
'Hello? World?'
```

Пример 3.6. Проверка /hi с помощью HTTPX, практически идентичная работе с Requests

```
>>> import httpx
>>> r = httpx.get("http://localhost:8000/hi")
>>> r.json()
'Hello? World?'
```



Неважно, используете ли вы Requests или HTTPX для тестирования маршрутов FastAPI. Но в главе 13 показаны случаи, когда HTTPX полезен при выполнении других асинхронных вызовов. Поэтому в остальных примерах в этой главе задействуются именно Requests.

Пример 3.7. Проверка /hi с помощью HTTPie

```
$ http localhost:8000/hi
HTTP/1.1 200 OK
content-length: 15
content-type: application/json
date: Thu, 30 Jun 2022 07:38:27 GMT
server: uvicorn

"Hello? World?"
```

В примере 3.8 используйте аргумент `-b`, чтобы пропустить заголовки ответа и вывести только тело запроса.

Пример 3.8. Проверка `/hi` с помощью HTTPie с выводом только тела ответа

```
$ http -b localhost:8000/hi
"Hello? World?"
```

Пример 3.9 позволяет получить полные заголовки запроса, а также ответ с помощью аргумента `-v`.

Пример 3.9. Проверка `/hi` с помощью HTTPie с получением всех данных

```
$ http -v localhost:8000/hi
GET /hi HTTP/1.1
Accept: /
Accept-Encoding: gzip, deflate
Connection: keep-alive
Host: localhost:8000
User-Agent: HTTPie/3.2.1

HTTP/1.1 200 OK
content-length: 15
content-type: application/json
date: Thu, 30 Jun 2022 08:05:06 GMT
server: uvicorn

"Hello? World?"
```

Одни примеры, приводимые в книге, показывают стандартный вывод HTTPie (заголовки и тело ответа), а другие — только тело.

HTTP-запросы

Пример 3.9 включает только один конкретный запрос: GET на URL `/hi` на сервер `localhost`, порт 8000.

Веб-запросы «бегают» по разным частям HTTP-запроса, а FastAPI позволяет получить к ним беспрепятственный доступ. В примере 3.10 показан HTTP-запрос из образца запроса в примере 3.9, отправленный командой `http` на веб-сервер.

Пример 3.10. HTTP-запрос

```
GET /hi HTTP/1.1
Accept: /
Accept-Encoding: gzip, deflate
```

```
Connection: keep-alive
Host: localhost:8000
User-Agent: HTTPie/3.2.1
```

Этот запрос содержит:

- глагол-оператор (GET) и путь (/hi);
- все *параметры запроса* (текст после любого символа ?, в данном случае отсутствует);
- другие HTTP-заголовки;
- содержимое тела запроса (отсутствует).

FastAPI разложит их по удобным определениям:

- **Header** — HTTP-заголовки;
- **Path** — URL-адрес;
- **Query** — параметры запроса (после символа ? в конце URL);
- **Body** — тело HTTP-сообщения.



То, как FastAPI предоставляет данные из различных частей HTTP-запросов, — одна из его лучших особенностей и улучшение по сравнению с тем, как это делают большинство веб-фреймворков Python. Все необходимые аргументы можно объявить и предоставить непосредственно внутри функции пути, используя определения из предыдущего списка (Path, Query и т. д.), а также с помощью написанных вами функций. Для этого применяется техника, называемая внедрением зависимостей. Ее мы рассмотрим по ходу повествования и расширим описание в главе 6.

Сделаем предыдущее приложение более личным, добавив параметр **who**, адресуя важный вопрос «Hello?» кому-то. Попробуем разные способы передачи этого нового параметра:

- в *пути* URL;
- в качестве *параметра запроса* после символа ? в URL;
- в *теле* HTTP-сообщения;
- в HTTP-заголовке.

Путь URL

Отредактируйте файл `hello.py` в примере 3.11.

Пример 3.11. Возврат пути к приветствию

```
from fastapi import FastAPI

app = FastAPI()

@app.get("/hi/{who}")
def greet(who):
    return f"Hello? {who}?"
```

Как только вы сохраните изменения в редакторе, Uvicorn должен перезапуститься. (В противном случае нам пришлось бы создавать файл `hello2.py` и так далее и каждый раз заново запускать Uvicorn.) Если вы допустили опечатку, продолжайте пытаться вводить код, пока не исправите ее, и Uvicorn не доставит вам хлопот.

Добавление слова `{who}` в URL-адрес (после выражения `@app.get`) приказывает FastAPI извлечь переменную под названием `who` в указанном местоположении в URL. Затем FastAPI присваивает ее аргументу `who` в следующей функции `greet()`. Это показывает координацию между декоратором пути и функцией пути.



Не следует здесь использовать f-строку Python для измененной строки URL (`"/hi/{who}"`). Фигурные скобки применяются самим FastAPI для сопоставления частей URL в качестве параметров пути.

В примерах 3.12–3.14 проверьте эту доработанную конечную точку с помощью различных методов, рассмотренных ранее.

Пример 3.12. Проверка `/hi/Mom` в браузере

```
localhost:8000/hi/Mom
```

Пример 3.13. Проверка `/hi/Mom` с помощью HTTPie

```
$ http localhost:8000/hi/Mom
HTTP/1.1 200 OK
content-length: 13
```

```
content-type: application/json
date: Thu, 30 Jun 2022 08:09:02 GMT
server: uvicorn
```

```
"Hello? Mom?"
```

Пример 3.14. Проверка /hi/Mom с помощью Requests

```
>>> import requests
>>> r = requests.get("http://localhost:8000/hi/Mom")
>>> r.json()
'Hello? Mom?'
```

Во всех случаях отправляемая как часть URL строка "Mom" передается в функцию пути `greet()` как переменная `who` и возвращается как часть ответа. Каждый раз ответом будет строка JSON "Hello? Mom?" (с одинарными или двойными кавычками в зависимости от того, какой тестовый клиент вы использовали).

Параметры запроса

Параметры запроса — это строки `name=value` после символа `?` в URL-адресе, разделенные символами `&`. Отредактируйте файл `hello.py` в примере 3.15.

Пример 3.15. Возврат параметра запроса приветствия

```
from fastapi import FastAPI

app = FastAPI()

@app.get("/hi")
def greet(who):
    return f"Hello? {who}?"
```

Функция конечной точки снова определяется как `greet(who)`, но выражение `{who}` на этот раз отсутствует в URL-адресе в предыдущей строке декоратора, поэтому FastAPI предполагает, что слово `who` — это параметр запроса. Проверьте код в примерах 3.16 и 3.17.

Пример 3.16. Проверка примера 3.15 с помощью браузера

```
localhost:8000/hi?who=Mom
```

Пример 3.17. Проверка примера 3.15 с помощью HTTPie

```
$ http -b localhost:8000/hi?who=Mom
"Hello? Mom?"
```

В примере 3.18 можно вызвать HTTPie с аргументом параметра запроса (обратите внимание на оператор `==`).

Пример 3.18. Проверка примера 3.15 с помощью HTTPie и параметров

```
$ http -b localhost:8000/hi who==Mom
"Hello? Mom?"
```

У вас может быть несколько таких аргументов для HTTPie, и их удобнее вводить через пробел.

В примерах 3.19 и 3.20 показаны те же альтернативы для веб-клиента Requests.

Пример 3.19. Проверка примера 3.15 с помощью Requests

```
>>> import requests
>>> r = requests.get("http://localhost:8000/hi?who=Mom")
>>> r.json()
'Hello? Mom?'
```

Пример 3.20. Проверка примера 3.15 с помощью Requests и параметров

```
>>> import requests
>>> params = {"who": "Mom"}
>>> r = requests.get("http://localhost:8000/hi", params=params)
>>> r.json()
'Hello? Mom?'
```

Во всех случаях вы предоставляете строку "Mom" новым способом, передаете ее в функцию пути и доводите до конечного ответа.

Тело запроса

Можно предоставить конечной точке GET путь или параметры запроса, но не значения из тела запроса. В HTTP запрос GET должен быть идемпотентным. *Идемпотентность* — вычислительный термин, означающий «задай один и тот же вопрос — получи один и тот же ответ». HTTP-запрос GET должен только выполнять возврат данных. Тело запроса используется для отправки данных на сервер при создании (POST) или обновлении (PUT или PATCH). В главе 9 показан способ обойти эту проблему.

Итак, в примере 3.21 изменим конечную точку с GET на POST. (Технически мы ничего не создаем, так что метод POST не является кошерным, но если владыки RESTful подадут на нас в суд, то оцените крутое здание суда.)

Пример 3.21. Возврат тела приветствия

```
from fastapi import FastAPI, Body

app = FastAPI()

@app.post("/hi")
def greet(who:str = Body(embed=True)):
    return f"Hello? {who}?"
```



Выражение `Body(embed=True)` требуется для того, чтобы сообщить FastAPI, что на этот раз мы получаем значение `who` из тела запроса в формате JSON. Часть выражения `embed` в скобках означает, что ответ должен выглядеть как `{"who": "Mom"}`, а не просто `"Mom"`.

В примере 3.22 попробуйте протестировать HTTPie, используя аргумент `-v` для отображения сгенерированного тела запроса (обратите внимание на единственный параметр `=` для указания данных тела в формате JSON).

Пример 3.22. Проверка примера 3.21 с помощью HTTPie

```
$ http -v localhost:8000/hi who=Mom
POST /hi HTTP/1.1
Accept: application/json, /;q=0.5
Accept-Encoding: gzip, deflate
Connection: keep-alive
Content-Length: 14
Content-Type: application/json
Host: localhost:8000
User-Agent: HTTPie/3.2.1
{
  "who": "Mom"
}

HTTP/1.1 200 OK
content-length: 13
content-type: application/json
date: Thu, 30 Jun 2022 08:37:00 GMT
server: uvicorn

"Hello? Mom?"
```

И наконец, проверьте пример 3.23 с помощью Requests, использующего свой аргумент `json` для передачи закодированных в формате JSON данных в теле запроса.

Пример 3.23. Проверка примера 3.21 с помощью Requests

```
>>> import requests
>>> r = requests.post("http://localhost:8000/hi", json={"who": "Mom"})
>>> r.json()
'Hello? Mom?'
```

HTTP-заголовок

Наконец, попробуем передать аргумент приветствия в качестве HTTP-заголовка в примере 3.24.

Пример 3.24. Возврат заголовка приветствия

```
from fastapi import FastAPI, Header

app = FastAPI()

@app.post("/hi")
def greet(who:str = Header()):
    return f"Hello? {who}?"
```

Проверим это с помощью HTTPie в примере 3.25. Для определения HTTP-заголовка используется выражение `name:value`.

Пример 3.25. Проверка примера 3.24 с помощью HTTPie

```
$ http -v localhost:8000/hi who:Mom
GET /hi HTTP/1.1
Accept: */*
Accept-Encoding: gzip, deflate
Connection: keep-alive
Host: localhost:8000
User-Agent: HTTPie/3.2.1
who: Mom

HTTP/1.1 200 OK
content-length: 13
content-type: application/json
date: Mon, 16 Jan 2023 05:14:46 GMT
server: uvicorn

"Hello? Mom?"
```

FastAPI переводит ключи HTTP-заголовков в нижний регистр и преобразует дефис (-) в нижнее подчеркивание (_). Поэтому вы можете вывести значение заголовка HTTP User-Agent, как показано в примерах 3.26 и 3.27.

Пример 3.26. Возврат заголовка User-Agent (hello.py)

```
from fastapi import FastAPI, Header

app = FastAPI()

@app.post("/agent")
def get_agent(user_agent:str = Header()):
    return user_agent
```

Пример 3.27. Возврат заголовка User-Agent с помощью HTTPie

```
$ http -v localhost:8000/agent
GET /agent HTTP/1.1
Accept: */*
Accept-Encoding: gzip, deflate
Connection: keep-alive
Host: localhost:8000
User-Agent: HTTPie/3.2.1

HTTP/1.1 200 OK
content-length: 14
content-type: application/json
date: Mon, 16 Jan 2023 05:21:35 GMT
server: uvicorn

"HTTPie/3.2.1"
```

Данные по нескольким запросам

В одной функции пути можно использовать более одного из этих методов. То есть вы можете получать данные из URL, параметров запроса, тела HTTP, HTTP-заголовков, cookie-файлов и т. д. Можете написать собственные функции зависимости, которые будут обрабатывать и объединять их особым образом, например, для пагинации или аутентификации. Некоторые из них вы увидите в главе 6 и различных главах части III.

Какой метод лучше?

Вот несколько рекомендаций, которые могут помочь сделать правильный выбор.

- При передаче аргументов в URL стандартной практикой стало следование рекомендациям RESTful.

- Строки запросов обычно применяются для предоставления дополнительных аргументов, таких как пагинация.
- Тело запроса обычно используется для больших объемов вводимых данных, например целых или частичных моделей.

Во всех случаях, если вы предоставите подсказки типов в определениях данных, ваши аргументы будут автоматически проверены библиотекой Pydantic. Это гарантирует их наличие и правильность.

HTTP-ответы

По умолчанию FastAPI преобразует все, что вы возвращаете из своей функции конечной точки, в формат JSON. HTTP-ответ содержит строку заголовка `Content-type: application/json`. Поэтому, несмотря на то что функция `greet()` первоначально возвращает строку `"Hello? World?"`, FastAPI преобразует ее в формат JSON. Это одно из значений по умолчанию, выбранных FastAPI для упрощения разработки API.

В этом случае строка Python `"Hello? World?"` будет преобразована в свой эквивалент строки в формате JSON `"Hello? World?"`, который представляет собой ту же самую строку. Но все, что вы возвращаете, преобразуется в формат JSON, будь то встроенные типы Python или модели Pydantic.

Код состояния

По умолчанию FastAPI возвращает код состояния `200`. Исключения вызывают коды группы `4xx`.

В декораторе пути необходимо указать возвращаемый в случае успеха код состояния HTTP (исключения будут генерировать собственные коды и перепределять это значение). Добавьте код из примера 3.28 куда-нибудь в свой файл `hello.py` (чтобы не показывать весь файл снова и снова) и проверьте его в примере 3.29.

Пример 3.28. Указание кода состояния HTTP (добавьте в файл `hello.py`)

```
@app.get("/happy")
def happy(status_code=200):
    return ":")
```

Пример 3.29. Указание кода состояния HTTP

```
$ http localhost:8000/happy
HTTP/1.1 200 OK
content-length: 4
content-type: application/json
date: Sun, 05 Feb 2023 04:37:32 GMT
server: uvicorn

":)"
```

Заголовки

Можно вводить заголовки HTTP-ответов, как в примере 3.30 (вам не нужно возвращать сообщения `response`).

Пример 3.30. Установка HTTP-заголовков (добавьте в файл `hello.py`)

```
from fastapi import Response

@app.get("/header/{name}/{value}")
def header(name: str, value: str, response: Response):
    response.headers[name] = value
    return "normal body"
```

Посмотрим, получилось ли (пример 3.31).

Пример 3.31. Проверка HTTP-заголовков ответа

```
$ http localhost:8000/header/marco/polo
HTTP/1.1 200 OK
content-length: 13
content-type: application/json
date: Wed, 31 May 2023 17:47:38 GMT
marco: polo
server: uvicorn

"normal body"
```

Типы ответов

Типы ответов (импортируйте эти классы из модуля `fastapi.responses`) бывают следующие:

- `JSONResponse` (по умолчанию);
- `HTMLResponse`;

- `PlainTextResponse`;
- `RedirectResponse`;
- `FileResponse`;
- `StreamingResponse`.

О двух последних я расскажу подробнее в главе 15.

Для других форматов вывода, известных также как *MIME-типы* или *медиа типы*, можно использовать общий класс `Response`, требующий следующие сущности:

- `content` — строка или байт;
- `media_type` — строка MIME-типа;
- `status_code` — целочисленный код состояния HTTP;
- `headers` — словарь (`dict`) строк.

Преобразование типов

Функция пути может возвращать что угодно, и по умолчанию (используя `JSONResponse`) FastAPI преобразует ее в строку JSON и возвращает с соответствующими заголовками HTTP-ответа `Content-Length` и `Content-Type`. Сюда входит любой класс модели Pydantic.

Но как это происходит? Если вы пользовались библиотекой Python JSON, то наверняка видели, что она вызывает исключение при предоставлении некоторых типов данных, таких как `datetime`. FastAPI задействует встроенную функцию `jsonable_encoder()` для преобразования любой структуры данных в JSON-подобную структуру данных Python, а затем вызывает обычную функцию `json.dumps()` для превращения этой структуры в JSON-строку. В примере 3.32 показан тест, выполняемый с помощью фреймворка `pytest`.

Пример 3.32. Используйте функцию `jsonable_encoder()`, чтобы избежать казусов в JSON

```
import datetime
import pytest
from fastapi.encoders import jsonable_encoder
import json
```

```
@pytest.fixture
def data():
    return datetime.datetime.now()
```

```
def test_json_dump(data):
    with pytest.raises(Exception):
        _ = json.dumps(data)

def test_encoder(data):
    out = jsonable_encoder(data)
    assert out
    json_out = json.dumps(out)
    assert json_out
```

Типы моделей и `response_model`

В программе могут быть разные классы с одинаковыми полями, но один из них будет специализирован для ввода данных пользователем, другой — для вывода, а третий — для внутреннего применения. Причины возникновения таких вариантов могут быть следующими.

- Удаление из выходных данных некоторой конфиденциальной информации — например, деидентификация личных медицинских данных, если вы столкнулись с требованиями закона о мобильности и подотчетности медицинского страхования (Health Insurance Portability and Accountability Act, HIPAA).
- Добавление поля для ввода пользователем, например даты и времени создания.

В примере 3.33 показаны три связанных класса для нестандартного случая.

- `TagIn` — это класс, определяющий, что должен предоставить пользователь (в данном случае просто строку под названием `tag`).
- `Tag` создается из класса `TagIn` и добавляет два поля: `created` (когда объект `Tag` был создан) и `secret` (внутренняя строка, которая может храниться в базе данных, но никогда не должна оказаться в широком доступе).
- `TagOut` — это класс, определяющий, что может быть возвращено пользователю (конечная точка определенного или неопределенного поиска). Он содержит поле `tag` из исходного объекта `TagIn` и производный от него объект `Tag`, а также поле `created`, созданное для объекта `Tag`, но не содержит поля `secret`.

Пример 3.33. Варианты моделей (`model/tag.py`)

```
from datetime import datetime
from pydantic import BaseClass
```

```
class TagIn(BaseClass):
    tag: str

class Tag(BaseClass):
    tag: str
    created: datetime
    secret: str

class TagOut(BaseClass):
    tag: str
    created: datetime
```

Из функции пути FastAPI можно возвращать типы данных, отличные от стандартного JSON, разными способами. Один из них — использовать аргумент `response_model` в декораторе пути, чтобы указать FastAPI вернуть что-то другое. FastAPI отбросит все поля возвращаемого объекта, не указанные в определенном аргументом `response_model` объекте.

Представьте, что в примере 3.34 вы написали новый сервисный модуль `service/tag.py` с функциями `create()` и `get()`, которые дают этому веб-модулю возможность вызывать что-то. Эти детали нижнего уровня здесь не имеют значения. Важными моментами являются функция пути `get_one()` в нижней части и выражение `response_model=TagOut` в декораторе пути. Это автоматически заменяет внутренний объект `Tag` очищенным объектом `TagOut`.

Пример 3.34. Возврат другого типа ответа с помощью аргумента `response_model` (`web/tag.py`)

```
import datetime
from model.tag import TagIn, Tag, TagOut
import service.tag as service

@app.post('/')
def create(tag_in: TagIn) -> TagIn:
    tag: Tag = Tag(tag=tag_in.tag, created=datetime.utcnow(),
                   secret="shhhh")
    service.create(tag)
    return tag_in

@app.get('/{tag_str}', response_model=TagOut)
def get_one(tag_str: str) -> TagOut:
    tag: Tag = service.get(tag_str)
    return tag
```

Несмотря на то что мы вернули объект `Tag`, `response_model` преобразует его в `TagOut`.

Автоматизированная документация

Здесь предполагается, что вы используете веб-приложение из примера 3.21 — версию, отправляющую параметр `who` в тело HTTP-запроса с помощью запроса POST к `http://localhost:8000/hi`.

Необходимо убедить браузер посетить URL-адрес **`http://localhost:8000/docs`**.

Вы увидите что-то похожее на рис. 3.1 (я обрезал следующие снимки экрана, чтобы подчеркнуть определенные области).

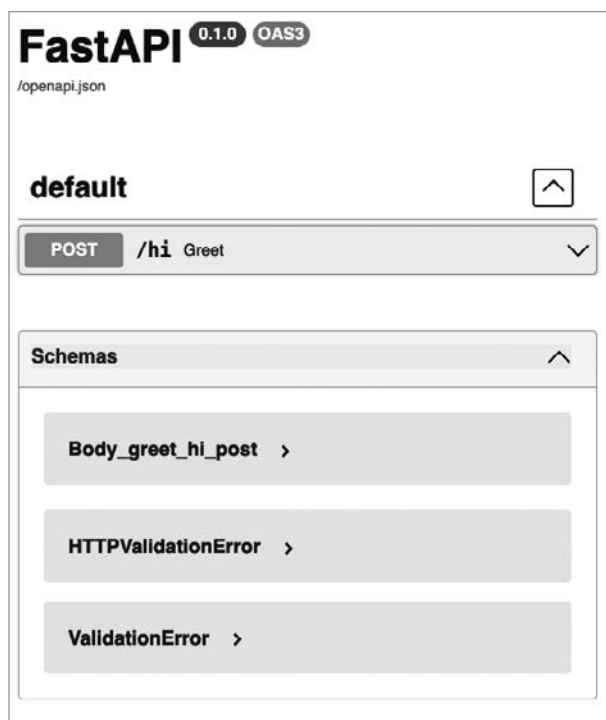


Рис. 3.1. Сгенерированная страница документации

Откуда это взялось?

FastAPI генерирует спецификацию OpenAPI из вашего кода и включает эту страницу для отображения *и тестирования* всех ваших конечных точек. Это лишь один из ингредиентов «секретного соуса».

Нажмите стрелку вниз в правой части зеленого поля, чтобы открыть его для тестирования (рис. 3.2).

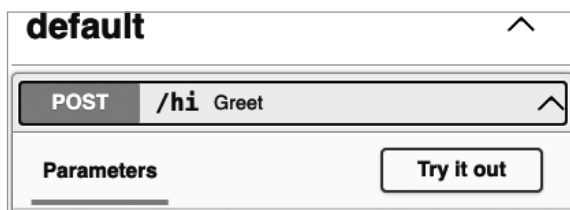


Рис. 3.2. Открытие страницы документации

Нажмите кнопку Try it out (Попробовать) справа. Теперь вы увидите область, которая позволит ввести значение в разделе тела запроса (рис. 3.3).

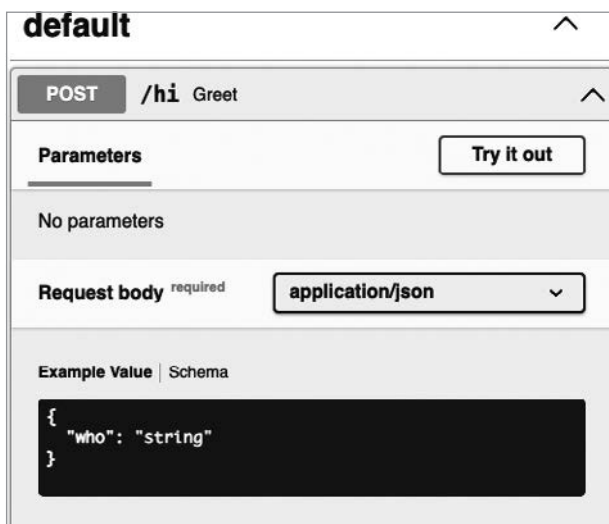
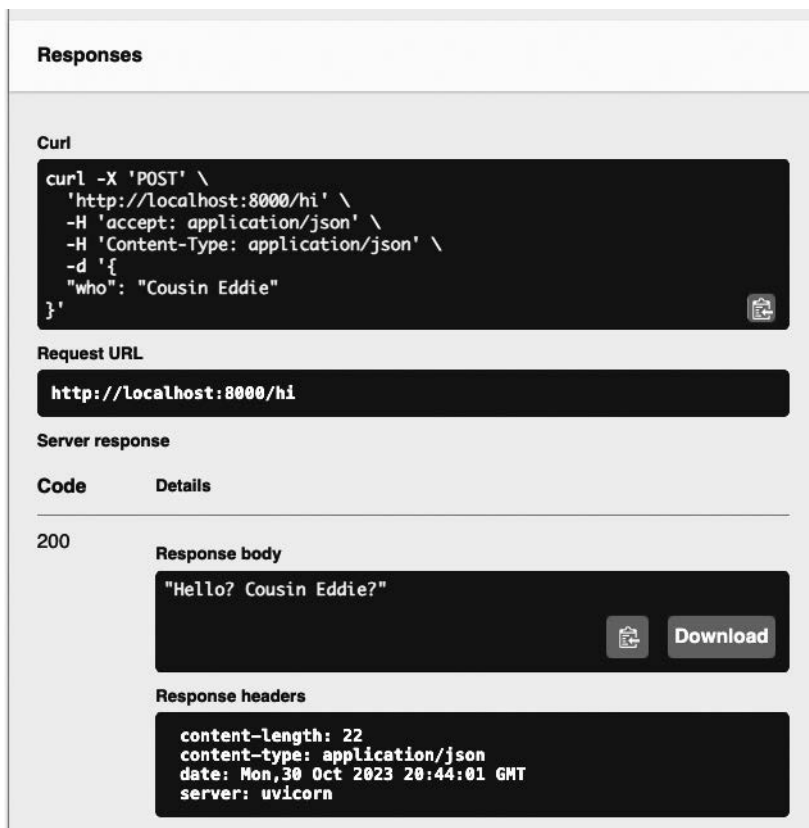


Рис. 3.3. Страница ввода данных

Щелкните левой кнопкой мыши на надписи "string". Измените ее на "Cousin Eddie" (Кузен Эдди) (она должна быть заключена в двойные кавычки). Затем нажмите нижнюю синюю кнопку Execute (Выполнить).

Теперь посмотрите на раздел Responses (Ответы) под кнопкой Execute (Выполнить) (рис. 3.4).



Responses

Curl

```
curl -X 'POST' \  
'http://localhost:8000/hi' \  
-H 'accept: application/json' \  
-H 'Content-Type: application/json' \  
-d '{  
  "who": "Cousin Eddie"  
}'
```

Request URL

```
http://localhost:8000/hi
```

Server response


Code	Details
200	<p>Response body</p> <pre>"Hello? Cousin Eddie?"</pre> <p> Download</p> <p>Response headers</p> <pre>content-length: 22 content-type: application/json date: Mon, 30 Oct 2023 20:44:01 GMT server: uvicorn</pre>

Рис. 3.4. Страница ответа

В поле Response body (Тело ответа) указано, что кузен Эдди объявился.

Этот процесс представляет собой еще один способ протестировать сайт (помимо предыдущих примеров с использованием браузера, HTTPie и Requests).

Кстати, в поле Curl в окне Responses (Ответы) видно, что применение инструмента curl для тестирования командной строки вместо HTTPie потребовало бы больше ввода. Здесь поможет автоматическое кодирование JSON в HTTPie.



Эта автоматизированная документация на самом деле представляет собой так называемую большую пушистую сделку. Когда ваш веб-сервис разрастется до сотен конечных точек, полезно иметь постоянно обновляемую страницу документации и тестирования.

Комплексные данные

В этих примерах показано, как передать конечной точке только одну строку. У многих конечных точек, особенно `GET` или `DELETE`, может быть несколько простых аргументов, таких как строки и числа, или не быть их вовсе. Но при создании (`POST`) или изменении (`PUT` или `PATCH`) ресурса нам обычно требуются более сложные структуры данных. В главе 5 показано, как FastAPI использует библиотеку `Pydantic` и модели данных для их чистой реализации.

Заключение

В этой главе мы задействовали FastAPI для создания веб-сайта с одной конечной точкой. Протестировали ее с помощью нескольких веб-клиентов: браузера, текстовой программы `HTTPie`, пакета `Requests Python` и пакета `HTTPX Python`. Начиная с простого вызова `GET`, аргументы запроса передавались на сервер через путь `URL`, параметр запроса и `HTTP`-заголовок. Затем тело `HTTP`-запроса применялось для отправки данных в конечную точку `POST`. Позже было показано, как возвращать различные типы `HTTP`-ответов. Наконец, автоматически сгенерированная страница с формами предоставила четвертому тестовому клиенту как документацию, так и действующие формы.

Этот обзор FastAPI будет расширен в главе 8.

ГЛАВА 4

Асинхронность, конкурентность и обзор библиотеки Starlette

Starlette — это легкий ASGI-фреймворк/инструментарий, он идеально подходит для создания асинхронных веб-сервисов на Python.

Том Кристи, создатель Starlette

Обзор

В предыдущей главе был приведен краткий обзор того, с чем может столкнуться разработчик при написании нового приложения FastAPI. Эта глава посвящена базовой для FastAPI библиотеке Starlette. В частности, мы рассмотрим возможность *асинхронной* обработки с ее помощью. После обзора различных способов делать больше дел одновременно в Python вы узнаете, как ключевые слова `async` и `await` были включены в Starlette и FastAPI.

Библиотека Starlette

Большая часть веб-кода FastAPI основана на созданном Томом Кристи пакете Starlette (<https://www.starlette.io>). Его можно применять в качестве самостоятельного веб-фреймворка или как библиотеку для других фреймворков, например

FastAPI. Как и любой другой веб-фреймворк, Starlette выполняет все обычные операции синтаксического анализа HTTP-запросов и генерации ответов. Он аналогичен лежащему в основе Flask пакету Werkzeug (<https://werkzeug.palletsprojects.com>).

Самая важная его особенность заключается в поддержке современного асинхронного веб-стандарта Python — ASGI (<https://asgi.readthedocs.io>). До сих пор большинство веб-фреймворков Python, например Flask и Django, основывались на традиционном синхронном стандарте WSGI (<https://wsgi.readthedocs.io>). ASGI позволяет избежать характерных для приложений на базе WSGI блокировок и напряженного ожидания. Проблемы такого типа связаны с частым подключением веб-приложений к гораздо более медленному коду, например, для доступа к базам данных, файлам и сетям. В результате Starlette и использующие его фреймворки стали самыми быстрыми веб-пакетами Python и составили конкуренцию даже приложениям на Go и Node.js.

Типы конкурентности

Прежде чем перейти к подробностям поддержки *асинхронности*, предоставляемой Starlette и FastAPI, полезно узнать, какими способами можно реализовать *конкурентность*.

При *параллельных* вычислениях задача распределяется между несколькими выделенными центральными процессорами (ЦП). Этот метод часто используется в приложениях для выполнения расчетов, таких как задачи обработки графики и машинное обучение.

При *конкурентных* вычислениях каждый ЦП переключается между несколькими задачами. Некоторые задачи из потока занимают больше времени, и необходимо сократить общее время выполнения. Считывание файла или доступ к удаленному сетевому сервису буквально в тысячи и миллионы раз медленнее, чем выполнение вычислений в ЦП.

Веб-приложения выполняют большую часть этой медленной работы. Как заставить их или любые другие серверы работать быстрее? В этом разделе рассматриваются некоторые возможности, начиная с общесистемных и заканчивая целевой темой этой главы — реализацией конструкций `async` и `await` в FastAPI для Python.

Распределенные и параллельные вычисления

Если у вас действительно большое приложение — такое, что на одном процессоре оно будет работать с трудом, — можете разбить его на части и указать им работать на отдельных процессорах на одной или нескольких машинах. Это можно реализовать множеством способов, и если у вас есть такое приложение, то вы уже знаете некоторые из них. Управление всеми этими частями сложнее и дороже, чем управление одним сервером.

Здесь основное внимание будем уделять приложениям малого и среднего размера, размещаемым на одной машине. В этих приложениях может быть смешан синхронный и асинхронный код, и им прекрасно можно управлять с помощью FastAPI.

Процессы в операционной системе

Операционная система (или ОС, потому что печатать длинные слова утомительно) планирует использование ресурсов: памяти, процессоров, устройств, сетей и т. д. Каждая запущенная программа выполняет свой код в одном или нескольких *процессах*. ОС предоставляет каждому из них управляемый, защищенный доступ к ресурсам, включая время работы ЦП.

Большинство систем применяют *вытесняющее* планирование процессов, не позволяя ни одному процессу занимать процессор, память или любой другой ресурс. ОС постоянно приостанавливает и возобновляет процессы в соответствии со своим системным дизайном и настройками.

Хорошая новость для разработчиков: это не ваша проблема! Но плохая новость (обычно затмевающая хорошую) такова: вы не сможете ничего сделать, чтобы изменить это, даже если захотите.

Для приложений Python, требовательных к производительности процессора, обычное решение заключается в использовании нескольких процессов, которые передаются под управление ОС. В Python для этого существует многопроцессорный модуль (<https://oreil.ly/YO4YE>).

Потоки в операционной системе

Вы также можете запускать *потоки* управления в рамках одного процесса. Для выполнения таких задач в Python есть пакет работы с потоками (<https://oreil.ly/xwVB1>).

Применять потоки рекомендуется, если ваша программа связана с операциями ввода-вывода. А использовать множество процессов — в том случае, когда выполнение программы ограничено средой процессора. Но потоки сложны в программировании и могут вызывать трудные для обнаружения ошибки. В книге *Introducing Python* я сравнивал потоки с призраками, которые бродят по дому с привидениями, — они свободные и невидимые, обнаружить их можно только по их воздействию. Ой, кто передвинул эту свечу?

Традиционно в Python библиотеки на основе процессов и библиотеки на основе потоков были разделены. Разработчикам требовалось изучать все тонкости и нюансы, чтобы использовать их. Более современный пакет под названием `concurrent.futures` (<https://oreil.ly/dT150>) представляет собой интерфейс более высокого уровня, упрощающий их применение.

Вскоре вы узнаете, что преимущества потоков легче получить с помощью новых асинхронных функций. FastAPI также управляет потоками для обычных синхронных функций (`def`, а не `async def`) с помощью пулов потоков.

Зеленые потоки

Более загадочный механизм представлен такими *зелеными потоками*, как `greenlet` (<https://greenlet.readthedocs.io>), `gevent` (<http://www.gevent.org>) и `Eventlet` (<https://eventlet.net>). Они являются *кооперативными* (невывесными). Зеленые потоки похожи на потоки ОС, но выполняются в пользовательском пространстве (то есть в вашей программе), а не в ядре ОС. Они работают путем применения к стандартным функциям Python подхода *monkey-patching* (модификации стандартных функций Python в процессе их выполнения), чтобы параллельный код выглядел как обычный последовательный код, — они отдают управление, когда блокируют ожидание ввода-вывода.

Потоки ОС легче (используют меньше памяти), чем процессы ОС, а зеленые потоки легче, чем потоки ОС. В некоторых бенчмарках (<https://oreil.ly/1NFYb>) все асинхронные методы в целом оказались быстрее своих синхронных аналогов.



После прочтения этой главы вы можете задать вопрос: какая библиотека лучше — `gevent` или `asyncio`? Я не думаю, что существует единое мнение относительно предпочтительности для всех видов использования. Зеленые потоки были реализованы ранее с помощью идей из многопользовательской игры Eve Online. В этой книге рассказывается о стандарте Python `asyncio`, применяемом в FastAPI. Он проще, чем потоки, и дает хорошую производительность.

Обратные вызовы

Разработчики интерактивных приложений, таких как игры и графические пользовательские интерфейсы, наверняка знакомы с *обратными вызовами*. Вы пишете функции и привязываете их к какому-либо событию, например к щелчку кнопкой мыши, нажатию клавиши или времени. Выдающимся пакетом Python в этой категории является Twisted (<https://twisted.org>). Его название говорит о том, что программы, основанные на обратных вызовах, немного «вывернуты наизнанку» и трудно следовать их потоку выполнения.

Генераторы Python

Как и большинство языков, Python обычно выполняет код последовательно. Когда вы вызываете функцию, Python запускает ее с первой строки до конца или до ключевого слова `return`.

Но в функции-генераторе Python вы можете останавливаться и возвращаться из любой точки, *а также возвращаться к этой точке позже*. Хитрость заключается в ключевом слове `yield`.

В одном из эпизодов мультсериала «Симпсоны» Гомер врзается на своей машине в статую оленя, после чего следуют три реплики диалога. Пример 4.1 определяет обычную функцию Python для возврата этих строк с помощью ключевого слова `return` в виде списка и их итерации вызывающей стороной.

Пример 4.1. Использование ключевого слова `return`

```
>>> def doh():
...     return ["Homer: D'oh!", "Marge: A deer!", "Lisa: A female deer!"]
...
>>> for line in doh():
...     print(line)
...
Homer: D'oh!
Marge: A deer!
Lisa: A female deer!
```

Этот подход отлично работает, когда списки относительно небольшие. Но что, если мы возьмем все диалоги из всех эпизодов «Симпсонов»? Списки занимают много памяти.

В примере 4.2 показано, как функция-генератор будет выдавать строки.

Пример 4.2. Использование ключевого слова `yield`

```
>>> def doh2():
...     yield "Homer: D'oh!"
...     yield "Marge: A deer!"
...     yield "Lisa: A female deer!"
...
>>> for line in doh2():
...     print(line)
...
Homer: D'oh!
Marge: A deer!
Lisa: A female deer!
```

Вместо итерации по списку, возвращаемому простой функцией `doh()`, мы выполняем итерации по *объекту-генератору*, возвращаемому *функцией-генератором* `doh2()`. Фактическая итерация (`for...in`) выглядит так же. Python возвращает первую строку из генератора `doh2()`, но отслеживает, где она находится, для следующей итерации, и так продолжается, пока функция не исчерпает диалог.

Любая функция, содержащая ключевое слово `yield`, — это функция-генератор. Учитывая эту возможность вернуться в середину функции и возобновить выполнение, следующий раздел выглядит логичной адаптацией.

Ключевые слова `async`, `await` и модуль `asyncio` из Python

Функциональные возможности библиотеки `asyncio` (<https://oreil.ly/cBMAc>) языка Python были представлены в различных выпусках. Вы используете как минимум Python версии 3.7, и здесь термины `async` и `await` стали зарезервированными ключевыми словами.

В следующих примерах показана шутка, смешная только при асинхронном выполнении. Выполните оба примера самостоятельно, потому что тайминг важен. Сначала запустите невеселый пример 4.3.

Пример 4.3. Уныло

```
>>> import time
>>>
>>> def q():
...     print("Why can't programmers tell jokes?")
```

```
...     time.sleep(3)
...
>>> def a():
...     print("Timing!")
...
>>> def main():
...     q()
...     a()
...
>>> main()
Why can't programmers tell jokes?
Timing!
```

Между вопросом и ответом будет трехсекундный промежуток. Скукота.

Но в асинхронном примере 4.4 все немного иначе.

Пример 4.4. Весело

```
>>> import asyncio
>>>
>>> async def q():
...     print("Why can't programmers tell jokes?")
...     await asyncio.sleep(3)
...
>>> async def a():
...     print("Timing!")
...
>>> async def main():
...     await asyncio.gather(q(), a())
...
>>> asyncio.run(main())
Why can't programmers tell jokes?
Timing!
```

На этот раз ответ должен появиться сразу после вопроса, затем наступит трехсекундная тишина — так, как будто это говорит программист. Ха-ха! Гм.



В примере 4.4 я задействовал функции `asyncio.gather()` и `asyncio.run()`, но существует несколько способов вызова асинхронных функций. При использовании FastAPI они вам не понадобятся.

Python при выполнении примера 4.4 думает так.

1. Выполню функцию `q()`. Ну, сейчас это только первая строчка.
2. Ладно, ты, ленивая асинхронная `q()`, я установил таймер и вернусь к тебе через три секунды.

3. А пока я выполняю функцию `a()` и сразу же выведу ответ.
4. Других ключевых слов `await` нет, так что следует вернуться к выполнению функции `q()`.
5. Скучный цикл событий! Я буду сидеть здесь и ждать все эти три секунды.
6. Хорошо, наконец-то я закончил.

В этом примере используется `asyncio.sleep()` для функции, занимающей некоторое время, например для считывания файла или обращения к веб-сайту. Ключевое слово `await` размещается перед функцией, которая может провести большую часть своего времени в ожидании. И в этой функции должно быть ключевое слово `async` до выражения `def`.



Если вы определили функцию с помощью `async def`, ее вызывающая сторона должна поместить слово `await` перед вызовом. Сам вызывающий модуль должен быть объявлен с помощью `async def`, а его вызывающий модуль должен ожидать с помощью слова `await` на протяжении всего времени выполнения.

Кстати, вы можете объявить функцию как `async` (асинхронную), даже если она не содержит `await`-вызова другой асинхронной функции. Это не повредит.

FastAPI и асинхронность

После долгого путешествия по холмам и долам давайте вернемся к FastAPI и к тому, почему все это важно.

Поскольку веб-серверы тратят много времени на ожидание, производительность можно повысить, избежав части этого ожидания — иными словами, с помощью конкурентности. Другие веб-серверы используют многие из упомянутых ранее методов: потоки, `gevent` и т. д. Одна из причин, по которой FastAPI является одним из самых быстрых веб-фреймворков Python, — это включение асинхронного кода благодаря поддержке протокола ASGI в пакете Starlette и некоторым собственным изобретениям.



Использование ключевых слов `async` и `await` само по себе не ускоряет выполнение кода. На самом деле такой код может оказаться немного медленнее из-за накладных расходов на асинхронную настройку. Основное назначение конструкций `async` заключается в том, чтобы избежать длительного ожидания ввода-вывода.

Теперь рассмотрим предыдущие вызовы конечных точек веб-приложения и поговорим о том, как сделать их асинхронными.

Функции, сопоставляющие URL с кодом, в документации FastAPI называются *функциями пути*. Я также называл их *конечными точками веб-приложения*, и вы видели их синхронные примеры в главе 3. Сделаем несколько асинхронных вариантов. Как и в предыдущих примерах, мы будем использовать простые типы, такие как числа и строки. В главе 5 представлены *подсказки типов* и Pydantic — они понадобятся для работы с более сложными структурами данных.

Пример 4.5 возвращает нас к первой программе FastAPI из предыдущей главы и делает ее асинхронной.

Пример 4.5. Робкая асинхронная конечная точка (greet_async.py)

```
from fastapi import FastAPI
import asyncio

app = FastAPI()

@app.get("/hi")
async def greet():
    await asyncio.sleep(1)
    return "Hello? World?"
```

Чтобы запустить этот фрагмент веб-кода, вам нужен веб-сервер, например Uvicorn. Первый способ — запустить Uvicorn в командной строке:

```
$ uvicorn greet_async:app
```

Второй, как в примере 4.6, заключается в вызове Uvicorn изнутри кода примера, когда он запускается как основная программа, а не как модуль.

Пример 4.6. Еще одна робкая асинхронная конечная точка (greet_async_uvicorn.py)

```
from fastapi import FastAPI
import asyncio
import uvicorn

app = FastAPI()

@app.get("/hi")
async def greet():
    await asyncio.sleep(1)
    return "Hello? World?"

