

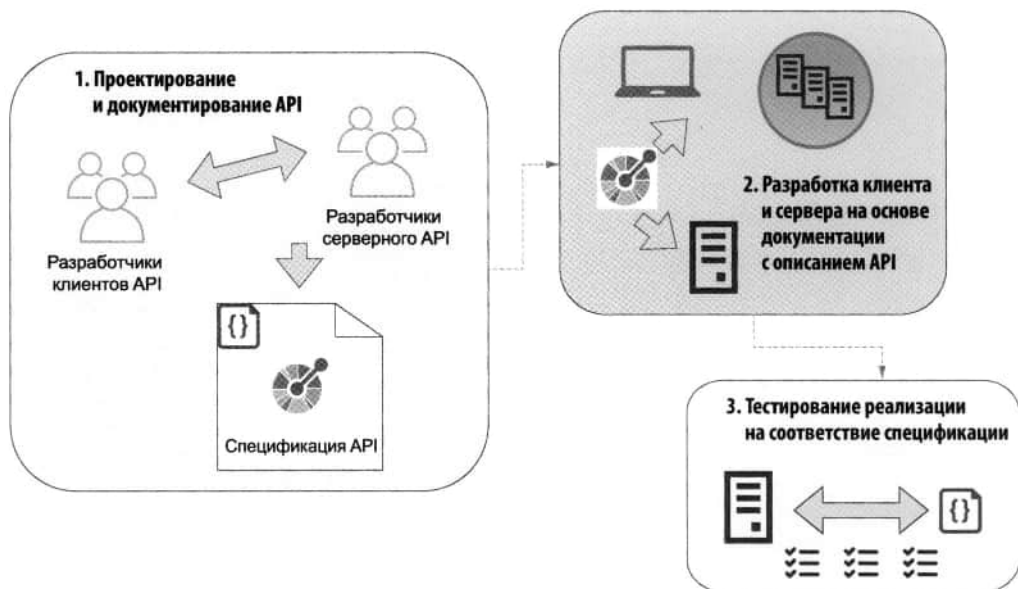
# Микросервисы и API

Хосе Аро Перальта

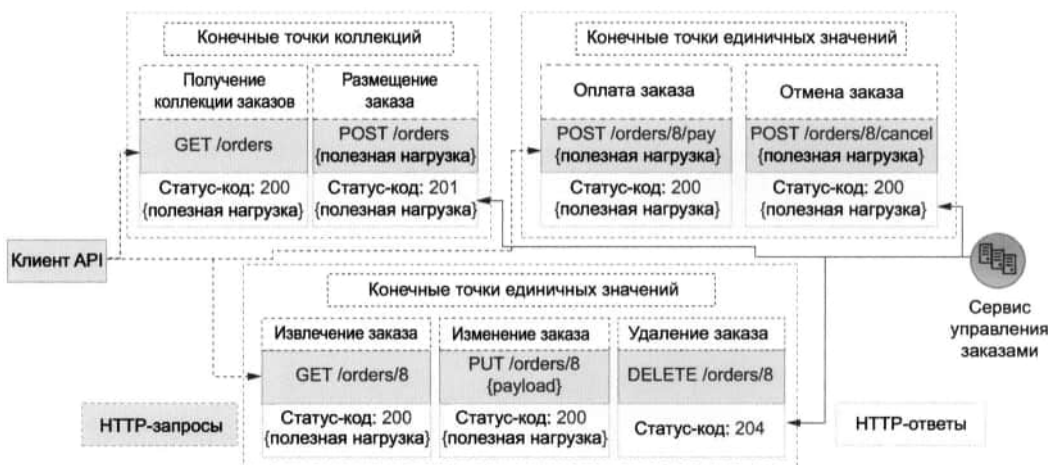


MANNING





Разработка через документирование (documentation-driven development) — это метод разработки, ориентированный на API, следуя которому API сначала проектируется и документируется, затем на основе документации создается серверный API и его клиент и в заключение документация используется для тестирования реализаций сервера и клиента. Разработка через документирование помогает уменьшить вероятность неудач при интеграции с API, а также обеспечивает больший контроль и лучшую выявляемость ошибок интеграции.



REST API структурированы вокруг конечных точек. Мы различаем конечные точки единичных значений, такие как GET /orders/8, и конечные точки коллекций, такие как GET /orders. REST API используют семантику методов HTTP для обозначения действий (например, POST для создания ресурсов) и статус-коды для сообщения результатов обработки запроса (например, 200 для сообщения об успешном выполнении операции).



# *Microservice APIs*

USING PYTHON, FLASK, FASTAPI,  
OPENAPI AND MORE

JOSÉ HARO PERALTA



MANNING

SHELTER ISLAND



# *Микросервисы и API*

ХОСЕ АРО ПЕРАЛЬТА

Выпущено  
при поддержке

**КРОК**

 **ПИТЕР®**

Санкт-Петербург • Москва • Минск

2024

# Хосе Аро Перальта

## Микросервисы и API

*Перевел с английского А. И. Ларин*

ББК 32.988.02-018

УДК 004.738.5

**Перальта Хосе Аро**

**П26** Микросервисы и API. — СПб.: Питер, 2024. — 464 с.: ил. — (Серия «Библиотека программиста»).

ISBN 978-5-4461-2094-9

Простые и понятные API — необходимое условие успеха микросервисных приложений. Хорошо продуманные API гарантируют надежную интеграцию сервисов и помогают упростить сопровождение, масштабирование и дальнейшее совершенствование. Познакомьтесь с паттернами, протоколами и стратегиями, которые помогут вам проектировать, реализовывать и развертывать эффективные микросервисы с REST и GraphQL API.

Книга наполнена проверенными советами и примерами кода на языке Python. Авторы фокусируются на реализации, а не на философии. Изучите проверенные методы проектирования простых в использовании API для микросервисных приложений. Создавайте надежные API микросервисов, тестируйте, защищайте и развертывайте их в облаке, следуя принципам и шаблонам, применимым в любом языке программирования.

**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 Manning Publications. 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. В книге возможны упоминания организаций, деятельность которых запрещена на территории Российской Федерации, таких как Meta Platforms Inc., Facebook, Instagram и др. 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 978-1617298417 англ.

Authorized translation of the English edition © 2023 Manning Publications.

This translation is published and sold by permission of Manning Publications, the owner of all rights to publish and sell the same.

ISBN 978-5-4461-2094-9

© Перевод на русский язык ООО «Прогресс книга», 2024

© Издание на русском языке, оформление ООО «Прогресс книга», 2024

© Серия «Библиотека программиста», 2024

Изготовлено в России. Изготовитель: ООО «Прогресс книга». Место нахождения и фактический адрес:

194044, Россия, г. Санкт-Петербург, Б. Сампсониевский пр., д. 29А, пом. 52. Тел.: +78127037373.

Дата изготовления: 04.2024. Наименование: книжная продукция. Срок годности: не ограничен.

Налоговая льгота — общероссийский классификатор продукции ОК 034-2014, 58.11.12 — Книги печатные профессиональные, технические и научные.

Импортер в Беларусь: ООО «ПИТЕР М», 220020, РБ, г. Минск, ул. Тимирязева, д. 121/3, к. 214, тел./факс: 208 80 01.

Подписано в печать 22.03.24. Формат 70×100/16. Бумага офсетная. Усл. п. л. 37,410. Тираж 1000. Заказ 1945.