if __name__ == "__main__":
    uvicorn.run("greet_async_uvicorn:app")
```

При запуске в качестве самостоятельной программы Python называет ее `main`. Выражение `if __name__...` — это указание Python запустить Uvicorn только при вызове в качестве основной программы. Да, это некрасиво.

Этот код сделает секундную паузу, после чего вернется к своему робкому приветствию. Единственное отличие от синхронной функции с применением стандартной функции `sleep(1)` заключается в том, что в асинхронном примере веб-сервер в это время может обрабатывать другие запросы.

Использование `asyncio.sleep(1)` имитирует реальную функцию, занимающую одну секунду, например вызов базы данных или загрузку веб-страницы. В последующих главах будут приведены примеры таких вызовов с веб-уровня на сервисный уровень, а оттуда на уровень данных, в результате чего время ожидания тратится на реальную работу.

FastAPI сам вызывает асинхронную функцию пути `greet()`, когда получает GET-запрос на URL `/hi`. Вам не нужно добавлять ключевое слово `await` куда-либо. Но для любых других определений функций `async def` вызывающая сторона должна поместить оператор `await` перед каждым вызовом.



FastAPI запускает асинхронный цикл событий, координирующий функции асинхронного пути выполнения, и пул потоков для функций синхронного пути. Разработчику не нужно разбираться в хитроумных деталях, что стало большим плюсом. Например, вам не нужно запускать такие методы, как `asyncio.gather()` или `asyncio.run()`, как в примере 4.4.

Непосредственное использование Starlette

FastAPI не так сильно раскрывает Starlette, как Pydantic. Starlette по большей части представляет собой механизм, который гудит в машинном отделении, обеспечивая бесперебойную работу корабля. Но если вам интересно, можно применять Starlette непосредственно для написания веб-приложения. Пример 3.1 из предыдущей главы может выглядеть как пример 4.7.

Пример 4.7. Использование Starlette: `starlette_hello.py`

```
from starlette.applications import Starlette
from starlette.responses import JSONResponse
from starlette.routing import Route
```

```
async def greeting(request):  
    return JSONResponse('Hello? World?')  
  
app = Starlette(debug=True, routes=[  
    Route('/hi', greeting),  
])
```

Запустите это веб-приложение с помощью команды:

```
$ uvicorn starlette_hello:app
```

На мой взгляд, дополнения FastAPI значительно упрощают разработку веб-интерфейсов.

Немного отвлечемся: уборка в доме из игры Clue

Вы владеете небольшой (очень небольшой, состоящей только из вас) компанией по уборке домов. Заработков вам хватало только на макароны, но только что вы заключили контракт, дающий возможность позволить себе гораздо более качественную пищу.

Ваш клиент купил старинный особняк, построенный в стиле настольной игры Clue, и хочет вскоре устроить там костюмированную вечеринку. Но в доме невероятный беспорядок. Если бы Мариэ Кондо увидела это место, она бы:

- закричала;
- прикрыла рот ладошкой;
- убежала;
- сделала все перечисленное.

Ваш контракт включает в себя бонус за скорость. Как тщательно убрать помещение за минимальное время? Лучше всего было бы получить больше блоков сохранения подсказок (Clue Preservation Units, CPU), но это уже дело ваше.

Поэтому вы можете попробовать один из следующих вариантов.

- Сделать все в одной комнате, затем все в следующей и т. д.
- Выполнить определенное задание в одной комнате, затем в другой и т. д. Например, отполировать серебро на кухне и в столовой или бильярдные шары в бильярдной.

Будет ли различаться общее время, затраченное вами при разных подходах? Может быть. Но, возможно, гораздо важнее рассмотреть вопрос о том, приходится ли вам ждать значительное время для выполнения какого-либо шага. Например, если взглянуть под ноги: после чистки ковров и натирания полов воском они должны сохнуть в течение нескольких часов, прежде чем на них можно будет поставить мебель. Итак, вот ваш план для каждой комнаты.

1. Очистить все статичные части (окна и т. п.).
2. Переместить всю мебель из комнаты в холл.
3. Удалить многолетнюю грязь с ковра и/или деревянного пола.
4. Выполнить любой из этих пунктов:
 - подождать, пока ковер или воск высохнут, и помахать на прощание своему бонусу;
 - перейти в следующую комнату и все повторить. После окончания работы в последней комнате занести мебель в первую комнату и т. д.

Подход «ждать, пока высохнет» — это синхронный подход, и он может быть лучшим, если время не играет роли и вам нужен перерыв. Второй вариант представляет собой асинхронную работу и экономит время ожидания для каждой комнаты.

Предположим, вы выбрали асинхронный путь, потому что так можно больше заработать. Вы заставите старую развалину сверкать и получите бонус от благодарного клиента. Поздняя вечеринка оказывается очень удачной, если не считать некоторых проблем.

1. Один забывчивый гость пришел в образе Марио.
2. Вы натерли воском танцпол в бальном зале, а подвыпивший профессор Плам катался в носках, пока не налетел на стол и не пролил шампанское на мисс Скарлет.

Мораль этой истории такова.

- Требования могут быть противоречивыми и/или странными.
- Оценка времени и усилий может зависеть от многих факторов.
- Последовательность выполнения задач может быть как искусством, так и наукой.
- Вы будете чувствовать себя прекрасно, когда все будет готово. М-м-м, макароны!

Заключение

После обзора способов увеличения конкурентности в этой главе были рассмотрены функции, использующие недавно появившиеся в Python ключевые слова `async` и `await`. Было показано, как FastAPI и Starlette работают и со старыми синхронными функциями, и с новыми асинхронными.

В следующей главе мы познакомимся со второй частью FastAPI — как Pydantic помогает определять данные.

Pydantic, подсказки типов и обзор моделей

Быстрый и расширяемый, Pydantic прекрасно сочетается с вашими линтерами/IDE/brain. Определите, как должны выглядеть данные в чистом, каноническом Python 3.6+. Проверьте их с помощью Pydantic.

Сэмюэл Колвин, разработчик Pydantic

Обзор

FastAPI во многом опирается на пакет Python с названием Pydantic. Для определения структур данных используются *модели* (объектные классы Python). Они широко применяются в приложениях FastAPI и становятся реальным преимуществом при написании больших приложений.

Подсказки типов данных

Пришло время узнать немного больше о подсказках типов в Python.

В главе 2 упоминалось, что во многих компьютерных языках переменная указывает непосредственно на значение в памяти. Это требует от программиста объявления типа значения, чтобы можно было определить его размер и рядность. В Python переменные — это просто имена, связанные с объектами, и именно у объектов есть типы.

В стандартном программировании переменная обычно связана с одним и тем же объектом. Если мы свяжем с этой переменной подсказку типа, то сможем избежать некоторых ошибок в программировании. Поэтому Python добавил подсказки типов к языку, в стандартный модуль типизации. Интерпретатор Python игнорирует синтаксис подсказки типа и выполняет программу так, как будто ее нет. Тогда в чем смысл?

В одной строке вы можете рассматривать переменную как строку, а потом забыть и присвоить ей объект другого типа. Компиляторы других языков будут жаловаться, а Python этого не сделает. Стандартный интерпретатор Python отлавливает обычные синтаксические ошибки и исключения времени выполнения, но не смешивает типы переменных. Инструменты-помощники, такие как *mypy*, обращают внимание на подсказки типов и предупреждают о любых несоответствиях.

Кроме того, подсказки доступны разработчикам Python, которые могут написать инструменты, выполняющие не только проверку ошибок типов. В следующих разделах описывается, как пакет *Pydantic* был разработан для удовлетворения неочевидных потребностей. Позже вы увидите, как его интеграция с FastAPI значительно упрощает решение многих вопросов веб-разработки.

Кстати, как выглядят подсказки? Существует один синтаксис для переменных и другой — для возвращаемых значений функций.

Подсказки типа переменной могут включать только тип:

```
name: type
```

или также инициализировать переменную значением:

```
name: type = value
```

Tuple может быть одним из стандартных простых типов Python, таких как `int` или `str`, или коллекцией, такой как `tuple`, `list` или `dict`:

```
thing: str = "yeti"
```



При использовании Python до версии 3.9 необходимо импортировать прописные версии стандартных имен типов из модуля типизации:

```
from typing import Str  
thing: Str = "yeti"
```


Вот несколько примеров с инициализацией:

```
physics_magic_number: float = 1.0/137.03599913
hp_lovecraft_noun: str = "ichor"
exploding_sheep: tuple = "sis", "boom", "bah!"
responses: dict = {"Marco": "Polo", "answer": 42}
```

Можно также включать подтипы коллекций:

```
name: dict[keytype, valtype] = {key1: val1, key2: val2}
```

Модуль типизации содержит полезные дополнения для подтипов. Наиболее распространенные из них следующие:

- Any — любой тип;
- Union — любой из указанных типов, например Union[str, int].



В Python, начиная с версии 3.10, можно написать `type1|type2`, а не `Union[type1, type2]`.

Примеры определений Pydantic для словарей (dict) в Python включают следующее:

```
from typing import Any
responses: dict[str, Any] = {"Marco": "Polo", "answer": 42}
```

Или, если быть более точными:

```
from typing import Union
responses: dict[str, Union[str, int]] = {"Marco": "Polo", "answer": 42}
```

либо (в Python 3.10 и более поздних версиях):

```
responses: dict[str, str | int] = {"Marco": "Polo", "answer": 42}
```

Обратите внимание на то, что в Python строка переменной с подсказкой типа является верной, а простая строка переменной — нет:

```
$ python
...
>>> thing0
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'thing0' is not defined
>>> thing0: str
```

Кроме того, некорректное использование типов не отлавливается обычным интерпретатором Python:

```
$ python
...
>>> thing1: str = "yeti"
>>> thing1 = 47
```

Но такие ошибки будут обнаружены `myru`. Если у вас еще не установлен этот статический анализатор, наберите команду `pip install mypy`. Сохраните две предыдущие строки в файле `stuff.py`¹, а затем попробуйте выполнить следующие команды:

```
$ mypy stuff.py
stuff.py:2: error: Incompatible types in assignment
(expression has type "int", variable has type "str")
Found 1 error in 1 file (checked 1 source file)
```

В подсказке типа возврата функции вместо двоеточия применяется стрелка:

```
function(args) -> type:
```

Вот пример возврата функции при использовании `Pydantic`:

```
def get_thing() -> str:
    return "yeti"
```

Можно задействовать любой тип, включая определенные классы или их комбинации. Вы увидите это через несколько страниц.

Группировка данных

Зачастую нам нужно сохранить связанную группу переменных, а не передавать множество отдельных переменных. Как объединить несколько переменных в группу и сохранить подсказки типа?

Давайте оставим в прошлом пример с простым приветствием из предыдущих глав и начнем использовать более богатые данные. Как и в остальных частях этой книги, я буду приводить примеры *криптидов* (воображаемых существ) и исследователей (тоже воображаемых), которые их ищут. Начальные определения *криптидов* будут включать в себя только строковые переменные для следующих параметров:

¹ Появились сомнения в наличии у меня воображения при именовании? Хм... нет.

- `name` — ключ;
- `country` — двухсимвольный код страны согласно стандарту ISO (3166-1 alpha 2) или *, что означает «все»;
- `area` (необязательный) — штат США или другое территориальное образование страны;
- `description` — в свободной форме;
- `aka` — обозначает «также известен как...» (also known as...).

А исследователи получают следующие параметры:

- `name` — ключ;
- `country` — двухсимвольный код страны согласно стандарту ISO;
- `description` — в свободной форме.

Исторические структуры группировки данных в Python (помимо базовых `int`, `string` и подобных им) приведены далее:

- `tuple` — неизменяемая последовательность объектов (кортеж);
- `list` — изменяемая последовательность объектов (список);
- `set` — изменяемые отдельные объекты (множество);
- `dict` — пары изменяемых объектов «ключ — значение» (ключ должен быть неизменяемого типа) (словарь).

Кортежи (пример 5.1) и списки (пример 5.2) позволяют обращаться к переменной-члену только по ее смещению, поэтому вам придется запоминать, что куда переместилось.

Пример 5.1. Использование кортежа

```
>>> tuple_thing = ("yeti", "CN", "Himalayas",  
                  "Hirsute Himalayan", "Abominable Snowman")  
>>> print("Name is", tuple_thing[0])  
Name is yeti
```

Пример 5.2. Использование списка

```
>>> list_thing = ["yeti", "CN", "Himalayas",  
                 "Hirsute Himalayan", "Abominable Snowman"]  
>>> print("Name is", list_thing[0])  
Name is yeti
```

Пример 5.3 показывает, что вы можете получить немного больше объяснений, определив имена для целочисленных смещений.

Пример 5.3. Использование кортежей и именованных смещений

```
>>> NAME = 0
>>> COUNTRY = 1
>>> AREA = 2
>>> DESCRIPTION = 3
>>> AKA = 4
>>> tuple_thing = ("yeti", "CN", "Himalayas",
...                 "Hirsute Himalayan", "Abominable Snowman")
>>> print("Name is", tuple_thing[NAME])
Name is yeti
```

В примере 5.4 словари выглядят немного лучше, предоставляя доступ по описательным ключам.

Пример 5.4. Использование словаря

```
>>> dict_thing = {"name": "yeti",
...               "country": "CN",
...               "area": "Himalayas",
...               "description": "Hirsute Himalayan",
...               "aka": "Abominable Snowman"}
>>> print("Name is", dict_thing["name"])
Name is yeti
```

Множества содержат только уникальные значения, поэтому они не очень полезны для кластеризации различных переменных.

В примере 5.5 *именованный кортеж* — это кортеж, предоставляющий вам доступ по целочисленному смещению *или* имени.

Пример 5.5. Использование именованного кортежа

```
>>> from collections import namedtuple
>>> CreatureNamedTuple = namedtuple("CreatureNamedTuple",
...                                 "name, country, area, description, aka")
>>> namedtuple_thing = CreatureNamedTuple("yeti",
...                                       "CN",
...                                       "Himalaya",
...                                       "Hirsute HImalayan",
...                                       "Abominable Snowman")
>>> print("Name is", namedtuple_thing[0])
Name is yeti
>>> print("Name is", namedtuple_thing.name)
Name is yeti
```



Нельзя написать `namedtuple_thing["name"]`. Это будет `tuple`, а не `dict`, поэтому индекс должен быть целым числом.

В примере 5.6 определяется новый класс Python под названием `class` и добавляются все атрибуты с помощью `self`. Но для их определения вам придется набрать много текста.

Пример 5.6. Использование стандартного класса

```
>>> class CreatureClass():
...     def __init__(self,
...         name: str,
...         country: str,
...         area: str,
...         description: str,
...         aka: str):
...         self.name = name
...         self.country = country
...         self.area = area
...         self.description = description
...         self.aka = aka
...
>>> class_thing = CreatureClass(
...     "yeti",
...     "CN",
...     "Himalayas"
...     "Hirsute Himalayan",
...     "Abominable Snowman")
>>> print("Name is", class_thing.name)
Name is yeti
```



Вы можете подумать: что в этом плохого? В обычном классе можно добавить больше данных (атрибутов), но особенно много поведения (методов). В один безумный день вы можете решить добавить метод для поиска любимых песен исследователя. (Это нельзя применить к существам¹.) Но в данном случае речь идет о том, чтобы просто без помех перемещать сборки данных между уровнями и проверять их на входе и выходе. Кроме того, методы — это квадратные детали, которые с трудом помещаются в круглые отверстия базы данных.

Есть ли в Python что-то похожее на то, что в других компьютерных языках называется *записью* (*record*) или *структурой* (*struct*) (группа имен и значений)? Недавно в Python появился *класс для хранения данных* (*dataclass*). В примере 5.7 показано, как все эти `self`-выражения исчезают при использовании классов данных.

¹ За исключением небольшой группы йодлингующих йети (хорошее название для группы).

Пример 5.7. Применение класса данных `dataclass`

```
>>> from dataclasses import dataclass
>>>
>>> @dataclass
... class CreatureDataClass():
...     name: str
...     country: str
...     area: str
...     description: str
...     aka: str
...
>>> dataclass_thing = CreatureDataClass(
...     "yeti",
...     "CN",
...     "Himalayas"
...     "Hirsute Himalayan",
...     "Abominable Snowman")
>>> print("Name is", dataclass_thing.name)
Name is yeti
```

Это очень хорошо для части описания, связанной с сохранением переменных вместе. Но нам требуется больше, так что давайте попросим у Дедушки Мороза вот что:

- объединение возможных альтернативных типов;
- отсутствующие/дополнительные значения;
- значения по умолчанию;
- проверку достоверности данных;
- сериализацию в форматы, такие как JSON, и из них.

Альтернативы

Очень заманчиво использовать встроенные структуры данных Python, особенно словари. Но вы неизбежно обнаружите, что словари слишком свободны. А за свободу приходится платить. Вам нужно будет проверить *абсолютно все*.

- Ключ необязателен?
- Если ключ отсутствует, есть ли значение по умолчанию?
- Существует ли ключ?
- Если да, то относится ли значение ключа к правильному типу?

- Если да, то находится ли значение в нужном диапазоне или соответствует ли оно шаблону?

По крайней мере три решения отвечают хотя бы некоторым из этих требований:

- *Dataclasses* (<https://oreil.ly/mxANA>) — часть стандартного языка Python;
- *attrs* (<https://www.attrs.org>) — сторонний пакет, но содержит супернабор классов данных;
- *Pydantic* (<https://docs.pydantic.dev>) — тоже сторонний продукт, но интегрированный в FastAPI, поэтому его легко выбрать, если вы уже используете FastAPI. И если вы читаете эту книгу, то вполне вероятно, что это именно так.

Удобное сравнение этих трех вариантов можно посмотреть на YouTube (<https://oreil.ly/pkQD3>). Одним из выводов является то, что Pydantic выделяется при проверке, а его интеграция с FastAPI позволяет выявить множество потенциальных ошибок в данных. Другое дело, что Pydantic полагается на наследование (от класса `BaseModel`), а два других используют декораторы Python для определения своих объектов. Это скорее вопрос стиля.

В другом сравнении (<https://oreil.ly/gU28a>) Pydantic превзошел более старые пакеты проверки, такие как *marshmallow* (<https://marshmallow.readthedocs.io>) и библиотека с интригующим названием *Voluptuous*¹ (<https://github.com/alecthomas/voluptuous>). Еще один большой плюс Pydantic в том, что он использует стандартный синтаксис подсказок типов Python — более старые библиотеки не применяли подсказки типов и создавали собственные.

В книге я остановился на Pydantic, но вы можете найти применение любой из альтернатив, если не используете FastAPI.

Pydantic предоставляет возможность задать любую комбинацию следующих проверок:

- обязательные и необязательные;
- значение по умолчанию, если не указано, но требуется;
- ожидаемый тип или типы данных;
- ограничения диапазона значений;
- другие проверки на основе функций, если необходимо;
- сериализацию и десериализацию.

¹ *Voluptuous* (англ.) — «чувственный». — *Примеч. пер.*

Простой пример

Вы уже видели, как передать простую строку в конечную точку веб-приложения через URL, параметр запроса или тело HTTP-запроса. Проблема в том, что обычно вы запрашиваете и получаете группы данных разных типов. Именно здесь в FastAPI впервые появляются модели Pydantic. В начальном примере будут использоваться три файла:

- `model.py` — определяет модель Pydantic;
- `data.py` — источник фиктивных данных, определяющих экземпляр модели;
- `web.py` — определяет конечную точку веб-приложения FastAPI, возвращающую фиктивные данные.

Для простоты в этой главе сохраним все файлы в одном каталоге. В последующих главах, посвященных более крупным веб-сайтам, мы разделим их на соответствующие уровни. Сначала определим *модель* существа в примере 5.8.

Пример 5.8. Определение модели существа: `model.py`

```
from pydantic import BaseModel

class Creature(BaseModel):
    name: str
    country: str
    area: str
    description: str
    aka: str

thing = Creature(
    name="yeti",
    country="CN",
    area="Himalayas",
    description="Hirsute Himalayan",
    aka="Abominable Snowman")
print("Name is", thing.name)
```

Класс `Creature` наследуется от класса `BaseModel` из Pydantic. Часть выражения `: str` после слов `name`, `country`, `area`, `description` и `aka` представляет собой под-сказку типа — каждое из значений относится к строковому типу данных Python.



В этом примере все поля обязательны для заполнения. В Pydantic, если слово `Optional` отсутствует в описании типа, поле должно содержать значение.

В примере 5.9 аргументы передаются в любом порядке, если вы указываете их имена.

Пример 5.9. Создание существа

```
>>> thing = Creature(
...     name="yeti",
...     country="CN",
...     area="Himalayas",
...     description="Hirsute Himalayan",
...     aka="Abominable Snowman")
>>> print("Name is", thing.name)
Name is yeti
```

Пока что в примере 5.10 определен небольшой источник данных. В последующих главах этим будут заниматься базы данных. Подсказка типа `list[Creature]` говорит Python, что это список только объектов `Creature`.

Пример 5.10. Определение фиктивных данных в файле `data.py`

```
from model import Creature

_creatures: list[Creature] = [
    Creature(name="yeti",
              country="CN",
              area="Himalayas",
              description="Hirsute Himalayan",
              aka="Abominable Snowman"),
    Creature(name="sasquatch",
              country="US",
              area="*",
              description="Yeti's Cousin Eddie",
              aka="Bigfoot")
]

def get_creatures() -> list[Creature]:
    return _creatures
```

(Мы использовали символ "*" для аргумента `area` объекта `Bigfoot`, потому что он может жить почти везде.)

Этот код импортирует написанный нами ранее файл `model.py`. Он немного скрывает данные, вызывая свой список объектов `Creature` `_creatures` и предоставляя функцию `get_creatures()` для их возврата.

В примере 5.11 приведен файл `web.py`, определяющий конечную точку веб-приложения FastAPI.

Пример 5.11. Определение конечной точки веб-приложения FastAPI: web.py

```
from model import Creature
from fastapi import FastAPI

app = FastAPI()

@app.get("/creature")
def get_all() -> list[Creature]:
    from data import get_creatures
    return get_creatures()
```

Теперь запустите этот сервер с одной конечной точкой в примере 5.12.

Пример 5.12. Запуск Uvicorn

```
$ uvicorn creature:app
INFO:      Started server process [24782]
INFO:      Waiting for application startup.
INFO:      Application startup complete.
INFO:      Uvicorn running on http://127.0.0.1:8000 (Press CTRL+C to quit)
```

В другом окне примера 5.13 осуществляется доступ к веб-приложению с помощью веб-клиента HTTPie (попробуйте использовать свой браузер или модуль Requests по желанию).

Пример 5.13. Проверка с помощью HTTPie

```
$ http http://localhost:8000/creature
HTTP/1.1 200 OK
content-length: 183
content-type: application/json
date: Mon, 12 Sep 2022 02:21:15 GMT
server: uvicorn
[
  {
    "aka": "Abominable Snowman",
    "area": "Himalayas",
    "country": "CN",
    "name": "yeti",
    "description": "Hirsute Himalayan"
  },
  {
    "aka": "Bigfoot",
    "country": "US",
    "area": "*",
    "name": "sasquatch",
    "description": "Yeti's Cousin Eddie"
  }
]
```

FastAPI и Starlette автоматически преобразуют исходный список объектов модели `Creature` в строку JSON. Это формат вывода по умолчанию в FastAPI, поэтому нам не нужно его указывать.

Кроме того, в окне, в котором вы первоначально запустили веб-сервер Uvicorn, должна быть выведена строка журнала:

```
INFO: 127.0.0.1:52375 - "GET /creature HTTP/1.1" 200 OK
```

Проверка типов

В предыдущем разделе было показано, как сделать следующее:

- применить подсказки типов к переменным и функциям;
- определить и использовать модель Pydantic;
- вернуть список моделей из источника данных;
- вернуть список моделей веб-клиенту, автоматически преобразовав его в JSON.

А теперь действительно применим этот план для проверки данных.

Попробуйте присвоить значение неправильного типа одному или нескольким полям объекта `Creature`. Для этого воспользуйтесь автономным тестом (Pydantic не применяется ни к какому веб-коду, он относится к данным).

В примере 5.14 показано содержимое файла `test1.py`.

Пример 5.14. Проверка модели Creature

```
from model import Creature

dragon = Creature(
    name="dragon",
    description=["incorrect", "string", "list"],
    country="*",
    area="*",
    aka="firedrake")
```

Теперь попробуйте выполнить тест из примера 5.15.

Он показывает, что мы присвоили полю `description` список строк, а ему нужна обычная строка.

Пример 5.15. Продолжение теста

```
$ python test1.py
Traceback (most recent call last):
  File ".../test1.py", line 3, in <module>
    dragon = Creature(
  File "pydantic/main.py", line 342, in
    pydantic.main.BaseModel.init
    pydantic.error_wrappers.ValidationError:
      1 validation error for Creature description
        str type expected (type=type_error.str)
```

Проверка значений

Даже если тип значения соответствует его спецификации в классе `Creature`, могут потребоваться дополнительные проверки. Некоторые ограничения могут быть наложены на само значение.

- Целочисленное значение (`conint`) или число с плавающей точкой:

`gt` — больше чем;

`lt` — меньше чем;

`ge` — больше или равно;

`le` — меньше или равно;

`multiple_of` — целое число, кратное значению.

- Строковое (`constr`) значение:

`min_length` — минимальная длина в символах (не в байтах);

`max_length` — максимальная длина в символах;

`to_upper` — преобразование в прописные буквы;

`to_lower` — преобразование в строчные буквы;

`regex` — сопоставление с регулярным выражением Python.

- Кортеж, список или множество:

`min_items` — минимальное количество элементов;

`max_items` — максимальное количество элементов.

Они указываются в типовых частях модели.

Пример 5.16 позволяет убедиться, что поле `name` всегда будет содержать не менее двух символов. В противном случае `""` (пустая строка) будет считаться допустимой.

Пример 5.16. Просмотр ошибки проверки

```
>>> from pydantic import BaseModel, constr
>>>
>>> class Creature(BaseModel):
...     name: constr(min_length=2)
...     country: str
...     area: str
...     description: str
...     aka: str
...
>>> bad_creature = Creature(name="!",
...     description="it's a raccoon",
...     area="your attic")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "pydantic/main.py", line 342,
    in pydantic.main.BaseModel.__init__
pydantic.error_wrappers.ValidationError:
1 validation error for Creature name
  ensure this value has at least 2 characters
  (type=value_error.any_str.min_length; limit_value=2)
```

Ключевое слово `constr` означает *ограниченную строку* (*constrained string*). В примере 5.17 используется альтернативный вариант — спецификация `Field` из библиотеки Pydantic.

Пример 5.17. Еще один сбой проверки, применена функция `Field`

```
>>> from pydantic import BaseModel, Field
>>>
>>> class Creature(BaseModel):
...     name: str = Field(..., min_length=2)
...     country: str
...     area: str
...     description: str
...     aka: str
...
>>> bad_creature = Creature(name="!",
...     area="your attic",
...     description="it's a raccoon")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "pydantic/main.py", line 342,
    in pydantic.main.BaseModel.__init__
pydantic.error_wrappers.ValidationError:
1 validation error for Creature name
  ensure this value has at least 2 characters
  (type=value_error.any_str.min_length; limit_value=2)
```

Аргумент ... функции `Field()` означает, что значение обязательное и значения по умолчанию не предусмотрено.

Это минимальное введение в Pydantic. Главное, что можно сделать, — автоматизировать проверку данных. Вы увидите, насколько это полезно, при получении данных с веб-уровня или уровня данных.

Заключение

Модели предоставляют лучший способ определить данные, передаваемые в вашем веб-приложении. Библиотека Pydantic использует подсказки типов Python для определения моделей, передаваемых в приложении данных. Далее — определение *зависимостей* для выделения конкретных деталей из общего кода.

Зависимости

Обзор

Одной из очень приятных особенностей дизайна FastAPI является техника, называемая *внедрением зависимостей*. Этот термин звучит технически и эзотерически, но это ключевой аспект FastAPI, и он удивительно полезен на многих уровнях. В этой главе рассматриваются встроенные возможности FastAPI, а также способы написания собственных.

Что такое зависимости

Зависимость — это конкретная информация, требующаяся вам в определенный момент. Обычный способ получить эту информацию — написать код, доставляющий ее именно тогда, когда она необходима.

При написании веб-сервиса в какой-то момент вам может понадобиться сделать следующее:

- получить входные параметры из HTTP-запроса;
- проверить вводимые данные;
- проверить аутентификацию и авторизацию пользователей для некоторых конечных точек;
- найти данные в источнике данных, часто в базе данных;
- выдавать параметры, журналы или информацию для отслеживания.

Веб-фреймворки преобразуют байты HTTP-запросов в структуры данных, а вы по мере необходимости извлекаете из них то, что вам нужно, в своих функциях веб-уровня.

Проблемы с зависимостями

Получение того, что вам нужно, именно тогда, когда нужно, причем внешнему коду не обязательно знать, как вы это получили, кажется вполне разумным. Но оказалось, что существуют последствия.

- *Тестирование* — вы не можете протестировать варианты функции, выполняющие поиск зависимостей по-другому.
- *Скрытые зависимости* — сокрытие подробной информации означает, что код, необходимый вашей функции, может прерваться при изменении внешнего кода.
- *Дублирование кода* — если зависимость является общей (например, поиск пользователя в базе данных или объединение значений из HTTP-запроса), код поиска может оказаться продублированным в нескольких функциях.
- *Видимость OpenAPI* — автоматическая тестовая страница, создаваемая FastAPI, нуждается в информации из механизма внедрения зависимостей.

Внедрение зависимостей

Термин «внедрение зависимостей» проще, чем кажется, — это передача функции любой требующейся ей *специфической* информации. Традиционный способ сделать это — передать вспомогательную функцию, которую вы затем вызываете для получения конкретных данных.

Зависимости FastAPI

FastAPI продвинут еще на один шаг вперед — он позволяет определить зависимости как аргументы функции, и они будут *автоматически* вызываться FastAPI и передавать возвращаемые ими *значения*. Например, зависимость `user_dep` может получать имя и пароль пользователя из HTTP-аргументов, искать их в базе данных и возвращать токен, применяемый для отслеживания в дальнейшем этого пользователя. Ваша функция веб-обработки никогда не вызывает зависимость напрямую — она обрабатывается во время вызова функции.

Вы уже видели некоторые зависимости, правда, прежде их так не называли, — это источники данных HTTP, такие как `Path`, `Query`, `Body` и `Header`. Это функции или классы Python, откапывающие запрашиваемые данные из различных областей HTTP-запроса. Они скрывают детали, такие как проверка валидности и форматы данных.

Почему бы не написать собственные функции для этого? Можно, но у вас не будет:

- проверки валидности данных;
- преобразования форматов;
- автоматического документирования.

Во многих других веб-фреймворках эти проверки выполняются внутри собственных функций. Примеры их работы приведены в главе 7, где FastAPI сравнивается с такими веб-фреймворками Python, как Flask и Django. Но в FastAPI можно работать с собственными зависимостями так же, как и со встроенными.

Написание зависимостей

В FastAPI зависимость — это то, что выполняется, поэтому объект зависимости должен относиться к типу `Callable`, включающему функции и классы — то, что вы вызываете, со скобками и необязательными аргументами.

В примере 6.1 показана функция зависимости `user_dep()`. Она принимает строковые аргументы имени и пароля и просто возвращает значение `True`, если пользователь прошел проверку на валидность. Для этой первой версии пусть функция возвращает значение `True` для всех данных.

Пример 6.1. Функция зависимости

```
from fastapi import FastAPI, Depends, Params

app = FastAPI()

# функция зависимости:
def user_dep(name: str = Params, password: str = Params):
    return {"name": name, "valid": True}

# функция пути/конечная точка веб-приложения:
@app.get("/user")
def get_user(user: dict = Depends(user_dep)) -> dict:
    return user
```

В этом фрагменте кода `user_dep()` — функция зависимости. Она действует как функция пути FastAPI (знает о таких вещах, как `Params` и т. д.), но не содержит декоратора пути над собой. Это помощник, а не сама конечная точка.

В функции пути `get_user()` говорится, что она ожидает переменную аргумента под названием `user` и эта переменная получит свое значение из функции зависимости `user_dep()`.



В аргументах функции `get_user()` нельзя написать `user = user_dep`, потому что `user_dep` — это объект функции Python. И нельзя написать `user = user_dep()`, потому что это вызвало бы функцию `user_dep()`, когда функция `get_user()` была определена, а не когда она используется. Поэтому нам нужна дополнительная вспомогательная функция FastAPI `Depends()`, чтобы вызывать `user_dep()` именно тогда, когда это необходимо.

В списке аргументов функции пути может быть несколько зависимостей.

Область действия зависимости

Вы можете определить зависимости для одной функции пути, их группы или всего веб-приложения.

Единый путь

Включите в *функцию пути* такой аргумент:

```
def pathfunc(name: depfunc = Depends(depfunc)):
```

или просто в таком виде:

```
def pathfunc(name: depfunc = Depends()):
```

name — это то, как вы хотите назвать значение (значения), возвращаемое *depfunc*. Из предыдущего примера:

- *pathfunc* — это `get_user()`;
- *depfunc* — это `user_dep()`;
- *name* — это `user`.

Пример 6.2 учитывает этот путь и зависимость для возврата фиксированного имени (*name*) пользователя и логического значения `valid`.

Пример 6.2. Возвращение зависимости пользователя

```

from fastapi import FastAPI, Depends, Params

app = FastAPI()

# функция зависимости:
def user_dep(name: str = Params, password: str = Params):
    return {"name": name, "valid": True}

# функция пути/конечная точка веб-приложения:
@app.get("/user")
def get_user(user: dict = Depends(user_dep)) -> dict:
    return user

```

Если функция зависимости просто проверяет что-то и не возвращает никаких значений, вы можете определить зависимость в *декораторе* пути (предыдущая строка, начинающаяся с @):

```
@app.method(url, dependencies=[Depends(depfunc)])
```

Попробуем сделать это в примере 6.3.

Пример 6.3. Определение зависимости проверки пользователя

```

from fastapi import FastAPI, Depends, Params

app = FastAPI()

# функция зависимости:
def check_dep(name: str = Params, password: str = Params):
    if not name:
        raise

# функция пути/конечная точка веб-приложения:
@app.get("/check_user", dependencies=[Depends(check_dep)])
def check_user() -> bool:
    return True

```

Множество путей

В главе 9 подробно рассказывается о том, как структурировать более крупное приложение FastAPI, включая определение нескольких объектов *маршрутизатора* (router) в приложении верхнего уровня, вместо того чтобы прикреплять каждую конечную точку к этому приложению. Пример 6.4 иллюстрирует эту концепцию.

Пример 6.4. Определение зависимости субмаршрута

```
from fastapi import FastAPI, Depends, APIRouter

router = APIRouter(..., dependencies=[Depends(depfunc)])
```

Это приведет к вызову функции *depfunc()* для всех функций пути ниже объекта *router*.

Способ глобального внедрения зависимостей

При определении объекта приложения FastAPI верхнего уровня можно добавить к нему зависимости, применяемые ко всем его функциям пути, как показано в примере 6.5.

Пример 6.5. Определение зависимости уровня приложения

```
from fastapi import FastAPI, Depends

def depfunc1():
    pass

def depfunc2():
    pass

app = FastAPI(dependencies=[Depends(depfunc1), Depends(depfunc2)])

@app.get("/main")
def get_main():
    pass
```

В этом случае используется инструкция *pass*, позволяющая проигнорировать другие детали, чтобы показать, как подключить зависимости.

Заключение

В этой главе мы обсудили зависимости и их внедрение — способы получения необходимых вам данных в нужный момент и простым способом. В следующей главе Flask, Django и FastAPI заходят в бар...

Сравнение фреймворков

Вам не нужен каркас. Вам нужна картина,
а не ее рама¹.

Клаус Кински, актер

Обзор

Для разработчиков, ранее использовавших Flask, Django или другие популярные веб-фреймворки Python, эта глава указывает на их сходство с FastAPI и отличия от него. Здесь не рассматриваются все утомительные подробности, потому что иначе клей для переплета не удержит эту книгу целой. Сравнения, приведенные здесь, могут быть полезны, если вы думаете о переносе приложения с одного из этих фреймворков на FastAPI или просто любопытствуете.

Одна из первых вещей, которую вы можете узнать о новом веб-фреймворке, — это как начать работу, и путь сверху вниз — это определение *маршрутов* (связки между URL-адресами и HTTP-методами и функциями). В следующем разделе мы сравним, как выполнить эту задачу с помощью FastAPI и Flask, поскольку они более похожи друг на друга, чем Django, и, скорее всего, будут рассматриваться вместе для похожих приложений.

¹ Цитата отражает некую игру слов, поскольку в английском языке слово *framework* означает каркас, а не только привычный разработчикам фреймворк. — *Примеч. пер.*

Flask

Разработчики Flask (<https://flask.palletsprojects.com>) называют его *микрофреймворком*. Он предоставляет базовые возможности, а вы загружаете сторонние пакеты, чтобы дополнить их по мере необходимости. Он меньше, чем Django, и в начале работы его можно быстрее освоить.

Flask относится к типу синхронных и создан на базе стандарта WSGI, а не ASGI. Новый проект под названием quart (<https://quart.palletsprojects.com>) воспроизводит Flask и добавляет в него поддержку стандарта ASGI.

Начнем с самого начала, показав, как Flask и FastAPI определяют веб-маршрутизацию.

Путь

На верхнем уровне Flask и FastAPI используют декоратор, чтобы связать маршрут с конечной веб-точкой. В примере 7.1 продублируем пример 3.11, в котором человек получает приветствие из URL-пути.

Пример 7.1. Путь FastAPI

```
from fastapi import FastAPI

app = FastAPI()

@app.get("/hi/{who}")
def greet(who: str):
    return f"Hello? {who}?"
```

По умолчанию FastAPI преобразует строку `f"Hello? {who}?"` в формат JSON и возвращает ее веб-клиенту.

В примере 7.2 показано, как это сделает Flask.

Пример 7.2. Путь Flask

```
from flask import Flask, jsonify

app = Flask(__name__)

@app.route("/hi/<who>", methods=["GET"])
def greet(who: str):
    return jsonify(f"Hello? {who}?")
```

Обратите внимание на то, что слово `who` в декораторе теперь заключено в угловые скобки (`<` и `>`). Во Flask метод должен быть включен в качестве аргумента, если только по умолчанию не используется `GET`. Следовательно, выражение `methods= ["GET"]` можно было бы и опустить, но ясность никогда не помешает.



Flask 2.0 поддерживает декораторы в стиле FastAPI, такие как `@app.get`, вместо `app.route`.

Функция `jsonify()` из Flask преобразует аргумент в строку формата JSON и возвращает ее вместе с заголовком HTTP-ответа, указывающим на то, что это формат JSON. Если вы возвращаете данные типа `dict` (а не другие типы данных), последние версии Flask автоматически конвертируют его в JSON и возвращают. Вызов функции `jsonify()` явно работает для всех типов данных, включая `dict`.

Параметр запроса

В примере 7.3 повторим пример 3.15, где `who` передается в качестве параметра запроса (после символа `?` в URL-адресе).

Пример 7.3. Параметр запроса FastAPI

```
from fastapi import FastAPI

app = FastAPI()

@app.get("/hi")
def greet(who):
    return f"Hello? {who}?"
```

Эквивалент Flask показан в примере 7.4.

Пример 7.4. Параметр запроса Flask

```
from flask import Flask, request, jsonify

app = Flask(__name__)

@app.route("/hi", methods=["GET"])
def greet():
    who: str = request.args.get("who")
    return jsonify(f"Hello? {who}?" )
```

Во Flask нам нужно получить значения запроса из объекта `request`. В данном случае аргумент `args` относится к типу `dict` и содержит параметры запроса.

Тело запроса

В примере 7.5 скопируем старый пример 3.21.

Пример 7.5. Тело запроса FastAPI

```
from fastapi import FastAPI

app = FastAPI()

@app.get("/hi")
def greet(who):
    return f"Hello? {who}?"
```

Версия Flask выглядит как в примере 7.6.

Пример 7.6. Тело запроса Flask

```
from flask import Flask, request, jsonify

app = Flask(__name__)

@app.route("/hi", methods=["GET"])
def greet():
    who: str = request.json["who"]
    return jsonify(f"Hello? {who}?" )
```

Flask хранит входные данные в формате JSON в файле `request.json`.

Заголовок

Наконец, повторим пример 3.24 в примере 7.7.

Пример 7.7. Заголовок FastAPI

```
from fastapi import FastAPI, Header

app = FastAPI()

@app.get("/hi")
def greet(who:str = Header()):
    return f"Hello? {who}?"
```

Версия Flask показана в примере 7.8.

Пример 7.8. Заголовок Flask