Отпечатано в полном соответствии с качеством предоставленных материалов в ООО «Фотоэксперт»

109316, г. Москва, Волгоградский проспект, д. 42, корп. 5, эт. 1, пом. 1, ком. 6.3-23Н

# *Краткое содержание*

---

Предисловие . . . . .	16
Благодарности . . . . .	18
О книге. . . . .	20
Об авторе . . . . .	25
Иллюстрация на обложке . . . . .	26
О научных редакторах русскоязычного издания. . . . .	27
От издательства . . . . .	28

## **Часть I. Представляем API микросервисов**

Глава 1. Что такое API микросервисов. . . . .	31
Глава 2. Разработка простого API . . . . .	50
Глава 3. Проектирование микросервисов . . . . .	76

## **Часть II. Проектирование и разработка REST API**

Глава 4. Принципы проектирования REST API . . . . .	93
Глава 5. Документирование REST API с помощью OpenAPI. . . . .	125
Глава 6. Создание REST API на Python . . . . .	147
Глава 7. Паттерны реализации сервисов . . . . .	183

### **Часть III. Проектирование и реализация GraphQL API**

<b>Глава 8.</b> Проектирование GraphQL API . . . . .	225
<b>Глава 9.</b> Использование GraphQL API . . . . .	253
<b>Глава 10.</b> Создание GraphQL API с помощью Python . . . . .	277

### **Часть IV. Защита, тестирование и развертывание API микросервисов**

<b>Глава 11.</b> Авторизация и аутентификация API. . . . .	315
<b>Глава 12.</b> Тестирование и валидация API . . . . .	352
<b>Глава 13.</b> Контейнеризация API микросервисов. . . . .	384
<b>Глава 14.</b> Развертывание API микросервисов с помощью Kubernetes . . . . .	396

### **Приложения**

<b>Приложение А.</b> Типы веб-API и протоколов . . . . .	434
<b>Приложение Б.</b> Управление жизненным циклом API . . . . .	446
<b>Приложение В.</b> Авторизация API с использованием поставщика удостоверений . . . . .	451

# Оглавление

---

Предисловие . . . . .	16
Благодарности . . . . .	18
О книге. . . . .	20
Кому стоит прочитать эту книгу. . . . .	20
Структура издания . . . . .	20
О коде . . . . .	23
Другие онлайн-ресурсы . . . . .	24
Об авторе . . . . .	25
Иллюстрация на обложке . . . . .	26
О научных редакторах русскоязычного издания. . . . .	27
От издательства . . . . .	28

## **Часть I. Представляем API микросервисов**

<b>Глава 1. Что такое API микросервисов. . . . .</b>	<b>31</b>
1.1. Что такое микросервисы . . . . .	32
1.1.1. Определение микросервисов . . . . .	32
1.1.2. Микросервисы против монолитов . . . . .	33
1.1.3. История появления современных микросервисов . . . . .	36
1.2. Что такое веб-API . . . . .	37
1.2.1. API. . . . .	37
1.2.2. Веб-API. . . . .	38
1.2.3. Как API помогают нам управлять интеграцией микросервисов . . . . .	38

1.3. Проблемы микросервисной архитектуры . . . . .	40
1.3.1. Оптимальная декомпозиция сервиса . . . . .	41
1.3.2. Интеграционные тесты микросервисов. . . . .	41
1.3.3. Обработка недоступности сервиса . . . . .	42
1.3.4. Трассировка распределенных транзакций . . . . .	43
1.3.5. Повышение сложности эксплуатации и накладных расходов на инфраструктуру . . . . .	44
1.4. Введение в разработку через документирование (DDD) . . . . .	45
1.5. Знакомство с приложением CoffeeMesh . . . . .	47
1.6. Для кого эта книга и чему вы научитесь . . . . .	47
Резюме. . . . .	48
<b>Глава 2. Разработка простого API . . . . .</b>	<b>50</b>
2.1. Знакомимся со спецификацией API сервиса заказов . . . . .	51
2.2. Высокоуровневая архитектура приложения заказов . . . . .	52
2.3. Реализация эндпоинтов API. . . . .	53
2.4. Реализация моделей валидации данных с помощью pydantic . . . . .	60
2.5. Валидация полезной нагрузки запроса с помощью pydantic. . . . .	65
2.6. Маршалинг и валидация полезной нагрузки ответа с помощью pydantic. . . . .	69
2.7. Добавление в API сохраненного в памяти списка заказов . . . . .	73
Резюме. . . . .	75
<b>Глава 3. Проектирование микросервисов . . . . .</b>	<b>76</b>
3.1. Представляем CoffeeMesh . . . . .	77
3.2. Принципы проектирования микросервисов. . . . .	77
3.2.1. Принцип «База данных для каждого сервиса» . . . . .	77
3.2.2. Принцип слабой связанности. . . . .	79
3.2.3. Принцип единственной ответственности . . . . .	80
3.3. Декомпозиция сервисов по бизнес-функциям . . . . .	80
3.3.1. Анализ бизнес-структуры CoffeeMesh . . . . .	81
3.3.2. Декомпозиция микросервисов по функциям . . . . .	81
3.4. Декомпозиция сервисов по поддоменам . . . . .	84
3.4.1. Что такое предметно-ориентированное проектирование. . . . .	84
3.4.2. Применение стратегического анализа в CoffeeMesh . . . . .	85
3.5. Декомпозиция по бизнес-функциям в сравнении с декомпозицией по поддоменам . . . . .	89
Резюме. . . . .	90

## Часть II. Проектирование и разработка REST API

<b>Глава 4. Принципы проектирования REST API</b>	93
4.1. Что такое REST	94
4.2. Архитектурные ограничения приложений REST	95
4.2.1. Разделение задач: принцип клиент-серверной архитектуры	96
4.2.2. Добавляем масштабирование: принцип отсутствия записи состояния	97
4.2.3. Оптимизация производительности: принцип кэшируемости	97
4.2.4. Упрощение для клиента: принцип многоуровневой системы	98
4.2.5. Расширяемые интерфейсы: принцип «код по запросу»	99
4.2.6. Сохранение согласованности: принцип единства интерфейса	100
4.3. Гипермедиа как двигатель состояния приложений	100
4.4. Анализ зрелости API с помощью модели Ричардсона	103
4.4.1. Уровень 0. Веб-API в стиле RPC	104
4.4.2. Уровень 1. Введение понятия ресурса	105
4.4.3. Уровень 2. Использование методов HTTP и статус-кодов	106
4.4.4. Уровень 3. Возможность обнаружения API	106
4.5. Структурированные URL-адреса ресурсов с методами HTTP	107
4.6. Использование статус-кодов HTTP для создания информативных HTTP-ответов	111
4.6.1. Что такое статус-коды HTTP	111
4.6.2. Использование статус-кодов HTTP для сообщения об ошибках клиента в запросе	112
4.6.3. Использование статус-кодов HTTP для сообщения об ошибках на сервере	116
4.7. Проектирование полезной нагрузки API	117
4.7.1. Что такое полезная нагрузка HTTP и когда мы ее используем	118
4.7.2. Модели проектирования полезной нагрузки HTTP	119
4.8. Проектирование параметров запроса URL	123
Резюме	124
<b>Глава 5. Документирование REST API с помощью OpenAPI</b>	125
5.1. Использование JSON Schema для моделирования данных	126
5.2. Структура спецификации OpenAPI	130
5.3. Документирование эндпоинтов API	132
5.4. Документирование параметров запроса URL	133
5.5. Документирование полезной нагрузки запросов	134
5.6. Рефакторинг определений схем во избежание повторов	136