```
from flask import Flask, request, jsonify

app = Flask(__name__)
```



```
@app.route("/hi", methods=["GET"])
def greet():
    who: str = request.headers.get("who")
    return jsonify(f"Hello? {who}?")
```

Как и в случае с параметрами запроса, Flask хранит данные запроса в объекте `request`. В данном случае это будет атрибут `headers` типа `dict`. Ключи заголовков должны быть нечувствительны к регистру.

Django

Django (<https://www.djangoproject.com>) — это более крупный и сложный проект, чем Flask или FastAPI, ориентированный на «перфекционистов со сроками», как утверждается на его сайте. Встроенное объектно-реляционное связывание (Object-Relational Mapper, ORM) полезно для сайтов с основными бэкендами баз данных. Это скорее монолит, чем набор инструментов. Оправданны ли трудности обучения и освоение дополнительных сложностей, зависит от ваших задач.

Хотя Django был традиционным WSGI-приложением, в версии 3.0 была добавлена поддержка стандарта ASGI.

В отличие от Flask и FastAPI, Django предпочитает определять маршруты (связывая URL с веб-функциями, которые он называет *функциями представления*) в одной таблице `URLConf`, а не с помощью декораторов. Это облегчает просмотр всех маршрутов в одном месте, но затрудняет понимание того, какой URL связан с функцией, когда вы смотрите только на саму функцию.

Другие функциональные возможности веб-фреймворка

В предыдущих разделах, посвященных сравнению трех фреймворков, я в основном сравнивал способы определения маршрутов. От веб-фреймворка можно ожидать помощи и в других областях.

- *Формы* — все три пакета поддерживают стандартные HTML-формы.
- *Файлы* — все эти пакеты работают с выгрузкой и скачиванием файлов, включая многокомпонентные HTTP-запросы и ответы.

- *Шаблоны* — язык шаблонов позволяет смешивать текст и код и полезен для контент-ориентированного сайта (HTML-текст с динамически вставляемыми данными), а не для сайта с API. Наиболее известным пакетом шаблонов Python является Jinja (<https://jinja.palletsprojects.com>), и он поддерживается Flask, Django и FastAPI. У Django есть и собственный язык шаблонов (<https://oreil.ly/OIbVJ>).

Если вы хотите использовать сетевые методы, выходящие за рамки базового HTTP, попробуйте следующие варианты.

- *Server-sent events (SSE)* — передает данные клиенту по мере необходимости. Поддерживается FastAPI (`sse-starlette`, <https://oreil.ly/Hv-QP>), Flask (Flask-SSE, <https://oreil.ly/oz518>) и Django (Django EventStream, <https://oreil.ly/NIBE5>).
- *Очереди* — очереди заданий, публикация-подписка и другие сетевые шаблоны поддерживаются такими внешними пакетами, как ZeroMQ, Celery, Redis и RabbitMQ.
- *WebSockets* — поддерживаются FastAPI (непосредственно), Django (Django Channels, <https://channels.readthedocs.io>) и Flask (сторонние пакеты).

Базы данных

В базовые пакеты Flask и FastAPI не включена работа с базами данных, но она является ключевой особенностью Django.

Уровень данных вашего сайта может обращаться к базе данных на разных уровнях:

- непосредственно SQL (PostgreSQL, SQLite);
- непосредственно NoSQL (Redis, MongoDB, Elasticsearch);
- *ORM*, генерирующее SQL;
- объектное связывание документов (Object Document Mapping, ODM), генерирующее NoSQL.

Для реляционных баз данных SQLAlchemy (<https://www.sqlalchemy.org>) — отличный пакет, включающий в себя несколько уровней доступа, от прямого SQL до ORM. Это обычный выбор для разработчиков Flask и FastAPI. Автор FastAPI использовал как SQLAlchemy, так и Pydantic для пакета SQLModel (<https://sqlmodel.tiangolo.com>) — о нем мы подробнее поговорим в главе 14.

Django часто выбирают в качестве фреймворка для сайтов с большими потребностями в базах данных. У него есть свои ORM (<https://oreil.ly/eFzZn>) и автоматизированная страница администрирования баз данных (https://oreil.ly/_al42). Хотя некоторые источники рекомендуют разрешить нетехническому персоналу использовать страницу администратора для рутинного управления данными, будьте осторожны. Однажды я видел, как неспециалист неправильно понял предупреждающее сообщение на странице администратора, в результате чего базу данных пришлось восстанавливать вручную из резервной копии.

В главе 14 более подробно рассматриваются FastAPI и базы данных.

Рекомендации

Для сервисов на базе API лучшим выбором кажется FastAPI. Flask и FastAPI примерно равны в плане скорости запуска сервиса. Чтобы разобраться в Django, потребуется больше времени, но он предоставляет множество возможностей, полезных для больших сайтов, особенно сильно зависящих от баз данных.

Другие веб-фреймворки Python

В настоящее время три основных веб-фреймворка на Python — это Flask, Django и FastAPI. Введите в Google запрос **python web frameworks**, и вы получите множество предложений, которые я не буду здесь приводить. Среди тех, которые, возможно, не выделяются в этих списках, но интересны по тем или иным причинам, можно назвать следующие:

- *Bottle* (<https://bottlepy.org/docs/dev>) — *минимальный* (один файл Python) пакет, хорошо подходящий для того, чтобы быстро доказать концепцию;
- *Litestar* (<https://litestar.dev>) — похож на FastAPI — основан на ASGI/Starlette и Pydantic, но представляет иной взгляд на поставленную задачу;
- *AIOHTTP* (<https://docs.aiohttp.org>) — клиент и сервер ASGI с полезным демонстрационным кодом;
- *Socketify.py* (<https://docs.socketify.dev>) — новый участник, потенциально очень высокопроизводительный.

Заключение

Flask и Django — самые популярные веб-фреймворки на Python, хотя популярность FastAPI растет быстрее. Все три они справляются с основными задачами веб-сервера, но скорость их освоения разная. У FastAPI, похоже, более чистый синтаксис для задания маршрутов, а поддержка ASGI позволяет ему во многих случаях работать быстрее своих конкурентов. Далее: давайте уже создадим сайт.

ЧАСТЬ III

Создание веб-сайта

В части II я дал краткий обзор FastAPI, чтобы быстро ввести вас в курс дела. В этой части книги будем углубляться в детали. Мы создадим веб-сервис среднего размера для доступа к данным о криптидах — выдуманных существах и таких же выдуманных исследователях, которые их ищут, а также для управления ими.

Полный сервис будет иметь три уровня, как я говорил ранее:

- *веб-уровень* — веб-интерфейс;
- *сервис* — бизнес-логика;
- *данные* — драгоценная ДНК всей конструкции.

Кроме того, веб-сервис будет содержать следующие межуровневые компоненты:

- *модель* — определения данных Pydantic;
- *тесты* — модульные, интеграционные и комплексные тесты.

В дизайне сайта будут учтены следующие моменты.

- Что должно располагаться на каждом из уровней?
- Что передается между уровнями?
- Можем ли мы позже изменить/добавить/удалить код, ничего не нарушив?
- Если работа чего-то прервется, как мне это найти и исправить?
- Как обстоит ситуация с безопасностью?
- Может ли сайт масштабироваться и сохранять работоспособность?
- Можно ли сделать все это как можно более понятным и простым?
- Почему я задаю так много вопросов? Почему, почему?

Веб-уровень

Обзор

В главе 3 мы вкратце поговорили о том, как определять конечные точки FastAPI, передавать им простые строковые данные и получать ответы. В этой главе подробнее рассмотрим верхний уровень приложения FastAPI — его также можно назвать уровнем *интерфейса* или *маршрутизации* — и его интеграцию с уровнями сервисов и данных.

Как и прежде, начну с небольших примеров. Затем введу некоторую структуру, разделив уровни на подуровни, чтобы обеспечить более чистое развитие и рост веб-приложения. Чем меньше кода мы пишем, тем меньше придется вспоминать и исправлять впоследствии.

Основные примеры данных в этой книге касаются воображаемых существ, или *криптидов*, и их исследователей. Но вы можете провести параллели с информацией из других областей.

Что мы вообще делаем с информацией? Как и на большинстве других сайтов, на нашем вы найдете способы выполнить следующие операции:

- получение;
- создание;
- изменение;
- замену;
- удаление.

Начав с самого верха, мы создадим конечные точки веб-приложения, способные выполнять эти функции с нашими данными. Сначала предоставим фиктивные данные, чтобы конечные точки работали с любым веб-клиентом. В следующих главах мы перенесем код этих данных на нижние уровни. На каждом этапе необходимо будет убедиться, что сайт по-прежнему работает и корректно передает данные.

Наконец, в главе 10 мы откажемся от фиктивных данных и будем хранить реальные данные в реальных базах данных, чтобы создать полноценный сайт (веб → сервис → данные).



Если позволить любому анонимному посетителю выполнить все эти действия, не стоит ожидать ничего хорошего. В главе 11 рассматриваются аутентификация и авторизация (auth), необходимые для определения ролей и ограничения того, кто что может делать. В оставшейся части этой главы мы обойдемся без аутентификации и просто рассмотрим, как работать с неопределенными веб-функциями.

Немного отвлечемся: сверху вниз, снизу вверх, от центра наружу?

При разработке веб-сайта можно реализовать один из таких вариантов, как:

- веб-уровень и работа дальше вниз;
- уровень данных и работа дальше вверх;
- сервисный уровень и работа в обоих направлениях.

У вас уже есть база данных, установленная и наполненная данными, и вы просто жаждете поделиться ею со всем миром? Если да, то, возможно, стоит сначала заняться кодом и тестами уровня данных, затем уровнем сервисов, а веб-уровень написать последним.

Если ваш подход — предметно-ориентированное проектирование (Domain-Driven Design, DDD) (<https://oreil.ly/iJu9Q>), можете начать со среднего, сервисного уровня, определяя основные сущности и модели данных. Или же сначала разработать веб-интерфейс, а к нижним уровням обратиться, когда поймете, чего от них ожидать.

Очень хорошие обсуждения и рекомендации по дизайну вы найдете в следующих книгах:

- *Clean Architectures in Python* (<https://oreil.ly/5KrL9>), автор Леонардо Джордани (Digital Cat Books);
- *Architecture Patterns with Python* (<https://www.cosmicpython.com>), авторы Гарри Персиваль и Боб Грегори (O'Reilly)¹;
- *Microservice APIs* (<https://oreil.ly/Gk0z2>), автор Хосе Аро Перальта (Manning)².

В этих и других источниках вы увидите такие термины, как «гексагональная архитектура», «порты» и «адаптеры». Выбор способа действий во многом зависит от того, какие данные у вас уже есть и как вы хотите подойти к созданию сайта.

Я предполагаю, что многие из вас в основном заинтересованы в том, чтобы попробовать FastAPI и связанные с ним технологии, и не обязательно имеют заранее определенный зрелый домен данных, который хочется сразу же задействовать. Поэтому в этой книге я использую подход под названием web-first — шаг за шагом, начиная с основных частей и добавляя другие по мере необходимости. Иногда эксперименты работают, иногда нет. Поначалу я постараюсь сдерживать желание запихнуть все в веб-уровень.



Веб-уровень — лишь один из способов передачи данных между пользователем и сервисом. Существуют и другие варианты, например с помощью интерфейса командной строки (CLI) или набора средств разработки программного обеспечения (SDK). В других фреймворках веб-уровень иногда называют уровнем представления или презентации.

Проектирование RESTful API

HTTP — это способ передачи команд и данных между веб-клиентами и серверами. Но, как и в случае с ингредиентами из холодильника, которые можно комбинировать, изготавливая блюда от отвратительных до изысканных, некоторые рецепты для HTTP работают лучше, чем другие.

В главе 1 я упоминал, что *RESTful* стал полезной, хотя иногда и нечеткой моделью для разработки HTTP. Проектирование RESTful включает в себя следующие основные компоненты:

- *ресурсы* — элементы данных, которыми управляет ваше приложение;
- *идентификаторы* — уникальные идентификаторы ресурсов;

¹ Персиваль Г., Грегори Б. Паттерны разработки на Python. — СПб.: Питер, 2022.

² Перальта Х. А. Микросервисы и API. — СПб.: Питер, 2024.

- *URL-адреса* — структурированные строки ресурсов и идентификаторов;
- *глагольные операторы* или *действия* — термины, сопровождающие URL-адреса для различных целей:
 - GET — получение ресурса;
 - POST — создание нового ресурса;
 - PUT — полная замена ресурса;
 - PATCH — частичная замена ресурса;
 - DELETE — ресурсы разлетаются в клочья.



Вы увидите разногласия по поводу относительных достоинств PUT в сравнении с PATCH. Если вам не нужно отличать частичную модификацию от полной (замены), то, возможно, оба оператора и не понадобятся.

Общие правила RESTful, касающиеся сочетания глаголов и URL-адресов, содержащих ресурсы и идентификаторы, предусматривают следующие шаблоны параметров пути (содержимое между / в URL):

- *verb/resource/* — применение глагольного оператора (*verb*) ко всем ресурсам типа *resource*;
- *verb/resource/id* — применение глагольного оператора (*verb*) к ресурсам (*resource*) с идентификатором *id*.

При использовании примера данных для этой книги запрос GET к конечной точке */thing* вернет данные обо всех исследователях, но запрос GET к */thing/abc* предоставит данные только для ресурса *thing* с идентификатором *abc*.

Наконец, веб-запросы часто содержат больше информации, указывая на необходимость следующих действий:

- сортировки результатов;
- пагинации результатов;
- выполнения другой функции.

Параметры для них иногда могут иметь вид параметров *пути* (добавляются в конец после еще одного символа /), но чаще всего они включаются как параметры *запроса* (*var=val* после знака ? в URL-адресе). Поскольку у URL-адресов есть ограничения по размеру, большие запросы часто передаются в теле HTTP.



Большинство авторов рекомендуют использовать множественное число при именовании ресурса и связанных с ним пространств имен, таких как разделы API и таблицы баз данных. Я долго следовал этому совету, но теперь считаю, что названия в единственном числе проще по многим причинам (включая странности английского языка):

- некоторые слова представляют множественное число самих себя — *series*, *fish*;
- у некоторых слов неправильное множественное число — *children*, *people*;
- вам нужен код преобразования единственного числа во множественное по требованию во многих местах.

По этим причинам во многих случаях в книге я использую схему именования в единственном числе. Это противоречит обычным рекомендациям RESTful, так что не стесняйтесь игнорировать эту мою особенность, если не согласны со мной.

Макет сайта с файлами и каталогами

Наши данные касаются в основном существ и исследователей. Изначально мы могли бы определить все URL-адреса и их функции пути FastAPI для доступа к данным в одном файле Python. Не будем поддаваться искушению и начнем так, как будто мы уже восходящая звезда в криптидном веб-пространстве. Имея хороший фундамент, гораздо легче добавлять новые крутые вещи.

Сначала выберите на своей машине каталог. Назовите его `fastapi` или как угодно, что поможет запомнить, где вы будете работать с кодом из этой книги. В нем создайте следующие подкаталоги:

- `src` — содержит весь код сайта;
- `web` — веб-уровень FastAPI;
- `service` — уровень бизнес-логики;
- `data` — уровень интерфейса хранения данных;
- `model` — для определения моделей Pydantic;
- `fake` — жестко указанные данные (заглушки) для ранних этапов.

В каждой из этих папок вскоре появится по три файла:

- `__init__.py` — необходим для восприятия этого каталога в качестве пакета;
- `creature.py` — код существа для этого уровня;
- `explorer.py` — код исследователя для этого уровня.

Существует *множество* мнений о том, как следует планировать страницы для разработки. Такой дизайн призван показать разделение уровней и оставить место для будущих дополнений.

Сейчас необходимо дать некоторые объяснения. Поначалу файлы `__init__.py` будут пустыми. Они представляют собой своего рода хакерскую уловку в отношении Python, поэтому к содержащей их папке следует относиться как к пакету Python, который можно импортировать. Во-вторых, папка `fake` предоставляет некоторые данные-заглушки для более высоких уровней по мере создания нижних.

Кроме того, логика *импорта* в Python не работает строго с иерархиями каталогов. Она опирается на *пакеты* и *модули* Python. Файлы с расширением `.py`, перечисленные в приведенной ранее древовидной структуре, являются модулями Python (исходными файлами). Их родительские каталоги будут считаться пакетами, *если* они содержат файл `__init__.py`. (Это соглашение необходимо, чтобы, если у вас есть каталог `sys` и вы набираете команду `import sys`, Python мог определить, вам нужен системный каталог или ваш локальный.)

Программы Python могут импортировать пакеты и модули. У интерпретатора Python есть встроенная переменная `sys.path`. Она содержит местоположение стандартного кода Python. Переменная окружения `PYTHONPATH` — это пустая или разделенная двоеточием строка имен каталогов. Она указывает Python, какие родительские каталоги проверять перед `sys.path`, чтобы найти импортированные модули или пакеты. Поэтому, если вы переходите в новую папку `fastapi`, введите следующую команду (в Linux или macOS), чтобы новый код в ней проверялся первым при импорте:

```
$ export PYTHONPATH=$PWD/src
```

Часть выражения `$PWD` означает «*вывести рабочий каталог*» и избавляет вас от необходимости вводить полный путь к каталогу `fastapi`, хотя вы можете это сделать, если хотите. А `src` означает, что искать модули и пакеты для импорта нужно только там.

Чтобы установить переменную окружения `PWD` в Windows, изучите раздел *Excursus: Setting Environment Variables* на сайте Python Software Foundation (<https://oreil.ly/9NRBA>).

Фух.

Первый код веб-сайта

В этом разделе мы рассмотрим, как использовать FastAPI для написания запросов и ответов для сайта RESTful API. Затем начнем применять их на нашем реальном все более и более странном сайте.

Начнем с примера 8.1. В каталоге `src` создайте новую программу верхнего уровня `main.py` — она будет запускать программу Uvicorn и пакет FastAPI.

Пример 8.1. Основная программа `main.py`

```
from fastapi import FastAPI

app = FastAPI()

@app.get("/")
def top():
    return "top here"

if __name__ == "__main__":
    import uvicorn
    uvicorn.run("main:app", reload=True)
```

Слово `app` здесь представляет собой объект FastAPI, связывающий все воедино. Первый аргумент Uvicorn — `"main:app"`, потому что файл называется `main.py`, а второй — `app`, имя объекта FastAPI.

Uvicorn будет продолжать работать и перезапустится, если в том же каталоге или в любых подкаталогах изменится код. Без аргумента `reload=True` придется перезапускать Uvicorn каждый раз, когда вы вносите изменения в код. Во многих следующих примерах просто вносите изменения в один и тот же файл `main.py` и принудительно перезапускайте его, вместо того чтобы создавать файлы `main2.py`, `main3.py` и т. д. Запустите файл `main.py` из примера 8.2.

Пример 8.2. Запуск основной программы

```
$ python main.py &
INFO:      Will watch for changes in these directories: [.../fastapi']
INFO:      Uvicorn running on http://127.0.0.1:8000 (Press CTRL+C to quit)
INFO:      Started reloader process [92543] using StatReload
INFO:      Started server process [92551]
INFO:      Waiting for application startup.
INFO:      Application startup complete.
```

Символ `&` в конце строки переводит программу в фоновый режим, и вы можете запускать другие программы в том же окне терминала, если хотите. Или опустите символ `&` и запустите другой код в другом окне или на другой вкладке.

Теперь вы можете получить доступ к сайту `localhost:8000` с помощью браузера или любой из приведенных ранее тестовых программ. В примере 8.3 используется HTTPie.

Пример 8.3. Тестирование основной программы

```
$ http localhost:8000
HTTP/1.1 200 OK
content-length: 8
content-type: application/json
date: Sun, 05 Feb 2023 03:54:29 GMT
server: uvicorn
```

```
"top here"
```

С этого момента при внесении изменений веб-сервер должен автоматически перезапускаться. Если ошибка останавливает его, перезапустите сервер с помощью команды `python main.py`.

В примере 8.4 добавлена еще одна тестовая конечная точка с использованием параметра пути (часть URL-адреса).

Пример 8.4. Добавление конечной точки

```
import uvicorn
from fastapi import FastAPI

app = FastAPI()

@app.get("/")
def top():
    return "top here"

@app.get("/echo/{thing}")
def echo(thing):
    return f"echoing {thing}"

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

Как только вы сохраните изменения в файле `main.py` в своем редакторе, в окне, где запущен веб-сервер, должно появиться что-то вроде этого:

```
WARNING: StatReload detected changes in 'main.py'. Reloading...
INFO: Shutting down
INFO: Waiting for application shutdown.
INFO: Application shutdown complete.
INFO: Finished server process [92862]
INFO: Started server process [92872]
INFO: Waiting for application startup.
INFO: Application startup complete.
```

Пример 8.5 показывает, правильно ли была обработана новая конечная точка (с помощью аргумента `-b` выводится только тело ответа).

Пример 8.5. Тестирование новой конечной точки

```
$ http -b localhost:8000/echo/argh
"echoing argh"
```

В следующих разделах мы добавим больше конечных точек в файл `main.py`.

Запросы

HTTP-запрос состоит из текстового *заголовка*, за которым следует один или несколько разделов *тела*.

Можете написать собственный код для разбора HTTP в структуры данных Python, но вы не будете первым. В веб-приложениях эти детали лучше поручить фреймворку.

Возможность реализовать внедрение зависимостей FastAPI здесь особенно полезна. Данные могут поступать из разных частей HTTP-сообщения, и вы уже видели, как можно указать одну или несколько таких зависимостей, чтобы сказать, где находятся данные:

- **Header** — в HTTP-заголовке;
- **Path** — в пути URL;
- **Query** — после символа `?` в URL;
- **Body** — в теле HTTP-сообщения.

К другим, более косвенным источникам можно отнести:

- переменные окружения;
- настройки конфигурации.

В примере 8.6 выполняется HTTP-запрос с использованием нашего старого друга HTTPie и игнорированием возвращаемых данных HTML-тела.

Пример 8.6. Заголовки HTTP-запросов и ответов

```
$ http -p NBh http://example.com/
GET / HTTP/1.1
Accept: /
Accept-Encoding: gzip, deflate
Connection: keep-alive
Host: example.com
User-Agent: HTTPie/3.2.1

HTTP/1.1 200 OK
Age: 374045
Cache-Control: max-age=604800
Content-Encoding: gzip
Content-Length: 648
Content-Type: text/html; charset=UTF-8
Date: Sat, 04 Feb 2023 01:00:21 GMT
Etag: "3147526947+gzip"
Expires: Sat, 11 Feb 2023 01:00:21 GMT
Last-Modified: Thu, 17 Oct 2019 07:18:26 GMT
Server: ECS (cha/80E2)
Vary: Accept-Encoding
X-Cache: HIT
```

Первая строка запрашивает верхнюю страницу по адресу `example.com` (бесплатный сайт, доступный для использования любому в качестве примера). Код запрашивает только URL, без каких-либо параметров. Первый блок строк — это заголовки HTTP-запроса, отправленного на сайт, а следующий блок содержит заголовки HTTP-ответа.



В большинстве тестовых примеров, начиная с данного момента, все эти заголовки запросов и ответов не понадобятся, поэтому вы будете чаще использовать выражение `http -b`.

Несколько маршрутизаторов

Большинство веб-сервисов работают с несколькими видами ресурсов. Хотя вы могли бы поместить весь код обработки путей в один файл и отправиться куда-нибудь побездельничать, часто удобно применять несколько *субмаршрутов* вместо одной переменной `app`, использованной в большинстве примеров ранее.

В каталоге `web` (в том же каталоге, где расположен ваш рабочий файл `main.py`) создайте файл `explorer.py`, как показано в примере 8.7.

Пример 8.7. Использование APIRouter в файле `web/explorer.py`

```
from fastapi import APIRouter

router = APIRouter(prefix = "/explorer")

@router.get("/")
def top():
    return "top explorer endpoint"
```

Теперь в примере 8.8 приложение верхнего уровня `main.py` узнает, что в системе появился новый субмаршрут, обрабатывающий все URL, начинающиеся со строки `/explorer`.

Пример 8.8. Подключение основного приложения (`main.py`) к субмаршруту

```
from fastapi import FastAPI
from .web import explorer

app = FastAPI()

app.include_router(explorer.router)
```

Uvicorn подхватит этот новый файл. Как обычно, проверьте в примере 8.9, а не предполагайте, что он будет работать.

Пример 8.9. Тестирование нового субмаршрута

```
$ http -b localhost:8000/explorer/
"top explorer endpoint"
```

Создание веб-уровня

Приступим к добавлению основных функций на веб-уровень. Изначально самим веб-функциям предоставим фиктивные данные. В главе 9 мы перенесем эти данные в соответствующие сервисные функции, а в главе 10 — в функции данных. Наконец, будет добавлена реальная база данных, к которой будет иметь доступ уровень данных. На каждом этапе разработки вызовы конечных веб-узлов должны работать.

Определение моделей данных

Сначала необходимо определить данные, которые будут передаваться между уровнями. Наша *предметная область* (или домен) содержит исследователей и существ, поэтому давайте определим для них минимальные начальные модели Pydantic. Позже могут появиться и другие идеи, например экспедиции, дневники или продажа кофейных кружек посредством электронной коммерции. Но пока просто включите две живые (обычно в случае с существами) модели в пример 8.10.

Пример 8.10. Определение модели в файле model/explorer.py

```
from pydantic import BaseModel

class Explorer(BaseModel):
    name: str
    country: str
    description: str
```

В примере 8.11 возрождается определение `Creature` из предыдущих глав.

Пример 8.11. Определение модели в файле model/creature.py

```
from pydantic import BaseModel

class Creature(BaseModel):
    name: str
    country: str
    area: str
    description: str
    aka: str
```

Это очень простые начальные модели. Здесь не были применены возможности Pydantic, такие как обязательные и необязательные или ограниченные значе-

ния. Этот простой код можно впоследствии усовершенствовать, не прибегая к масштабным логическим перестройкам.

Для значений `country` будут использоваться двухсимвольные коды стран по стандарту ISO. Это позволяет немного сэкономить на вводе текста, однако приходится искать необычные коды.

Заглушки и фиктивные данные

Заглушки, известные также как *макеты данных*, представляют собой фиксированные результаты, возвращаемые без вызова обычных активных модулей. Это быстрый способ проверить свои маршруты и ответы.

Фиктивные данные — это аналог настоящего источника данных, выполняющий по крайней мере некоторые из тех же функций. Примером может служить класс в оперативной памяти, имитирующий базу данных. В этой и следующих главах вам предстоит создать некоторое количество фиктивных данных, по мере того как вы будете заполнять код, определяющий уровни и их взаимодействие. В главе 10 вы определите реальное хранилище данных (базу данных), и оно заменит фиктивные данные.

Создание общих функций с помощью стека

Как и в примерах с данными, подход к созданию этого сайта является исследовательским. Часто бывает неясно, что в итоге понадобится, поэтому давайте начнем с некоторых характерных для подобных сайтов элементов. Для обеспечения доступа фронтенда к данным обычно требуются способы выполнения следующих запросов:

- *получить* один, некоторые, все (get one, some, all);
- *создать* (create);
- *заменить* (replace) полностью;
- *изменить* (modify) частично;
- *удалить* (delete).

По сути, это основы CRUD из баз данных, хотя я разделил букву U (модификация) на частичные (*изменение*) и полные (*замена*) функции. Возможно, это различие окажется излишним! Это зависит от того, куда ведут данные.

Создание фиктивных данных

Работая сверху вниз, вы будете дублировать некоторые функции на всех трех уровнях. Чтобы сэкономить на вводе текста, в примере 8.12 введу каталог верхнего уровня под названием `fake`. В нем находятся модули, предоставляющие фиктивные данные об исследователях и существах.

Пример 8.12. Новый модуль в файле `fake/explorer.py`

```
from model.explorer import Explorer

# фиктивные данные, в главе 10 они будут заменены на реальную базу данных и SQL
_explorers = [
    Explorer(name="Claude Hande",
              country="FR",
              description="Scarce during full moons"),
    Explorer(name="Noah Weiser",
              country="DE",
              description="Myopic machete man"),
]

def get_all() -> list[Explorer]:
    """Возврат всех исследователей"""
    return _explorers

def get_one(name: str) -> Explorer | None:
    for _explorer in _explorers:
        if _explorer.name == name:
            return _explorer
    return None

# Приведенные ниже варианты пока не функциональны,
# поэтому они просто делают вид, что работают,
# не изменяя реальный фиктивный список
def create(explorer: Explorer) -> Explorer:
    """Добавление исследователя"""
    return explorer

def modify(explorer: Explorer) -> Explorer:
    """Частичное изменение записи исследователя"""
    return explorer

def replace(explorer: Explorer) -> Explorer:
    """Полная замена записи исследователя"""
    return explorer
```

```
def delete(name: str) -> bool:
    """Удаление записи исследователя; возврат значения None,
    если запись существовала"""
    return None
```

Настройка существа в примере 8.13 аналогична.

Пример 8.13. Новый модуль в файле `fake/creature.py`

```
from model.creature import Creature

# фиктивные данные, пока не произойдет замена на реальную базу данных и SQL
_creatures = [
    Creature(name="Yeti",
              aka="Abominable Snowman",
              country="CN",
              area="Himalayas",
              description="Hirsute Himalayan"),
    Creature(name="Bigfoot",
              description="Yeti's Cousin Eddie",
              country="US",
              area="*",
              aka="Sasquatch"),
]

def get_all() -> list[Creature]:
    """Возврат всех существ"""
    return _creatures

def get_one(name: str) -> Creature | None:
    """Возврат одного существа"""
    for _creature in _creatures:
        if _creature.name == name:
            return _creature
    return None

# Приведенные ниже варианты пока не функциональны,
# поэтому они просто делают вид, что работают,
# не изменяя реальный фиктивный список
def create(creature: Creature) -> Creature:
    """Добавление существа"""
    return creature

def modify(creature: Creature) -> Creature:
    """Частичное изменение записи существа"""
    return creature
```

```
def replace(creature: Creature) -> Creature:
    """Полная замена записи существа"""
    return creature

def delete(name: str):
    """Удаление записи существа; возврат значения None,
    если запись существовала"""
    return None
```



Да, функции модулей практически идентичны. Они изменятся позже, когда появится настоящая база данных — она будет обрабатывать различные поля двух моделей. Кроме того, я использовал отдельные функции, а не определил абстрактный или класс Fake. У модуля собственное пространство имен, так что это эквивалентный способ объединения данных и функций.

Теперь изменим веб-функции из примеров 8.12 и 8.13. Готовясь к созданию последующих уровней (сервисного и данных), импортируйте только что определенный фиктивный провайдер данных, но назовите его `service` в строке `import fake.explorer as service` (пример 8.14). В главе 9 вы сделаете следующее:

- создадите новый файл `service/explorer.py`;
- импортируете туда фиктивные данные;
- укажете коду файла `web/explorer.py` импортировать новый модуль сервиса вместо фиктивного модуля.

В главе 10 вы проделаете то же самое на уровне данных. Все это сводится к добавлению частей программы и их соединению, при этом код переделывается как можно реже. Электричество (то есть настоящую базу данных и постоянные данные) вы включите позже, в главе 10.

Пример 8.14. Новые конечные точки в файле `web/explorer.py`

```
from fastapi import APIRouter
from model.explorer import Explorer
import fake.explorer as service

router = APIRouter(prefix = "/explorer")

@router.get("/")
def get_all() -> list[Explorer]:
    return service.get_all()
```

```

@router.get("/{name}")
def get_one(name) -> Explorer | None:
    return service.get_one(name)

# все остальные конечные точки пока ничего не делают:
@router.post("/")
def create(explorer: Explorer) -> Explorer:
    return service.create(explorer)

@router.patch("/")
def modify(explorer: Explorer) -> Explorer:
    return service.modify(explorer)

@router.put("/")
def replace(explorer: Explorer) -> Explorer:
    return service.replace(explorer)

@router.delete("/{name}")
def delete(name: str):
    return None

```

Теперь сделайте то же самое для конечных точек `/creature` (пример 8.15). Да, пока это похоже на вырезанный и вставленный код, но если сделать все заранее, это упростит внесение изменений в дальнейшем — а они всегда будут.

Пример 8.15. Новые конечные точки в файле `web/creature.py`

```

from fastapi import APIRouter
from model.creature import Creature
import fake.creature as service

router = APIRouter(prefix = "/creature")

@router.get("/")
def get_all() -> list[Creature]:
    return service.get_all()

@router.get("/{name}")
def get_one(name) -> Creature:
    return service.get_one(name)

# все остальные конечные точки пока ничего не делают:
@router.post("/")
def create(creature: Creature) -> Creature:
    return service.create(creature)

```

```
@router.patch("/")
def modify(creature: Creature) -> Creature:
    return service.modify(creature)

@router.put("/")
def replace(creature: Creature) -> Creature:
    return service.replace(creature)

@router.delete("/{name}")
def delete(name: str):
    return service.delete(name)
```

В последний раз мы обращались к файлу `main.py`, чтобы добавить субмаршрут для URL-адресов `/explorer`. Теперь добавим еще один для модуля `/creature` (пример 8.16).

Пример 8.16. Добавление субмаршрута существа в файл `main.py`

```
import uvicorn
from fastapi import FastAPI
from web import explorer, creature

app = FastAPI()

app.include_router(explorer.router)
app.include_router(creature.router)

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

Все сработало? Если вы набрали или вставили все точно, `Uvicorn` должен был перезапустить приложение. Попробуем провести несколько тестов вручную.

Тестируем!

В главе 12 будет показано, как использовать `pytest` для автоматизации тестирования на разных уровнях. В примерах 8.17–8.21 вручную выполняются несколько тестов веб-уровня для конечных точек исследователя с помощью `HTTPie`.

Пример 8.17. Тестирование конечной точки с инструкцией `Get All`

```
$ http -b localhost:8000/explorer/
[
  {
```



```

    "country": "FR",
    "name": "Claude Hande",
    "description": "Scarce during full moons"
  },
  {
    "country": "DE",
    "name": "Noah Weiser",
    "description": "Myopic machete man"
  }
]

```

Пример 8.18. Тестирование конечной точки с инструкцией Get One

```

$ http -b localhost:8000/explorer/"Noah Weiser"
{
  "country": "DE",
  "name": "Noah Weiser",
  "description": "Myopic machete man"
}

```

Пример 8.19. Тестирование конечной точки с инструкцией Replace

```

$ http -b PUT localhost:8000/explorer/"Noah Weiser"
{
  "country": "DE",
  "name": "Noah Weiser",
  "description": "Myopic machete man"
}

```

Пример 8.20. Тестирование конечной точки с инструкцией Modify

```

$ http -b PATCH localhost:8000/explorer/"Noah Weiser"
{
  "country": "DE",
  "name": "Noah Weiser",
  "description": "Myopic machete man"
}

```

Пример 8.21. Тестирование конечной точки с инструкцией Delete

```

$ http -b DELETE localhost:8000/explorer/Noah%20Weiser
true

$ http -b DELETE localhost:8000/explorer/Edmund%20Hillary
false

```

То же самое можно сделать для конечных точек в части /creature.

Использование форм автоматизированного тестирования FastAPI

Помимо выполняемых вручную тестов, которые я применял в большинстве примеров, FastAPI предоставляет очень хорошие автоматизированные формы тестирования в конечных точках `/docs` и `/redocs`. Это два разных стиля для одних и тех же сведений, поэтому я просто покажу немного информации, размещаемой на страницах `/docs` (рис. 8.1).

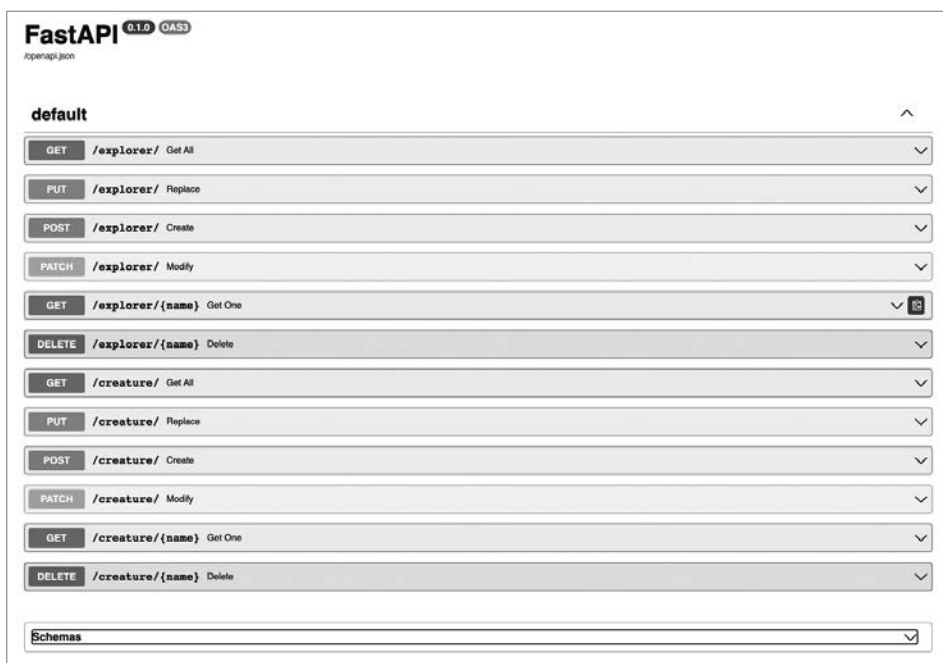


Рис. 8.1. Сгенерированная страница документации

Попробуйте выполнить первый тест.

1. Нажмите стрелку вниз, находящуюся справа под верхним разделом `GET /explorer/`. Откроется большая светло-голубая форма.
2. Нажмите синюю кнопку `Execute` (Выполнить) слева. На рис. 8.2 вы видите верхнюю часть результатов.

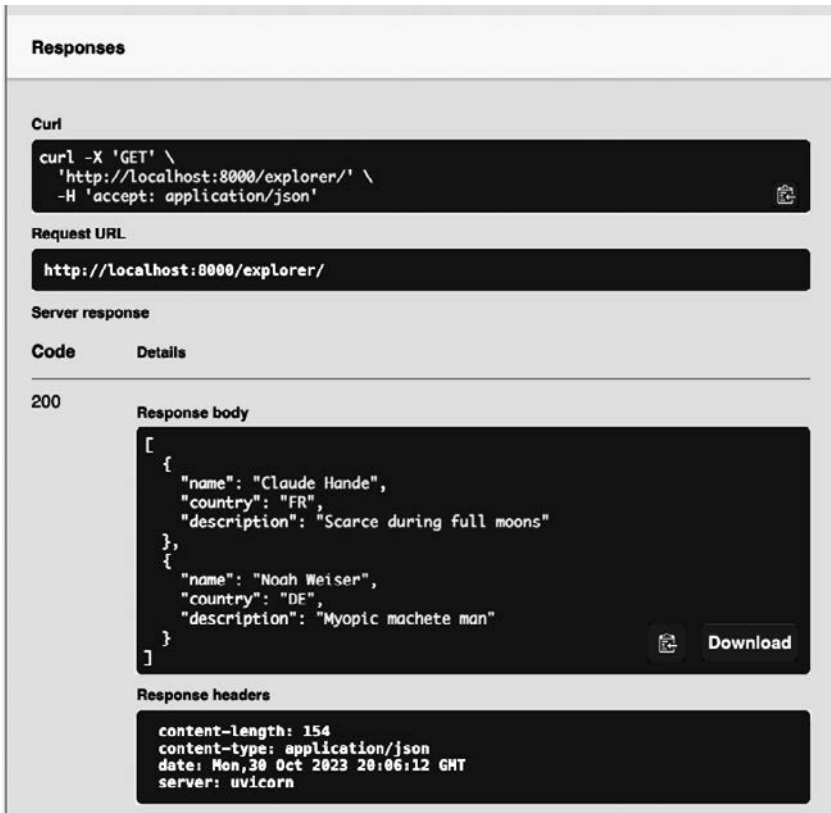


Рис. 8.2. Генерируемая страница результатов для GET /explorer/

В разделе Response body (Тело ответа) выводится текст в формате JSON, возвращаемый для фиктивных данных исследователя. Их мы определили ранее:

```
[
  {
    "name": "Claude Hande",
    "country": "FE",
    "description": "Scarce during full moons"
  },
  {
    "name": "Noah Weiser",
    "country": "DE",
    "description": "Myopic machete man"
  }
]
```

Попробуйте выполнить все остальные тесты. Для некоторых, таких как `GET /explorer/{name}`, нужно будет указать входное значение. Вы получите ответ на каждый из тестов (правда, некоторые так и останутся без ответа, пока не будет добавлен код базы данных). Можно повторить эти тесты в конце глав 9 и 10, чтобы убедиться, что никакие конвейеры данных не были повреждены при внесении изменений в код.

Общение с уровнями сервисов и данных

Когда функции на веб-уровне нужны данные, находящиеся под управлением уровня данных, она должна попросить уровень сервисов стать посредником. Это требует больше кода и может показаться ненужным, но это хорошая идея.

- Как гласит этикетка на банке, веб-уровень работает с Интернетом, а уровень данных — с внешними хранилищами данных и сервисами. Гораздо безопаснее хранить их данные по отдельности.
- Уровни можно тестировать независимо друг от друга. Механизм разделения уровней позволяет это сделать.



Для очень маленького сайта можно пропустить сервисный слой, если он ничего не делает. В главе 9 изначально определены сервисные функции, выполняющие лишь передачу запросов и ответов между веб-уровнем и уровнем данных. Хотя бы эти уровни необходимо разделить.

Что делает функция сервисного уровня? Узнаете в следующей главе. Подсказка: она разговаривает с уровнем данных, но тихим голосом, чтобы веб-уровень не понял, что именно она говорит. Также она определяет любую специфическую бизнес-логику, например взаимодействие между ресурсами. Чаще всего веб-уровень и уровень данных не должны заботиться о том, что происходит внутри.

(Уровень сервисов — это секретная служба.)

Пагинация и сортировка

В веб-интерфейсах, когда возвращаются многие или все сущности с URL-шаблонами, такими как `GET /resource`, часто требуется запросить поиск и возврат ресурсов:

- только одного;
- возможно, многих;
- всех.

Как заставить наш благонамеренный, но крайне прямолинейно мыслящий компьютер выполнять такие задачи? В первом случае, согласно описанному ранее шаблону RESTful, в URL-путь нужно включить идентификатор ресурса. При получении нескольких ресурсов может потребоваться увидеть результаты в определенном порядке.

- *Сортировка* — упорядочение всех результатов, даже если за один раз вы получите только часть из них.
- *Пагинация* — возвращение лишь некоторых результатов за раз с соблюдением любого типа сортировки.

В каждом случае группа параметров, задаваемых пользователем, указывает на то, что вам нужно. Обычно их задают в качестве параметров запроса. Вот несколько примеров.

- *Сортировка* GET /explorer?sort=country — получение всех исследователей, отсортированных по коду страны.
- *Пагинация* GET /explorer?offset=10&size=10 — возвращение из всего списка лишь записей исследователей (в данном случае неотсортированных), находящихся на позициях с 10-й по 19-ю.
- *Оба запроса* — GET /explorer?sort=country&offset=10&size=10.

Их можно задать в виде отдельных параметров запроса, и в этом вам поможет внедрение зависимостей FastAPI.

- Определите параметры сортировки и пагинации как модель Pydantic.
- Предоставьте модель параметров функции пути `get_all()` с функциональной возможностью `Depends` в аргументах функции пути.

Где должны располагаться сортировка и пагинация? Поначалу может показаться, что проще всего передавать все результаты запросов к базе данных на веб-уровень и использовать Python для обработки данных там. Но это не очень эффективный подход. Эти задачи обычно лучше всего решаются на уровне данных, потому что базы данных хорошо справляются с задачами такого типа. Я займусь кодом для них в главе 17. В ней содержится больше информации о базах данных, чем в главе 10.

Заключение

В этой главе вы подробнее узнали о том, о чем говорилось в главе 3 и др. С нее начался процесс создания полноценного сайта, содержащего информацию о воображаемых существах и их исследователях. Начиная с веб-уровня, вы определяете конечные точки с помощью декораторов путей FastAPI и функций пути. Последние собирают данные запроса, где бы они ни находились в байтах HTTP-запроса. Данные модели автоматически проверяются и подтверждаются Pydantic. Функции пути обычно передают аргументы соответствующим сервисным функциям, о которых речь пойдет в следующей главе.

Сервисный уровень

Что это было в середине?

*Отто Уэст (фильм
«Рыбка по имени Ванда»)*

Обзор

В этой главе рассказывается о сервисном уровне — среднем. Ущерб от протекающей крыши здания может вылиться в кругленькую сумму. Утечки в программном обеспечении не так очевидны, но исправление вызванных ими проблем может потребовать много времени и усилий. Как построить приложение так, чтобы на уровнях не возникало утечек? В частности, что должно и что не должно попадать на сервисный уровень, расположенный посередине?

Определение сервиса

Сервисный уровень — это сердце сайта, смысл его существования. Он принимает запросы из разных источников, получает доступ к данным, представляющим собой ДНК сайта, и возвращает ответы.

Общие шаблоны сервисов включают в себя сочетание следующих элементов:

- создать/извлечь/изменить (частично или полностью)/удалить;
- один/несколько элементов.

На уровне маршрутизатора RESTful существительные — это *ресурсы*. В этой книге ресурсы изначально будут включать в себя криптидов (воображаемых существ) и людей (исследователей криптидов). Позже можно будет определить связанные ресурсы, подобные следующим:

- локации;
- события (например, экспедиции, наблюдения).

Макет

Вот текущее расположение файлов и каталогов:

```
main.py web
├── __init__.py
├── creature.py ── explorer.py service
├── __init__.py
├── creature.py ── explorer.py
data
├── __init__.py
├── creature.py ── explorer.py model
├── __init__.py
├── creature.py ── explorer.py
fake
├── __init__.py
├── creature.py
├── explorer.py
└── test
```

В этой главе вы будете работать с файлами, находящимися в каталоге `service`.

Защита

Одна из приятных особенностей уровней заключается в том, что вам не нужно беспокоиться обо всем. Сервисный уровень заботится только о том, что входит на уровень данных и выходит с него. В главе 11 вы увидите, что более высокий уровень (здесь — веб-уровень) может справиться со всеми сложностями аутентификации и авторизации. Функции создания, изменения и удаления не должны быть широко открытыми, и даже для функций `get` со временем могут потребоваться некоторые ограничения.

Функции

Начнем с файла `creature.py`. На этом этапе потребности файла `explorer.py` будут почти такими же, и мы можем позаимствовать почти весь ранее написанный код. Так заманчиво написать один сервисный файл, работающий с обоими типами ресурсов, но почти неизбежно то, что в какой-то момент нам понадобится работать с ними по-разному.

Кроме того, сейчас сервисный файл представляет собой практически сквозной уровень. Это тот случай, когда создание небольшой дополнительной структуры на начальном этапе окупится впоследствии. Точно так же, как это делалось для файлов `web/creature.py` и `web/explorer.py` в главе 8, вы определите сервисные модули для обоих ресурсов и подключите их к соответствующим модулям фиктивных данных (примеры 9.1 и 9.2).

Пример 9.1. Начальный файл `service/creature.py`

```
from models.creature import Creature
import fake.creature as data

def get_all() -> list[Creature]:
    return data.get_all()

def get_one(name: str) -> Creature | None:
    return data.get(id)

def create(creature: Creature) -> Creature:
    return data.create(creature)

def replace(id, creature: Creature) -> Creature:
    return data.replace(id, creature)

def modify(id, creature: Creature) -> Creature:
    return data.modify(id, creature)

def delete(id, creature: Creature) -> bool:
    return data.delete(id)
```

Пример 9.2. Начальный файл `service/explorer.py`

```
from models.explorer import Explorer
import fake.explorer as data

def get_all() -> list[Explorer]:
    return data.get_all()
```

```
def get_one(name: str) -> Explorer | None:
    return data.get(name)

def create(explorer: Explorer) -> Explorer:
    return data.create(explorer)

def replace(id, explorer: Explorer) -> Explorer:
    return data.replace(id, explorer)

def modify(id, explorer: Explorer) -> Explorer:
    return data.modify(id, explorer)

def delete(id, explorer: Explorer) -> bool:
    return data.delete(id)
```



Синтаксис функции `get_one()`, возвращающий значение (`Creature | None`), требует наличия сборки Python начиная с версии 3.9. Для более ранних версий вам потребуется код `Optional`:

```
from typing import Optional
...
def get_one(name: str) -> Optional[Creature]:
...
```

Тестируем!

Теперь, когда кодовая база немного наполнилась, самое время внедрить автоматизированные тесты. (Все веб-тесты в предыдущей главе выполнялись вручную.) Итак, создадим несколько каталогов:

- **test** — каталог верхнего уровня наряду с **web**, **service**, **data** и **model**;
- **unit** — проверка отдельных функций без пересечения границ уровня;
- **web** — модульные тесты веб-уровня;
- **service** — модульные тесты сервисного уровня;
- **data** — модульные тесты уровня данных;
- **full** — также известны как *сквозные* или *контрактные* тесты, они охватывают все уровни сразу и обращаются к конечным точкам API на веб-уровне.

У каталогов будет префикс `test_` или суффикс `_test` для использования `pytest`, что показано в примере 9.4 (в нем выполняется тест из примера 9.3).

Прежде чем приступить к тестированию, необходимо выбрать несколько вариантов дизайна API. Что должна возвращать функция `get_one()` при отсутствии совпадений для ресурсов `Creature` или `Explorer`? Можно вернуть значение `None`, как в примере 9.2. Или вызвать исключение. Ни один из встроенных типов исключений Python не работает напрямую с отсутствующими значениями.

- `TypeError` может оказаться наиболее близким по смыслу, поскольку типы `None` и `Creature` различаются.
- `ValueError` больше подходит для неправильного значения для данного типа, но, наверное, можно сказать, что передача отсутствующей строки `id` в функцию `get_one(id)` подходит.
- Вы можете определить собственный тип `MissingError`, если действительно хотите этого.

Какой бы способ вы ни выбрали, результат будет сказываться на верхнем уровне.

Пока остановимся на варианте с `None`, а не на исключении. В конце концов, слово *none* означает именно отсутствие чего-то. Пример 9.3 представляет собой тест.

Пример 9.3. Сервисный тест `test/unit/service/test_creature.py`

```
from model.creature import Creature
from service import creature as code

sample = Creature(name="yeti",
                  country="CN",
                  area="Himalayas",
                  description="Hirsute Himalayan",
                  aka="Abominable Snowman",
                  )

def test_create():
    resp = code.create(sample)
    assert resp == sample

def test_get_exists():
    resp = code.get_one("yeti")
    assert resp == sample

def test_get_missing():
    resp = code.get_one("boxturtle")
    assert data is None
```

Запустите тест из примера 9.4.

Пример 9.4. Запуск сервисного теста

```
$ pytest -v test/unit/service/test_creature.py
test_creature.py::test_create PASSED [ 16%]
test_creature.py::test_get_exists PASSED [ 50%]
test_creature.py::test_get_missing PASSED [ 66%]

===== 3 passed in 0.06s =====
```



В главе 10 функция `get_one()` больше не будет возвращать значение `None` для отсутствующего существа, а выполнение теста `test_get_missing()` из примера 9.4 станет выдавать сбой. Но это будет исправлено.

Другие нюансы сервисного уровня

Сейчас мы находимся в середине стека — в части, действительно определяющей цель нашего сайта. И пока мы использовали этот уровень только для пересылки веб-запросов на уровень данных (см. следующую главу).

До сих пор в этой книге сайт развивался итеративно, создавая минимальную базу для дальнейшей работы. Узнав больше об имеющейся информации, о том, что вы способны с ней делать и что может понадобиться пользователям, можете развивать сайт и экспериментировать. Некоторые идеи могут оказаться полезными только для крупных сайтов, но вот несколько технических идей для сайта-помощника:

- ведение журналов;
- получение метрик;
- мониторинг;
- трассировка.

В этом разделе рассмотрим каждую из них. И вернемся к этим параметрам в разделе «Устранение неполадок» (в главе 13), чтобы узнать, могут ли они помочь в диагностике проблем.

Ведение журналов

FastAPI регистрирует каждый вызов API к конечной точке, включая метку времени, метод и URL-адрес, но не любые данные, переданные в теле или заголовках.

Метрики, мониторинг, наблюдаемость

Если у вас есть сайт, вы наверняка хотите знать, как он работает. Для веб-сайта с API может потребоваться узнать, к каким конечным точкам обращаются, сколько человек их посещают и т. д. Статистические данные о таких факторах называются *метриками*, а их сбор — *мониторингом* или *наблюдением*.

В настоящее время популярными инструментами для работы с метриками являются Prometheus (<https://prometheus.io>), предназначенный для сбора метрик, и Grafana (<https://grafana.com>) для их отображения.

Трассировка

Хорошо ли работает ваш сайт? Часто бывает, что метрики в целом хороши, но результаты то тут, то там разочаровывают. Или весь сайт может работать неудовлетворительно. В любом случае полезно иметь инструмент, измеряющий количество времени, затраченное на вызов API от начала и до конца, и не только общую продолжительность, но и длительность каждого промежуточного этапа. Если что-то работает медленно, вы можете найти слабое звено в цепи. Это называется *трассировкой*.

Новый проект с открытым исходным кодом взял на вооружение более ранние продукты для трассировки, такие как Jaeger (<https://www.jaegertracing.io>), и назвал их OpenTelemetry (<https://opentelemetry.io>). Он включает в себя Python API (<https://oreil.ly/gyL70>) и по крайней мере одну интеграцию с FastAPI (<https://oreil.ly/L6RXV>).

Чтобы установить и настроить OpenTelemetry с помощью Python, следуйте инструкциям, приводимым в документации OpenTelemetry Python (<https://oreil.ly/MBgd5>).

Другие возможности

Эксплуатационные вопросы будут рассмотрены в главе 13. Как насчет наших доменов-криптидов и всего, что с ними связано? Помимо голых подробностей об исследователях и существах, что еще вы могли бы взять на вооружение? У вас могут появиться новые идеи, требующие внесения изменений в модели и другие уровни. Можете попробовать вот такие:

- связь исследователей с существами, которых они ищут;
- данные наблюдений;
- экспедиции;

- фото и видео;
- кружки и футболки с изображением снежного человека.



Каждая из этих категорий, как правило, требует определения одной или нескольких новых моделей, а также новых модулей и функций. Некоторые из них будут добавлены в часть IV книги, представляющую собой галерею приложений, добавленных к базе, созданной в части III.

Заключение

В этой главе вы повторили некоторые функции из веб-слоя и перенесли фиктивные данные, с которыми они работали. Цель заключалась в том, чтобы инициировать создание нового сервисного слоя. До сих пор это был стандартный процесс, но теперь он будет развиваться и расходиться. В следующей главе создается уровень данных, в результате чего получается по-настоящему живой сайт.

Уровень данных

Если я не ошибаюсь, Дейта сыграла комика в сериале.

Брент Спайнер (фильм «Звездный путь: Следующее поколение»)

Обзор

В этой главе мы создаем постоянный дом для данных нашего сайта, наконец-то соединяя три уровня. В нем используется реляционная база данных SQLite и представлен API базы данных Python, метко названный DB-API. Базы данных, включая пакет SQLAlchemy и нереляционные базы данных, более подробно рассматриваются в главе 14.

DB-API

Уже более 20 лет в Python существует базовое определение интерфейса реляционной базы данных, называемое DB-API: PEP 249 (<https://oreil.ly/4Gp9T>). Любой, кто пишет Python-драйвер для реляционной базы данных, должен как минимум включить поддержку DB-API, хотя могут быть задействованы и другие возможности. Вот основные функции DB-API.

- Создание соединения `conn` с базой данных с помощью функции `connect()`.
- Создание курсора `curs` с помощью функции `conn.cursor()`.
- Выполнение строки SQL `stmt` с помощью функции `curs.execute(stmt)`.

Функции семейства `execute...()` выполняют строку SQL-оператора `stmt` с дополнительными параметрами, перечисленными далее:

- `execute(stmt)`, если параметров нет;
- `execute(stmt, params)` с параметрами `params` в одной последовательности (в списке или кортеже) или словаре;
- `executemany(stmt, params_seq)` с несколькими группами параметров в последовательности `params_seq`.

Существует пять способов указания параметров, но не все они поддерживаются всеми драйверами баз данных. Если оператор `stmt` начинается с выражения `"select * from creature where"` и необходимо задать строковые параметры `name` или `country` существа, оставшаяся часть строки `stmt` и ее параметры будут выглядеть так, как показано в табл. 10.1.

Таблица 10.1. Указание оператора и параметров

Тип	Часть, отображающая оператор	Часть, отображающая параметры
qmark	<code>name=? or country=?</code>	<code>(name, country)</code>
numeric	<code>name=:0 or country=:1</code>	<code>(name, country)</code>
format	<code>name=%s or country=%s</code>	<code>(name, country)</code>
named	<code>name=:name or country=:country</code>	<code>{"name": name, "country": country}</code>
pyformat	<code>name=%(name)s</code> or <code>country=%(country)s</code>	<code>{"name": name, "country": country}</code>

Первые три принимают аргумент в виде кортежа, где порядок параметров соответствует `?`, `:N` или `%s` в описании оператора. Последние два принимают словарь, в котором ключи соответствуют именам в операторе.

Таким образом, полный вызов в *именованном* стиле будет выглядеть так, как в примере 10.1.

Пример 10.1. Использование параметров в именованном стиле

```
stmt = """select * from creature where
    name=:name or country=:country"""
params = {"name": "yeti", "country": "CN"}
curs.execute(stmt, params)
```

Возвращаемое для SQL-операторов `INSERT`, `DELETE` и `UPDATE` из функции `execute()` значение рассказывает, как это работает. В случае с оператором `SELECT`

необходимо выполнить итерации над возвращаемыми строками данных как над кортежами Python с помощью метода `fetch`:

- `fetchone()` возвращает один кортеж, или значение `None`;
- `fetchall()` возвращает последовательность кортежей;
- `fetchmany(num)` возвращает до *num* кортежей.

SQLite

В стандартных пакетах Python есть поддержка одной базы данных (SQLite, <https://www.sqlite.org>) с помощью модуля `sqlite3` (<https://oreil.ly/CcYtJ>).

SQLite необычен — в нем нет отдельного сервера баз данных. Весь код находится в библиотеке, а хранение реализовано в одном файле. Другие базы данных работают на отдельных серверах, и клиенты общаются с ними с помощью TCP/IP, используя специальные протоколы. Задействуем SQLite в качестве первого физического хранилища данных для этого веб-сайта. В главе 14 речь пойдет о других базах данных, реляционных и нереляционных, а также о более продвинутых пакетах, таких как `SQLAlchemy`, и методах, подобных ORM.

Сначала необходимо определить, как структуры данных, использованные на сайте (*модели*), могут быть представлены в базе данных. До сих пор наши единственные модели были простыми и похожими, но не идентичными: `Creature` и `Explorer`. Они станут меняться по мере того, как мы будем придумывать, что с ними делать, и позволять данным развиваться без масштабных изменений кода.

В примере 10.2 показан голый код DB-API и SQL для создания первых таблиц и работы с ними. Он использует *именованные* строки аргументов (значения представляются как `name`), поддерживаемые пакетом `sqlite3`.

Пример 10.2. Создание файла `data/creature.py` с помощью `sqlite3`

```
import sqlite3
from model.creature import Creature

DB_NAME = "cryptid.db"
conn = sqlite3.connect(DB_NAME)
curs = conn.cursor()

def init():
    curs.execute("create table creature(name, description, country, area, aka")
```

```
def row_to_model(row: tuple) -> Creature:
    name, description, country, area, aka = row
    return Creature(name, description, country, area, aka)

def model_to_dict(creature: Creature) -> dict:
    return creature.dict()

def get_one(name: str) -> Creature:
    qry = "select * from creature where name=:name"
    params = {"name": name}
    curs.execute(qry, params)
    row = curs.fetchone()
    return row_to_model(row)

def get_all(name: str) -> list[Creature]:
    qry = "select * from creature"
    curs.execute(qry)
    rows = list(curs.fetchall())
    return [row_to_model(row) for row in rows]

def create(creature: Creature):
    qry = """insert into creature values
        (:name, :description, :country, :area, :aka)"""
    params = model_to_dict(creature)
    curs.execute(qry, params)

def modify(creature: Creature):
    return creature

def replace(creature: Creature):
    return creature

def delete(creature: Creature):
    qry = "delete from creature where name = :name"
    params = {"name": creature.name}
    curs.execute(qry, params)
```

В самом верху функция `init()` устанавливает соединение с `sqlite3` и базой данных `cryptid.db`. Она хранит его в переменной `conn` — глобальной для модуля `data/creature.py`. Далее переменная `curs` — это *курсор* для итерации по данным, возвращаемым при выполнении SQL-оператора `SELECT`. Она также является глобальной для модуля.

Две служебные функции выполняют перевод между моделями Pydantic и DB-API:

- `row_to_model()` преобразует кортеж, возвращаемый функцией `fetch`, в объект модели;
- `model_to_dict()` переводит Pydantic-модель в словарь, пригодный для использования в качестве именованного параметра запроса.

Фиктивные функции CRUD, хранящиеся на каждом уровне (веб → сервис → данные), теперь будут заменены. Они применяют только обычный SQL и методы DB-API в `sqlite3`.

Макет

До настоящего момента данные (фиктивные) изменялись поэтапно:

- в главе 8 мы составили фиктивный список *creatures* в файле `web/creature.py`;
- в главе 8 составили фиктивный список *explorers* в файле `web/explorer.py`;
- в главе 9 перенесли подделку *creatures* в каталог `service/creature.py`;
- в главе 9 перенесли подделку *explorers* в каталог `service/explorer.py`.

Теперь данные переместились в последний раз — в файл `data/creature.py`. Но это уже не подделка — это настоящие живые данные, хранящиеся в файле базы данных SQLite `cryptids.db`. Данные о существах, опять же из-за отсутствия воображения, хранятся в SQL-таблице `creature` в этой базе данных.

Как только вы сохраните этот новый файл, Unicorn должен перезапуститься из верхнего файла `main.py`, вызывающего `web/creature.py`, который вызывает файл `service/creature.py`, и наконец перейдет к новому файлу `data/creature.py`.

Заставляем все это работать

У нас есть одна небольшая проблема — этот модуль никогда не вызывает свою функцию `init()`, поэтому в SQLite нет переменных `conn` и `curs`, которые могли бы использовать другие функции. Это вопрос конфигурации: как предоставить информацию о базе данных при запуске? Возможны следующие варианты.

- Жесткое подключение информации о базе данных в коде, как в примере 10.2.
- Передача информации по уровням. Но это нарушило бы принцип разделения уровней — уровни веб и сервиса не должны знать о внутреннем устройстве уровня данных.
- Передача информации из другого внешнего источника:
 - файла конфигурации;
 - переменной окружения.

Переменная окружения проста и поддерживается такими рекомендациями, как Twelve-Factor App (<https://12factor.net/config>). Код может включать значение по умолчанию, если переменная окружения не определена. Этот подход можно использовать также при тестировании, чтобы создать отдельную от рабочей тестовую базу данных.

В примере 10.3 определим переменную окружения `CRYPTID_SQLITE_DB` и присвоим ей значение по умолчанию `cryptid.db`. Создайте новый файл `data/init.py` для нового кода инициализации базы данных, чтобы его можно было использовать и для кода исследователя.

Пример 10.3. Новый модуль инициализации базы данных `data/init.py`

```
"""Инициализация базы данных SQLite"""

import os
from pathlib import Path
from sqlite3 import connect, Connection, Cursor, IntegrityError

conn: Connection | None = None
curs: Cursor | None = None

def get_db(name: str|None = None, reset: bool = False):
    """Подключение к файлу БД SQLite"""
    global conn, curs
    if conn:
        if not reset:
            return
        conn = None
    if not name:
        name = os.getenv("CRYPTID_SQLITE_DB")
        top_dir = Path(__file__).resolve().parents[1] # repo top
        db_dir = top_dir / "db"
        db_name = "cryptid.db"
        db_path = str(db_dir / db_name)
        name = os.getenv("CRYPTID_SQLITE_DB", db_path)
    conn = connect(name, check_same_thread=False)
    curs = conn.cursor()

get_db()
```

Модуль Python — это *синглтон*, вызываемый только один раз, несмотря на многократный импорт. Таким образом, код инициализации в файле `init.py` запускается всего один раз, когда происходит его первый импорт.

Наконец, измените файл `data/creature.py` в примере 10.4, чтобы вместо него использовать новый модуль.

- Главное, уберите строки с четвертой по восьмую.
- О, в первую очередь создайте таблицу `creature`!
- Все поля таблицы являются строками `text SQL`. Это тип столбца по умолчанию в `SQLite`, в отличие от большинства баз данных `SQL`, поэтому вам не нужно было включать `text` ранее, но указать явно не помешает.
- Выражение `if not exists` позволяет избежать разрушения таблицы после ее создания.
- Поле `name` служит явным первичным ключом (`primary key`) для этой таблицы. Если в ней будет храниться много данных исследователя, этот ключ будет необходим для быстрого поиска. Альтернативой может стать ужасное *сканирование таблицы*, когда код базы данных должен просмотреть каждую строку, пока не найдет совпадение с полем `name`.

Пример 10.4. Добавление конфигурации базы данных в файл `data/creature.py`

```
from .init import conn, curs
from model.creature import Creature

curs.execute("""create table if not exists creature(
    name text primary key,
    description text,
    country text,
    area text,
    aka text)""")

def row_to_model(row: tuple) -> Creature:
    (name, description, country, area, aka) = row
    return Creature(name, description, country, area, aka)

def model_to_dict(creature: Creature) -> dict:
    return creature.dict()

def get_one(name: str) -> Creature:
    qry = "select * from creature where name=:name"
    params = {"name": name}
    curs.execute(qry, params)
    return row_to_model(curs.fetchone())

def get_all() -> list[Creature]:
    qry = "select * from creature"
    curs.execute(qry)
    return [row_to_model(row) for row in curs.fetchall()]
```

```
def create(creature: Creature) -> Creature:
    qry = "insert into creature values"
        "(:name, :description, :country, :area, :aka)"
    params = model_to_dict(creature)
    curs.execute(qry, params)
    return get_one(creature.name)

def modify(creature: Creature) -> Creature:
    qry = """update creature
        set country=:country,
            name=:name,
            description=:description,
            area=:area,
            aka=:aka
        where name=:name_orig"""
    params = model_to_dict(creature)
    params["name_orig"] = creature.name
    _ = curs.execute(qry, params)
    return get_one(creature.name)

def delete(creature: Creature) -> bool:
    qry = "delete from creature where name = :name"
    params = {"name": creature.name}
    res = curs.execute(qry, params)
    return bool(res)
```

При импорте объектов `conn` и `curs` из файла `init.py` файлу `data/creature.py` больше нет необходимости импортировать сам модуль `sqlite3`. Если только однажды не потребуется вызвать другой метод `sqlite3`, не являющийся методом объекта `conn` или `curs`.

Опять же эти изменения должны указать Uvicorn перезагрузить все. С этого момента тестирование с помощью любого из описанных ранее методов (HTTPIe и подобные ему или автоматические формы `/docs`) будет показывать сохраняемые данные. Если вы добавите существо, то оно появится в следующий раз, когда вы соберете их всех.

Сделаем то же самое для исследователей в примере 10.5.

Пример 10.5. Добавление конфигурации базы данных в файл `data/explorer.py`

```
from .init import curs
from model.explorer import Explorer
```

```
curs.execute("""create table if not exists explorer(
    name text primary key,
    country text,
    description text)""")

def row_to_model(row: tuple) -> Explorer:
    return Explorer(name=row[0], country=row[1], description=row[2])

def model_to_dict(explorer: Explorer) -> dict:
    return explorer.dict() if explorer else None

def get_one(name: str) -> Explorer:
    qry = "select * from explorer where name=:name"
    params = {"name": name}
    curs.execute(qry, params)
    return row_to_model(curs.fetchone())

def get_all() -> list[Explorer]:
    qry = "select * from explorer"
    curs.execute(qry)
    return [row_to_model(row) for row in curs.fetchall()]

def create(explorer: Explorer) -> Explorer:
    qry = """insert into explorer (name, country, description)
        values (:name, :country, :description)"""
    params = model_to_dict(explorer)
    _ = curs.execute(qry, params)
    return get_one(explorer.name)

def modify(name: str, explorer: Explorer) -> Explorer:
    qry = """update explorer
        set country=:country,
        name=:name,
        description=:description
        where name=:name_orig"""
    params = model_to_dict(explorer)
    params["name_orig"] = explorer.name
    _ = curs.execute(qry, params)
    explorer2 = get_one(explorer.name)
    return explorer2

def delete(explorer: Explorer) -> bool:
    qry = "delete from explorer where name = :name"
    params = {"name": explorer.name}
    res = curs.execute(qry, params)
    return bool(res)
```

Тестируем!

Было введено очень много кода без тестов. Все ли работает? Я бы удивился, если бы это было так. Итак, создадим несколько тестов.

Сделайте в каталоге `test` следующие подкаталоги:

- `unit` — внутри уровня;
- `full` — по всем уровням.

Какой тип следует написать и запустить первым? Большинство людей сначала пишут автоматизированные модульные тесты — они меньше, а всех остальных частей слоя может еще не существовать. В этой книге разработка велась сверху вниз, и сейчас мы завершаем последний слой. Кроме того, в главах 8 и 9 мы тестировали вручную с помощью HTTPie и похожих инструментов. Они помогают быстро выявить ошибки и упущения. Автоматизированные тесты гарантируют, что вы не будете повторять те же ошибки в дальнейшем. Поэтому я рекомендую:

- провести несколько тестов вручную в процессе написания кода;
- выполнить модульные тесты после исправления синтаксических ошибок Python;
- провести полное тестирование после того, как будет получен полный поток данных на всех уровнях.

Полные тесты

Они вызывают конечные веб-точки, спускающие лифт кода вниз, через сервисный уровень к уровню данных, и поднимающие обратно вверх. Иногда их называют *сквозными* или *контрактными* тестами.

Получение всех исследователей

Окунуться в тестовые воды, еще не зная, кишат ли они пираньями, сможет смелый доброволец — пример 10.6.

Пример 10.6. Тестирование получения всех исследователей

```
$ http localhost:8000/explorer
HTTP/1.1 405 Method Not Allowed
allow: POST
```



```
content-length: 31
content-type: application/json
date: Mon, 27 Feb 2023 20:05:18 GMT
server: uvicorn
```

```
{
  "detail": "Method Not Allowed"
}
```

Ух ты! Что же произошло?

Ох. Тест запрашивает путь `/explorer`, а не `/explorer/`, а также отсутствует GET-метод функции пути для URL-адреса `/explorer` (без завершающей косой черты). В файле `web/explorer.py` декоратор пути для функции пути `get_all()` имеет вид:

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

Это плюс предыдущий код:

```
router = APIRouter(prefix = "/explorer")
```

означает, что функция пути `get_all()` предоставляет URL-адрес, содержащий выражение `/explorer/`.

Пример 10.7 показывает, что на одну функцию пути может приходиться более одного декоратора пути.

Пример 10.7. Добавление декоратора пути без косой черты для функции пути `get_all()`

```
@router.get("")
@router.get("/")
def get_all() -> list[Explorer]:
    return service.get_all()
```

Протестируем оба URL-адреса в примерах 10.8 и 10.9.

Пример 10.8. Тестирование конечной точки без косой черты в конце

```
$ http localhost:8000/explorer
HTTP/1.1 200 OK
content-length: 2
content-type: application/json
date: Mon, 27 Feb 2023 20:12:44 GMT
server: uvicorn
```

```
[]
```

Пример 10.9. Тестирование конечной точки с косой чертой в конце

```
$ http localhost:8000/explorer/  
HTTP/1.1 200 OK  
content-length: 2  
content-type: application/json  
date: Mon, 27 Feb 2023 20:14:39 GMT  
server: uvicorn
```

```
[]
```

Теперь, когда оба варианта работают, создайте объект исследователя и повторите тест получения всех ресурсов. В примере 10.10 предпринята подобная попытка, но с поворотом сюжета.

Пример 10.10. Создание тестового исследователя с ошибкой ввода

```
$ http post localhost:8000/explorer name="Beau Buffette", contry="US"  
HTTP/1.1 422 Unprocessable Entity  
content-length: 95  
content-type: application/json  
date: Mon, 27 Feb 2023 20:17:45 GMT  
server: uvicorn
```

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

Я написал слово `country` с ошибкой, хотя моя орфография обычно безупречна. Pydantic обнаружил это на веб-уровне, вернув HTTP код состояния 422 и описание проблемы. В общем, если FastAPI возвращает код 422, велика вероятность того, что Pydantic нашел виновника сбоя. Часть выражения `"loc"` указывает, где произошла ошибка: поле `"country"` ошибочно, потому что я так плохо печатаю.

Исправьте орфографию и проведите повторный тест (пример 10.11).

Пример 10.11. Создание исследователя
с исправленным значением

```
$ http post localhost:8000/explorer name="Beau Buffette" country="US"
```

```
HTTP/1.1 201 Created
```

```
content-length: 55
```

```
content-type: application/json
```

```
date: Mon, 27 Feb 2023 20:20:49 GMT
```

```
server: uvicorn
```

```
{  
  "name": "Beau Buffette",  
  "country": "US",  
  "description": ""  
}
```

На этот раз вызов возвращает код статуса **201**. Он традиционно получается при создании ресурса (все коды статуса группы 2xx считаются признаком успеха, а наиболее распространен простой код **200**). Ответ также содержит JSON-версию только что созданного объекта Explorer.

А теперь вернемся к начальному тесту: появится ли имя Beau в тестировании по получению всех записей исследователей? Пример 10.12 отвечает на этот животрепещущий вопрос.

Пример 10.12. Работает ли последняя функция create()?

```
$ http localhost:8000/explorer
```

```
HTTP/1.1 200 OK
```

```
content-length: 57
```

```
content-type: application/json
```

```
date: Mon, 27 Feb 2023 20:26:26 GMT
```

```
server: uvicorn
```

```
[  
  {  
    "name": "Beau Buffette",  
    "country": "US",  
    "description": ""  
  }  
]
```

Отлично!

Получите записи одного исследователя

Что произойдет, если вы попытаетесь найти Beau с помощью конечной точки для получения одной записи — Get One (пример 10.13)?

Пример 10.13. Тестирование конечной точки с инструкцией Get One

```
HTTP/1.1 200 OK
content-length: 55
content-type: application/json
date: Mon, 27 Feb 2023 20:28:48 GMT
server: uvicorn
```

```
{
  "name": "Beau Buffette",
  "country": "US",
  "description": ""
}
```

Я использовал кавычки, чтобы сохранить пробел между именем и фамилией. В URL-адресах вы можете применять также написание вида `Beau%20Buffette`. Выражение `%20` означает символ пробела в шестнадцатеричном коде стандарта ASCII.

Отсутствующие и дублированные данные

До сих пор я игнорировал два основных класса ошибок:

- *отсутствующие данные* — если вы пытаетесь получить, изменить или удалить запись исследователя с именем, которого нет в базе данных;
- *дублированные данные* — если попытаетесь создать запись исследователя с одним и тем же именем более одного раза.

Что же делать, если вы запрашиваете запись несуществующего или продублированного исследователя? Пока что код слишком оптимистичен, и исключения будут всплывать из бездны.

Наш друг по имени Beau только что был добавлен в базу данных. Представьте, что его злобный клон, который носит то же имя, замышляет подменить его темной ночью, используя пример 10.14.

Пример 10.14. Ошибка дублирования — попытка создать исследователя более одного раза

```
$ http post localhost:8000/explorer name="Beau Buffette" country="US"
HTTP/1.1 500 Internal Server Error
```

```
content-length: 3127
content-type: text/plain; charset=utf-8
date: Mon, 27 Feb 2023 21:04:09 GMT
server: uvicorn
```

```
Traceback (most recent call last):
  File ".../starlette/middleware/errors.py", line 162, in call
... (lots of confusing innards here) ...
  File ".../service/explorer.py", line 11, in create
    return data.create(explorer)
           ^^^^^^^
  File ".../data/explorer.py", line 37, in create
    curs.execute(qry, params)
sqlite3.IntegrityError: UNIQUE constraint failed: explorer.name
```

Я опустил большинство строк в этой трассировке ошибок и заменил некоторые части многоточиями, потому что данные содержат в основном внутренние вызовы, выполняемые FastAPI и базовым Starlette. Но вот последняя строка — исключение SQLite в веб-слое! Где кушетка для обмороков?

По пятам за этим следует еще один ужас — исчезновение исследователя (пример 10.15).

Пример 10.15. Получение несуществующего исследователя

```
$ http localhost:8000/explorer/"Beau Buffalo"
HTTP/1.1 500 Internal Server Error
content-length: 3282
content-type: text/plain; charset=utf-8
date: Mon, 27 Feb 2023 21:09:37 GMT
server: uvicorn
```

```
Traceback (most recent call last):
  File ".../starlette/middleware/errors.py", line 162, in call
... (many lines of ancient cuneiform) ...
  File ".../data/explorer.py", line 11, in row_to_model
    name, country, description = row
    ^^^^^^^
```

```
TypeError: cannot unpack non-iterable NoneType object
```

Каков хороший способ выявить их на нижнем (данные) уровне и передать детали на верхний (веб)? Возможны следующие варианты.

- Пусть SQLite выкашливает комок волос (исключение) и разбирается с ним на веб-уровне.

Но! Это смешивает уровни, что *плохо*. Веб-уровень не должен ничего знать о конкретных базах данных.

- Сделайте так, чтобы все функции на сервисном уровне и уровне данных возвращали `Explorer | None` там, где раньше они возвращали объект `Explorer`. В таком случае `None` будет означать отказ. (Вы можете урезать это, определив `OptExplorer = Explorer | None` в файле `model/explorer.py`.)

Но! Функция могла не сработать по нескольким причинам, и вам могут понадобиться подробности. А это требует редактирования большого количества кода.

- Определите исключения для утраченных (`Missing`) и продублированных (`Duplicate`) данных, включая более детальное описание проблемы. Они будут проходить через все уровни без изменений в коде, пока функции пути веб-уровня не поймут их. Кроме того, они зависят от приложения, а не от базы данных, что позволяет сохранить неприкосновенность уровней.

Но! На самом деле мне нравится этот вариант, так что он пойдет в пример 10.16.

Пример 10.16. Определение нового файла `errors.py` верхнего уровня

```
class Missing(Exception):
    def __init__(self, msg:str):
        self.msg = msg

class Duplicate(Exception):
    def __init__(self, msg:str):
        self.msg = msg
```

У каждого из этих исключений есть строковый атрибут `msg`. Он может сообщить коду более высокого уровня о том, что произошло.

Чтобы реализовать это, в примере 10.17 импортируйте исключение `DB-API` в файл `data/init.py`. `SQLite` вызовет его при дублировании данных.

Пример 10.17. Добавление импорта исключений `SQLite` в файл `data/init.py`

```
from sqlite3 import connect, IntegrityError
```

Импортируйте и отловите эту ошибку в примере 10.18.

Пример 10.18. Изменение `data/explorer.py`, чтобы получить возможность отлавливать и выбрасывать исключения

```
from init import (conn, curs, IntegrityError)
from model.explorer import Explorer
from error import Missing, Duplicate

curs.execute("""create table if not exists explorer(
    name text primary key,
```

```

        country text,
        description text)""")

def row_to_model(row: tuple) -> Explorer:
    name, country, description = row
    return Explorer(name=name,
                    country=country, description=description)

def model_to_dict(explorer: Explorer) -> dict:
    return explorer.dict()

def get_one(name: str) -> Explorer:
    qry = "select * from explorer where name=:name"
    params = {"name": name}
    curs.execute(qry, params)
    row = curs.fetchone()
    if row:
        return row_to_model(row)
    else:
        raise Missing(msg=f"Explorer {name} not found")

def get_all() -> list[Explorer]:
    qry = "select * from explorer"
    curs.execute(qry)
    return [row_to_model(row) for row in curs.fetchall()]

def create(explorer: Explorer) -> Explorer:
    if not explorer: return None
    qry = """insert into explorer (name, country, description) values
        (:name, :country, :description)"""
    params = model_to_dict(explorer)
    try:
        curs.execute(qry, params)
    except IntegrityError:
        raise Duplicate(msg=
            f"Explorer {explorer.name} already exists")
    return get_one(explorer.name)

def modify(name: str, explorer: Explorer) -> Explorer:
    if not (name and explorer): return None
    qry = """update explorer
        set name=:name,
        country=:country,
        description=:description
        where name=:name_orig"""
    params = model_to_dict(explorer)
    params["name_orig"] = explorer.name
    curs.execute(qry, params)
    if curs.rowcount == 1:

```

```
        return get_one(explorer.name)
    else:
        raise Missing(msg=f"Explorer {name} not found")

def delete(name: str):
    if not name: return False
    qry = "delete from explorer where name = :name"
    params = {"name": name}
    curs.execute(qry, params)
    if curs.rowcount != 1:
        raise Missing(msg=f"Explorer {name} not found")
```

Это избавляет от необходимости объявлять, что все функции возвращают выражение `Explorer | None` или `Optional[Explorer]`.

Вы указываете подсказки типов только для обычных типов возврата, но не для исключений. Поскольку исключения распространяются вверх независимо от стека вызовов до тех пор, пока кто-то их не отловит, вам не придется ничего менять на сервисном уровне. А вот новый файл `web/explorer.py` с обработчиками исключений и соответствующим возвратом кода состояния HTTP (пример 10.19).

Пример 10.19. Обработка Missing и Duplicate исключений в файле `web/explorer.py`

```
from fastapi import APIRouter, HTTPException
from model.explorer import Explorer
from service import explorer as service
from error import Duplicate, Missing

router = APIRouter(prefix = "/explorer")

@router.get("")
@router.get("/")
def get_all() -> list[Explorer]:
    return service.get_all()

@router.get("/{name}")
def get_one(name) -> Explorer:
    try:
        return service.get_one(name)
    except Missing as exc:
        raise HTTPException(status_code=404, detail=exc.msg)

@router.post("", status_code=201)
@router.post("/", status_code=201)
def create(explorer: Explorer) -> Explorer:
    try:
        return service.create(explorer)
```



```

except Duplicate as exc:
    raise HTTPException(status_code=404, detail=exc.msg)

@router.patch("/")
def modify(name: str, explorer: Explorer) -> Explorer:
    try:
        return service.modify(name, explorer)
    except Missing as exc:
        raise HTTPException(status_code=404, detail=exc.msg)

@router.delete("/{name}", status_code=204)
def delete(name: str):
    try:
        return service.delete(name)
    except Missing as exc:
        raise HTTPException(status_code=404, detail=exc.msg)

```

Проверьте эти изменения в примере 10.20.

Пример 10.20. Повторное тестирование конечной точки Get One с отсутствующей записью исследователя и с новым исключением Missing

```
$ http localhost:8000/explorer/"Beau Buffalo"
```

```

HTTP/1.1 404 Not Found
content-length: 44
content-type: application/json
date: Mon, 27 Feb 2023 21:11:27 GMT
server: uvicorn

```

```

{
  "detail": "Explorer Beau Buffalo not found"
}

```

Хорошо. Теперь попробуйте повторить попытку создания злого клона (пример 10.21).

Пример 10.21. Тестирование исправления дублирования

```
$ http post localhost:8000/explorer name="Beau Buffette" country="US"
```

```

HTTP/1.1 404 Not Found
content-length: 50
content-type: application/json
date: Mon, 27 Feb 2023 21:14:00 GMT
server: uvicorn

```

```

{
  "detail": "Explorer Beau Buffette already exists"
}

```

Тестирование запросов отсутствующих данных будет применяться также к конечным точкам изменения (**Modify**) и удаления (**Delete**). Можете попробовать написать для них аналогичные тесты самостоятельно.

Модульное тестирование

Модульное тестирование работает только с уровнем данных, проверяя вызовы базы данных и синтаксис SQL. Я поместил этот раздел после полных тестов, потому что хотел, чтобы исключения **Missing** и **Duplicate** уже были определены, объяснены и закодированы в файле `data/creature.py`. В примере 10.22 приведен скрипт тестирования `test/unit/data/test_creature.py`. Вот на что стоит обратить внимание.

- Вы присваиваете переменной окружения `CRYPTID_SQLITE_DATABASE` значение `":memory:"` до импорта `init` или `creature` из `data`. Это значение указывает SQLite, что необходимо работать исключительно в памяти, не захламляя существующий файл базы данных и даже не создавая файл на диске. Оно проверяется в файле `data/init.py` при первом импорте этого модуля.
- *Фикстура* под названием `sample` передается функциям, которым нужен объект `Creature`.
- Тесты выполняются по порядку. В этом случае одна и та же база данных сохраняется на протяжении всего времени выполнения, а не сбрасывается между функциями. Это необходимо для того, чтобы изменения, внесенные предыдущими функциями, сохранились. В `pytest` у фикстуры могут быть следующие параметры:
 - *область действия функции* (по умолчанию) — вызывается заново перед каждой тестовой функцией;
 - *область действия сессии* — вызывается только один раз, в самом начале.
- Некоторые тесты принудительно вызывают исключения **Missing** или **Duplicate** и проверяют возможность их отлавливания.

Итак, каждый из тестов получает совершенно новый неизменный объект `Creature` с именем `sample` (пример 10.22).

Пример 10.22. Модульные тесты для файла `data/creature.py`

```
import os
import pytest
from model.creature import Creature
from error import Missing, Duplicate
```

```

# set this before data imports below for data.init
os.environ["CRYPTID_SQLITE_DB"] = ":memory:"
from data import creature

@pytest.fixture
def sample() -> Creature:
    return Creature(name="yeti", country="CN", area="Himalayas",
                    description="Harmless Himalayan",
                    aka="Abominable Snowman")

def test_create(sample):
    resp = creature.create(sample)
    assert resp == sample

def test_create_duplicate(sample):
    with pytest.raises(Duplicate):
        _ = creature.create(sample)

def test_get_one(sample):
    resp = creature.get_one(sample.name)
    assert resp == sample

def test_get_one_missing():
    with pytest.raises(Missing):
        _ = creature.get_one("boxturtle")

def test_modify(sample):
    creature.area = "Sesame Street"
    resp = creature.modify(sample.name, sample)
    assert resp == sample

def test_modify_missing():
    thing: Creature = Creature(name="snurfle", country="RU", area="",
                               description="some thing", aka="")
    with pytest.raises(Missing):
        _ = creature.modify(thing.name, thing)

def test_delete(sample):
    resp = creature.delete(sample.name)
    assert resp is None

def test_delete_missing(sample):
    with pytest.raises(Missing):
        _ = creature.delete(sample.name)

```

Подсказка: можете сделать собственную версию `test/unit/data/test_explorer.py`.

Заключение

В этой главе был представлен простой уровень обработки данных с несколькими переходами вверх и вниз по стеку уровней по мере необходимости. В главе 12 рассматриваются модульные тесты для каждого уровня, а также тесты межуровневой интеграции и полные сквозные тесты. Глава 14 посвящена более глубокому изучению баз данных и подробным примерам.

Аутентификация и авторизация

Уважай мою власть!

Эрик Картман
(мультсериал «Южный парк»)

Обзор

Иногда веб-сайт предоставляет широкий доступ, и любой посетитель может зайти на любую страницу. Но если содержимое сайта может быть изменено, некоторые конечные точки будут ограничены для определенных людей или групп. Если бы каждый мог изменять страницы на Amazon, представьте, сколько странных товаров появилось бы на них и какие удивительные покупки получили бы некоторые люди. К сожалению, такова человеческая природа — некоторые люди пользуются остальными, уплачивающими скрытый налог за свои действия.

Стоит ли оставить наш сайт с криптодами открытым для доступа любых пользователей к любой конечной точке? Нет! Практически любой крупный веб-сервис в конечном счете должен решать следующие задачи.

- *Аутентификация* (authn). Кто вы?
- *Авторизация* (authz). Что вам нужно?

Должен ли код аутентификации и авторизации (auth) получить собственный новый уровень, скажем, между сервисным и веб-уровнями? Или же все должно решаться сервисным или веб-уровнем самостоятельно? В этой главе вы узнаете о методах аутентификации и о том, где их использовать.

Часто описания веб-безопасности кажутся более запутанными, чем нужно.

Злоумышленники могут быть очень, очень хитрыми, а меры противодействия — непростыми.



Как я уже не раз упоминал, официальная документация по FastAPI превосходна. Попробуйте изучить раздел Security section (<https://oreil.ly/oYsKl>), если информация данной главы покажется вам недостаточно подробной.

Итак, давайте разберемся с этим вопросом пошагово. Я начну с простых техник, предназначенных только для подключения auth-системы к конечной точке веб-сайта для тестирования, но на публичном сайте все это применяться не будет.

Немного отвлечемся. Нужна ли вам аутентификация?

Повторю: *аутентификация* связана с *идентификацией* — кто вы? Чтобы реализовать аутентификацию, необходимо сопоставить секретную информацию с уникальной идентификацией. Существуют разные способы реализовать это со *множеством* уровней сложности. Давайте начнем с малого и будем работать на усложнение.

Часто в книгах и статьях по веб-разработке сразу же переходят к деталям аутентификации и авторизации, иногда путая их. Порой в таких материалах пропущен первый вопрос: действительно ли вам нужно и то и другое?

Вы можете предоставить полностью анонимный доступ ко всем страницам своего сайта. Но это оставит открытую возможность для таких уязвимостей, как атаки типа «отказ в обслуживании» (DoS). Хотя некоторые меры защиты, например ограничение количества запросов, можно реализовать вне веб-сервера (см. главу 13), почти все поставщики публичных API требуют наличия хотя бы какой-то аутентификации. Помимо безопасности, необходимо знать, насколько эффективны веб-сайты.

- Сколько уникальных посетителей?
- Какие страницы пользуются наибольшей популярностью?
- Увеличивают ли определенные изменения количество просмотров?
- Какие последовательности посещения страниц часто встречаются?

Ответы на эти вопросы требуют аутентификации конкретных посетителей. В противном случае вы можете получить только общее количество значений по всем показателям.



Если ваш сайт требует аутентификации или авторизации, доступ к нему должен быть зашифрован (с использованием HTTPS вместо HTTP), чтобы злоумышленники не смогли извлечь секретные данные из обычного текста. Подробные сведения о настройке HTTPS см. в главе 13.

Методы аутентификации

Существует множество методов и инструментов веб-аутентификации:

- *имя пользователя/электронная почта и пароль* — применение классических HTTP Basic и дайджест-аутентификации;
- *ключ API* — неясная длинная строка с сопутствующим секретом;
- *OAuth2* — набор стандартов для аутентификации и авторизации;
- *веб-токены JavaScript* (JavaScript Web Tokens, JWT) — формат кодирования, содержащий криптографически подписанную информацию о пользователе.

В этом разделе я рассмотрю первые два метода и покажу их традиционную реализацию. Но остановлюсь перед тем, как заполнить код API и базы данных. Вместо этого мы полностью реализуем более современную схему с OAuth2 и JWT.

Глобальная аутентификация — секретный ключ или общий секрет (Shared Secret)

Самый простой метод аутентификации заключается в передаче секрета, который обычно известен только веб-серверу. Если он совпадает, доступ открывается. Это небезопасно, если ваш сайт API открыт для общего доступа по протоколу HTTP, а не HTTPS. Если он скрыт за открытым фронтенд-сайтом, фронтенд- и бэкенд-части системы могут взаимодействовать, используя общий постоянный секрет. Но если фронтенд-сайт взломают, всем конец. Давайте посмотрим, как FastAPI обрабатывает простую аутентификацию.

Создайте новый файл верхнего уровня под названием `auth.py`. Убедитесь, что у вас нет другого сервера FastAPI, запущенного из одного из постоянно

меняющихся файлов `main.py` из предыдущих глав. В примере 11.1 реализован сервер, просто возвращающий все записи `username` и `password`, отправленные ему с помощью HTTP Basic Authentication — метода из первых дней существования Интернета.

Пример 11.1. Применение HTTP Basic Auth для получения информации о пользователе: `auth.py`