5.7. Документирование ответов API . . . . .	139
5.8. Создание общих ответов . . . . .	142
5.9. Определение схемы аутентификации API . . . . .	144
Резюме . . . . .	146
<b>Глава 6. Разработка REST API на Python . . . . .</b>	<b>147</b>
6.1. Обзор API сервиса заказов . . . . .	148
6.2. Параметры URL-запроса для API сервиса заказов . . . . .	149
6.3. Валидация полезной нагрузки с неизвестными полями . . . . .	152
6.4. Переопределение динамически генерируемой спецификации FastAPI . . . . .	156
6.5. Обзор API сервиса кухни . . . . .	157
6.6. Вводим в работу flask-smorest . . . . .	159
6.7. Инициализация веб-приложения для API . . . . .	161
6.8. Реализация эндпоинтов API . . . . .	163
6.9. Реализация моделей проверки полезной нагрузки с помощью marshmallow . . . . .	167
6.10. Проверка параметров запроса URL . . . . .	171
6.11. Проверка данных перед сериализацией ответа . . . . .	174
6.12. Реализация списка расписаний в памяти . . . . .	178
6.13. Переопределение динамически генерируемой спецификации API flask-smorest . . . . .	180
Резюме . . . . .	181
<b>Глава 7. Паттерны реализации сервисов . . . . .</b>	<b>183</b>
7.1. Гексагональные архитектуры для микросервисов . . . . .	184
7.2. Настройка среды и структуры проекта . . . . .	187
7.3. Реализация моделей баз данных . . . . .	189
7.4. Реализация паттерна Репозиторий для доступа к данным . . . . .	195
7.4.1. Паттерн Репозиторий: что это и чем он полезен . . . . .	195
7.4.2. Реализация паттерна Репозиторий . . . . .	197
7.5. Реализация уровня бизнес-логики . . . . .	202
7.6. Внедрение паттерна Единица работы . . . . .	212
7.7. Интеграция уровня API и уровня сервиса . . . . .	217
Резюме . . . . .	221
 <b>Часть III. Проектирование и реализация GraphQL API</b>	
<b>Глава 8. Проектирование GraphQL API . . . . .</b>	<b>225</b>
8.1. Знакомство с GraphQL . . . . .	226
8.2. Представляем API сервиса продукции . . . . .	229



8.3. Знакомство с системой типов GraphQL . . . . .	232
8.3.1. Создание определений свойств с помощью скалярных величин . . . . .	232
8.3.2. Моделирование ресурсов с использованием объектных типов . . . . .	233
8.3.3. Создание кастомных скалярных величин . . . . .	235
8.4. Представление коллекций элементов с помощью списков . . . . .	236
8.5. Мыслите графически: выстраивание значимых связей между объектными типами . . . . .	237
8.5.1. Соединение типов с помощью свойств-ребер . . . . .	237
8.5.2. Создание соединений со сквозными типами . . . . .	239
8.6. Сочетание различных типов с помощью объединений и интерфейсов . . . . .	241
8.7. Ограничение значений свойств с помощью перечислений . . . . .	244
8.8. Определение запросов для получения данных из API . . . . .	245
8.9. Изменение состояния сервера с помощью мутаций . . . . .	248
Резюме . . . . .	251
<b>Глава 9. Использование GraphQL API . . . . .</b>	<b>253</b>
9.1. Запуск mock-сервера GraphQL . . . . .	254
9.2. Запросы GraphQL . . . . .	256
9.2.1. Выполнение простых запросов . . . . .	256
9.2.2. Выполнение запросов с параметрами . . . . .	257
9.2.3. Ошибки запросов . . . . .	258
9.3. Использование фрагментов в запросах . . . . .	260
9.4. Выполнение запросов с входными параметрами . . . . .	262
9.5. Навигация по графу API . . . . .	262
9.6. Выполнение нескольких запросов и алиасинг . . . . .	264
9.6.1. Выполнение нескольких запросов за один раз . . . . .	264
9.6.2. Алиасинг запросов . . . . .	265
9.7. Выполнение мутаций GraphQL . . . . .	268
9.8. Выполнение параметризованных запросов и мутаций . . . . .	269
9.9. Раскрываем тайны запросов GraphQL . . . . .	273
9.10. Вызов GraphQL API с помощью кода на Python . . . . .	274
Резюме . . . . .	276
<b>Глава 10. Создание GraphQL API с помощью Python . . . . .</b>	<b>277</b>
10.1. Анализ требований к API . . . . .	278
10.2. Технологический стек . . . . .	278
10.3. Фреймворк Ariadne . . . . .	280
10.4. Реализация API сервиса продукции . . . . .	286
10.4.1. Разработка структуры проекта . . . . .	286

10.4.2. Создание точки входа для сервера GraphQL . . . . .	287
10.4.3. Реализация резольверов запросов . . . . .	288
10.4.4. Реализация резольверов типов . . . . .	291
10.4.5. Обработка параметров запроса . . . . .	298
10.4.6. Реализация резольверов мутаций . . . . .	301
10.4.7. Создание резольверов для кастомных скалярных типов . . . . .	304
10.4.8. Реализация резольверов полей . . . . .	308
Резюме . . . . .	311