```
import uvicorn
from fastapi import Depends, FastAPI
from fastapi.security import HTTPBasic, HTTPBasicCredentials

app = FastAPI()

basic = HTTPBasic()

@app.get("/who")
def get_user(
    creds: HTTPBasicCredentials = Depends(basic)):
    return {"username": creds.username, "password": creds.password}

if __name__ == "__main__":
    uvicorn.run("auth:app", reload=True)
```

В примере 11.2 укажите HTTPie выполнить этот запрос Basic Auth (для этого требуются аргументы `-a name:password`). Здесь мы используем название `me` и пароль `secret`.

Пример 11.2. Проверка с помощью HTTPie

```
$ http -q -a me:secret localhost:8000/who
{
  "password": "secret",
  "username": "me"
}
```

Тестирование с помощью пакета `Requests` в примере 11.3 аналогично, используется параметр `auth`.

Пример 11.3. Проверка с помощью `Requests`

```
>>> import requests
>>> r = requests.get("http://localhost:8000/who",
    auth=("me", "secret"))
>>> r.json()
{'username': 'me', 'password': 'secret'}
```


Вы также можете протестировать пример 11.1 с помощью автоматической страницы документации (<http://localhost:8000/docs>), показанной на рис. 11.1.



Рис. 11.1. Страница документации по простой аутентификации

Нажмите стрелку вниз, расположенную справа, затем кнопку Try It Out (Пробовать) и кнопку Execute (Выполнить). Вы увидите форму, запрашивающую имя пользователя и пароль. Введите что угодно. Форма документации обратится к этой конечной точке сервера и покажет эти значения в ответе.

Эти тесты показывают, что вы можете получить имя пользователя и пароль к серверу и обратно (хотя ни один из них на самом деле ничего не проверял). Что-то на сервере должно проверить, что имя и пароль соответствуют утвержденным значениям. Так, в примере 11.4 я включу в веб-сервер одно секретное имя пользователя и пароль. Вводимые имя пользователя и пароль должны совпадать (каждый из них представляет собой *секретный ключ*), иначе будет выброшено исключение. Код состояния HTTP 401 официально называется Unauthorized (Не авторизован), но на самом деле он означает «*неаутентифцированный*».



Вместо того чтобы запоминать все коды статуса HTTP, можно импортировать модуль статуса FastAPI, который сам импортируется непосредственно из Starlette. Поэтому вы можете использовать более понятное `status_code=HTTP_401_UNAUTHORIZED` в примере 11.4 вместо простой строки `status_code=401`.

Пример 11.4. Добавление секретного имени пользователя и пароля в auth.py

```
import uvicorn
from fastapi import Depends, FastAPI, HTTPException
from fastapi.security import HTTPBasic, HTTPBasicCredentials

app = FastAPI()

secret_user: str = "newphone"
secret_password: str = "whodis?"

basic: HTTPBasicCredentials = HTTPBasic()

@app.get("/who")
def get_user(
    creds: HTTPBasicCredentials = Depends(basic)) -> dict:
    if (creds.username == secret_user and
        creds.password == secret_password):
        return {"username": creds.username,
                "password": creds.password}
    raise HTTPException(status_code=401, detail="Hey!")

if __name__ == "__main__":
    uvicorn.run("auth:app", reload=True)
```

Неправильное введение имени пользователя и пароля приведет к мягкому упреку 401, как показано в примере 11.5.

Пример 11.5. Тест с помощью HTTPie и с несовпадающими именем пользователя/паролем

```
$ http -a me:secret localhost:8000/who
HTTP/1.1 401 Unauthorized
content-length: 17
content-type: application/json
date: Fri, 03 Mar 2023 03:25:09 GMT
server: uvicorn

{
  "detail": "Hey!"
}
```

Применение магической комбинации возвращает имя пользователя и пароль, как показано в примере 11.6.

Пример 11.6. Тестирование с помощью HTTPie и с правильными именем пользователя/паролем

```
$ http -q -a newphone:whodis? localhost:8000/who
{
  "password": "whodis?",
  "username": "newphone"
}
```

Простая индивидуальная аутентификация

В предыдущем разделе было показано, как можно использовать секретный ключ для контроля доступа. Это широко применяемый, но не очень надежный подход. И это ничего не говорит вам о конкретном посетителе, только то, что он (или разумный ИИ) знает секретный ключ. Многим сайтам требуется обеспечить следующее:

- определение каким-либо образом отдельных посетителей;
- идентификацию конкретных посетителей при получении ими доступа к определенным конечным точкам (аутентификация);
- возможность назначения различных прав доступа для некоторых посетителей и конечных точек (авторизация);
- возможность сохранения определенной информации о каждом посетителе (интересы, покупки и т. д.).

Если ваши посетители — люди, можете попросить их указать имя пользователя или электронную почту и пароль. Если это внешние программы, можно попросить их предоставить ключ и секрет API.



В дальнейшем я буду применять просто имя пользователя для обозначения либо выбранного им имени, либо электронной почты.

Чтобы аутентифицировать реальных, а не фиктивных пользователей, вам потребуется сделать немного больше.

- Передать значения пользователя (имя и пароль) конечным точкам сервера API в виде HTTP-заголовков.

- Использовать HTTPS вместо HTTP, чтобы никто не смог подсмотреть текст этих заголовков.
- *Хешировать* пароль в отдельную строку. Результат не является «дехешируемым» — из его хеша нельзя извлечь оригинальный пароль.
- Реальная база данных должна хранить таблицу `User`, содержащую имя пользователя и хешированный пароль (ни в коем случае не оригинальный пароль в виде простого текста).
- Хешировать только что введенный пароль и сравнить результат с хешированным паролем в базе данных.
- Если имя пользователя и хешированный пароль совпадают, передать соответствующий объект `User` вверх по стеку. Если они не совпадают, вернуть значение `None` или вызвать исключение.
- На уровне сервисов запустить все метрики/ведение журналов/что угодно, относящиеся к аутентификации отдельных пользователей.
- На веб-уровне отправить информацию об аутентифицированном пользователе всем функциям, которым она требуется.

В следующих разделах я покажу вам, как сделать все эти вещи, применяя такие современные инструменты, как OAuth2 и JWT.

Более сложная индивидуальная аутентификация

Если требуется выполнить аутентификацию отдельных пользователей, вам нужно где-то хранить информацию о них — например, в базе данных, содержащей записи, хранящие как минимум ключ (имя пользователя или ключ API) и секрет (пароль или секрет API). Посетители вашего сайта будут указывать их при обращении к защищенным URL-адресам, и вам нужны данные в базе данных, чтобы сопоставить их.

Официальные документы по безопасности FastAPI (вводные (<https://oreil.ly/kkTUB>) и продвинутые (<https://oreil.ly/biKwy>)) содержат полные описания того, как настроить аутентификацию для нескольких пользователей, задействуя локальную базу данных. Но пример веб-функции подделывает фактический доступ к базе данных.

Здесь вы поступите наоборот — начнете с уровня данных и будете работать вверх. Укажите, как пользователь/посетитель определяется, хранится и как к нему осуществляется доступ. Затем вы перейдете к веб-уровню и узнаете, как передается, оценивается и аутентифицируется идентификация пользователя.

OAuth2

OAuth 2.0, что расшифровывается как Open Authorization («открытая авторизация»), — это стандарт, позволяющий веб-сайту или приложению получать доступ к ресурсам, размещенным другими веб-приложениями, от имени пользователя.

Auth0

В ранние времена полного доверия к Интернету вы могли предоставить логин и пароль от веб-сайта (назовем его Б) другому сайту (А, конечно же), и он получал доступ к материалам, размещенным на Б, для вас. В результате А получит *полный* доступ к Б, хотя ему будет позволено иметь доступ только к тому, что ему положено. Примерами Б и ресурсов могут служить подписчики в Twitter, друзья в Facebook, контакты по электронной почте и т. д. Конечно, это не могло продолжаться долго, поэтому различные компании и группы объединились, чтобы определить стандарт OAuth. Изначально он был разработан только для того, чтобы позволить сайту А получить доступ к определенным (не всем) ресурсам сайта Б.

OAuth2 (<https://oauth.net/2>) — это популярный, но сложный стандарт *авторизации*, его применение выходит за рамки примера А/Б. Для него существует множество объяснений, от простых (<https://oreil.ly/ehmuf>) до сложных (<https://oreil.ly/qAUaM>).



Раньше существовал стандарт OAuth1 (<https://oauth.net/1>), но он больше не используется. Некоторые из первоначальных рекомендаций OAuth2 уже устарели (другими словами — не применяйте их). На горизонте уже виден стандарт OAuth2.1 (<https://oauth.net/2.1>) и где-то дальше в тумане — txauth (<https://oreil.ly/5PW2T>).

OAuth предлагает различные потоки (flows) (<https://oreil.ly/kRiWh>) для разных обстоятельств. Здесь я буду использовать *Authorization Code Flow* (поток кода авторизации). В этом разделе мы рассмотрим реализацию, по одному среднему этапу за раз.

Сначала вам нужно установить сторонние пакеты Python:

- *JWT handling* — `pip install python-jose[cryptography]`;
- *Secure password handling* — `pip install passlib`;
- *Form handling* — `pip install python-multipart`.

Следующие разделы начинаются с модели пользовательских данных и управления базой данных, а затем по знакомым уровням поднимаются к сервисному и веб-уровню, где появляется OAuth.

Модель пользователя

Начнем с самых минимальных определений пользовательской модели в примере 11.7. Они будут применяться во всех слоях.

Пример 11.7. Определение пользователя: model/user.py

```
from pydantic import BaseModel

class User(BaseModel):
    name: str
    hash: str
```

Объект `User` содержит произвольное поле `name` и строку `hash` — хешированный пароль, а не оригинальный в виде простого текста, и именно он сохраняется в базе данных. Для аутентификации посетителя нам понадобятся оба варианта.

Уровень пользовательских данных

Пример 11.8 содержит код базы данных пользователя.



Код содержит таблицы `user` (активные пользователи) и `xuser` (удаленные пользователи). Часто разработчики добавляют булево поле `deleted` в таблицу пользователей, чтобы указать, что пользователь больше не активен, не удаляя запись из таблицы. Я предпочитаю перемещать данные удаленного пользователя в другую таблицу. Это позволяет избежать повторной проверки поля `deleted` во всех пользовательских запросах. Также это может помочь ускорить запросы — создание индекса для поля с низкой мощностью, например булева, не принесет пользы.

Пример 11.8. Уровень данных: data/user.py

```
from model.user import User
from .init import (conn, curs, get_db, IntegrityError)
from error import Missing, Duplicate

curs.execute("""create table if not exists
                user(
                    name text primary key,
                    hash text)""")
```

```
curs.execute("""create table if not exists
                xuser(
                    name text primary key,
                    hash text)""")

def row_to_model(row: tuple) -> User:
    name, hash = row
    return User(name=name, hash=hash)

def model_to_dict(user: User) -> dict:
    return user.dict()

def get_one(name: str) -> User:
    qry = "select * from user where name=:name"
    params = {"name": name}
    curs.execute(qry, params)
    row = curs.fetchone()
    if row:
        return row_to_model(row)
    else:
        raise Missing(msg=f"User {name} not found")

def get_all() -> list[User]:
    qry = "select * from user"
    curs.execute(qry)
    return [row_to_model(row) for row in curs.fetchall()]

def create(user: User, table:str = "user"):
    """Добавление <пользователя> в таблицу user или xuser"""
    qry = f"""insert into {table}
                (name, hash)
                values
                (:name, :hash)"""
    params = model_to_dict(user)
    try:
        curs.execute(qry, params)
    except IntegrityError:
        raise Duplicate(msg=
            f"{table}: user {user.name} already exists")

def modify(name: str, user: User) -> User:
    qry = """update user set
                name=:name, hash=:hash
                where name=:name0"""
    params = {
        "name": user.name,
        "hash": user.hash,
        "name0": name}
    curs.execute(qry, params)
```

```
if curs.rowcount == 1:
    return get_one(user.name)
else:
    raise Missing(msg=f"User {name} not found")

def delete(name: str) -> None:
    """Отбрасывание пользователя с именем <name> из таблицы пользователей,
    добавление его в таблицу xuser"""
    user = get_one(name)
    qry = "delete from user where name = :name"
    params = {"name": name}
    curs.execute(qry, params)
    if curs.rowcount != 1:
        raise Missing(msg=f"User {name} not found")
    create(user, table="xuser")
```

Уровень фиктивных данных пользователя

Модуль из примера 11.9 применяется в тестах, исключающих базу данных, но нуждается в пользовательских данных.

Пример 11.9. Уровень фиктивных данных: fake/user.py

```
from model.user import User
from error import Missing, Duplicate

# (в этом модуле нет проверки хешированного пароля)
fakes = [
    User(name="kwijobo",
          hash="abc"),
    User(name="ermagerd",
          hash="xyz"),
]

def find(name: str) -> User | None:
    for e in fakes:
        if e.name == name:
            return e
    return None

def check_missing(name: str):
    if not find(name):
        raise Missing(msg=f"Missing user {name}")

def check_duplicate(name: str):
    if find(name):
        raise Duplicate(msg=f"Duplicate user {name}")
```



```
def get_all() -> list[User]:
    """Возврат всех пользователей"""
    return fakes

def get_one(name: str) -> User:
    """Возврат одного пользователя"""
    check_missing(name)
    return find(name)

def create(user: User) -> User:
    """Добавление пользователя"""
    check_duplicate(user.name)
    return user

def modify(name: str, user: User) -> User:
    """Частичное изменение пользователя"""
    check_missing(name)
    return user

def delete(name: str) -> None:
    """Удаление пользователя"""
    check_missing(name)
    return None
```

Сервисный уровень пользователя

Пример 11.10 определяет сервисный уровень для пользователей. Отличием от других модулей сервисного уровня является добавление функций OAuth2 и JWT. Я думаю, что лучше оставить их здесь, чем в веб-слое, хотя несколько функций веб-слоя OAuth2 уже есть в готовящемся проекте `web/user.py`.

Функции CRUD пока что остаются проходными, но в будущем их можно будет разнообразить метриками по своему вкусу. Обратите внимание на то, что, как и сервисы существ и исследователей, эта конструкция поддерживает применение во время выполнения либо фиктивных, либо настоящих уровней данных для доступа к пользовательским данным.

Пример 11.10. Сервисный уровень: `service/user.py`

```
from datetime import timedelta, datetime
import os
from jose import jwt
from model.user import User

if os.getenv("CRYPTID_UNIT_TEST"):
    from fake import user as data
```

```
else:
    from data import user as data

# --- Новые данные auth

from passlib.context import CryptContext

# Измените SECRET_KEY для среды эксплуатации!
SECRET_KEY = "keep-it-secret-keep-it-safe"
ALGORITHM = "HS256"
pwd_context = CryptContext(schemes=["bcrypt"], deprecated="auto")

def verify_password(plain: str, hash: str) -> bool:
    """Хеширование строки <plain> и сравнение с записью <hash> из базы данных"""
    return pwd_context.verify(plain, hash)

def get_hash(plain: str) -> str:
    """Возврат хеша строки <plain>"""
    return pwd_context.hash(plain)

def get_jwt_username(token: str) -> str | None:
    """Возврат имени пользователя из JWT-доступа <token>"""
    try:
        payload = jwt.decode(token, SECRET_KEY, algorithms=[ALGORITHM])
        if not (username := payload.get("sub")):
            return None
    except jwt.JWTError:
        return None
    return username

def get_current_user(token: str) -> User | None:
    """Декодирование токена <token> доступа OAuth и возврат объекта User"""
    if not (username := get_jwt_username(token)):
        return None
    if (user := lookup_user(username)):
        return user
    return None

def lookup_user(username: str) -> User | None:
    """Возврат совпадающего пользователя из базы данных для строки <name>"""
    if (user := data.get(username)):
        return user
    return None

def auth_user(name: str, plain: str) -> User | None:
    """Аутентификация пользователя <name> и <plain> пароль"""
    if not (user := lookup_user(name)):
        return None
```

```

    if not verify_password(plain, user.hash):
        return None
    return user

def create_access_token(data: dict,
    expires: timedelta | None = None
):
    """Возвращение токена доступа JWT"""
    src = data.copy()
    now = datetime.utcnow()
    if not expires:
        expires = timedelta(minutes=15)
    src.update({"exp": now + expires})
    encoded_jwt = jwt.encode(src, SECRET_KEY, algorithm=ALGORITHM)
    return encoded_jwt

# --- CRUD-пассивный материал

def get_all() -> list[User]:
    return data.get_all()

def get_one(name) -> User:
    return data.get_one(name)

def create(user: User) -> User:
    return data.create(user)

def modify(name: str, user: User) -> User:
    return data.modify(name, user)

def delete(name: str) -> None:
    return data.delete(name)

```

Веб-уровень пользователей

Пример 11.11 определяет базовый пользовательский модуль на веб-уровне. Он применяет новый код авторизации из модуля `service/user.py` из примера 11.10.

Пример 11.11. Веб-уровень: `web/user.py`

```

import os
from fastapi import APIRouter, HTTPException
from fastapi.security import OAuth2PasswordBearer, OAuth2PasswordRequestForm
from model.user import User
if os.getenv("CRYPTID_UNIT_TEST"):
    from fake import user as service

```

```
else:
    from service import user as service
from error import Missing, Duplicate

ACCESS_TOKEN_EXPIRE_MINUTES = 30

router = APIRouter(prefix = "/user")

# --- Новые данные auth

# Эта зависимость создает сообщение в каталоге
# "/user/token" (из формы с именем пользователя и паролем)
# и возвращает токен доступа.
oauth2_dep = OAuth2PasswordBearer(tokenUrl="token")

def unauthed():
    raise HTTPException(
        status_code=401,
        detail="Incorrect username or password",
        headers={"WWW-Authenticate": "Bearer"},
    )

# К этой конечной точке направляется любой вызов,
# содержащий зависимость oauth2_dep():
@app.post("/token")
async def create_access_token(
    form_data: OAuth2PasswordRequestForm = Depends()
):
    """Получение имени пользователя и пароля
    из формы OAuth, возврат токена доступа"""
    user = service.auth_user(form_data.username, form_data.password)
    if not user:
        unauthed()
    expires = timedelta(minutes=ACCESS_TOKEN_EXPIRE_MINUTES)
    access_token = service.create_access_token(
        data={"sub": user.username}, expires=expires
    )
    return {"access_token": access_token, "token_type": "bearer"}

@app.get("/token")
def get_access_token(token: str = Depends(oauth2_dep)) -> dict:
    """Возврат текущего токена доступа"""
    return {"token": token}

# --- предыдущий материал CRUD

@app.get("/")
def get_all() -> list[User]:
    return service.get_all()
```

```
@router.get("/{name}")
def get_one(name) -> User:
    try:
        return service.get_one(name)
    except Missing as exc:
        raise HTTPException(status_code=404, detail=exc.msg)

@router.post("/", status_code=201)
def create(user: User) -> User:
    try:
        return service.create(user)
    except Duplicate as exc:
        raise HTTPException(status_code=409, detail=exc.msg)

@router.patch("/")
def modify(name: str, user: User) -> User:
    try:
        return service.modify(name, user)
    except Missing as exc:
        raise HTTPException(status_code=404, detail=exc.msg)

@router.delete("/{name}")
def delete(name: str) -> None:
    try:
        return service.delete(name)
    except Missing as exc:
        raise HTTPException(status_code=404, detail=exc.msg)
```

Тестируем!

Модульные и полные тесты для этого нового пользовательского компонента похожи на те, что вы уже видели для существ и исследователей. Вместо того чтобы изучать печатный текст, можно посмотреть их на сайте, сопровождающем эту книгу¹.

Верхний уровень

В предыдущем разделе была определена новая переменная `router` для URL, начинающихся с пути `/user`, поэтому в примере 11.12 добавляется этот суб-маршрут.

¹ Если бы мне платили за количество строк, моя судьба могла бы измениться.

Пример 11.12. Верхний уровень: `main.py`

```
from fastapi import FastAPI
from web import explorer, creature, user

app = FastAPI()
app.include_router(explorer.router)
app.include_router(creature.router)
app.include_router(user.router)
```

При автозагрузке Uvicorn конечные точки папки `/user/...` теперь должны быть доступны.

Итак, мы создали пользовательский код, а теперь давайте дадим ему повод для работы.

Этапы аутентификации

Вот обзор массы кода из предыдущих разделов.

- Если у конечной точки есть зависимость `oauth2_dep()` (в файле `web/user.py`), то форма, содержащая поля имени пользователя и пароля, генерируется и отправляется клиенту.
- После того как клиент заполнит и отправит эту форму, имя пользователя и пароль (хешированные тем же алгоритмом, что и те, которые уже хранятся в локальной базе данных) будут сопоставлены с находящимися в локальной базе данных.
- При их совпадении токен доступа (в формате JWT) генерируется и возвращается.
- Этот токен доступа передается обратно веб-серверу в виде HTTP-заголовка `Authorization` при последующих запросах. JWT-токен декодируется на локальном сервере в имя пользователя и другие данные. Это имя не нужно снова искать в базе данных.
- Имя пользователя аутентифицировано, и сервер может делать с ним все, что захочет.

Что может сделать сервер с полученной с таким трудом информацией об аутентификации? Он может:

- генерировать метрики (пользователя, конечной точки, времени), чтобы изучить, что просматривается, кем, как долго и т. д.;
- сохранять информацию о пользователе.

JWT

Этот раздел содержит некоторые подробности о JWT. На самом деле эти токены не нужны, чтобы применять весь предыдущий код из этой главы, но если вам любопытно...

JWT (<https://jwt.io>) — это схема кодирования, а не метод аутентификации. Низкоуровневые детали определены в стандарте RFC 7519 (https://oreil.ly/_op1j). Его можно использовать для передачи информации об аутентификации для OAuth2 и других методов, я покажу пример такой реализации.

JWT — это читаемая строка, состоящая из трех разделов, разделенных точками:

- *заголовок* — используемый алгоритм шифрования и тип токена;
- *полезная нагрузка* — ...
- *подпись* — ...

Каждый раздел состоит из строки JSON, закодированной в формате Base 64 URL (<https://www.base64url.com>). Вот пример (он был разбит на позициях точек, чтобы поместиться на ширине этой страницы):

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.  
eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiwiaWF0IjoxNTE2MjM5MDIyfQ.  
SflKxwRJSMeKKF2QT4fwpMeJf36POk6yJV_adQssw5c
```

Как обычная строка ASCII, которую можно использовать также в URL, она может передаваться веб-серверам как часть URL-адреса, параметр запроса, HTTP-заголовок, куки-файлы и т. д.

JWT позволяет избежать поиска в базе данных, но это означает также, что вы не сможете обнаружить аннулированную авторизацию напрямую.

Сторонняя аутентификация: OIDC

Часто можно встретить сайты, позволяющие войти в систему с помощью идентификатора и пароля или войти через свой аккаунт на другом сайте, например Google, Facebook/Meta, LinkedIn и многих других. В таком случае часто используется стандарт OpenID Connect (OIDC) (<https://openid.net/connect>), созданный поверх OAuth2. Когда вы подключаетесь к внешнему сайту с поддержкой OIDC, вы получаете в ответ маркер доступа OAuth2 (как в примерах в этой главе), а также *ID token*.

Официальная документация FastAPI не содержит примера кода для интеграции с OIDC. Если вы хотите попробовать, то сэкономить время на создание собственной реализации позволят некоторые сторонние пакеты (как специфичные для FastAPI, так и более общие):

- FastAPI OIDC (<https://oreil.ly/TDABr>);
- fastapi-third-party-auth (<https://oreil.ly/yGaO6>);
- FastAPI Resource Server (<https://oreil.ly/THByF>);
- oauthlib (<https://oreil.ly/J-pDB>);
- oic (<https://oreil.ly/AgYKZ>);
- OIDC Client (<https://oreil.ly/e9QGb>);
- oidc-op (<https://oreil.ly/cJCF4>);
- OpenID Connect (<https://oreil.ly/WH49I>).

Страница репозитория с проблемами FastAPI (<https://oreil.ly/ztR3r>) содержит множество примеров кода, а также комментариев от пользователя tiangelo (Себастьян Рамирес) о том, что в будущем примеры FastAPI OIDC будут включены в официальную документацию и учебники.

Авторизация

Аутентификация отвечает за то, *кто* (личность), а авторизация — за то, *что*: к каким ресурсам (конечным точкам веб-страниц) вам разрешен доступ и каким образом? Количество комбинаций ответов на вопросы «кто?» и «что?» может быть огромным.

В этой книге основными ресурсами стали исследователи и существа. Поиск исследователя или их общего списка — обычно более открытый процесс, чем добавление или изменение существующей записи. Если веб-сайт должен быть надежным интерфейсом для доступа к данным, доступ к записи должен быть более ограниченным, чем доступ к чтению. Потому что, бр-р, люди.

Если доступ ко всем конечным точкам полностью открыт, авторизация не требуется, можете пропустить этот раздел. Простейшая авторизация может быть простой булевой функцией (является этот пользователь администратором или нет?). Для примеров в этой книге вам может потребоваться авторизация уровня администратора для добавления, удаления или изменения исследователя или существа. Если в вашей базе данных много записей, возможно, вы захотите

ограничить функции `get_all()` с дополнительными правами для неадминистраторов. По мере усложнения сайта разрешения могут становиться все более детализированными.

Рассмотрим несколько вариантов авторизации. Берем таблицу `User`, в которой поле `name` может быть электронной почтой, именем пользователя или ключом API. Парные таблицы — это способ реляционной базы данных сопоставить записи из двух отдельных таблиц:

- Если вы хотите отслеживать только посетителей-администраторов, а остальных оставить анонимными, то задействуйте таблицу `Admin` аутентифицированных имен пользователей. Найдите в ней имя и, если оно совпадает, сравните хешированные пароли в таблице `User`.
- Если *все* посетители должны проходить аутентификацию, но вам нужно авторизовать администраторов только для некоторых конечных точек, то аутентифицируйте всех, как в предыдущих примерах (из таблицы `User`), а затем проверьте таблицу `Admin`, чтобы узнать, является ли этот пользователь также администратором.
- Для более чем одного типа разрешения (например, только чтение, чтение, запись):
 - задействуйте таблицу определения уровней допуска `Permission`;
 - возьмите таблицу `UserPermission`, в которой сопоставляются пользователи и уровни допуска. Иногда ее называют *списком управления доступом*.
- Если комбинации уровней управления доступом сложны, добавьте уровень и определите *роли* (независимые наборы разрешений):
 - создайте таблицу ролей `Role`;
 - создайте таблицу `UserRole`, сопоставляющую пары сущностей из таблиц пользователей и ролей — `User` и `Role` соответственно. Иногда ее называют *управлением доступом на основе ролей* (Role-Based Access Control, RBAC).

Промежуточное программное обеспечение

FastAPI позволяет вставлять на веб-уровень код, выполняющий:

- перехват запроса;
- операции с запросом;
- передачу запроса функции пути;

- перехват ответа, возвращаемого исполняющей функцией;
- операции с ответом;
- возврат ответа вызывающей стороне.

Это похоже на то, что декоратор в Python делает с «оборачиваемой» функцией.

В некоторых случаях можно использовать либо промежуточное ПО, либо внедрение зависимостей с помощью функции `Depends()`. Промежуточное ПО удобнее для решения более глобальных вопросов безопасности, таких как CORS, что приводит к...

CORS

Совместное использование ресурсов разными источниками (Cross-Origin Resource Sharing, CORS) предполагает связь между другими доверенными серверами и вашим сайтом. Если на сайте весь код фронтенда и бэкенда находится в одном месте, то проблем не возникнет. Но в наши дни часто встречается ситуация, когда фронтенд на JavaScript общается с бэкендом, написанным на чем-то вроде FastAPI. Эти серверы не будут иметь одинакового происхождения:

- *протокол* — HTTP или HTTPS;
- *домен* — интернет-домен, например google.com или localhost;
- *порт* — числовой TCP/IP-порт в этом домене, например 80, 443 или 8000.

Как бэкенд может отличить надежный фронтенд от коробки с заплесневелой редиской или злоумышленника, крутящего усы? Это работа для CORS — технологии, определяющей, чему доверяет бэкенд. Наиболее известными способами являются следующие:

- заголовки запросов Origin;
- HTTP-методы;
- HTTP-заголовки;
- тайм-аут кэша CORS.

Вы подключаетесь к CORS на веб-уровне. В примере 11.13 показано, как разрешить только один фронтенд-сервер (с доменом <https://ui.cryptids.com>), а также любые HTTP-заголовки и методы.

Пример 11.13. Активация промежуточного ПО CORS

```
from fastapi import FastAPI, Request
from fastapi.middleware.cors import CORSMiddleware

app = FastAPI()

app.add_middleware(
    CORSMiddleware,
    allow_origins=["https://ui.cryptids.com"],
    allow_credentials=True,
    allow_methods=["*"],
    allow_headers=["*"],
)

@app.get("/test_cors")
def test_cors(request: Request):
    print(request)
```

После этого любой другой домен, пытающийся связаться с бэкендом сайта на-прямую, будет отклонен.

Пакеты сторонних разработчиков

Вы ознакомились с примерами того, как создавать решения для аутентификации и авторизации с помощью FastAPI. Но, возможно, не нужно делать все самостоятельно. Экосистема FastAPI быстро развивается, и могут появиться пакеты, выполняющие большую часть работы за вас.

Приведу несколько непроверенных примеров. Нет никаких гарантий, что через некоторое время любой из пакетов из этого списка будет продолжать существовать и поддерживаться, но, возможно, на них стоит обратить внимание:

- FastAPI Users (<https://oreil.ly/ueVfq>);
- FastAPI JWT Auth (<https://oreil.ly/ooGSK>);
- FastAPI-Login (<https://oreil.ly/oWA3p>);
- fastapi-auth0 (<https://oreil.ly/fHfkU>);
- AuthX (<https://authx.yezz.me>);
- FastAPI-User-Auth (<https://oreil.ly/J57xu>);
- fastapi-authz (<https://oreil.ly/aAGzW>);
- fastapi-opa (<https://oreil.ly/Bvzv3>);

- FastAPI-key-auth (<https://oreil.ly/s-Ui5>);
- FastAPI Auth Middleware (<https://oreil.ly/jnR-s>);
- fastapi-jwt (<https://oreil.ly/RrxUZ>);
- fastapi_auth2 (<https://oreil.ly/5DXkB>);
- fastapi-sso (<https://oreil.ly/GLTdt>);
- Fief (<https://www.fief.dev>).

Заключение

Эта глава была труднее остальных. В ней были показаны способы аутентификации посетителей и их авторизации для выполнения определенных действий. Это два аспекта веб-безопасности. Здесь также обсуждается CORS — еще одна важная тема веб-безопасности.

Тестирование

Инженер по контролю качества заходит в бар. Заказывает пиво. Заказывает 0 пива. Заказывает 99 999 999 999 пива. Заказывает ящерицу. Заказывает −1 пива. Заказывает ueicbksjdhd.

Первый настоящий клиент заходит и спрашивает, где находится туалет. В баре вспыхивает пламя, и все погибают.

Бренан Келлер (Twitter)

Обзор

В этой главе рассматриваются виды тестирования, выполняемые на сайте FastAPI: *модульное*, *интеграционное* и *полное*. Здесь будут применяться модуль *pytest* и автоматическая разработка тестов.

Тестирование Web API

Вы уже видели несколько инструментов для тестирования API вручную по мере добавления конечных точек:

- HTTPie;
- Requests;
- HTTPX;
- браузер.

Существует и множество других инструментов для тестирования.

- Инструмент Curl (<https://curl.se>) очень хорошо известен, хотя в этой книге я использовал HTTPie, чтобы синтаксис был проще.
- Служба Httpbin (<http://httpbin.org>) написана автором библиотеки Requests. Она представляет собой бесплатный тестовый сервер, дающий возможность изучить множество представлений о вашем HTTP-запросе.
- Postman (<https://www.postman.com>) — полноценная платформа для тестирования API.
- Chrome DevTools (https://oreil.ly/eUK_R) — это богатый набор инструментов, входящий в состав браузера Chrome.

Все они могут использоваться для выполнения полных (сквозных) тестов, подобных тем, что вы видели в предыдущих главах. Эти выполняемые вручную тесты были полезны для быстрой проверки кода сразу после его ввода.

Но что, если внесенное позже изменение нарушит один из этих ранних тестов, выполняемых вручную (*регрессия*)? Вы же не хотите повторять десятки тестов после каждого изменения кода. Именно поэтому большое значение приобретают *автоматизированные* тесты. Остальная часть этой главы посвящена им и тому, как создавать их с помощью pytest.

Где тестировать

Я уже называл разновидности тестов:

- *модульные* — внутри уровня, тестируются отдельные функции;
- *интеграционные* — различные уровни, тестирование взаимосвязей;
- *полные* — тестирование полного API и стека под ним.

Иногда их называют *пирамидой тестов*, причем ее ширина указывает на относительное количество тестов, требующихся в каждой группе (рис. 12.1).



Рис. 12.1. Пирамида тестирования

Что тестировать

Что нужно тестировать в процессе написания кода? По сути, необходимо подтвердить, что при заданных входных данных вы получите правильные выходные данные.

Можно проверить следующие моменты:

- ошибочный ввод;
- дублирование ввода;
- неправильные типы ввода;
- неправильный порядок ввода;
- недопустимые значения ввода;
- огромные входные и выходные массивы данных.

Ошибки могут возникнуть где угодно.

- *Веб-уровень* — Pydantic отловит любое несоответствие модели и вернет код состояния HTTP 422.
- *Уровень данных* — база данных будет выдавать исключения при отсутствии или дублировании данных, а также при ошибках синтаксиса SQL-запросов. При передаче результата данных огромного размера одним куском, а не фрагментами с помощью генератора или пагинации могут возникать тайм-ауты или истощение памяти.
- *Любой уровень* — могут быть допущены обычные ошибки и недочеты.

Главы 8–10 содержат некоторые из этих тестов:

- *полное тестирование вручную* с использованием таких инструментов, как HTTPie;
- модульное тестирование вручную в виде фрагментов Python;
- *автоматизированные тесты* с помощью скриптов pytest.

В следующих нескольких разделах мы подробно рассмотрим pytest.

Pytest

В Python уже давно существует стандартный пакет `unittest` (https://oreil.ly/3u0M_). Более поздний пакет стороннего производителя под названием `nose` (<https://nose.readthedocs.io>) представляет собой попытку улучшить его. Большинство разработчиков Python сейчас предпочитают фреймворк `pytest` (<https://docs.pytest.org>), который выполняет больше задач, чем любой из перечисленных ранее, и более прост в использовании. Он не встроен в Python, поэтому при необходимости нужно будет выполнить команду `pip install pytest`. Также запустите команду `pip install pytest-mock`, чтобы получить автоматическую фикстуру `mock` — вы увидите ее позже в этой главе.

Что предлагает `pytest`? Приятные автоматические функции включают следующее.

- *Обнаружение тестирования* — тест для файла Python с префиксом `test_` или суффиксом `_test` в имени будет запущен автоматически. Это сопоставление имен файлов переходит в подкаталоги, выполняя столько тестов, сколько в них содержится.
- *Подробности отказа с конструкцией `Assert`* — оператор выявления ошибок `assert` выводит то, что ожидалось, и то, что произошло на самом деле.
- *Фикстуры* — эти функции могут запускаться один раз для всего тестового скрипта или выполняться для каждого теста (его *области видимости*), предоставляя тестовым функциям такие параметры, как стандартные тестовые данные или инициализация базы данных. Фикстуры — это своего рода внедрение зависимости, подобное предлагаемому FastAPI для функций пути веб-приложения, — конкретные данные передаются общей тестовой функции.
- *Параметризация* — обеспечивает несколько тестовых данных для тестовой функции.

Макет

Где разместить тесты? Похоже, что единого мнения нет, но вот два разумных варианта:

- каталог `test` на верхнем уровне с подкаталогами для тестируемой области кода, например `web`, `service` и т. д.;
- каталог `test` под каждым каталогом кода, например `web`, `service` и т. д.

Кроме того, в рамках конкретного подкаталога `test/web` следует создать дополнительные каталоги для различных типов тестов — *модульных, интеграционных и полных*. В книге я использую такую иерархию:

```
test
├── unit
│   ├── web
│   ├── service
│   └── data
├── integration
└── full
```

Отдельные тестовые скрипты находятся в каталогах нижнего уровня. Они рассматриваются в этой главе.

Автоматизированные модульные тесты

Модульный тест должен проверять что-то одно в пределах одного уровня. Обычно это означает передачу параметра (-ов) в функцию и утверждение того, что должно быть возвращено.

Модульные тесты требуют *изоляции* тестируемого кода. Если ее нет, значит, вы тестируете и еще какую-то часть кода. Итак, как же изолировать код для модульных тестов?

Макетирование

В стеке кода этой книги обращение к URL-адресу через веб-интерфейс обычно вызывает функцию на веб-уровне, вызывающую функцию на сервисном уровне. Далее уже она вызывает функцию на уровне данных, обращающуюся к базе данных. Результаты возвращаются по цепочке, в конечном счете обратно с веб-уровня до вызывающей стороны.

Модульное тестирование на первый взгляд выглядит просто. Для каждой функции, имеющейся в вашей кодовой базе, передаются тестовые аргументы, и требуется убедиться, что она возвращает ожидаемые значения. Это хорошо работает для *чистой функции*, принимающей входные аргументы и возвращающей ответы без обращения к какому-либо внешнему коду. Но большинство функций также вызывают другие функции. Каким же образом можно контролировать то, что делают другие функции? А как насчет данных, поступающих из внешних источников? Наиболее распространенным внешним фактором,

требующим контроля, является доступ к базе данных, но на самом деле это может быть что угодно.

Один из методов заключается в том, чтобы создавать *макет* каждого вызова внешней функции. Поскольку функции в Python являются объектами первого класса, можно одну функцию заменить другой. В пакете unittest есть модуль mock, он выполняет эту операцию.

Многие разработчики считают, что макетирование — это лучший способ изолировать модульные тесты. Сначала я покажу примеры макетирования, а также приведу аргумент, что часто макеты требуют слишком много знаний о том, как работает ваш код, а не о результатах. Вам могут быть знакомы термины «*структурное тестирование*» (как в макетировании, где тестируемый код вполне нагляден) и «*поведенческое тестирование*» (внутренняя структура кода не нужна). Примеры 12.1 и 12.2 определяют модули mod1.py и mod2.py соответственно.

Пример 12.1. Вызываемый модуль (mod1.py)

```
def preamble() -> str:
    return "The sum is "
```

Пример 12.2. Вызывающий модуль (mod2.py)

```
import mod1

def summer(x: int, y:int) -> str:
    return mod1.preamble() + f"{x+y}"
```

Функция `summer()` вычисляет сумму своих аргументов и возвращает строку с результатом функции `preamble` и суммой. Пример 12.3 представляет собой минимальный скрипт `pytest` для проверки функции `summer()`.

Пример 12.3. Скрипт Pytest `test_summer1.py`

```
import mod2

def test_summer():
    assert "The sum is 11" == mod2.summer(5,6)
```

В примере 12.4 тест выполняется успешно.

Пример 12.4. Запуск скрипта `pytest`

```
$ pytest -q test_summer1.py
.
1 passed in 0.04s
```

[100%]

(Аргумент `-q` выполняет тест тихо, не выводя лишних деталей.) Хорошо, он оказался успешным. Но функция `summer()` получила текст из функции `preamble`. А если нам просто требуется проверить, что добавление прошло успешно?

Можно написать новую функцию, возвращающую строковую сумму двух чисел, а затем переписать функцию `summer()`, чтобы она возвращала эту сумму, добавленную к строке в `preamble()`. Или можно создать макет функции `preamble()`, чтобы убрать ее влияние, как показано в примере 12.5.

Пример 12.5. Скрипт Pytest с макетом (`test_summer2.py`)

```
from unittest import mock
import mod1
import mod2

def test_summer_a():
    with mock.patch("mod1.preamble", return_value=""):
        assert "11" == mod2.summer(5,6)

def test_summer_b():
    with mock.patch("mod1.preamble") as mock_preamble:
        mock_preamble.return_value=""
        assert "11" == mod2.summer(5,6)

@mock.patch("mod1.preamble", return_value="")
def test_summer_c(mock_preamble):
    assert "11" == mod2.summer(5,6)

@mock.patch("mod1.preamble")
def test_caller_d(mock_preamble):
    mock_preamble.return_value = ""
    assert "11" == mod2.summer(5,6)
```

Эти тесты показывают, что макеты можно создавать более чем одним способом. Функция `test_caller_a()` использует `mock.patch()` в качестве *менеджера контекста* Python (оператор `with`). Его аргументы приведены далее:

- `"mod1.preamble"` — полное строковое имя функции `preamble()` в модуле `mod1`;
- `return_value=""` — указывает макетированной версии возвращать пустую строку.

Функция `test_caller_b()` представляет собой почти то же самое, но добавляет выражение `as mock_preamble`, чтобы использовать объект макета в следующей строке.

Функция `test_caller_c()` определяет макет с помощью *декоратора* Python. Объект макета передается в качестве аргумента в функции `test_caller2()`.

Функция `test_caller_d()` подобна `test_caller_b()` и задает аргумент `return_value` в отдельном вызове к `mock_preamble`.

В каждом случае строковое имя объекта макета должно совпадать с тем, как он вызывается в тестируемом коде, — в данном случае `summer()`. Библиотека макетов преобразует это строковое имя в переменную, перехватывающую все ссылки на исходную переменную с таким именем. (Помните, что в Python переменные — это просто ссылки на реальные объекты.)

Таким образом, в примере 12.6 во всех четырех тестовых функциях `summer()` при вызове `summer(5, 6)` вместо настоящей функции вызывается изменяющийся макет `preamble()`. В макетированной версии эта строка отбрасывается, поэтому тест может убедиться, что функция `summer()` возвращает строковую версию суммы двух своих аргументов.

Пример 12.6. Запуск макетированного скрипта `pytest`

```
$ pytest -q test_summer2.py
....
4 passed in 0.13s
```

[100%]



Это был выдуманный случай, для простоты. Макетирование может быть довольно сложным. Наглядные примеры можно изучить в таких статьях, как *Understanding the Python Mock Object Library* Алекса Ронкильо (<https://oreil.ly/I0bkd>). А пугающие подробности есть в официальной документации Python (<https://oreil.ly/hN9lZ>).

Тестовые дублиеры и фиктивные объекты

Чтобы выполнить макетирование, вам нужно знать, что функция `summer()` импортирует функцию `preamble()` из модуля `mod1`. Это был структурный тест, требующий знания специфических имен переменных и модулей.

Есть ли способ провести поведенческий тест, в котором это не требуется?

Один из способов — определить *дублер* (иногда его называют *дубликатом* или *двойником*). Это отдельный код, выполняющий то, что мы хотим получить в тесте, — в данном случае, чтобы функция `preamble()` возвращала пустую строку. Одна из возможностей сделать это — импорт. Сначала примените этот подход

к данному примеру, а затем используйте его для модульных тестов на уровнях следующих трех разделов.