## Часть IV. Защита, тестирование и развертывание API микросервисов

<b>Глава 11. Авторизация и аутентификация API . . . . .</b>	<b>315</b>
11.1. Настройка среды для примеров этой главы . . . . .	316
11.2. Протоколы аутентификации и авторизации . . . . .	317
11.2.1. Что такое Open Authorization . . . . .	317
11.2.2. Что такое OpenID Connect . . . . .	323
11.3. Работа с JSON Web Tokens . . . . .	325
11.3.1. Заголовок JWT . . . . .	327
11.3.2. Утверждения JWT . . . . .	328
11.3.3. Создание JWT . . . . .	330
11.3.4. Проверка JWT . . . . .	332
11.3.5. Валидация JWT . . . . .	334
11.4. Добавление авторизации на API сервера . . . . .	335
11.4.1. Создание модуля авторизации . . . . .	337
11.4.2. Создание промежуточного программного обеспечения для авторизации . . . . .	338
11.4.3. Добавление промежуточного программного обеспечения CORS . . . . .	341
11.5. Авторизация доступа к ресурсам . . . . .	343
11.5.1. Обновление базы данных для привязки пользователей и заказов . . . . .	343
11.5.2. Ограничение доступа пользователей к их собственным ресурсам . . . . .	346
Резюме . . . . .	350
<b>Глава 12. Тестирование и валидация API . . . . .</b>	<b>352</b>
12.1. Настройка среды для тестирования API . . . . .	353
12.2. Тестирование REST API с помощью Dredd . . . . .	354
12.2.1. Что такое Dredd . . . . .	355
12.2.2. Установка и запуск набора тестов Dredd по умолчанию . . . . .	356
12.2.3. Настройка набора тестов Dredd с помощью хуков . . . . .	358
12.2.4. Использование Dredd для тестирования API . . . . .	366

12.3. Введение в тестирование на основе свойств . . . . .	366
12.3.1. Что такое тестирование на основе свойств . . . . .	366
12.3.2. Традиционный подход к тестированию API . . . . .	367
12.3.3. Тестирование на основе свойств с использованием Hypothesis . . . . .	369
12.3.4. Использование Hypothesis для тестирования эндпоинта REST API . . . . .	371
12.4. Тестирование REST API с помощью Schemathesis . . . . .	374
12.4.1. Запуск набора тестов Schemathesis по умолчанию . . . . .	375
12.4.2. Использование ссылок для расширения набора тестов Schemathesis . . . . .	375
12.5. Тестирование GraphQL API . . . . .	379
12.5.1. Тестирование GraphQL API с помощью Schemathesis . . . . .	380
12.6. Разработка стратегии тестирования API . . . . .	382
Резюме . . . . .	382
<b>Глава 13. Контейнеризация API микросервисов . . . . .</b>	<b>384</b>
13.1. Настройка среды для этой главы . . . . .	385
13.2. Контейнеризация микросервиса . . . . .	386
13.3. Запуск приложений с помощью Docker Compose . . . . .	391
13.4. Публикация Docker-образов в реестре контейнеров . . . . .	393
Резюме . . . . .	395
<b>Глава 14. Развертывание API микросервисов с помощью Kubernetes . . . . .</b>	<b>396</b>
14.1. Настройка среды для этой главы . . . . .	398
14.2. Как работает Kubernetes: версия CliffsNotes . . . . .	399
14.3. Создание кластера Kubernetes с помощью EKS . . . . .	401
14.4. Использование ролей IAM для учетных записей сервисов Kubernetes . . . . .	405
14.5. Развертывание балансировщика нагрузки Kubernetes . . . . .	406
14.6. Развертывание микросервисов в кластере Kubernetes . . . . .	409
14.6.1. Создание объекта развертывания . . . . .	410
14.6.2. Создание объекта сервиса . . . . .	413
14.6.3. Предоставление доступа к сервисам с помощью ingress-объектов . . . . .	415
14.7. Настройка бессерверной базы данных с помощью AWS Aurora . . . . .	417
14.7.1. Создание бессерверной базы данных Aurora . . . . .	417
14.7.2. Управление секретами в Kubernetes . . . . .	420
14.7.3. Запуск миграции базы данных и подключение сервиса к базе данных . . . . .	424
14.8. Обновление спецификации OpenAPI с указанием имени хоста ALB . . . . .	427
14.9. Удаление кластера Kubernetes . . . . .	430
Резюме . . . . .	432