В первую очередь переопределите файл `mod2.py` (пример 12.7).

Пример 12.7. Укажите файлу `mod2.py` импортировать дублер при модульном тестировании

```
import os
if os.getenv("UNIT_TEST"):
    import fake_mod1 as mod1
else:
    import mod1

def summer(x: int, y: int) -> str:
    return mod1.preamble() + f"{x+y}"
```

Пример 12.8 определяет этот модуль-двойник `fake_mod1.py`.

Пример 12.8. Дублер `fake_mod1.py`

```
def preamble() -> str:
    return ""
```

А пример 12.9 представляет собой тест...

Пример 12.9. Тестовый скрипт `test_summer_fake.py`

```
import os
os.environ["UNIT_TEST"] = "true"
import mod2

def test_summer_fake():
    assert "11" == mod2.summer(5,6)
```

...Запускаемый примером 12.10.

Пример 12.10. Запуск нового модульного теста

```
$ pytest -q test_summer_fake.py
.
1 passed in 0.04s [100%]
```

Этот метод переключения импорта требует добавления проверки переменной окружения, но позволяет избежать необходимости писать специальные макеты для вызовов функций. Вы сами можете решить, что вам больше нравится. В следующих разделах мы будем применять метод `import` — он отлично работает с *фиктивным* пакетом, который я использовал при определении слоев кода.

Подводя итог, можно сказать, что в этих примерах функция `preamble()` была заменена на макет в тестовом скрипте или импортирован *дублер*. Вы можете изолировать тестируемый код и другими способами, но приведенные варианты работают и не содержат особых хитростей, как другие, которые вам может предложить Google.

Веб-уровень

Этот уровень реализует API сайта. В идеале для каждой функции пути (конечной точки) должен быть как минимум один тест, а то и больше, если функция может не сработать более чем одним способом. На веб-уровне обычно требуется проверить, существует ли конечная точка, работает ли она с правильными параметрами и возвращает ли правильный код состояния и данные.



Это поверхностные API-тесты, тестирующие исключительно веб-слой. Таким образом, вызовы сервисного уровня, который, в свою очередь, будет обращаться к уровню данных и базе данных, должны быть перехвачены, как и любые другие вызовы, выходящие с веб-уровня.

Используя идею с реализацией конструкции `import` из предыдущего раздела, задействуйте переменную окружения `CRYPTID_UNIT_TEST`, чтобы импортировать *фиктивный* пакет, такой как `service`, вместо настоящего `service`. Это не позволяет веб-функциям вызывать сервисные функции, а вместо этого замыкает их на *фиктивную* (дублированную) версию. Тогда нижний уровень данных и база данных также не будут задействованы. Мы получаем то, что хотим, — модульные тесты. В примере 12.11 представлен измененный файл `web/creature.py`.

Пример 12.11. Измененный файл `web/creature.py`

```
import os
from fastapi import APIRouter, HTTPException
from model.creature import Creature
if os.getenv("CRYPTID_UNIT_TEST"):
    from fake import creature as service
else:
    from service import creature as service
from error import Missing, Duplicate

router = APIRouter(prefix = "/creature")

@router.get("/")
def get_all() -> list[Creature]:
    return service.get_all()
```

```

@router.get("/{name}")
def get_one(name) -> Creature:
    try:
        return service.get_one(name)
    except Missing as exc:
        raise HTTPException(status_code=404, detail=exc.msg)

@router.post("/", status_code=201)
def create(creature: Creature) -> Creature:
    try:
        return service.create(creature)
    except Duplicate as exc:
        raise HTTPException(status_code=409, detail=exc.msg)

@router.patch("/")
def modify(name: str, creature: Creature) -> Creature:
    try:
        return service.modify(name, creature)
    except Missing as exc:
        raise HTTPException(status_code=404, detail=exc.msg)

@router.delete("/{name}")
def delete(name: str) -> None:
    try:
        return service.delete(name)
    except Missing as exc:
        raise HTTPException(status_code=404, detail=exc.msg)

```

В примере 12.12 приведены тесты, использующие две фикстуры pytest:

- `sample()` — новый объект `Creature`;
- `fakes()` — список существ.

Фиктивные данные получаются из модуля нижнего уровня. Установив переменную окружения `CRYPTID_UNIT_TEST`, веб-модуль из примера 12.11 импортирует фиктивную версию сервиса (предоставляющую фиктивные данные, а не вызывающую базу данных) вместо настоящей. Это позволяет изолировать тесты, в чем и заключается смысл.

Пример 12.12. Модульные тесты веб-уровня для существ с использованием фикстур

```

from fastapi import HTTPException
import pytest
import os
os.environ["CRYPTID_UNIT_TEST"] = "true"
from model.creature import Creature
from web import creature

```

```
@pytest.fixture
def sample() -> Creature:
    return Creature(name="dragon",
                    description="Wings! Fire! Aieee!",
                    country="*")

@pytest.fixture
def fakes() -> list[Creature]:
    return creature.get_all()

def assert_duplicate(exc):
    assert exc.value.status_code == 404
    assert "Duplicate" in exc.value.msg

def assert_missing(exc):
    assert exc.value.status_code == 404
    assert "Missing" in exc.value.msg

def test_create(sample):
    assert creature.create(sample) == sample

def test_create_duplicate(fakes):
    with pytest.raises(HTTPException) as exc:
        _ = creature.create(fakes[0])
    assert_duplicate(exc)

def test_get_one(fakes):
    assert creature.get_one(fakes[0].name) == fakes[0]

def test_get_one_missing():
    with pytest.raises(HTTPException) as exc:
        _ = creature.get_one("bobcat")
    assert_missing(exc)

def test_modify(fakes):
    assert creature.modify(fakes[0].name, fakes[0]) == fakes[0]

def test_modify_missing(sample):
    with pytest.raises(HTTPException) as exc:
        _ = creature.modify(sample.name, sample)
    assert_missing(exc)

def test_delete(fakes):
    assert creature.delete(fakes[0].name) is None

def test_delete_missing(sample):
    with pytest.raises(HTTPException) as exc:
        _ = creature.delete("emu")
    assert_missing(exc)
```


Сервисный уровень

В некотором смысле сервисный уровень самый важный и может быть связан с различными уровнями данных и веб-уровнями. Пример 12.13 похож на пример 12.11, отличаясь в основном инструкцией `import` и использованием модуля данных нижнего уровня. Также он не перехватывает исключения, возникающие на уровне данных, оставляя их для обработки на веб-уровне.

Пример 12.13. Измененный файл `service/creature.py`

```
import os
from model.creature import Creature
if os.getenv("CRYPTID_UNIT_TEST"):
    from fake import creature as data
else:
    from data import creature as data

def get_all() -> list[Creature]:
    return data.get_all()

def get_one(name) -> Creature:
    return data.get_one(name)

def create(creature: Creature) -> Creature:
    return data.create(creature)

def modify(name: str, creature: Creature) -> Creature:
    return data.modify(name, creature)

def delete(name: str) -> None:
    return data.delete(name)
```

В примере 12.14 приведены соответствующие модульные тесты.

Пример 12.14. Сервисный тест в файле `test/unit/service/test_creature.py`

```
import os
os.environ["CRYPTID_UNIT_TEST"] = "true"
import pytest

from model.creature import Creature
from error import Missing, Duplicate
from data import creature as data

@pytest.fixture
def sample() -> Creature:
    return Creature(name="yeti",
                    aka:"Abominable Snowman",
```

```
        country="CN",
        area="Himalayas",
        description="Handsome Himalayan")

def test_create(sample):
    resp = data.create(sample)
    assert resp == sample

def test_create_duplicate(sample):
    resp = data.create(sample)
    assert resp == sample
    with pytest.raises(Duplicate):
        resp = data.create(sample)

def test_get_exists(sample):
    resp = data.create(sample)
    assert resp == sample
    resp = data.get_one(sample.name)
    assert resp == sample

def test_get_missing():
    with pytest.raises(Missing):
        _ = data.get_one("boxturtle")

def test_modify(sample):
    sample.country = "CA" # Canada!
    resp = data.modify(sample.name, sample)
    assert resp == sample

def test_modify_missing():
    bob: Creature = Creature(name="bob", country="US", area="",
        description="some guy", aka="??")
    with pytest.raises(Missing):
        _ = data.modify(bob.name, bob)
```

Уровень данных

Уровень данных проще тестировать изолированно, поскольку можно не беспокоиться о случайном вызове функции на еще более низком уровне. Модульные тесты должны охватывать как функции этого уровня, так и конкретные используемые запросы к базе данных. До сих пор SQLite был «сервером» баз данных, а SQL — языком запросов. Но вы можете решить работать с таким пакетом, как SQLAlchemy, и задействовать его возможности в виде SQLAlchemy Expression Language или ORM. Тогда потребуется полное тестирование. Пока что я придерживаюсь самого низкого уровня — DB-API Python и обычные SQL-запросы.

В отличие от модульных тестов веб- и сервисного уровней, на этот раз нам не нужны фиктивные модули для замены существующих модулей уровня данных. Вместо этого задайте другую переменную окружения, чтобы указать уровню данных использовать экземпляр SQLite, основанный только на памяти, а не на файлах. Это не требует никаких изменений в существующих модулях данных, нужна просто настройка в тесте примера 12.15 *перед* импортом любых модулей данных.

Пример 12.15. Модульные тесты данных для файла data/creature.py

```
import os
import pytest
from model.creature import Creature
from error import Missing, Duplicate

# Установите этот параметр перед импортом данных
os.environ["CRYPTID_SQLITE_DB"] = ":memory:"
from data import creature

@pytest.fixture
def sample() -> Creature:
    return Creature(name="yeti",
                    aka="Abominable Snowman",
                    country="CN",
                    area="Himalayas",
                    description="Hapless Himalayan")

def test_create(sample):
    resp = creature.create(sample)
    assert resp == sample

def test_create_duplicate(sample):
    with pytest.raises(Duplicate):
        _ = creature.create(sample)

def test_get_one(sample):
    resp = creature.get_one(sample.name)
    assert resp == sample

def test_get_one_missing():
    with pytest.raises(Missing):
        resp = creature.get_one("boxturtle")

def test_modify(sample):
    creature.country = "JP" # Япония!
    resp = creature.modify(sample.name, sample)
    assert resp == sample
```

```
def test_modify_missing():
    thing: Creature = Creature(name="snurfle",
                                description="some thing", country="somewhere")
    with pytest.raises(Missing):
        _ = creature.modify(thing.name, thing)

def test_delete(sample):
    resp = creature.delete(sample.name)
    assert resp is None

def test_delete_missing(sample):
    with pytest.raises(Missing):
        _ = creature.delete(sample.name)
```

Автоматизированные интеграционные тесты

Интеграционные тесты показывают, хорошо ли различные фрагменты кода взаимодействуют между уровнями. Но если вы поищите примеры, то получите множество разных ответов. Нужно ли тестировать частичные маршруты вызовов, такие как веб → сервис, веб → данные и т. д.?

Чтобы полностью протестировать каждое соединение в конвейере $A \rightarrow B \rightarrow V$, потребуется проверить следующие комбинации:

- $A \rightarrow B$;
- $B \rightarrow V$;
- $A \rightarrow V$.

А у вас будет полный колчан таких стрелок, если в системе более трех пересечений. Или интеграционные тесты должны быть по сути сквозными тестами, но с макетом самой последней части — хранения данных на диске?

До сих пор вы использовали SQLite в качестве базы данных и можете задействовать режим работы SQLite In-Memory в качестве дублера (подделки) для базы данных SQLite на диске. Если ваши запросы представляют собой *стандартный* SQL, то SQLite-In-Memory может быть подходящим вариантом макета и для других баз данных. Если нет, то для создания макетов конкретных баз данных адаптированы следующие модули:

- *PostgreSQL* — pgmock (<https://pgmock.readthedocs.io>);
- *MongoDB* — Mongomock (<https://github.com/mongomock/mongomock>);

- *множество ресурсов Pytest Mock* (<https://pytest-mock-resources.readthedocs.io>), которые запускают различные тестовые базы данных в контейнерах Docker, и они интегрированы с pytest.

Наконец, можно просто запустить тестовую базу данных того же типа, что и рабочая. Переменная окружения может иметь специфику, подобную тому, как вы использовали трюк с юнит-тестом/подделкой.

Паттерн «Репозиторий»

Хотя я не использовал его в этой книге, но паттерн «Репозиторий» (<https://oreil.ly/3JMKH>) представляет собой интересный подход. *Репозиторий* — это простое промежуточное хранилище данных в оперативной памяти, подобное представленному ранее уровню фиктивных данных. Затем он связывается с подключаемыми бэкендами для реальных баз данных. Репозиторий сопровождается *паттерном Unit of Work* (<https://oreil.ly/jHGV8>), гарантирующим, что либо будет зафиксирована группа операций в одной *сессии*, либо выполнен откат, как для единого фрагмента.

До сих пор запросы к базе данных в этой книге были атомарными. На практике для работы с базами данных вам могут понадобиться многоэтапные запросы и определенная обработка сессий. Паттерн «Репозиторий» сочетается также с внедрением зависимостей (<https://oreil.ly/0f0Q3>), с которым вы уже сталкивались в других частях этой книги и которое, вероятно, уже немного оценили.

Автоматизированные полные тесты

При полном или сквозном тестировании используются все слои одновременно — этот метод максимально приближен к производственному применению. Большинство описанных ранее тестов были полными: вызов конечной точки веб-приложения, путешествие через Сервисград в центр Даннобурга и возвращение с продуктами. Это закрытые тесты. Все происходит в реальном времени, и вам неважно, как это происходит, — важно, что это происходит.

Вы можете полностью протестировать каждую конечную точку в общем API двумя способами.

- *С помощью HTTP/HTTPS* напишите отдельные обращающиеся к серверу тестовые клиенты Python. Во многих примерах, приведенных в книге, это

сделано с помощью автономных клиентов, таких как HTTPie, или в скриптах, использующих Requests.

- С помощью `TestClient` примените встроенный объект FastAPI/Starlette для получения прямого доступа к серверу, без открытого TCP-соединения.

Однако эти подходы требуют написания одного или нескольких тестов для каждой конечной точки. Это может превратиться в Средневековье, а мы уже на несколько веков ушли вперед. Более современный подход основан на тестировании на основе свойств (Property-Based Testing, PBT). При этом используется преимущество автоматически генерируемой документации FastAPI. *Схема* OpenAPI под названием `openapi.json` создается FastAPI каждый раз, когда вы изменяете функцию пути или декоратор пути на веб-уровне. В этой схеме подробно описано все о каждой конечной точке — аргументы, возвращаемые значения и т. д. Для этого и существует спецификация OpenAPI (OpenAPI Specification, OAS), приведенная на странице OpenAPI Initiative's FAQ (<https://www.openapis.org/faq>): «OAS определяет стандартное, не зависящее от языка программирования описание интерфейса для REST API, которое позволяет людям и компьютерам обнаружить и понять возможности сервиса, не требуя доступа к исходному коду, дополнительной документации или изучения сетевого трафика».

Для работы потребуется два пакета:

- Hypothesis (<https://hypothesis.works>) — `pip install hypothesis`;
- Schemathesis (<https://schemathesis.readthedocs.io>) — `pip install schemathesis`.

Hypothesis — это базовая библиотека, а Schemathesis применяет ее к схеме OpenAPI 3.0, которую генерирует FastAPI. При запуске инструмент Schemathesis считывает эту схему, генерирует множество тестов с различными данными (и вам не нужно их придумывать!) и работает с pytest.

Чтобы не затягивать, в примере 12.16 сначала сократим код в файле `main.py` до базовых конечных точек существа и исследователя — `creature` и `explorer`.

Пример 12.16. Основа файла `main.py`

```
from fastapi import FastAPI
from web import explorer, creature

app = FastAPI()
app.include_router(explorer.router)
app.include_router(creature.router)
```

В примере 12.17 выполняются тесты.

Пример 12.17. Запуск тестов Schemathesis

```
$ schemathesis http://localhost:8000/openapi.json
===== Schemathesis test session starts =====
Schema location: http://localhost:8000/openapi.json
Base URL: http://localhost:8000/
Specification version: Open API 3.0.2
Workers: 1
Collected API operations: 12

GET /explorer/ . [ 8%]
POST /explorer/ . [ 16%]
PATCH /explorer/ F [ 25%]
GET /explorer . [ 33%]
POST /explorer . [ 41%]
GET /explorer/{name} . [ 50%]
DELETE /explorer/{name} . [ 58%]
GET /creature/ . [ 66%]
POST /creature/ . [ 75%]
PATCH /creature/ F [ 83%]
GET /creature/{name} . [ 91%]
DELETE /creature/{name} . [100%]
```

Я получил две пометки F, обе при вызове PATCH (функций `modify()`). Как же неприятно.

За этой секцией вывода следует секция с пометкой FAILURES (отказы), содержащая подробные трассировки стека всех тестов, завершившихся неудачей. Их необходимо исправить. Заключительный раздел обозначен как SUMMARY (краткое описание):

```
Performed checks:
not_a_server_error 717 / 727 passed FAILED
```

Hint: You can visualize test results in Schemathesis.io
by using `--report` in your CLI command.

Это было быстро, и не требовалось проводить множество тестов для каждой конечной точки, придумывая, какие входные данные могут нарушить их работу. Тестирование на основе свойств считывает типы и ограничения входных аргументов из схемы API и генерирует диапазон значений для проверки в каждой конечной точке.

Это еще одно неожиданное преимущество подсказок типов, которые поначалу казались просто приятными вещами: подсказки типа `→` схема OpenAPI `→` `→` сгенерированная документация *и* тесты.

Тестирование безопасности

Безопасность — это не что-то одно, а все. Вам нужно защищаться не только от злого умысла, но и от обычных ошибок, и даже от неподвластных вам событий. Отложим вопросы масштабирования до следующего раздела, а здесь займемся в основном анализом потенциальных угроз.

В главе 11 обсуждались аутентификация и авторизация. Эти факторы всегда беспорядочны и подвержены ошибкам. Заманчиво использовать умные методы для противодействия умным атакам и всегда сложно разработать легкую для понимания и реализации защиту.

Но теперь, когда вы знаете о Schemathesis, прочитайте его документацию (https://oreil.ly/v_O-Q) о тестировании на основе свойств для аутентификации. Так же как он значительно упростил тестирование большей части API, он может автоматизировать большую часть тестов для конечных точек, требующих аутентификации.

Нагрузочное тестирование

То, как ваше приложение справляется с большим трафиком, показывают нагрузочные тесты, проверяющие:

- вызовы API;
- чтение или запись в базу данных;
- использование памяти;
- использование дискового пространства;
- время ожидания и пропускную способность сети.

Некоторые из них могут представлять собой *сквозные* тесты, имитирующие армию пользователей, жаждущих воспользоваться вашим сервисом. Вам стоит быть готовыми к тому, что такой день наступит. Содержание этого раздела частично совпадает с содержанием разделов «Производительность» и «Устранение неполадок» главы 13.

Существует много хороших нагрузочных тестеров, но здесь я рекомендую инструмент под названием Locust (<https://locust.io>). При его использовании можно определить все тесты с помощью обычных скриптов на языке Python. Он может имитировать работу сотен тысяч пользователей, одновременно запрашивающих ваш сайт или даже несколько серверов.

Установите его локально с помощью команды `pip install locust`. Первым тестом может стать проверка возможного количества единовременных посетителей вашего сайта. Это похоже на проверку того, насколько экстремальные погодные условия может выдержать здание во время урагана, землетрясения, снежной бури или наступления другого страхового случая. Поэтому вам нужны структурные тесты сайта. В документации (<https://docs.locust.io>) Locust можно найти более подробную информацию.

Но, как говорят по телевизору, это еще не все! Недавно разработчики инструмента Grasshopper (<https://github.com/alteryx/locust-grasshopper>) расширили возможности Locust для выполнения таких задач, как измерение времени в нескольких HTTP-вызовах. Чтобы опробовать это расширение, установите его с помощью команды `pip install locust-grasshopper`.

Заключение

В этой главе были подробно рассмотрены типы тестирования, приведены примеры выполнения `pytest` автоматизированного тестирования кода на уровне модулей, интеграции и полного тестирования. Тесты API можно автоматизировать с помощью инструмента Schemathesis. Здесь также обсуждалось, как выявить проблемы безопасности и производительности до того, как они возникнут.

ГЛАВА 13

Запуск в эксплуатацию

Если бы строители строили здания так, как программисты пишут программы, то первый же появившийся дятел уничтожил бы цивилизацию.

Джеральд Вайнберг, ученый в области IT

Обзор

У вас есть приложение, работающее на локальной машине, и теперь вы хотите поделиться им. В этой главе представлено множество сценариев того, как перенести приложение в среду эксплуатации и поддерживать его правильную и эффективную работу. Поскольку часть описаний *очень* подробные, в некоторых случаях я буду ссылаться на полезные сторонние документы, а не выкладывать их здесь.

Развертывание

До сих пор во всех примерах кода в этой книге использовался один экземпляр `uvicorn`, запущенный на адресе `localhost` на порте `8000`. Для обработки большого количества трафика вам потребуется несколько серверов, работающих на нескольких ядрах, предоставляемых современным оборудованием. Кроме того, понадобится что-то поверх этих серверов для выполнения следующих действий:

- поддержания их в рабочем состоянии (*супервайзер*);
- сбора и отправки внешних запросов (*обратный прокси*);

- возврата ответов;
- обеспечения HTTPS-терминации (расшифровка SSL).

Множество процессов

Вы наверняка видели сервер Python под названием Gunicorn (<https://gunicorn.org>). Он может контролировать несколько процессов, но это сервер WSGI, а FastAPI основан на ASGI. К счастью, существует специальный класс процессов Uvicorn, которым может управлять Gunicorn.

В примере 13.1 рассматриваются эти процессы Uvicorn на `localhost`, порт 8000 (взято из официальной документации, <https://oreil.ly/Svdhx>). Кавычки защищают оболочку от любой специальной интерпретации.

Пример 13.1. Использование Gunicorn с процессами Uvicorn

```
$ pip install "uvicorn[standard]" gunicorn
$ gunicorn main:app --workers 4 --worker-class \
  uvicorn.workers.UvicornWorker --bind 0.0.0.0:8000
```

Вы увидите множество строк, когда Gunicorn будет выполнять ваши запросы. Будет запущен процесс Gunicorn верхнего уровня, который станет общаться с четырьмя рабочими подпроцессами Uvicorn, совместно использующими порт 8000 на `local host (0.0.0.0)`. Измените хост, порт или количество процессов, если вам нужно что-то другое. Выражение `main:app` ссылается на файл `main.py` и объект FastAPI с именем переменной `app`. В документации (<https://oreil.ly/TxYIy>) Gunicorn утверждается: «Для обработки сотен или тысяч запросов в секунду Gunicorn потребуется всего 4–12 рабочих процессов».

Оказывается, сам Uvicorn также может запускать несколько процессов Uvicorn, как показано в примере 13.2.

Пример 13.2. Использование Uvicorn с рабочими процессами Uvicorn

```
$ uvicorn main:app --host 0.0.0.0 --port 8000 --workers 4
```

Но этот метод не позволяет управлять процессами, поэтому обычно предпочтение отдается методу Gunicorn. Для Uvicorn существуют и другие менеджеры процессов — см. официальную документацию (<https://www.uvicorn.org/deployment>).

Это позволяет выполнять три из четырех задач, упомянутых в предыдущем разделе, но не шифрование HTTPS.

HTTPS

Официальная документация FastAPI по HTTPS (<https://oreil.ly/HYRW7>), как и вся остальная, чрезвычайно информативна. Я рекомендую прочитать ее, а затем описание (<https://oreil.ly/zcUWS>) Рамиресом того, как добавить поддержку HTTPS в FastAPI с помощью обратного прокси под названием Traefik (<https://traefik.io>). Он располагается над вашими веб-серверами, подобно nginx в качестве обратного прокси и балансировщика нагрузки, но включает в себя магию HTTPS.

Хотя этот процесс состоит из множества этапов, он все же намного проще, чем прежде. В частности, раньше вам приходилось регулярно платить большие деньги центру сертификации за цифровой сертификат, который можно было использовать для поддержки протокола HTTPS на своем сайте. К счастью, на смену этим органам пришел бесплатный сервис Let's Encrypt (<https://letsencrypt.org>).

Docker

Когда Docker появился на сцене (он был упомянут в молниеносном пятиминутном докладе (<https://oreil.ly/25oef>) Соломона Хайкса из dotCloud на PyCon 2013), большинство из нас впервые услышали о контейнерах для Linux. Со временем мы поняли, что Docker быстрее и легче виртуальных машин. Вместо эмуляции полноценной операционной системы все контейнеры совместно используют ядро Linux сервера, а процессы и сети изолируются в собственных пространствах имен. Внезапно у вас появилась возможность с помощью бесплатного программного обеспечения Docker разместить несколько независимых сервисов на одной машине, не беспокоясь о том, что они будут пересекаться.

Десять лет спустя Docker получил всеобщее признание и поддержку. Если вы хотите разместить свое приложение FastAPI на облачном сервисе, обычно нужно сначала создать его *образ в Docker*. В официальной документации FastAPI (<https://oreil.ly/QnwOW>) содержится подробное описание того, как сделать Docker-версию вашего FastAPI-приложения. Одним из этапов будет написание *Dockerfile* — текстового файла, содержащего информацию о конфигурации Docker, например, какой код приложения использовать и какие процессы запускать. Чтобы доказать, что по уровню сложности это не операция на мозге во время запуска космической ракеты, приведу Dockerfile с этой страницы:

```
FROM python:3.9
WORKDIR /code
COPY ./requirements.txt /code/requirements.txt
```

```
RUN pip install --no-cache-dir --upgrade -r /code/requirements.txt
COPY ./app /code/app
CMD ["uvicorn", "app.main:app", "--host", "0.0.0.0", "--port", "80"]
```

Я рекомендую прочитать официальную документацию или другие ссылки, выдаваемые поисковой системой Google по запросу `fastapi docker`, например *The Ultimate FastAPI Tutorial Part 13 — Using Docker to Deploy Your App* (<https://oreil.ly/7TUpR>) Кристофера Самиулы.

Облачные сервисы

В Сети можно найти множество источников платного или бесплатного хостинга. Вот некоторые примеры сведений о том, как разместить FastAPI с их помощью:

- статья *FastAPI — Deployment* на сайте Tutorials Point (<https://oreil.ly/DBZcm>);
- материалы *The Ultimate FastAPI Tutorial Part 6b — Basic Deployment on Linode*, сформированные инженером Кристофером Самиулой (<https://oreil.ly/s8iar>);
- статья *How to Deploy a FastAPI App on Heroku for Free* Шиничи Окады (<https://oreil.ly/A6gij>).

Kubernetes

Платформа Kubernetes выросла из внутреннего кода Google для управления внутренними системами, которые становились просто ужасающе сложными. Системные администраторы (так их тогда называли) вручную настраивали такие инструменты, как балансировщики нагрузки, обратные прокси, хьюмидоры¹ и т. д. Kubernetes стремился взять большую часть этих знаний и автоматизировать их — не говорите мне, *как* это сделать, а скажите, *чего* вы хотите. Сюда входят такие задачи, как поддержание работоспособности сервиса или запуск дополнительных серверов при резком увеличении трафика.

Существует множество описаний того, как развернуть FastAPI на Kubernetes, в том числе статья Суманты Мукхопадхья *Deploying a FastAPI Application on Kubernetes* (<https://oreil.ly/ktTNu>).

¹ Погодите, они же сохраняют сигары свежими.

Производительность

В настоящее время производительность FastAPI одна из самых высоких (<https://oreil.ly/mxabf>) среди всех веб-фреймворков на Python и даже сравнима с производительностью фреймворков на более быстрых языках, таких как Go. Но во многом это связано с ASGI, позволяющим избежать ожидания ввода-вывода с помощью асинхронности. Сам по себе Python — довольно медленный язык. Далее приведены некоторые советы и рекомендации по улучшению общей производительности.

Асинхронность

Часто веб-серверу не нужно быть очень быстрым. Большую часть своего времени он тратит на получение сетевых HTTP-запросов и возврат результатов (в этой книге он представлен веб-уровнем). Между ними веб-сервис выполняет бизнес-логику (сервисный уровень), получает доступ к источникам данных (уровень данных) и снова тратит большую часть своего времени на сетевой ввод-вывод.

Когда код в веб-сервисе должен ждать ответа, лучше всего использовать асинхронную функцию (`async def`, а не `def`). Это позволяет FastAPI и Starlette планировать работу асинхронной функции и выполнять другие действия в ожидании ее ответа. Это одна из причин того, почему бенчмарки FastAPI лучше, чем фреймворки на базе WSGI, такие как Flask и Django. У производительности есть два аспекта:

- время обработки одного запроса;
- количество одновременно обрабатываемых запросов.

Кэши

Если у вас есть конечная точка веб-приложения, получающая данные из статичного источника (например, записи в базе данных, которые меняются редко или не меняются никогда), можно кэшировать данные в функции. Это может быть на любом уровне. В Python представлен стандартный модуль `functools` (<https://oreil.ly/8Kg4V>), а также функции `cache()` и `lru_cache()`.

Базы данных, файлы и память

Одна из самых распространенных причин медленной работы веб-сайта — отсутствие индекса для таблицы базы данных достаточного размера. Часто вы не замечаете проблемы до тех пор, пока ваша таблица не вырастет до определенного размера, и тогда запросы внезапно становятся намного медленнее. В SQL любой столбец в операторе `WHERE` должен быть проиндексирован.

Во многих примерах, приведенных в книге, первичный ключ таблиц `creature` и `explorer` был представлен текстовым полем `name`. При создании таблиц поле `name` было объявлено в качестве первичного ключа — `primary key`. Для крошечных таблиц, приведенных ранее, SQLite в любом случае проигнорирует этот ключ, поскольку быстрее будет просто просканировать таблицу. Но как только она достигает приличного размера — скажем, миллиона строк, отсутствие индекса становится заметным. Решением может стать запуск оптимизатора запросов (<https://oreil.ly/YPR3Q>).

Даже если у вас небольшая таблица, можете провести нагрузочное тестирование базы данных с помощью скриптов Python или инструментов с открытым исходным кодом. Если выполняется множество последовательных запросов к базе данных, возможно, их стоит объединить в один пакет. Если вы выгружаете или скачиваете большой файл, используйте потоковые версии, а не гигантский фрагмент данных.

Очереди

Если вы выполняете какую-либо задачу, занимающую больше доли секунды, например отправку письма с подтверждением или уменьшение изображения, возможно, стоит передать ее в очередь заданий, например в Celery (<https://docs.celeryq.dev>).

Непосредственно Python

Если веб-сервис кажется медленным, потому что выполняет значительные вычисления с помощью Python, вам может понадобиться «более быстрый Python». Альтернативные варианты:

- использовать PyPy (<https://www.pypy.org>) вместо стандартной реализации CPython;

- написать расширение (<https://oreil.ly/BEIJa>) для Python на C, C++ или Rust;
- преобразовать медленный код Python в язык Cython (<https://cython.org>), используемый Pydantic и Uvicorn.

Недавно был сделан очень интригующий анонс языка Mojo (<https://oreil.ly/C96kx>). Он стремится стать полным супернабором Python с новыми возможностями (применяя тот же дружественный синтаксис Python), способными ускорить примеры на Python в тысячи раз. Основной автор Крис Латтнер ранее работал над такими инструментами компиляции, как LLVM (<https://llvm.org>), Clang (<https://clang.llvm.org>) и MLIR (<https://mlir.llvm.org>), а также языком Swift (<https://www.swift.org>) для Apple.

Mojo стремится стать одноязычным решением для разработки ИИ, для чего сейчас (в PyTorch и TensorFlow) требуются сборки Python/C/C++, сложные в разработке, управлении и отладке. Но Mojo был бы хорошим языком общего назначения не только в сфере ИИ.

Я много лет писал на C и все ждал преемника, обладающего производительностью и простотой применения языка Python. Возможными вариантами были D, Go, Julia, Zig и Rust, но если Mojo сможет оправдать свои цели (<https://oreil.ly/EojvA>), я бы активно его использовал.

Устранение неполадок

Смотрите снизу вверх с того момента и места, где вы столкнулись с проблемой. К ним относятся проблемы производительности во времени и пространстве, а также логические и асинхронные ловушки.

Виды проблем

Какой код ответа HTTP вы получили в первую очередь?

- 404 — ошибка аутентификации или авторизации.
- 422 — обычно это жалоба Pydantic на использование модели.
- 500 — отказ сервиса, расположенного за вашим FastAPI.

Ведение журналов

Uvicorn и другие веб-серверы обычно пишут журналы в файл `stdout`. Вы можете проверить журнал, чтобы узнать, какой вызов был сделан на самом деле, включая HTTP-глагол и URL-адрес, но не данные в теле, заголовках или файлах cookies.

Если определенная конечная точка возвращает код состояния семейства **400**, можно попробовать подать те же данные еще раз и посмотреть, не повторится ли ошибка. Если да, то у меня срабатывает первый пещерный инстинкт отладчика — добавить операторы `print()` в соответствующие функции веб, сервисного и уровня данных.

Кроме того, везде, где вы инициируете выброс исключения, добавляйте подробное описание. Если поиск в базе данных не удался, укажите входные значения и конкретную ошибку, например попытку дублировать уникальное ключевое поле.

Метрики

Может показаться, что значения терминов «метрика», «мониторинг», «наблюдаемость» и «телеметрия» частично совпадают. В стране Python принято использовать:

- Prometheus (<https://prometheus.io>) — для получения метрик;
- Grafana (<https://grafana.com>) — для их отображения;
- OpenTelemetry (<https://opentelemetry.io>) — для измерения времени.

Вы можете применить их ко всем уровням своего сайта: веб-уровню, сервисному и уровню данных. Сервисные уровни могут быть более бизнес-ориентированными, а другие — более техническими, полезными при разработке и сопровождении сайтов.

Вот несколько ссылок для сбора метрик FastAPI:

- Prometheus FastAPI Instrumentator (<https://oreil.ly/EYJwR>);
- Getting Started: Monitoring a FastAPI App with Grafana and Prometheus — A Step-by-Step Guide, автор Зю Кодекс (<https://oreil.ly/Gs90t>);
- страница FastAPI Observability на сайте Grafana Labs (<https://oreil.ly/spKwe>);

- OpenTelemetry FastAPI Instrumentation (<https://oreil.ly/wDSNv>);
- OpenTelemetry FastAPI Tutorial — Complete Implementation Guide, автор Анкит Ананд (<https://oreil.ly/ZpSXs>);
- документация OpenTelemetry Python (<https://oreil.ly/nSD4G>).

Заключение

Понятно, что производство — дело непростое. Среди проблем — сама веб-техника, перегрузка сети и дисков, а также проблемы с базой данных. В этой главе вы найдете подсказки о том, как получить нужную информацию и где начать искать, если возникли проблемы.

ЧАСТЬ IV

Галерея

В части III вы создали минимальный веб-сайт с помощью базового кода. А теперь давайте сделаем с ним что-нибудь интересное. В следующих главах FastAPI применяется для обычных веб-приложений: форм, файлов, баз данных, диаграмм, графиков, карт и игр.

Чтобы связать эти приложения и сделать их более интересными, чем обычные сухие примеры из книг по информатике, возьмем из необычных источников данные, часть которых вы уже видели: вымышленных существ из мирового фольклора и исследователей, которые их преследуют. Здесь будут не только йети, но и менее известные, хотя и не менее яркие представители этого мира.

ГЛАВА 14

Базы данных, наука о данных и немного искусственного интеллекта

Обзор

В этой главе рассказывается о том, как использовать FastAPI для хранения и получения данных. Здесь расширяются простые примеры SQLite, приведенные в главе 10:

- другие базы данных с открытым исходным кодом (реляционные и нереляционные);
- использование SQLAlchemy на более высоком уровне;
- улучшенная проверка ошибок.

Альтернативные варианты хранения данных



Термин «база данных», к сожалению, используется для обозначения трех вещей:

- типа сервера, например PostgreSQL, SQLite или MySQL;
- работающего экземпляра этого сервера;
- коллекции таблиц на этом сервере.

Чтобы избежать путаницы, называя экземпляр последнего из перечисленных пунктов базой данных PostgreSQL, я буду использовать другие термины, чтобы указать, какой из них имею в виду.

Обычный бэкенд для веб-сайта — это база данных. Веб-сайты и базы данных — это как арахисовое масло и желе, и хотя вы можете хранить свои данные и другими способами (или сочетать арахисовое масло с огурцами), в этой книге мы будем использовать базы данных.

Базы данных решают многие проблемы, которые в противном случае вам пришлось бы решать самостоятельно с помощью кода, например такие:

- множественный доступ;
- индексирование;
- согласованность данных.

В целом выбор баз данных выглядит следующим образом:

- реляционные базы данных с языком запросов SQL;
- нереляционные базы данных с различными языками запросов.

Реляционные базы данных и SQL

В Python есть стандартное определение реляционного API под названием DB-API (<https://oreil.ly/StbE4>). Оно поддерживается пакетами драйверов Python для всех основных баз данных. В табл. 14.1 перечислены некоторые известные реляционные базы данных и их основные пакеты драйверов для Python.

Таблица 14.1. Реляционные базы данных и драйверы Python

База данных	Драйвер Python
<i>С открытым исходным кодом</i>	
SQLite (https://www.sqlite.org)	sqlite3 (https://oreil.ly/TNNaA)
PostgreSQL (https://www.postgresql.org)	psycopg2 (https://oreil.ly/nLn5x) и asyncpg (https://oreil.ly/90pvK)
MySQL (https://www.mysql.com)	MySQLdb (https://oreil.ly/yn1fn) и PyMySQL (https://oreil.ly/Cmup-)
<i>Коммерческие</i>	
Oracle (https://www.oracle.com)	python-oracledb (https://oreil.ly/gynvX)
SQL Server (https://www.microsoft.com/en-us/sql-server)	pyodbc (https://oreil.ly/_UEYq) и pymssql (https://oreil.ly/FkKUn)
IBM Db2 (https://www.ibm.com/products/db2)	ibm_db (https://oreil.ly/3uwpD)

Основные пакеты Python для работы с реляционными базами данных и SQL:

- *SQLAlchemy* (<https://www.sqlalchemy.org>) — полнофункциональная библиотека, которую можно использовать на разных уровнях;
- *SQLModel* (<https://sqlmodel.tiangolo.com>) — комбинация SQLAlchemy и Pydantic от автора FastAPI;
- *Records* (<https://github.com/kennethreitz/records>) — от автора пакета Requests — простой API для запросов.

SQLAlchemy

Самым популярным SQL-пакетом для Python стал SQLAlchemy. Хотя во многих объяснениях SQLAlchemy обсуждаются только возможности ORM этой библиотеки, она содержит несколько слоев, и я буду рассматривать их снизу вверх.

Core

Основа SQLAlchemy, называемая *Core*, включает в себя следующее:

- объект *Engine*, реализующий стандарт DB-API;
- URL-адреса, выражающие тип и драйвер SQL-сервера, а также конкретную коллекцию баз данных на этом сервере;
- пулы соединений «клиент — сервер»;
- транзакции (COMMIT и ROLLBACK);
- различия в *диалектах* SQL для различных типов баз данных;
- прямые запросы SQL (текстовые строки);
- запросы на языке выражений SQLAlchemy.

Некоторые из этих возможностей, например работа с диалектами, делают SQLAlchemy оптимальным пакетом для работы с различными типами серверов. С его помощью можно выполнять обычные SQL-запросы DB-API или использовать язык выражений SQLAlchemy.

До этого момента я работал с базовым драйвером DB-API SQLite и буду продолжать делать это. Но для больших сайтов или при необходимости вос-

пользоваться специальными возможностями сервера стоит взять SQLAlchemy (применяя базовый DB-API, SQLAlchemy Expression Language или полноценный ORM).

Язык выражений SQLAlchemy

Язык выражений SQLAlchemy (SQLAlchemy Expression Language) — это *не* ORM, а другой способ выражения запросов к реляционным таблицам. Он отображает базовые структуры хранения данных на классы Python, такие как `Table` и `Column`, и операции с методами Python, такими как `select()` и `insert()`. Эти функции преобразуются в обычные строки SQL, и вы можете обратиться к ним, чтобы посмотреть, что произошло. Язык не зависит от типов SQL-серверов. Если вам трудно дается SQL, возможно, стоит попробовать этот вариант.

Сравним несколько примеров. В примере 14.1 показана версия исключительно на языке SQL.

Пример 14.1. Прямой код SQL для функции `get_one()` в файле `data/explorer.py`

```
def get_one(name: str) -> Explorer:
    qry = "select * from explorer where name=:name"
    params = {"name": name}
    curs.execute(qry, params)
    return row_to_model(curs.fetchone())
```

В примере 14.2 показан частичный эквивалент SQLAlchemy Expression Language для настройки базы данных, создания таблицы и выполнения вставки.

Пример 14.2. SQLAlchemy Expression Language для функции `get_one()`

```
from sqlalchemy import Metadata, Table, Column, Text
from sqlalchemy import connect, insert

conn = connect("sqlite:///cryptid.db")
meta = Metadata()
explorer_table = Table(
    "explorer",
    meta,
    Column("name", Text, primary_key=True),
    Column("country", Text),
    Column("description", Text),
)
insert(explorer_table).values(
    name="Beau Buffette",
    country="US",
    description="...")
```

Для получения большего количества примеров можно воспользоваться альтернативной документацией (<https://oreil.ly/ZGCHv>) — она читается немного легче, чем официальные страницы.

ORM

ORM выражает запросы в терминах моделей данных домена, а не реляционных таблиц и логики SQL, лежащих в основе механизма базы данных. В официальной документации (<https://oreil.ly/x4DCi>) приведена подробная информация. ORM гораздо сложнее, чем язык выражений SQL. Разработчики, предпочитающие полностью *объектно-ориентированные* модели, обычно выбирают ORM.

Во многих книгах и статьях о FastAPI, начиная раздел, посвященный базам данных, авторы сразу же переходят к ORM SQLAlchemy. Я понимаю, что это привлекательно, но также знаю, что это требует изучения еще одной абстракции. SQLAlchemy — отличный пакет, но если его абстракции не всегда работают, то у вас возникают две проблемы. Самым простым решением может быть использование SQL и переход к языку выражений или ORM, если SQL становится слишком сложным.

SQLModel

Автор FastAPI объединил аспекты FastAPI, Pydantic и SQLAlchemy, чтобы создать библиотеку SQLModel (<https://sqlmodel.tiangolo.com>). Он переносит некоторые методы разработки из веб-мира в реляционные базы данных. SQLModel сочетает в себе ORM от SQLAlchemy с определением и проверкой данных от Pydantic.

SQLite

Пакет SQLite был представлен в главе 10, я использовал его в примерах уровня данных. Это общественное достояние — более открытого исходного кода и не придумаешь. SQLite применяется в каждом браузере и в каждом смартфоне, что делает его одним из самых распространенных программных пакетов в мире. При выборе реляционной базы данных этот пакет часто упускают из виду, но вполне возможно, что несколько «серверов» SQLite смогут поддерживать некоторые крупные сервисы не хуже, чем такой мощный сервер, как PostgreSQL.

PostgreSQL

На заре развития реляционных баз данных пионером была система System R от IBM, а за новый рынок боролись ее ответвления — в основном Ingres с открытым исходным кодом и коммерческий продукт Oracle. В Ingres был применен язык запросов QUEL, а в System R — SQL. Хотя некоторые считали, что QUEL лучше, чем SQL, принятие Oracle SQL в качестве стандарта, а также влияние IBM помогли Oracle и SQL добиться успеха.

Спустя годы Майкл Стоунбрейкер вернулся, чтобы осуществить переход от Ingres к PostgreSQL (<https://www.postgresql.org>). В настоящее время разработчики систем с открытым исходным кодом чаще всего выбирают PostgreSQL, хотя система MySQL была популярна несколько лет назад и до сих пор не потеряла своей актуальности.

EdgeDB

Несмотря на многолетний успех SQL, у него есть некоторые недостатки, делающие запросы неудобными. В отличие от математической теории, на которой основан SQL (*реляционное исчисление* Э. Ф. Кодда), сама конструкция языка SQL не является *композиционной*. В основном это означает, что сложно вложить запросы в большие запросы, что порождает более сложный и многословный код.

Поэтому просто для развлечения я создам здесь новую реляционную базу данных. EdgeDB (<https://www.edgedb.com>) была написана (на Python!) автором библиотеки `asyncio` для языка Python. Она описывается как *Post-SQL* или *граф-реляционная*. В ядре БД используется PostgreSQL для обработки сложных системных задач. Компания Edge привнесла в эту сферу свой продукт EdgeQL (<https://oreil.ly/sdK4J>) — новый язык запросов, стремясь избежать острых граней SQL. На самом деле он переводится на SQL для выполнения PostgreSQL. В статье Ивана Данилюка *My Experience with EdgeDB* (<https://oreil.ly/ciNfg>) приведено удобное сравнение EdgeQL и SQL. Приятная для чтения иллюстрированная официальная документация (<https://oreil.ly/ce6y3>) проводит параллели с книгой «Дракула».

Может ли EdgeQL распространиться за пределы EdgeDB и стать альтернативой SQL? Время покажет.

Нереляционные (NoSQL) базы данных

Крупные игроки в мире NoSQL или NewSQL с открытым исходным кодом перечислены в табл. 14.2.

Таблица 14.2. Базы данных NoSQL и драйверы Python

База данных	Драйвер Python
Redis (https://redis.io)	redis-py (https://github.com/redis/redis-py)
MongoDB (https://www.mongodb.com)	PyMongo (https://pymongo.readthedocs.io), Motor (https://oreil.ly/Cmgtl)
Apache Cassandra (https://cassandra.apache.org)	DataStax Driver for Apache Cassandra (https://github.com/datastax/python-driver)
Elasticsearch (https://www.elastic.co/elasticsearch)	Python Elasticsearch Client (https://oreil.ly/e_bDI)

Иногда NoSQL означает буквально «*отсутствие SQL*», а иногда «*не только SQL*». Реляционные базы данных накладывают структуру на данные. Часто она визуализируется в виде прямоугольных таблиц с полями — столбцами и строками данных, подобно электронным таблицам. Чтобы уменьшить избыточность и повысить производительность, реляционные базы данных *нормализуются* с помощью *нормальных форм* (правил для данных и структур), например допускают только одно значение в ячейке (на пересечении строки и столбца).

Базы данных NoSQL делают эти правила менее строгими, иногда позволяя варьировать типы столбцов/полей в отдельных строках данных. Часто *схемы* (дизайн баз данных) могут представлять собой не реляционные ячейки, а разрозненные структуры, которые можно выразить на JSON или Python.

Redis

Redis — это сервер структур данных, работающий исключительно в оперативной памяти, хотя он может сохранять данные на диск и восстанавливать их с диска. Он полностью соответствует собственным структурам данных Python и стал очень популярным.

MongoDB

MongoDB — это своего рода PostgreSQL для NoSQL-серверов. *Коллекция* — это эквивалент таблицы SQL, а *документ* — эквивалент строки таблицы SQL. Еще одно отличие — и главная причина, по которой база данных NoSQL является основной, — заключается в том, что вам не нужно определять, как выглядит документ. Другими словами, нет никакой фиксированной *схемы*. Документ — это как словарь Python, ключом в нем может быть любая строка.

Cassandra

Cassandra — это крупномасштабная база данных, ее можно распределить между сотнями узлов. Она написана на языке Java.

Альтернативная база данных называется ScyllaDB (<https://www.scylladb.com>). Она написана на C++, и утверждается, что она совместима с Cassandra, но имеет бóльшую производительность.

Elasticsearch

Elasticsearch (<https://www.elastic.co/elasticsearch>) больше похожа на индекс базы данных, чем на саму базу данных. Она часто используется для полнотекстового поиска.

Возможности NoSQL в базах данных SQL

Как отмечалось ранее, реляционные базы данных традиционно нормализуются и ограничиваются различными уровнями правил, называемых *нормальными формами*. Одним из основных правил было то, что значение в каждой ячейке должно быть *скаляром* — никаких массивов или других структур.

Базы данных NoSQL (или документоориентированные) поддерживали JSON напрямую и обычно оказывались единственным выбором, если у вас были неравномерные или неровные структуры данных. Часто они были *денормализованными* — все данные, необходимые для документа, были включены в него. В SQL для создания полного документа часто требовалось выполнить *объединение* данных из разных таблиц.

Однако последние изменения в стандарте SQL позволили хранить данные JSON и в реляционных базах данных. Некоторые из таких БД позволяют хранить сложные (нескалярные) данные в ячейках таблиц и даже выполнять в них поиск и индексирование. Функции JSON поддерживаются различными способами для SQLite (https://oreil.ly/h_FNn), PostgreSQL (<https://oreil.ly/awYrc>), MySQL (https://oreil.ly/OA_sT), Oracle (<https://oreil.ly/osOYk>) и других систем.

SQL с JSON может быть лучшим из двух миров. Базы данных SQL существуют гораздо дольше и поддерживают действительно полезные функции, такие как внешние ключи и вторичные индексы. Кроме того, SQL довольно хорошо стандартизирован до определенного момента, а языки запросов NoSQL все разные.

Наконец, новые языки проектирования данных и запросов пытаются объединить преимущества SQL и NoSQL, как, например, EdgeQL, о котором я упоминал ранее. Поэтому, если вы не можете уместить свои данные в прямоугольную реляционную коробку, обратите внимание на базу данных NoSQL, реляционную базу данных с поддержкой JSON или базу данных Post-SQL.

Нагрузочное тестирование баз данных

Эта книга в основном посвящена FastAPI, но веб-сайты слишком часто связаны с базами данных.

Примеры данных в этой книге были крошечными. Чтобы действительно протестировать базу данных на стрессоустойчивость, было бы неплохо использовать миллионы элементов. Вместо того чтобы их придумывать и добавлять вручную, проще воспользоваться пакетом Python, например Faker (<https://faker.readthedocs.io>). Он может быстро генерировать различные типы данных — имена, места или определяемые вами специальные типы.

В примере 14.3 функция Faker выкачивает имена и страны, а затем они загружаются с помощью функции `load()` в SQLite.

Пример 14.3. Загрузка фиктивных исследователей в файл `test_load.py`

```
from faker import Faker
from time import perf_counter

def load():
    from error import Duplicate
    from data.explorer import create
    from model.explorer import Explorer
```

```
f = Faker()
NUM = 100_000
t1 = perf_counter()
for row in range(NUM):
    try:
        create(Explorer(name=f.name(),
                        country=f.country(),
                        description=f.description))
    except Duplicate:
        pass
t2 = perf_counter()
print(NUM, "rows")
print("write time:", t2-t1)

def read_db():
    from data.explorer import get_all

    t1 = perf_counter()
    _ = get_all()
    t2 = perf_counter()
    print("db read time:", t2-t1)

def read_api():
    from fastapi.testclient import TestClient
    from main import app

    t1 = perf_counter()
    client = TestClient(app)
    _ = client.get("/explorer/")
    t2 = perf_counter()
    print("api read time:", t2-t1)

load()
read_db()
read_db()
read_api()
```

Код будет улавливать исключение `Duplicate` в функции `load()`, но его стоит игнорировать, потому что `Faker` генерирует имена из ограниченного списка и, скорее всего, время от времени повторяет некоторые из них. Таким образом, в результате может быть загружено менее 100 000 исследователей.

Кроме того, вы вызываете функцию `read_db()` дважды, чтобы исключить время запуска процесса, пока `SQLite` выполняет запрос. Тогда время выполнения `read_api()` должно получиться честным. Пример 14.4 запускает тестирование.

Пример 14.4. Проверка производительности запросов к базе данных

```
$ python test_load.py
100000 rows
write time: 14.868232927983627
db read time: 0.4025074450764805
db read time: 0.39750714192632586
api read time: 2.597553930943832
```

Время чтения API для всех исследователей было намного медленнее, чем время чтения уровня данных. Вероятно, часть этих расходов связана с преобразованием ответа в JSON с помощью FastAPI. Кроме того, время первоначальной записи в базу данных было не очень быстрым. Код записывает по одному исследователю за раз, потому что в API уровня данных есть единственная функция `create()`, но нет функции `create_many()`. В части считывания API может вернуть один (`get_one()`) или все (`get_all()`) результаты. Поэтому, если вы хотите выполнять массовую загрузку, возможно, стоит добавить новую функцию загрузки данных и новую конечную точку веб-приложения (с ограниченной авторизацией). Кроме того, если вы ожидаете, что любая таблица в базе данных вырастет до 100 000 строк, возможно, не стоит позволять случайным пользователям получать их все за один вызов API. Не помешала бы пагинация или возможность загрузки одного CSV-файла из таблицы.

Наука о данных и искусственный интеллект

Python стал самым популярным языком в области науки о данных в целом и машинного обучения (Machine Learning, ML) в частности. Там необходимо много работать с данными, и Python отлично справляется с этой задачей.

Иногда разработчики используют сторонние инструменты (<https://oreil.ly/WFH09>), такие как pandas, для манипулирования слишком сложными в SQL-представлении данными.

PyTorch (<https://pytorch.org>) — один из самых популярных инструментов ML, поскольку в работе с данными он использует сильные стороны Python. Для повышения скорости базовые вычисления могут выполняться на C или C++, а для высших задач интеграции данных хорошо подходят Python или Go. Язык Mojo (<https://www.modular.com/mojo>) — супернабор Python — сможет справиться с задачами и большой, и малой сложности, если все получится создать так, как задумано. Хотя это язык общего назначения, он специально предназначен для решения некоторых текущих проблем при разработке ИИ.

Новый инструмент Python под названием Chroma (<https://www.trychroma.com>) — это база данных, похожая на SQLite, но предназначенная для машинного обучения, в частности для больших языковых моделей (Large Language Models, LLM). Прочтите страницу Getting Started page (<https://oreil.ly/W59nn>), чтобы быстрее начать работу.

Хотя разработка ИИ сложна и идет быстрыми темпами, вы можете опробовать искусственный интеллект с помощью Python на собственной машине, не тратя мегасредства, затраченные на создание GPT-4 и ChatGPT. Создадим небольшой веб-интерфейс FastAPI для небольшой модели искусственного интеллекта.



Понятие «модель» имеет разные значения в области ИИ и Pydantic/FastAPI. В Pydantic модель — это класс Python, объединяющий связанные поля данных. Модели ИИ охватывают широкий спектр методов определения закономерностей в данных.

Платформа Hugging Face (<https://huggingface.co>) предоставляет бесплатные модели искусственного интеллекта, наборы данных и код на Python для их использования. Сначала установите PyTorch и код Hugging Face:

```
$ pip install torch torchvision
$ pip install transformers
```

В примере 14.5 показано приложение FastAPI, использующее модуль трансформеров с платформы Hugging Face для доступа к предварительно обученной модели машинного языка среднего размера с открытым исходным кодом. Эта программа попытается ответить на ваши вопросы. (Код был адаптирован из примера командной строки, приведенного на YouTube-канале CodeToTheMoon.)

Пример 14.5. Тесты высокого уровня для LLM (ai.py)

```
from fastapi import FastAPI

app = FastAPI()

from transformers import (AutoTokenizer,
                          AutoModelForSeq2SeqLM, GenerationConfig)
model_name = "google/flan-t5-base"
tokenizer = AutoTokenizer.from_pretrained(model_name)
model = AutoModelForSeq2SeqLM.from_pretrained(model_name)
config = GenerationConfig(max_new_tokens=200)

@app.get("/ai")
def prompt(line: str) -> str:
```

```
tokens = tokenizer(line, return_tensors="pt")
outputs = model.generate(**tokens,
                          generator_config=config)
result = tokenizer.batch_decode(outputs,
                                skip_special_tokens=True)
return result[0]
```

Запустите этот код с помощью выражения `uvicorn ai:app` (как всегда, сначала убедитесь, что у вас нет другого все еще запущенного веб-сервера на адресе `localhost`, порт `8000`). Задавайте вопросы конечной точке `/ai` и получайте ответы, например, так (обратите внимание на двойной знак равенства `==` для параметра запроса HTTPie):

```
$ http -b localhost:8000/ai line=="What are you?"
"a sailor"
```

Это довольно маленькая модель, и, как вы можете видеть, она не особенно хорошо отвечает на вопросы. Я попробовал другие задания (`line`-аргументы) и получил не менее достойные ответы.

В: Лучше ли кошки, чем собаки?

О: Нет.

В: Что йети ест на завтрак?

О: Кальмара.

В: Кто спускается по дымоходу?

О: Визжащий поросенок.

В: В какой группе состоял Джон Клиз?

О: The Beatles.

В: У чего есть противные острые зубы?

О: Плюшевый мишка.

В разное время на эти вопросы можно получить разные ответы! Однажды та же самая конечная точка ответила, что йети ест на завтрак песок. В среде ИИ подобные ответы называются *галлюцинациями*. Вы можете получить более точные ответы, задействуя более крупную модель, например `google/flan-75xl`, но для загрузки данных модели на персональный компьютер и получения ответа потребуется больше времени. И конечно, такие модели, как ChatGPT, были обучены на всех данных, которые смогли найти (с использованием всех ЦП, ГП, ТП и любых других видов процессоров), и дадут отличные ответы.

Заключение

В этой главе мы распространили возможности применения SQLite, описанные в главе 10, на другие базы данных SQL и даже NoSQL. Здесь также показано, как некоторые базы данных SQL могут выполнять трюки NoSQL с поддержкой JSON. Наконец, речь шла об использовании баз данных и специальных инструментов для работы с данными, которые становятся все более важными по мере того, как машинное обучение продолжает бурно развиваться.

ГЛАВА 15

Файлы

Обзор

Помимо обработки API-запросов и традиционного контента, например HTML, веб-серверы должны обрабатывать передачу файлов в обоих направлениях. Очень большие файлы могут передаваться *частями*, чтобы не занимать много памяти системы. Вы также можете предоставить доступ к папке с файлами (и подчиненными папками любой глубины) с помощью статических файлов — *Static Files*.

Поддержка Multipart

Чтобы обрабатывать большие файлы, функции загрузки и скачивания FastAPI нуждаются в следующих дополнительных модулях:

- *Python-Multipart* (<https://oreil.ly/FUBk7>) — `pip install python-multipart`;
- *aio-files* (<https://oreil.ly/OZYR>) — `pip install aiofiles`.

Выгрузка файлов

FastAPI нацелен на разработку API, и в большинстве примеров в этой книге используются запросы и ответы в формате JSON. Но в следующей главе вы познакомитесь с формами, которые обрабатываются по-другому. Здесь же рассказывается о файлах, по некоторым параметрам похожих на формы.

FastAPI предлагает два способа загрузки файлов: функцию `File()` и класс `UploadFile`.

Функция File()

Функция `File()` применяется в качестве типа для прямой загрузки файла. Ваша функция пути может быть синхронной (`def`) или асинхронной (`async def`), но асинхронная версия лучше, потому что она не будет нагружать веб-сервер во время загрузки файла.

FastAPI будет извлекать файл по частям и собирать его в памяти, поэтому функцию `File()` следует использовать только для относительно небольших файлов. Вместо того чтобы считать, что входные данные представлены в формате JSON, FastAPI кодирует файл как элемент формы.

Напишем код для запроса файла и протестируем его. Вы можете взять любой файл на своей машине для тестирования или загрузить его с такого сайта, как Fastest Fish (<https://oreil.ly/EnIH->). Я взял оттуда файл размером 1 Кбайт и сохранил его локально под названием `1KB.bin`. В примере 15.1 добавьте эти строки в файл `main.py` верхнего уровня.

Пример 15.1. Обработка загрузки небольшого файла с помощью FastAPI

```
from fastapi import File

@app.post("/small")
async def upload_small_file(small_file: bytes = File()) -> str:
    return f"file size: {len(small_file)}"
```

После перезапуска Uvicorn попробуйте выполнить HTTPie-тест в примере 15.2.

Пример 15.2. Выгрузка небольшого файла с помощью HTTPie

```
$ http -f -b POST http://localhost:8000/small small_file@1KB.bin
"file size: 1000"
```

Приведу несколько замечаний по этому тесту.

- Необходимо добавить аргумент `-f` или `--form`, потому что файлы выгружаются как формы, а не как JSON-текст.
- `small_file@1KB.bin`:
 - `small_FILE` соответствует имени переменной `small_file` в функции пути FastAPI в примере 15.1;
 - `@` — сокращение HTTPie для создания формы;
 - `1KB.bin` — выгружаемый файл.

Пример 15.3 представляет собой эквивалентный программный тест.

Пример 15.3. Выгрузка небольшого файла с помощью Requests

```
$ python
>>> import requests
>>> url = "http://localhost:8000/small"
>>> files = {'small_file': open('1KB.bin', 'rb')}
>>> resp = requests.post(url, files=files)
>>> print(resp.json())
file size: 1000
```

Класс UploadFile

Для больших файлов лучше использовать класс `UploadFile`. Он создает объект Python под названием `SpooledTemporary File`, обычно на диске сервера, а не в памяти. Это *файлоподобный* объект Python, и он поддерживает методы `read()`, `write()` и `seek()`. Пример 15.4 показывает реализацию этого подхода; в нем также используется объявление `async def` вместо `def`, чтобы избежать блокировки веб-сервера во время выгрузки частей файла.

Пример 15.4. Выгрузка большого файла с помощью FastAPI
(добавить к файлу `main.py`)

```
from fastapi import UploadFile

@app.post("/big")
async def upload_big_file(big_file: UploadFile) -> str:
    return f"file size: {big_file.size}, name: {big_file.filename}"
```



Функция `File()` создает объект `bytes` и нуждается в круглых скобках. `UploadFile` — это объект другого класса.

Если стартер `Uvicorn` еще не износился, значит, пришло время испытаний. На этот раз в примерах 15.5 и 15.6 используется файл размером 1 Гбайт (`1GB.bin`), я взял его на сайте `Fastest.Fish`.

Пример 15.5. Тестирование выгрузки большого файла с помощью HTTPie

```
$ http -f -b POST http://localhost:8000/big big_file@1GB.bin
"file size: 1000000000, name: 1GB.bin"
```

Пример 15.6. Тестирование выгрузки большого файла с помощью Requests

```
>>> import requests
>>> url = "http://localhost:8000/big"
>>> files = {'big_file': open('1GB.bin', 'rb')}
>>> resp = requests.post(url, files=files)
>>> print(resp.json())
file size: 1000000000, name: 1GB.bin
```

Загрузка файлов

К сожалению, гравитация не ускоряет скачивание файлов. Вместо этого мы будем использовать эквиваленты методов выгрузки.

Класс FileResponse

Первым (пример 15.7) представлен вариант «все и сразу», класс `FileResponse`.

Пример 15.7. Загрузка небольшого файла с помощью `FileResponse` (добавить в файл `main.py`)

```
from fastapi.responses import FileResponse

@app.get("/small/{name}")
async def download_small_file(name):
    return FileResponse(name)
```

Где-то здесь есть тест. Сначала поместите файл `1KB.bin` в тот же каталог, что и `main.py`. Теперь запустите пример 15.8.

Пример 15.8. Загрузка небольшого файла с помощью `HTTPie`

```
$ http -b http://localhost:8000/small/1KB.bin
```

```
-----
| NOTE: binary data not shown in terminal |
-----
```

Если вы не доверяете этому сообщению об ограничениях, то пример 15.9 направляет вывод в утилиту типа `ws`, чтобы убедиться, что вы получили все 1000 байт.

Пример 15.9. Загрузка небольшого файла с помощью HTTPie с подсчетом байтов

```
$ http -b http://localhost:8000/small/1KB.bin | wc -c
1000
```

Класс `StreamingResponse`

Как и в случае с модулем `FileUpload`, большие файлы лучше загружать с помощью класса `StreamingResponse`, возвращающего файл по частям. Пример 15.10 показывает такой подход к реализации с помощью функции пути, определенной как `async def`. Он позволяет избежать блокировки, когда процессор не используется. Я пока пропускаю проверку ошибок. Если файла `path` не существует, вызов функции `open()` выбросит исключение.

Пример 15.10. Возврат большого файла с помощью класса `StreamingResponse` (добавить в файл `main.py`)

```
from pathlib import Path
from typing import Generator
from fastapi.responses import StreamingResponse
```

```
def gen_file(path: str) -> Generator:
    with open(file=path, mode="rb") as file:
        yield file.read()
```

```
@app.get("/download_big/{name}")
async def download_big_file(name:str):
    gen_expr = gen_file(file_path=Path(name))
    response = StreamingResponse(
        content=gen_expr,
        status_code=200,
    )
    return response
```

`gen_expr` — *выражение-генератор*, возвращаемое *функцией-генератором* `gen_file()`. Класс `StreamingResponse` использует его для своего итерируемого аргумента `content`, чтобы загружать файл по частям.

Пример 15.11 представляет собой сопутствующий тест. (Для этого сначала потребуется разместить файл `1GB.bin` рядом с файлом `main.py`, процесс займет *немного* больше времени.)

Пример 15.11. Загрузка большого файла с помощью HTTPie

```
$ http -b http://localhost:8000/big/1GB.bin | wc -c
1000000000
```

Предоставление статических файлов

Традиционные веб-серверы могут обращаться с файлами сервера так, как будто они находятся в обычной файловой системе. FastAPI позволяет делать это с помощью класса `StaticFiles`.

Для этого примера создадим каталог скучных бесплатных файлов для загрузки пользователями.

- Создайте каталог `static` на том же уровне, что и файл `main.py`. (У этого хранилища может быть любое название, я называю его `static` (статическим) только для того, чтобы не забыть, зачем я его сделал.)
- Поместите в него текстовый файл `abc.txt` с текстовым содержимым `abc` :).

Пример 15.12 предоставит любой URL, начинающийся с выражения `/static` (вы могли бы использовать здесь любую текстовую строку), с файлами из каталога `static`.

Пример 15.12. Предоставление всего содержимого в каталоге с помощью `StaticFiles` (добавить в файл `main.py`)

```
from pathlib import Path
from fastapi import FastAPI
from fastapi.staticfiles import StaticFiles

# Каталог, содержащий файл main.py:
top = Path(__file__).resolve().parent

app.mount("/static",
          StaticFiles(directory=f"{top}/static", html=True),
          name="free")
```

Расчет `top` гарантирует, что вы разместите каталог `static` рядом с файлом `main.py`. Переменная `__file__` представляет собой полное имя пути к этому файлу.

Пример 15.13 — это один из способов проверки примера 15.12 вручную.

Пример 15.13. Получение статического файла

```
$ http -b localhost:8000/static/abc.txt
abc :)
```

А что насчет аргумента `html=True`, который мы передаем в функцию `StaticFiles()`? Это делает ее работу немного более похожей на работу традиционного сервера,

возвращая файл `index.html`, если он существует в этом каталоге, но вы не запрашивали файл `index.html` в явном виде в URL. Итак, создадим в каталоге `static` файл `index.html` с содержимым `Oh. Hi!`, а затем протестируем работу кода в примере 15.14.

Пример 15.14. Получение файла `index.html` из каталога `/static`

```
$ http -b localhost:8000/static/  
Oh. Hi!
```

У вас может быть любое необходимое количество файлов (и подкаталогов с файлами и т. д.). Создайте подкаталог `xyz` в каталоге `static` и поместите туда два файла:

- `xyz.txt` — содержит текст: `xyz : (;`
- `index.html` — содержит текст `How did you find me?`.

Я не буду приводить здесь примеры. Попробуйте запустить их сами (надеюсь, у вас более богатое воображение).

Заключение

В этой главе было показано, как выгружать и скачивать файлы — маленькие, большие и даже гигантские. Кроме того, вы научились предоставлять *статические файлы* в ностальгическом (не API) веб-стиле из каталога.

Формы и шаблоны

Обзор

Хотя акроним *API* в названии *FastAPI* — это намек на его основную направленность, FastAPI может работать и с традиционным веб-контентом. В этой главе рассказывается о стандартных HTML-формах и шаблонах для вставки данных в HTML.

Формы

Как вы уже поняли, FastAPI был разработан в основном для создания API, и его входной информацией по умолчанию будут данные в формате JSON. Но это не значит, что он не может служить стандартным базовым HTML-формам и их друзьям.

FastAPI поддерживает данные из HTML-форм так же, как и из других источников, таких как `Query` и `Path`, используя зависимость `Form`.

Для работы с формами FastAPI вам потребуется пакет `Python-Multipart`, поэтому при необходимости выполните команду `pip install python-multipart`. Кроме того, каталог `static` из главы 15 понадобится для размещения тестовых форм из нее.

Повторим пример 3.11, но предоставим значение `who` через форму, а не в виде JSON-строки. (Вызовите функцию пути `greet2()`, чтобы избежать нарушения работы старой функции пути `greet()`, если она все еще существует.) Добавьте пример 16.1 в файл `main.py`.

Пример 16.1. Получение значения из формы GET

```
from fastapi import FastAPI, Form

app = FastAPI()

@app.get("/who2")
def greet2(name: str = Form()):
    return f"Hello, {name}?"
```

Основное отличие заключается в том, что значение поступает от объекта `Form`, а не `Path`, `Query` и остальных из главы 3.

Попробуйте (пример 16.2) провести начальный тест формы с помощью `HTTPie` (вам потребуется аргумент `-f`, чтобы выгрузка происходила в кодировке формы, а не в формате JSON).

Пример 16.2. Формирование запроса GET с помощью `HTTPie`

```
$ http -f -b GET localhost:8000/who2 name="Bob Frapples"
"Hello, Bob Frapples?"
```

Можете также отправить запрос из стандартного файла HTML-формы. В главе 15 было показано, как создать каталог `static` (доступ к нему осуществляется по URL `/static`) для хранения любых данных, включая HTML-файлы, поэтому в примере 16.3 поместим туда этот файл (`form1.html`).

Пример 16.3. Формирование запроса GET (`static/form1.html`)

```
<form action="http://localhost:8000/who2" method="get">
Say hello to my little friend:
<input type="text" name="name" value="Bob Frapples">
<input type="submit">
</form>
```

Если вы попросите браузер загрузить страницу `http://localhost:8000/static/form1.html`, то увидите форму. Если введете любую тестовую строку, получите следующее сообщение:

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

А?

Посмотрите в окно, где запущен `Uvicorn`, чтобы увидеть, что написано в его журнале:

```
INFO:      127.0.0.1:63502 -  
      "GET /who2?name=rr23r23 HTTP/1.1"  
      422 Unprocessable Entity
```

Почему форма отправила переменную `name` в качестве параметра запроса, когда мы поместили ее в поле формы? Это оказалось странностью HTML, задокументированной на веб-сайте W3C (<https://oreil.ly/e6CJb>). Кроме того, если в вашем URL были параметры запроса, он сотрет их и заменит значением `name`.

Почему же HTTPie справился с этим, как и ожидалось? Мне это неизвестно. Это несоответствие, о котором следует знать.

Официальная магическая формула HTML заключается в том, чтобы изменить действие с GET на POST. Так что добавим конечную точку POST для `/who2` в файл `main.py` (пример 16.4).

Пример 16.4. Получение значения из формы POST

```
from fastapi import FastAPI, Form  
  
app = FastAPI()  
  
@app.post("/who2")  
def greet3(name: str = Form()):  
    return f"Hello, {name}?"
```

Пример 16.5 представляет собой файл `stuff/form2.html`, но с оператором `get`, замененным на `post`.

Пример 16.5. Формирование запроса POST (`static/form2.html`)

```
<form action="http://localhost:8000/who2" method="post">  
Say hello to my little friend:  
<input type="text" name="name">  
<input type="submit">  
</form>
```

Разбудите свой браузер и попросите его получить эту новую форму. Внесите в нее текст **Bob Frapples** и подтвердите отправку формы. На этот раз вы получите тот же результат, что и при использовании HTTPie:

```
"Hello, Bob Frapples?"
```

Поэтому, если отправляете формы из HTML-файлов, задействуйте метод POST.

Шаблоны

Возможно, вам знакома игра в слова *Mad Libs*. Игрокам дают последовательность слов — существительных, глаголов или чего-то более конкретного, они вставляют их в отмеченные места на странице текста. Вставив все слова, нужно прочитать текст — и начинается веселье, иногда сопровождаемое неловкостью.

Веб-шаблон — это то же самое, но, как правило, без неловкости. Шаблон содержит кучу текста со слотами для данных, вставляемых сервером. Его обычное назначение — генерировать HTML с переменным содержимым, в отличие от *статического* HTML из главы 15.

Пользователи Flask хорошо знакомы с его сопутствующим проектом — шаблонизатором Jinja (<https://jinja.palletsprojects.com>) (его часто называют также *Jinja2*). FastAPI поддерживает Jinja и другие шаблонизаторы.

Создайте каталог `template` рядом с файлом `main.py` для размещения HTML-файлов с поддержкой Jinja. Внутри создайте файл `list.html` (пример 16.6).

Пример 16.6. Определение шаблона файла (template/list.html)

```
<html>
<table bgcolor="#eeeeee">
  <tr>
    <th colspan=3>Creatures</th>
  </tr>
  <tr>
    <th>Name</th>
    <th>Description</th>
    <th>Country</th>
    <th>Area</th>
    <th>AKA</th>
  </tr>
  {% for creature in creatures: %}
    <tr>
      <td>{{ creature.name }}</td>
      <td>{{ creature.description }}</td>
      <td>{{ creature.country }}</td>
      <td>{{ creature.area }}</td>
      <td>{{ creature.aka }}</td>
    </tr>
  {% endfor %}
</table>

<br>
```

```
<table bgcolor="#dddddd">
  <tr>
    <th colspan=2>Explorers</th>
  </tr>
  <tr>
    <th>Name</th>
    <th>Country</th>
    <th>Description</th>
  </tr>
  {% for explorer in explorers: %}
    <tr>
      <td>{{ explorer.name }}</td>
      <td>{{ explorer.country }}</td>
      <td>{{ explorer.description }}</td>
    </tr>
  {% endfor %}
</table>
</html>
```

Неважно, как это выглядит, поэтому здесь не используется формальный язык CSS, только древний, существовавший еще до появления CSS, атрибут таблицы `bgcolor`, чтобы обеспечить различия между двумя таблицами.

Переменные Python, которые необходимо вставить, заключены в двойные фигурные скобки, а в наборы символов `{%}` и `%}` заключают операторы `if`, циклы `for` и другие структуры управления. В документации Jinja (<https://jinja.palletsprojects.com>) можно получить полную информацию по синтаксису и примерам.

Этот шаблон ожидает, что ему будут переданы переменные Python `creatures` и `explorers`, представляющие собой списки объектов `Creature` и `Explorer`.

В примере 16.7 показано, что нужно добавить в файл `main.py`, чтобы установить шаблоны и использовать данные из примера 16.6. Код подает переменные `creatures` и `explorers` в шаблон, применяя модули в *фиктивном* каталоге из предыдущих глав — эта папка предоставляла тестовые данные, если БД была пуста или не подключена.

Пример 16.7. Настройка шаблонов и использование одного из них (`main.py`)

```
from pathlib import Path
from fastapi import FastAPI, Request
from fastapi.templating import Jinja2Templates

app = FastAPI()

top = Path(__file__).resolve().parent
```

```

template_obj = Jinja2Templates(directory=f"{top}/template")

# Получение нескольких небольших предопределенных списков наших приятелей:
from fake.creature import fakes as fake_creatures
from fake.explorer import fakes as fake_explorers

@app.get("/list")
def explorer_list(request: Request):
    return template_obj.TemplateResponse("list.html",
        {"request": request,
         "explorers": fake_explorers,
         "creatures": fake_creatures})

```

Задайте своему любимому браузеру или даже тому, который вам не очень нравится, адрес `http://localhost:8000/list`, и вы получите в ответ рис. 16.1.

Creatures				
Name	Description	Country	Area	AKA
Yeti	Hirsute Himalayan	CN	Himalayas	Abominable Snowman
Bigfoot	New world Cousin Eddie of the yeti	US	*	Sasquatch

Explorers		
Name	Country	Description
Claude Hande	FR	Scarce during full moons
Noah Weiser	DE	Myopic machete man

Рис. 16.1. Вывод из каталога `/list`

Заключение

В этой главе был дан краткий обзор того, как FastAPI работает с темами, не относящимися к API, такими как формы и шаблоны. Наряду с рассмотренным в предыдущей главе о файлах, это традиционные минимально необходимые веб-задачи, с ними вы часто сталкиваетесь.

Обнаружение и визуализация данных

Обзор

Несмотря на то что в названии фреймворка FastAPI фигурирует акроним *API*, он может служить не только для API. В этой главе вы узнаете, как создавать таблицы, графики, диаграммы и карты на основе данных, используя небольшую базу данных о воображаемых существах со всего мира.

Python и данные

В последние несколько лет Python стал очень популярным по многим причинам:

- он легок в обучении;
- у него прозрачный синтаксис;
- он имеет богатую стандартную библиотеку;
- в нем огромное количество высококачественных пакетов сторонних разработчиков;
- особое внимание в нем уделяется манипулированию данными, преобразованию и самостоятельной проверке.

Последний пункт всегда был актуален для традиционных ETL-задач по созданию баз данных. Некоммерческая группа PyData (<https://pydata.org>) даже организует конференции и разрабатывает инструменты для анализа данных с открытым исходным кодом на Python. Популярность Python отражают также недавний всплеск развития искусственного интеллекта и потребность в инструментах для подготовки данных, используемых в моделях ИИ.

В этой главе мы попробуем применить несколько пакетов данных Python и посмотрим, как они связаны с современной веб-разработкой на Python и FastAPI.

Текстовый вывод с помощью PSV

В этом разделе мы будем использовать существ, перечисленных в приложении Б. Данные находятся в репозитории GitHub этой книги, в файле `cryptid.psv` с разделителем в виде вертикальной черты и в базе данных SQLite `cryptid.db`. Файлы, где используется разделение запятыми (`.csv`) и табуляцией (`.tsv`), широко распространены, но запятые используются в самих ячейках данных, а табуляцию иногда трудно отличить от других пробельных символов. Символ вертикальной черты (`|`) отличается от прочих и достаточно редко встречается в стандартном тексте, чтобы служить хорошим разделителем.

Сначала попробуем задействовать текстовый файл с расширением `.psv`, для простоты используя только примеры вывода текста, а затем перейдем к полноценным веб-примерам с применением базы данных SQLite.

В начальной строке заголовка файла `.psv` содержатся имена полей:

- `name`;
- `country` (символ `*` означает множество стран);
- `area` (не обязательно, штат США или другое территориальное образование страны);
- `description`;
- `aka` (обозначает «также известен как»).

В остальных строках файла описывается по одному существу, поля располагаются в таком порядке и разделяются символом `|`.

Модуль csv

Пример 17.1 считывает данные о существе в структуры данных Python. Во-первых, файл `cryptids.psv`, где используется разделение символами вертикальной черты, можно считать с помощью стандартного пакета `csv` Python, получив список кортежей, где каждый кортеж представляет собой строку данных из файла. (Пакет `csv` включает также класс `DictReader`, возвращающий список словарей.) Первая строка этого файла представляет собой заголовок с именами столбцов. Без этого мы могли бы предоставлять заголовки через аргументы для функций `csv`.

Я включаю в примеры подсказки типов, но вы можете отказаться от них, если у вас более старая версия Python, — код все равно будет работать. Напечатаем только заголовок и первые пять строк, чтобы сохранить несколько деревьев¹.

Пример 17.1. Считывание файла PSV с помощью `csv` (`load_csv.py`)

```
import csv
import sys

def read_csv(fname: str) -> list[tuple]:
    with open(fname) as file:
        data = [row for row in csv.reader(file, delimiter="|")]
    return data

if __name__ == "__main__":
    data = read_csv(sys.argv[1])
    for row in data[0:5]:
        print(row)
```

Теперь запустите тест из примера 17.2.

Пример 17.2. Тестирование загрузки базы данных CSV

```
$ python load_csv.py cryptid.psv
['name', 'country', 'area', 'description', 'aka']
['Abaia', 'FJ', ' ', 'Lake eel', ' ']
['Afanc', 'UK', 'CYM', 'Welsh lake monster', ' ']
['Agropeliter', 'US', 'ME', 'Forest twig flinger', ' ']
['Akkorokamui', 'JP', ' ', 'Giant Ainu octopus', ' ']
['Albatwitch', 'US', 'PA', 'Apple stealing mini Bigfoot', ' ']
```

¹ Если есть деревья, похожие на энтов из книг Толкина, не хотелось бы, чтобы они ночью подошли к дверям нашего дома для небольшой беседы.

Модуль python-tabulate

Опробуем еще один инструмент с открытым исходным кодом, python-tabulate (<https://oreil.ly/L0f6k>). Он специально разработан для табличного вывода. Сначала потребуется запустить команду `pip install tabulate`. В примере 17.3 показан код.

Пример 17.3. Считывание файла PSV с помощью python-tabulate (load_tabulate.py)

```
from tabulate import tabulate
import sys

def read_csv(fname: str) -> list[tuple]:
    with open(fname) as file:
        data = [row for row in csv.reader(file, delimiter="|")]
    return data

if __name__ == "__main__":
    data = read_csv(sys.argv[1])
    print(tabulate(data[0:5]))
```

Выполните пример 17.3 в примере 17.4.

Пример 17.4. Запуск скрипта загрузки tabulate

```
$ python load_tabulate.py cryptid.psv
```

Name	Country	Area	Description	AKA
Abaia	FJ		Lake eel	
Afanc	UK	CYM	Welsh lake monster	
Agropelter	US	ME	Forest twig flinger	
Akkorokamui	JP		Giant Ainu octopus	

Модуль pandas

Два предыдущих примера представляли собой в основном форматоры вывода. Библиотека pandas (<https://pandas.pydata.org>) — это отличный инструмент для нарезки данных. Он выходит за рамки стандартных структур данных Python, используя такие продвинутые конструкции, как DataFrame (<https://oreil.ly/j-8eh>) — комбинацию таблицы, словаря и серии. Он может читать .csv и другие файлы с разделителями в виде символов. Пример 17.5 похож на предыдущие примеры, но вместо списка кортежей pandas возвращает DataFrame.

Пример 17.5. Считывание файла PSV с помощью pandas (load_pandas.py)

```
import pandas
import sys

def read_pandas(fname: str) -> pandas.DataFrame:
    data = pandas.read_csv(fname, sep="|")
    return data

if __name__ == "__main__":
    data = read_pandas(sys.argv[1])
    print(data.head(5))
```

Выполните пример 17.5 в примере 17.6.

Пример 17.6. Запуск скрипта загрузки pandas

```
$ python load_pandas.py cryptid.psv
```

	name	country	area		description aka
0	Abaia	FJ			Lake eel
1	Afanc	UK	CYM		Welsh lake monster
2	Agropelter	US	ME		Forest twig flinger
3	Akkorokamui	JP			Giant Ainu octopus
4	Albatwitch	US	PA	Apple stealing mini	Bigfoot

В библиотеке pandas есть множество интересных функций, так что стоит изучить ее более внимательно.

Источник данных SQLite и веб-вывод

В остальных примерах этой главы вы будете считывать данные о существах из базы данных SQLite, используя определенные фрагменты кода веб-сайта из предыдущих глав. Затем нарежете данные на кусочки, кубики и замаринуете их по разным рецептам. Вместо простого вывода текста вы будете устанавливать каждый пример на наш постоянно растущий сайт о криптоидах. Вам понадобится несколько дополнений к существующим сервисному, веб-уровню и уровню данных.

Во-первых, потребуется функция веб-уровня и соответствующий HTTP GET-маршрут, чтобы вернуть все данные о существе. И у вас уже есть такой! Сделаем веб-вызов, чтобы получить все данные, но снова покажем лишь первые несколько строк (деревья, знаете ли) в примере 17.7.

Пример 17.7. Запустите тест загрузки существ (в усеченном виде — деревья наблюдают)

```
$ http -b localhost:8000/creature
```

```
[
  {
    "aka": "AKA",
    "area": "Area",
    "country": "Country",
    "description": "Description",
    "name": "Name"
  },
  {
    "aka": " ",
    "area": " ",
    "country": "FJ",
    "description": "Lake eel",
    "name": "Abaia"
  },
  ...
]
```

Пакеты диаграмм и графиков

Теперь мы можем перейти от текста к графическим интерфейсам (Graphical User Interface, GUI). Среди наиболее полезных и популярных пакетов Python для графического отображения данных можно назвать следующие:

- *Matplotlib* (<https://matplotlib.org>) — обширный, но требует некоторого вмешательства для получения красивых результатов;
- *Plotly* (<https://plotly.com/python>) — аналогичен Matplotlib и Seaborn, но с акцентом на интерактивных графиках;
- *Dash* (<https://dash.plotly.com>) — построен на основе пакета Plotly как своего рода информационная панель;
- *Seaborn* (<https://seaborn.pydata.org>) — построен на основе пакета Matplotlib и предлагает интерфейс более высокого уровня, но с меньшим количеством типов графов;
- *Bokeh* (<http://bokeh.org>) — интегрируется с JavaScript для создания информационных панелей для просмотра очень больших наборов данных.

Как же сделать правильный выбор? Стоит рассмотреть следующие критерии:

- типы графиков (например, диаграмма рассеяния, столбчатая диаграмма, линейный график);
- стилизация;
- простота использования;
- производительность;
- ограничения в данных.

Такие сравнительные исследования, как *Top 6 Python Libraries for Visualization: Which One to Use?* (<https://oreil.ly/10Nsw>) пользователя khuyentran1476, могут помочь вам с выбором. В конце концов выбор часто сводится к тому варианту, о котором вы узнаете больше всего. Для этой главы я выбрал пакет Plotly, позволяющий создавать привлекательные графики без написания лишнего кода.

Пример диаграммы 1. Тестирование

Plotly — это библиотека Python с открытым исходным кодом (бесплатная) и несколькими уровнями контроля и детализации:

- *Plotly Express* (<https://plotly.com/python/plotly-express>) — минимальная библиотека Plotly;
- *Plotly* (<https://plotly.com/python>) — основная библиотека;
- *Dash* (<https://dash.plotly.com>) — инструменты для работы с данными.

Существует также платформа Dash Enterprise (<https://dash.plotly.com/dash-enterprise>). Она, как и почти все, что имеет в названии слово enterprise — «корпоративный» (включая модели космических кораблей), стоит денег, обычно больших.

Что мы можем показать на основе данных о существах? У диаграмм и графиков есть несколько общих форм:

- столбцовая;
- рассеяния;

- линейная;
- коробчатая (статистическая);
- гистограмма.

Все наши поля данных — строки намеренно минимального размера, чтобы примеры не перегружали логику и этапы интеграции. Для каждого примера будем считывать все данные о существах из базы данных SQLite, используя код из предыдущих глав, а также добавлять функции веб- и сервисного уровня для выбора определенных данных для передачи в функции библиотеки графиков. Сначала установите пакет Plotly и библиотеку, необходимую ему для экспорта изображений:

- `pip install plotly;`
- `pip install kaleido.`

Затем (пример 17.8) добавьте тестовую функцию в файл `web/creature.py`, чтобы проверить, есть ли у нас нужные фрагменты кода в нужных местах.

Пример 17.8. Добавление тестовой конечной точки
(редактирование файла `web/creature.py`)

(добавьте эти строки в файл `web/creature.py`)

```
from fastapi import Response
import plotly.express as px

@router.get("/test")
def test():
    df = px.data.iris()
    fig = px.scatter(df, x="sepal_width", y="sepal_length", color="species")
    fig_bytes = fig.to_image(format="png")
    return Response(content=fig_bytes, media_type="image/png")
```

В документации обычно рекомендуется вызывать функцию `fig.show()`, чтобы показать только что созданное изображение, но мы пытаемся соответствовать тому, как это делают FastAPI и Starlette.

Итак, сначала вы получаете `fig_bytes` (актуальное содержимое `bytes` изображения). Затем возвращаете пользовательский объект `Response`.

После того как вы добавили конечную точку в файл `web/creature.py` и перезапустили веб-сервер (автоматически, если запустили Uvicorn с аргументом

--reload), попробуйте получить доступ к новой конечной точке, набрав текст **localhost:8000/creature/test** в адресной строке браузера. На экране должно появиться изображение, приведенное на рис. 17.1.

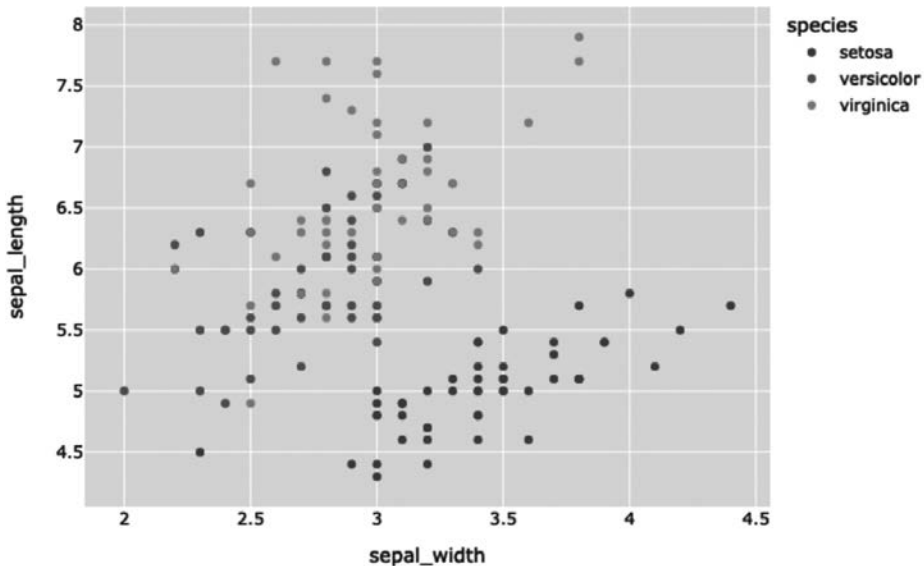


Рис. 17.1. Тестовое изображение Plotly

Если вы получили от Uvicorn странную ошибку, например `ValueError: 'not' is not a valid parameter name`, обновите Pydantic, чтобы исправить это: `pip install -U pydantic`.

Пример диаграммы 2. Гистограмма

Если все в порядке, начнем работать с данными о существах. Добавим функцию `plot()` в файл `web/creature.py`. Мы получим все данные о существах из базы данных с помощью функции `get_all()` в файлах `service/creature.py` и `data/creature.py`. Затем извлечем то, что нам нужно, и с помощью возможностей Plotly выведем различные изображения результатов.

Для первого приема (пример 17.9) просто используем поле `name` и построим гистограмму, показывающую количество имен существ, начинающихся на каждую букву.

Пример 17.9. Столбчатая диаграмма инициалов имен существ

(добавьте эти строки в файл `web/creature.py`)

```
from collections import Counter
from fastapi import Response
import plotly.express as px
from service.creature import get_all

@router.get("/plot")
def plot():
    creatures = get_all()
    letters = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
    counts = Counter(creature.name[0] for creature in creatures)
    y = { letter: counts.get(letter, 0) for letter in letters }
    fig = px.histogram(x=list(letters), y=y, title="Creature Names",
                      labels={"x": "Initial", "y": "Initial"})
    fig_bytes = fig.to_image(format="png")
    return Response(content=fig_bytes, media_type="image/png")
```

Введите `localhost:8000/creature/plot` в адресную строку своего браузера. Вы должны увидеть изображение, приведенное на рис. 17.2.

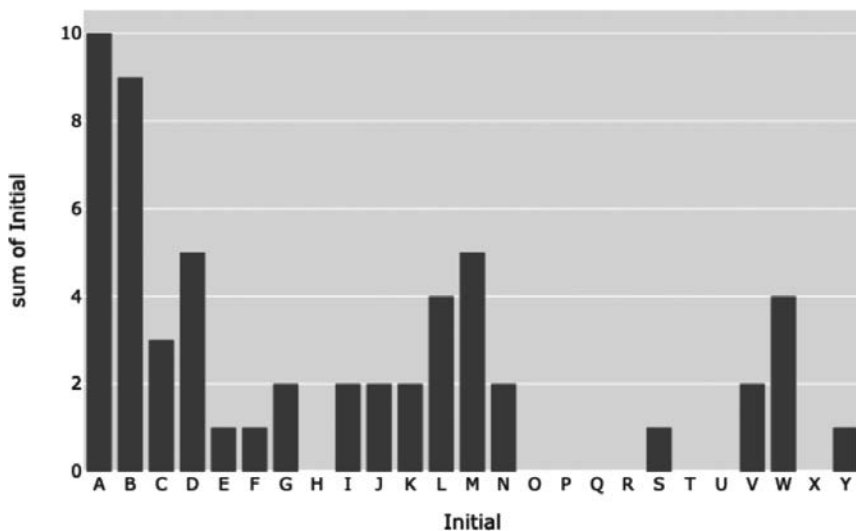
Creature Names

Рис. 17.2. Гистограмма инициалов имен существ

Пакеты для работы с картами

Если вы введете в поисковую строку Google слова *Python* и *maps*, то получите множество ссылок о словарях Python, которые являются встроенным в язык *типом отображения*¹, но это не то, о чем сейчас пойдет речь. Поэтому вам, возможно, придется попробовать такие синонимы, как *GIS*, *geo*, *cartography*, *spatial* и т. д. Некоторые популярные пакеты, приведенные далее, созданы на основе других пакетов из списка:

- *PyGIS* (<https://oreil.ly/3QvCz>) — ссылки по обработке пространственных данных в Python;
- *PySAL* (<https://pysal.org>) — библиотека пространственного анализа Python;
- *Cartopy* (<https://oreil.ly/YnUow>) — анализирует и наносит на карту геопространственные данные;
- *Folium* (<https://oreil.ly/72luj>) — интегрирован с JavaScript;
- *Python Client for Google Maps Services* (<https://oreil.ly/LWfS5>) — API-доступ к Google Maps;
- *Geemap* (<https://geemap.org>) — с поддержкой Google Earth;
- *Geoplot* (<https://oreil.ly/Slfvc>) — расширяет пакеты Cartopy и Matplotlib;
- *GeoPandas* (<https://geopandas.org>) — расширение для нашей любимой библиотеки pandas;
- *ArcGIS and ArcPy* (<https://oreil.ly/l7M5C>) — интерфейс Esri с открытым исходным кодом.

Как и в случае с пакетами диаграмм/графиков, выбор может зависеть от таких факторов, как:

- типы карт (например, фоновая, векторная, растровая);
- стилизация;
- простота использования;
- производительность;
- ограничения в данных.

¹ В английском языке слово *map* означает и «карта», и «сопоставлять». Перевод различается только по контексту. — *Примеч. пер.*

Как и диаграммы и графики, карты бывают разных типов и могут использоваться для различных целей.

Пример карты

Для примеров из области картографии я снова использую пакет Plotly — он не слишком прост и не слишком сложен и помогает показать, как интегрировать небольшую веб-карту с FastAPI.

Пример 17.10 иллюстрирует получение двухбуквенных кодов стран ISO для наших существ. Но оказалось, что функция, создающая карты Plotly (фоновая или *хороплет*, что само по себе звучит как криптид, меняющий форму), хочет использовать вместо этого другой, трехбуквенный стандарт кодов стран ISO. Бр-р-р. Мы могли бы переделать все коды в базе данных и PSV-файле, но проще выполнить команду `pip install country_converter` и сопоставить один набор кодов стран с другим.

Пример 17.10. Карта стран с криптидами (редактирование файла `web/creature.py`)

(добавьте эти строки в файл `web/creature.py`)

```
import plotly.express as px
import country_converter as coco

@router.get("/map")
def map():
    creatures = service.get_all()
    iso2_codes = set(creature.country for creature in creatures)
    iso3_codes = coco.convert(names=iso2_codes, to="ISO3")
    fig = px.choropleth(
        locationmode="ISO-3",
        locations=iso3_codes)
    fig_bytes = fig.to_image(format="png")
    return Response(content=fig_bytes, media_type="image/png")
```

Введите запрос браузеру на получение ответа по адресу `localhost:8000/creature/map`, и, если повезет, вы увидите карту, на которой выделяются страны с криптидами (рис. 17.3).

Можете увеличить масштаб этой карты, чтобы сосредоточиться на США, используя поле `area`, представляющее собой двухсимвольный код государства, где `country` — это US. Задействуйте выражение `locationmode="USA-states"` и присвойте значение `area` параметру `locations` функции `px.choropleth()`.

ГЛАВА 18

Игры

Обзор

Игры бывают очень разными, от простых текстовых до многопользовательских 3D-феерий. В этой главе я продемонстрирую простую игру и то, как конечная точка веб-приложения может взаимодействовать с пользователем на нескольких этапах. Этот процесс отличается от привычных вам по этой книге одноразовых запросов-ответов конечных точек веб-приложения.

Игровые пакеты в Python

Если вы действительно хотите освоить Python для игр, вот несколько полезных инструментов:

- текст — Adventurelib (<https://adventurelib.readthedocs.io>);
- графика:
 - PyGame (<https://www.pygame.org>), primer (<https://realpython.com/pygame-a-primer>);
 - pygamelet (<https://pygamelet.org>);
 - Python Arcade (<https://api.arcade.academy>);
 - HARFANG (<https://www.harfang3d.com>);
 - Panda3D (<https://docs.panda3d.org>).

Но в этой главе я не буду использовать ни один из них. Код примеров может стать настолько большим и сложным, что отодвинет на второй план цель этой книги — максимально упрощенное создание сайтов (API и традиционного контента) с помощью FastAPI.

Разделение игровой логики

Существует так много способов написать игру. Но как определить, кто что делает и кто где хранит данные? У веб-приложения нет статистики, и каждый раз, когда клиент обращается к серверу, того поражает полная амнезия и он клянется, что никогда не видел этого клиента раньше. Поэтому нам нужно где-то хранить *состояние* — данные, сохраняемые на всех этапах игры, чтобы связать их воедино.

Мы могли бы написать игру полностью на JavaScript на стороне клиента и хранить все состояния там. Если вы хорошо знаете JavaScript, это хорошее решение, но если не знаете (что вполне возможно, ведь вы читаете книгу по языку Python), дадим Python тоже кое-что сделать.

В то же время мы могли бы написать слишком тяжелое для сервера приложение. Допустим, генерировать некий отдельный идентификатор для конкретной игры при первом веб-вызове, передавать этот идентификатор вместе с другими данными серверу на последующих этапах игры и хранить все изменяющиеся состояния в каком-нибудь хранилище данных на стороне сервера, например в базе данных.

Наконец, мы могли бы структурировать игру как последовательность вызовов клиент-серверной конечной точки веб-приложения в так называемом одностраничном приложении (Single-Page Application, SPA). При написании SPA обычно JavaScript выполняет Ajax-вызовы на сервер и нацеливает веб-ответы на обновление отдельных частей страницы, а не всего экрана. Клиентские JavaScript и HTML выполняют часть работы, а сервер обрабатывает часть логики и данных.

Гейм-дизайн

Во-первых, что это за игра? Мы создадим простую игру, похожую на Wordle (<https://oreil.ly/PuD-Y>), но в ней будут использоваться только названия существ из базы данных `cryptid.db`. Это намного проще, чем Wordle, особенно если

схитрить и заглянуть в приложение Б. Применим окончательный сбалансированный дизайнерский подход, описанный ранее.

1. Задействуем ванильный JavaScript в клиенте вместо известных библиотек JavaScript, таких как React, Angular или даже jQuery.
2. Новая конечная точка FastAPI, `GET /game`, инициализирует игру. Она получает имя случайного существа из нашей базы криптоидов и возвращает его, встроенное в качестве скрытого значения в файл шаблона Jinja, состоящий из HTML, CSS и JavaScript.
3. На стороне клиента вновь созданные HTML и JavaScript отображают интерфейс типа Wordle. Появится последовательность окошек, по одному на каждую букву в названии скрытого существа.
4. Игрок вводит букву в каждое поле, а затем отправляет свою догадку и скрытое истинное имя на сервер. Это происходит в Ajax-вызове с помощью функции JavaScript `fetch()`.
5. Вторая новая конечная точка FastAPI, `POST /game`, принимает эту догадку и реальное секретное имя и в сравнении с ним оценивает догадку. Она возвращает клиенту угаданное значение и результат.
6. Клиентская часть отображает отгадку и результат соответствующими цветами CSS во вновь созданной строке таблицы: зеленый — буква в правильном месте, желтый — буква в имени, но в другой позиции и серый — буква, не встречающаяся в скрытом имени. Результат — строка одиночных символов, используемых как имена классов CSS для отображения правильных цветов букв угадайки.
7. Если все буквы зеленые, можно праздновать. В противном случае клиент отображает новую последовательность полей ввода текста для следующей отгадки и повторяет шаг 4 и последующие до тех пор, пока имя не будет угадано или вы не сдадитесь. Большинство названий криптоидов — это не общеупотребительные слова, поэтому при необходимости сверьтесь с приложением Б.

Эти правила немного отличаются от официального варианта игры Wordle, где разрешены только словарные слова из пяти букв и есть ограничение в шесть ходов.

Не надейтесь на это. Как и в большинстве примеров в книге, логика и дизайн игры минимальны — их достаточно, чтобы части работали совместно. Вы можете придать игре гораздо больше стиля и изыска, если у вас есть рабочая база.

Первая веб-часть — инициализация игры

Нам нужны две новые конечные точки веб-приложения. Мы используем имена существ, поэтому можно назвать конечные точки следующим образом: `GET /creature/game` и `POST /creature/game`. Но это не сработает, потому что у нас уже есть похожие конечные точки — `GET /creature/{name}` и `POST /creature/{name}` и FastAPI выберет их в качестве совпадения в первую очередь. Поэтому давайте создадим новое пространство имен маршрутизации верхнего уровня `/game` и поместим в него обе новые конечные точки.

Первая конечная точка в примере 18.1 выполняет инициализацию игры. Она должна получить случайное имя существа из базы данных и вернуть его вместе со всем клиентским кодом для реализации многоходовой игровой логики. Для этого мы используем шаблон Jinja (он приведен в главе 16), содержащий HTML, CSS и JavaScript.

Пример 18.1. Инициализация веб-игры (web/game.py)

```
from pathlib import Path

from fastapi import APIRouter, Body, Request
from fastapi.templating import Jinja2Templates

from service import game as service

router = APIRouter(prefix = "/game")

# Первоначальный запрос на игру
@router.get("")
def game_start(request: Request):
    name = service.get_word()
    top = Path(__file__).resolve().parents[1] # прародитель
    templates = Jinja2Templates(directory=f"{top}/template")
    return templates.TemplateResponse("game.html",
        {"request": request, "word": name})

# Последующие игровые запросы
@router.post("")
async def game_step(word: str = Body(), guess: str = Body()):
    score = service.get_score(word, guess)
    return score
```

FastAPI требуется функция пути `game_start()`, чтобы получить параметр `request` и передать его в шаблон в качестве аргумента.

Далее, в примере 18.2, подключите субмаршрут `/game` к основному модулю, контролирующему маршруты `/explorer` и `/creature`.

Пример 18.2. Добавление субмаршрута `/game` (`web/main.py`)

```
from fastapi import FastAPI
from web import creature, explorer, game

app = FastAPI()

app.include_router(explorer.router)
app.include_router(creature.router)
app.include_router(game.router)

if __name__ == "__main__":
    import uvicorn
    uvicorn.run("main:app",
                host="localhost", port=8000, reload=True)
```

Вторая веб-часть — этапы игры

Самый крупный компонент шаблона на стороне клиента (HTML, CSS и JavaScript) показан в примере 18.3.

Пример 18.3. Рабочий файл шаблона Jinja (`template/game.html`)

```
<head>
<style>
html * {
    font-size: 20pt;
    font-family: Courier, sans-serif;
}
body {
    margin: 0 auto;
    max-width: 700px;
}
input[type=text] {
    width: 30px;
    margin: 1px;
    padding: 0px;
    border: 1px solid black;
}
td, th {
    cell-spacing: 4pt;
    cell-padding: 4pt;
    border: 1px solid black;
}
```



```

.H { background-color: #00EE00; } /* hit (green) */
.C { background-color: #EEEE00; } /* close (yellow) */
.M { background-color: #EEEEEE; } /* miss (gray) */
</style>
</head>
<body>
<script>
function show_score(guess, score){
    var table = document.getElementById("guesses");
    var row = table.insertRow(row);
    for (var i = 0; i < guess.length; i++) {
        var cell = row.insertCell(i);
        cell.innerHTML = guess[i];
        cell.classList.add(score[i]);
    }
    var word = document.getElementById("word").value;
    if (guess.toLowerCase() == word.toLowerCase()) {
        document.getElementById("status").innerHTML = "&#x1F600";
    }
}

async function post_guess() {
    var word = document.getElementById("word").value;
    var vals = document.getElementsByName("guess");
    var guess = "";
    for (var i = 0; i < vals.length; i++) {
        guess += vals[i].value;
    }
    var req = new Request("http://localhost:8000/game", {
        method: "POST",
        headers: {"Content-Type": "application/json"},
        body: JSON.stringify({"guess": guess, "word": word})
    })
    fetch(req)
        .then((resp) => resp.json())
        .then((score) => {
            show_score(guess, score);
            for (var i = 0; i < vals.length; i++) {
                vals[i].value = "";
            }
        });
}
</script>
<h2>Cryptonomicon</h2>

<table id="guesses">
</table>
<span id="status"></span>
<hr>

```

```
<div>
{% for letter in word %}<input type=text name="guess">{% endfor %}
<input type=hidden id="word" value="{{word}}">
<br><br>
<input type=submit onclick="post_guess()">
</div>

</body>
```

Первая сервисная часть — инициализация

В примере 18.4 показан сервисный код для связи функции запуска игры на веб-уровне с функцией предоставления случайного имени существа на уровне данных.

Пример 18.4. Расчет результата (service/game.py)

```
import data.game as data

def get_word() -> str:
    return data.get_word()
```

Вторая сервисная часть — определение результатов

Добавьте код из примера 18.5 к коду из примера 18.4. Результат представляет собой строку одиночных символов, указывающих, совпала ли введенная буква с правильной позицией, с другой позицией или указана неверно. Отгадываемые буквы и слово преобразуются в нижний регистр, чтобы сопоставление не зависело от регистра. Если длина отгадки не совпадает с длиной скрытого слова, возвращается пустая строка.

Пример 18.5. Расчет результата (service/game.py)

```
from collections import Counter, defaultdict

HIT = "H"
MISS = "M"
CLOSE = "C" # (буква находится в слове, но в другой позиции)

def get_score(actual: str, guess: str) -> str:
    length: int = len(actual)
    if len(guess) != length:
        return ERROR
```

```

actual_counter = Counter(actual) # {буква: подсчет, ...}
guess_counter = defaultdict(int)
result = [MISS] * длина
for pos, letter in enumerate(guess):
    if letter == actual[pos]:
        result[pos] = HIT
        guess_counter[letter] += 1
for pos, letter in enumerate(guess):
    if result[pos] == HIT:
        continue
    guess_counter[letter] += 1
    if (letter in actual and
        guess_counter[letter] <= actual_counter[letter]):
        result[pos] = CLOSE
result = ''.join(result)
return result

```

Тестируем!

Пример 18.6 содержит несколько упражнений pytest для расчета оценки сервиса. В коде используем функциональную возможность pytest под названием `parametrize` для передачи последовательности тестов, вместо того чтобы писать цикл внутри самой тестовой функции. Помните из примера 18.5, что `H` — точное попадание, `C` — близко (неверная позиция) и `M` означает, что игрок вообще не угадал.

Пример 18.6. Тестирование расчета результата (`test/unit/service/test_game.py`)

```

import pytest
from service import game

word = "bigfoot"
guesses = [
    ("bigfoot", "ННННННН"),
    ("abcdefg", "МСМММСС"),
    ("toofgib", "СССНССС"),
    ("wronglength", ""),
    ("", ""),
]

@pytest.mark.parametrize("guess,score", guesses)
def test_match(guess, score):
    assert game.get_score(word, guess) == score

```

Запускаем:

```

$ pytest -q test_game.py
.....
5 passed in 0.05s

```

[100%]

Данные — инициализация

В новом модуле `data/game.py` потребуется только одна функция, показанная в примере 18.7.

Пример 18.7. Получение случайного имени существа (`data/game.py`)

```
from .init import curs
```

```
def get_word() -> str:
    qry = "select name from creature order by random() limit 1"
    curs.execute(qry)
    row = curs.fetchone()
    if row:
        name = row[0]
    else:
        name = "bigfoot"
    return name
```

Давайте поиграем в «Криптономикон»

(Кто-нибудь, пожалуйста, придумайте название получше.)

В браузере перейдите по адресу `http://localhost:8000/game`. На экране должно появиться следующее изображение.

Cryptonomicon

Submit

Введем несколько букв и отправим их в качестве отгадки, чтобы посмотреть, что получится.

Cryptonomicon

Submit

Буквы *b*, *f* и *g* выделены желтым (если вы не видите это в цвете, вам придется поверить мне на слово!). Это говорит о том, что они есть в скрытом имени, но стоят не на своих местах.

Cryptonomicon

a	b	c	d	e	f	g
---	---	---	---	---	---	---

--	--	--	--	--	--	--

Попробуем придумать название, но изменим последнюю букву. Во второй строке мы видим много зеленого цвета. Ого, так близко!

Cryptonomicon

a	b	c	d	e	f	g
b	i	g	f	o	o	d

--	--	--	--	--	--	--

Исправим последнюю букву и ради интереса сделаем некоторые буквы прописными, чтобы убедиться, что сопоставление работает без учета регистра. Подтверждаем отправку, ну и дела!

Cryptonomicon

a	b	c	d	e	f	g
b	i	g	f	o	o	d
B	i	g	f	O	O	T



--	--	--	--	--	--	--

Заключение

Мы использовали HTML, JavaScript, CSS и FastAPI, чтобы создать простую (очень!) интерактивную игру в стиле Wordle. В этом разделе было показано, как управлять многопоточным взаимодействием между веб-клиентом и сервером с помощью JSON и Ajax.

Дополнительная литература

Если вы захотите узнать больше и заполнить знаниями те области, которые я осветил недостаточно глубоко или вообще не затронул, то сможете найти множество замечательных ресурсов. В этом приложении перечислены ресурсы для изучения Python, FastAPI, Starlette и Pydantic.

Python

Вот некоторые известные сайты, посвященные Python:

- *Python Software Foundation* (<https://www.python.org>) — базовый материал;
- *Real Python Tutorials* (<https://realpython.com>) — подробные учебные пособия по Python;
- *Reddit* (<https://www.reddit.com/r/Python>) — раздел основного форума на сайте reddit.com о Python;
- *Stack Overflow* (<https://stackoverflow.com/questions/tagged/python>) — вопросы с тегом Python;
- *PyCoder's Weekly* (<https://pycoders.com>) — еженедельная рассылка по электронной почте;
- *Anaconda* (<https://www.anaconda.com>) — распространение научной информации.

А вот лишь некоторые из книг по Python, показавшиеся мне полезными во время написания этой книги:

- *Любанович Б.* Простой Python. Изд. 2-е. — СПб.: Питер, 2021;
- *Бизли Д.* Python. Исчерпывающее руководство. — СПб.: Питер, 2023;

- *Рамальо Л.* Python. К вершинам мастерства. — 2016;
- *Виафоре П.* Надежный Python. — 2023;
- *Персиваль Г., Грегори Б.* Паттерны разработки на Python. — СПб.: Питер, 2022.

FastAPI

Далее перечислены некоторые веб-сайты FastAPI:

- *Home* (<https://fastapi.tiangolo.com>) — официальный сайт и лучшая техническая документация, которую я видел;
- *External links and articles* (<https://fastapi.tiangolo.com/external-links>) — с официального сайта;
- *FastAPI GitHub* (<https://github.com/tyangolo/fastapi>) — репозиторий кода FastAPI;
- *Awesome FastAPI* (<https://github.com/mjhea0/awesome-fastapi>) — список ресурсов;
- *The Ultimate FastAPI Tutorial* (<https://oreil.ly/vfvS3>) — подробное описание из множества частей;
- *The Blue Book: FastAPI* (<https://lyz-code.github.io/blue-book/fastapi>) — подробный обзор FastAPI;
- *Medium* (<https://medium.com/tag/fastapi>) — статьи с тегом FastAPI;
- *Using FastAPI to Build Python Web APIs* (<https://realpython.com/fastapi-python-webapis>) — сокращенная документация по FastAPI;
- *Twitter* (https://oreil.ly/kHJm_) — твиты с отметкой @FastAPI или #FastAPI;
- *Gitter* (<https://oreil.ly/-56rC>) — просьбы о помощи и ответы на них;
- *GitHub* (<https://oreil.ly/NXTU1>) — репозитории со словом FastAPI в названиях.

Несмотря на то что FastAPI появился в конце 2018 года, книг по нему пока не так уж много. Я смог извлечь полезные уроки из прочтения следующих книг:

- *Building Data Science Applications with FastAPI*, автор Франсуа Ворон (Packt);
- *Building Python Microservices with FastAPI*, автор Шервин Джон Трагура (Packt);
- *Microservice APIs*, автор Хосе Аро Перальта (Manning).

Starlette

Основные ссылки для Starlette:

- Home (<https://www.starlette.io>);
- GitHub (<https://github.com/encode/starlette>).

Pydantic

Основные ссылки для Pydantic:

- Home (<https://pydantic.dev>);
- Docs (<https://docs.pydantic.dev>);
- GitHub (<https://github.com/pydantic/pydantic>).

ПРИЛОЖЕНИЕ Б

Существа и люди

От упырей, от призраков,
От тварей долголапых
И от существ, рыщущих в ночи,
Избави нас, Боже!

Строфа из Корнуольской литании

Сообщения о воображаемых существах, или *криптидах*, поступают отовсюду. Некоторые животные, когда-то считавшиеся воображаемыми, — панда, утконос и черный лебедь — оказались реальными. Поэтому мы не будем строить догадки. Отважные исследователи ищут их. Все вместе они обеспечивают данные для примеров, приводимых в этой книге.



Существа

В табл. Б.1 перечислены существа, которых мы будем исследовать.

Таблица Б.1. Мини-бестиарий

Название	Страна	Область	Описание	Ака
Abaia (абайя)	FJ		Из озера Иил	
Afanc (аванк)	UK	CYM	Валлийский озерный монстр	
Agropelter (агропельтер)	US	ME	Бросается ветками с деревьев в лесу	
Akkorokamui (аккорокамуи)	JP		Гигантский осьминог из фольклора айну	
Albatwitch (альбатвич)	US	PA	Миниатюрный йети, ворующий яблоки	
Alicanto (аликанто)	CL		Птица, питающаяся золотом	
Altamata-ha (альтамата-ха)	US	GA	Болотное существо	Алти
Amarok (амарок)	CA		Дух волка инуитов	
Auli (аули)	CY		Морское чудовище Айя-Напы	Дружелюбный монстр
Azeban (азебан)	CA		Дух шутника	Енот
Batsquatch (бэтсквотч)	US	WA	Летающий снежный человек	
Beast of Bladenboro (зверь из Бладенборо)	US	NC	Собака-кровопийца	
Beast of Bray Road (зверь из Брей-Роуд)	US	WI	Оборотень из Висконсина	
Beast of Busco (зверь из Буско)	US	IN	Гигантская черепаха	
Beast of Gevaudan (жеводанский зверь)	FR		Французский оборотень	

Продолжение ➞

Таблица Б.1 (продолжение)

Название	Страна	Область	Описание	Ака
Beaver Eater (пожиратель бобров)	CA		Переворачивает жилища	Сайтоэчин
Bigfoot (бигфут)	US		Кузен йети — Эдди	Сасквоч
Bukavac (букавац)	HR		Озерный душитель	
Bunyip (баньип)	AU		Водный австралийский монстр	
Cadborosaurus (кадборозавр)	CA	BC	Морской змей	Кадди
Champ (шамп)	US	VT	Затаившийся на озере Шамплейн	Шампи
Chupacabra (чупакабра)	MX		Убийца коз	
Dahu (даху)	FR		Французский кузен вам-пахуфуса	
Doyarchu (довар-ху)	IE		Водяная собака	Ирландский крокодил
Dragon (дракон)	*		Крылья! Огонь!	
Drop Bear (падающий медведь)	AU		Плотоядный коала	
Dungavenhooter (дунгавенхутер)	US		Измельчает добычу до состояния пыли, а затем вдыхает	
Encantado (энкантадо)	BR		Резвый речной дельфин	
Fouke Monster (монстр Фуке)	US	AR	Вонючий бигфут	Монстр болотного ручья
Glocester Ghoul (упырь из Глостера)	US	RI	Дракон Род-Айленда	
Gloucester Sea Serpent (морской змей из Глостера)	US	MA	Американское чудовище Несси	
Igorogo (игопого)	CA	ON	Канадское чудовище Несси	
Isshii (исси)	JP			Исси
Jackalope (джекалоп)	US		Рогатый заяц	
Jersey Devil (дьявол из Джерси)	US	NJ	Прыгун по снежным крышам	

Название	Страна	Область	Описание	Ака
Kodiak Dinosaur (динозавр из Кадьяка)	US	AK	Гигантский океанский ящер	
Kraken (кракен)	*		Мегакальмар	
Lizard Man (человек-ящерица)	US	SC	Болотное существо	
LLaammaa (ламия)	CL		Голова ламы, тело ламы. Но это не та лама	
Loch Ness Monster ¹ (лохнесское чудовище)	UK	SC	Знаменитый лохнесский зверь	Несси
Luska (луска)	BS		Гигантский осьминог	
Maero (маэро)	NZ		Гиганты	
Menehune (менехуне)	US	HI	Гавайские эльфы	
Mokele-mbembe (моке-мбембе)	CG		Болотный монстр	
Mongolian Death Worm (олгой-хорхой, или монголь- ский смертоносный червь)	MN		Пришелец из Арракиса	
Mothman (человек-мотылек)	US	WV	Единственный криптид в фильме Ричарда Гира	
Snarly Yow (ворчливый яу)	US	MD	Адская гончая	
Vampire (вампир)	*		Кровопийца	
Vlad the Impala (Влад бессмертный)	KE		Вампир из Саванны	
Wendigo (вендиги)	CA		Бигфут-каннибал	
Werewolf (оборотень)	*		Перевертыш	Лугару, ругару ²
Wyvern (виверна)	UK		Дракон с одной парой лап	
Wampahaofus (вампахуфус)	US	VT	Асимметричный обитатель гор	Сайдхил гоуджер
Yeti (йети)	CN		Косматый гималайский монстр	Отвратительный снежный человек

¹ Однажды я встретил Питера Макнаба, который сделал одну из предполагаемых фотографий Несси.

² От французского слова. Или от реплики Скуби-Ду: «Ruh-roh! Rougarou!»

Исследователи

Наша команда исследователей, собравшаяся из разных уголков мира, представлена в табл. Б.2.



Таблица Б.2. Люди

Имя	Страна	Описание
Claude Hande (Клод Ханде)	UK	Его не хватает в полнолуние
Helena Hande-Basquette (Хелена Ханде-Баскетт)	UK	Дама ¹ с претензией на славу
Beau Buffette (Бо Баффет)	US	Никогда не снимает шлем
O. B. Juan Cannoli (О. Б. Хуан Канноли)	MX	Мудрый в делах лесных
Simon N. Glorfindel (Саймон Глорфиндейл)	FR	Курчавый остроухий дровосек
«Па» Tuohy («Па» Туохи)	IE	Исследователь
Radha Tuohy (Радха Туохи)	IN	Мистическая мать-Земля
Noah Weiser (Ноа Вайзер)	DE	Близорукий человек с мачете

¹ В смысле благородства, а не знатности.

Публикации исследователей

Вот воображаемые публикации наших воображаемых исследователей:

- *The Secret of Rat Island*, В. Buffette (*Баффет В.* Тайна Крысиного острова);
- *What Was I Thinking?*, О. В. J. Cannoli (*Канноли О. В. Х.* О чем я думал?);
- *Spiders Never Sleep, Journal of Disturbing Results*, N. Weiser (*Байзер Н.* «Пауки никогда не спят», Журнал тревожных результатов);
- «*Sehr Böse Spinnen*», *Zeitschrift für Vergleichende Kryptozoologie*, N. Weiser (*Байзер Н.* «Очень плохие пауки», временной сценарий для сравнительной криптидной зоологии).

Другие источники

У преданий о криптидах много источников. Некоторых криптидов можно отнести к воображаемым существам, а некоторых можно увидеть на нечетких фотографиях, сделанных на большом расстоянии. Среди моих источников были следующие:

- страница «Википедии» *List of Cryptids* (<https://oreil.ly/7e1ED>);
- страница «Википедии» *List of Legendary Creatures by Type* (<https://oreil.ly/1AVfx>);
- веб-сайт *The Cryptid Zoo: A Menagerie of Cryptozoology* (<http://www.newanimal.org>);
- книга *The United States of Cryptids*, автор Дж. У. Окер (Quirk Books);
- книга *In the Wake of the Sea-Serpents*, автор Бернар Эйвельманс (Hill & Wang);
- книга *Abominable Snowmen: Legend Come to Life*, Иван Сандерсон (Chilton);
- видео *Every Country Has a Monster* (<https://oreil.ly/yQP7Q>), Mystery Science Theater;
- ресурсы о наблюдениях за бигфутом:
 - *Data on Bigfoot sightings by Tim Renner* (<https://oreil.ly/1wMDb>);
 - *Bigfoot Sightings Dash App* (<https://oreil.ly/b5IKt>);
 - *Finding Bigfoot with Dash Part 1* (<https://oreil.ly/0gjCT>), *Part 2* (<https://oreil.ly/Lespw>), *Part 3* (<https://oreil.ly/aDV8K>);
 - *If It's There, Could It Be a Bear?* (<https://oreil.ly/ТИYn7>), автор Флоу Фоксон.

Об авторе

Билл Любанович занимается разработкой программного обеспечения уже более 40 лет, специализируясь на Linux, веб-технологиях и Python. Выступил соавтором книги издательства O'Reilly «Системное администрирование в Linux» и написал оба издания книги «Простой Python». Несколько лет назад он открыл для себя FastAPI и вместе со своей командой использовал его для переписывания большого API для биомедицинских исследований. Опыт оказался настолько положительным, что они решили применять FastAPI для всех новых проектов. Билл живет со своей семьей и кошками в горах Сангре-де-Саскуатч в штате Миннесота.

Иллюстрация на обложке

Животное на обложке — шипохвостая игуана (род *Ctenosaura*). Название *Ctenosaura* происходит от двух греческих слов: *ctenos*, что означает «гребень» (за гребнеподобные шипы на спине и хвосте), и *saura* — «ящерица». Существует 15 признанных видов шипохвостых игуан, включая пятиклювую, черногрудую, мотагуа, оахаку, роатану и утилу.

Длина шипохвостых игуан может варьироваться от 12 до 100 см. Каждый вид окрашен в свой цвет, который может меняться в зависимости от температуры тела, настроения, состояния здоровья животного и температуры среды обитания. Шипохвостые игуаны всеядны и питаются разнообразными фруктами, цветами, листвой и мелкими животными.

Игуан можно встретить в самых разных местах обитания. Родиной шипохвостых игуан являются Мексика и Центральная Америка. Их можно встретить в тропических и субтропических сухих лесах, зарослях кустарников, а иногда и в измененных человеком местах обитания и городских районах. Некоторые виды, такие как роатан (встречается только на острове Роатан в Гондурасском заливе), утила (встречается только на Утиле — архипелаге в заливе у карибского побережья Гондураса, в болотах и мангровых экосистемах) и мотагуа (встречается только в Гватемале), являются эндемиками очень специфических территорий.

Несколько видов шипохвостых игуан внесены в список исчезающих или находящихся под критической угрозой исчезновения. Некоторые из них (западная и черная шипохвостая игуаны) известны как инвазивные виды в США. Они сталкиваются с рядом угроз, включая утрату среды обитания из-за сельского хозяйства и выпаса скота, незаконную торговлю домашними животными и браконьерство, фрагментацию среды обитания, интродуцированных хищников и убийство от страха.

Многие животные, изображаемые на обложках книг издательства O'Reilly, находятся под угрозой исчезновения, хотя все они важны для мира. Иллюстрация на обложке выполнена Карен Монтгомери на основе старинной гравюры из Музея живой природы.

Алфавитный указатель

A

Asynchronous Server Gateway Interface, ASGI 41
Authorization Code Flow 173

C

Command-Line Interface, CLI 29
Cross-Origin Resource Sharing, CORS 186
CRUD 24

D

Domain-Driven Design, DDD 112

E

Extract, Transform, Load, ETL 32

F

FastAPI 38

G

Graph Query Language (GraphQL) 26

H

HTML 23
HTTP 23
HTTPIe 38
HTTPX 38
HTTP-глагол 24
HTTP-заголовок 55

J

JavaScript Object Notation, JSON 26
JWT 183

M

Machine Learning, ML 32
MIME-типы 59

N

NoSQL 226

O

OAuth2 173

R

Remote procedure call, RPC 23
Representational State Transfer, REST 24
Requests 38
RESTful 24
Role-Based Access Control, RBAC 185

T

Type hints 40

U

URL 23
Uvicorn 38

W

Web Server Gateway Interface, WSGI 41

А

Авторизация 165
Асинхронный режим 27
Аутентификация 165

Б

База данных 28
Блочная модель 29
Бэкенд 22

В

Веб 19
Веб-клиент 28
Веб-сервис 22
Веб-уровень 28
Веб-шаблон 244
Ведение журналов 217
Виртуальные среды 36
Внедрение зависимостей 96
Время ожидания 27
Всемирная паутина 19
Вытесняющее планирование 68

Г

Галлюцинации 232
Группы данных 82

Д

Декоратор пути 46, 99
Дублированные данные 156

З

Зависимости данных 95
Заглушки 123
Запрос — ответ 23

И

Идемпотентность 53
Издатель-подписчик 23
Изменяемый объект 39
Именованный кортеж 84
Интеграционное тестирование 190
Интерфейс командной строки 29
Искусственный интеллект, ИИ 32

К

Код-дублер 196
Код состояния 25
Конечная точка 24
Конкурентность 27
Конкурентные вычисления 67
Конструкция `async/await` 71
Контрактные тесты 138
Кооперативные потоки 69
Кортеж 83
Криптиды 82

М

Макет 194
Макетирование 194
Макеты данных 123
Маршрут 24, 101
Машинное обучение 32
Менеджер контекста 195
Метрики 141
Множество 83
Модель данных 28
Модель ресурса 88
Модульное тестирование 190
Мониторинг 141

Н

Наблюдение 141
Нагрузочное тестирование 208
Неизменяемый объект 39
Нормальные формы 226

О

Область видимости переменной 40
Обозначение объектов JavaScript 26
Образ Docker 212
Обратные вызовы 70
Объект 39
Объект-генератор 71
Объект-курсор 146
Ограниченная строка 93
Отсутствующие данные 156
Очереди сообщений 23
Очередь заданий 215

П

Пагинация результатов 133
Пакет Poetry 37
Параллельные вычисления 67
Параметризация 192
Параметры запроса 50, 52, 114
Параметры пути 114
Паттерн «Репозиторий» 205
Передача репрезентативного состояния 24
Переменная 39
Переменная окружения 116
Пирамида тестов 190
Подсказки типа 80

Подсказки типов 40
Подход web-first 113
Полное тестирование 190
Потоки управления 68
Предметная область 122
Предметно-ориентированное проектирование 112
Промежуточное ПО 186
Пропускная способность 27
Протокол API 22
Процессы ОС 68

Р

Ресурс 24
Роли 185

С

Секрет 167
Секретный ключ 169
Сервисный уровень 28
Синглтон 148
Синхронизированный режим 27
Система передачи сообщений 23
Система pip 35
Сквозные тесты 138
Словарь 83
События 23
Совместное использование ресурсов разными источниками 186
Сортировка результатов 133
Список 83
Строгая типизация 39
Схемы БД 226

Т

Тело запроса 53
Тело сообщения 25
Типы ответов 58
Трассировка 141

У

Удаленные вызовы процедур 23
Уровень данных 28

Ф

Файл конфигурации 37
Фикстуры 192
Фиктивные данные 123

Формат API 22

Фронтенд 22

Функция-генератор 70

Функция пути 46, 74

Х

Хеширование 172

Ц

Цикл событий 27

Я

Язык выражений
SQLAlchemy 223

Билл Любанович
FastAPI: веб-разработка на Python

Перевел с английского Я. Голуб

Руководитель дивизиона	<i>Ю. Сергиенко</i>
Руководитель проекта	<i>А. Питиримов</i>
Ведущий редактор	<i>Н. Гринчик</i>
Литературный редактор	<i>Н. Рощина</i>
Художественный редактор	<i>В. Мостипан</i>
Корректоры	<i>Е. Павлович, Н. Терех</i>
Верстка	<i>Г. Блинов</i>

Изготовлено в Казахстане. Изготовитель: ТОО «Спринт Бук».

Место нахождения и фактический адрес:

010000, Казахстан, город Астана, район Алматы,
Проспект Рахымжан Кошкарбаев, д. 10/1, н. п. 18.

Дата изготовления: 06.2024.

Наименование: книжная продукция.

Срок годности: не ограничен.

Подписано в печать 23.04.24. Формат 70×100/16. Бумага офсетная.

Усл. п. л. 23,220. Тираж 700. Заказ 0000.

Отпечатано в ТОО «ФАРОС Графикс».
100004, РК, г. Караганда, ул. Молокова, 106/2.



Джейк Вандер Плас

PYTHON ДЛЯ СЛОЖНЫХ ЗАДАЧ: НАУКА О ДАННЫХ. 2-е междунар. изд.

Python — первоклассный инструмент, и в первую очередь благодаря наличию множества библиотек для хранения, анализа и обработки данных. Отдельные части стека Python описываются во многих источниках, но только в новом издании «Python для сложных задач» вы найдете подробное описание: IPython, NumPy, pandas, Matplotlib, Scikit-Learn и др.

Специалисты по обработке данных, знакомые с языком Python, найдут во втором издании решения таких повседневных задач, как обработка, преобразование и подготовка данных, визуализация различных типов данных, использование данных для построения статистических моделей и моделей машинного обучения. Проще говоря, эта книга является идеальным справочником по научным вычислениям в Python.

КУПИТЬ



Оливье Келен,
Мари-Алис Блете

РАЗРАБОТКА ПРИЛОЖЕНИЙ НА БАЗЕ GPT-4 И CHATGPT

Эта небольшая книга представляет собой подробное руководство для разработчиков на Python, желающих научиться создавать приложения с использованием больших языковых моделей. Авторы расскажут об основных возможностях и преимуществах GPT-4 и ChatGPT, а также о принципах их работы. Здесь же вы найдете пошаговые инструкции по разработке приложений с использованием библиотеки поддержки GPT-4 и ChatGPT для Python, в том числе инструментов для генерирования текста, отправки вопросов и получения ответов и обобщения контента.

«Разработка приложений на базе GPT-4 и ChatGPT» содержит множество легковоспроизводимых примеров, которые помогут освоить особенности применения моделей в своих проектах. Все примеры кода на Python доступны в репозитории GitHub. Решили использовать возможности LLM в своих приложениях? Тогда вы выбрали правильную книгу.

КУПИТЬ