## Приложения

<b>Приложение А.</b> Типы веб-API и протоколов . . . . .	434
А.1. Рассвет API: RPC, XML-RPC и JSON-RPC. . . . .	435
А.2. SOAP и появление стандартов API . . . . .	437
А.3. RPC снова наносит удар: быстрый обмен данными через gRPC . . . . .	439
А.4. API HTTP и REST . . . . .	441
А.5. Детализированные запросы с использованием GraphQL . . . . .	443
<b>Приложение Б.</b> Управление жизненным циклом API . . . . .	446
Б.1. Стратегии управления версиями для развивающихся API . . . . .	447
Б.2. Вывод API из эксплуатации. . . . .	449
<b>Приложение В.</b> Авторизация API с использованием поставщика удостоверений . . . . .	451
В.1. Использование поставщика удостоверений как услуги . . . . .	452
В.2. Использование схемы авторизации PKCE . . . . .	458
В.3. Использование схемы учетных данных клиента . . . . .	459
В.4. Авторизация запросов в пользовательском интерфейсе Swagger . . . . .	461

*Дживон, без чьей поддержки и ободрения я бы  
не смог написать эту книгу, и Айви, которая  
делает меня счастливым каждый день.*

# Предисловие

---

API и микросервисы захватили индустрию программного обеспечения. Под давлением растущей сложности программного обеспечения и необходимости масштабирования все больше организаций переходят от монолитной архитектуры к микросервисной. Отчет O'Reilly «Внедрение микросервисов в 2020 году» показал, что 77 % респондентов внедрили микросервисы, и эта тенденция, как ожидается, и дальше будет расти.

При использовании микросервисов стоит задача интеграции сервисов через API. По данным Nordic APIs, 90 % разработчиков применяют API и тратят 30 % своего времени на их создание<sup>1</sup>. Внедрение API привело к изменению способа разработки приложений. Сегодня все чаще создаются продукты и сервисы, которые предоставляются исключительно через API, например Twilio и Stripe. Даже такие традиционные отрасли, как банковское дело и страхование, находят новые направления деятельности, открывая свои API и интегрируясь в экосистему Open Banking. Широкое распространение продуктов, направленных на предоставление API, означает, что при разработке собственных приложений мы можем сосредоточиться на основных бизнес-функциях, используя внешние API для решения таких типовых задач, как аутентификация пользователей и рассылка электронной почты.

Быть частью этой растущей экосистемы очень интересно. Однако прежде, чем мы перейдем к микросервисам и API, необходимо понять, как разрабатывать архитектуру микросервисов, проектировать API, как спланировать стратегию развития API, как убедиться в надежности предоставляемых интеграций, как выбрать модель развертывания и защитить наши системы. По моему опыту, большинство организаций сталкиваются с одним или несколькими из этих вопросов, а недавний отчет IBM показал, что 31 % организаций не внедрили микросервисы из-за отсутствия экспертизы внутри компании<sup>2</sup>. Кроме того, в отчете *Postman's 2022 State of the API*

---

<sup>1</sup> Simpson J. 20 Impressive API Economy Statistics («20 впечатляющих статистических данных по экономике за счет API»). <https://nordicapis.com/20-impressive-api-economy-statistics/>.

<sup>2</sup> Microservices in the enterprise, 2021: Real benefits, worth the challenges («Микросервисы на предприятии, 2021 год: реальные преимущества, возникающие проблемы»). <https://www.ibm.com/downloads/cas/OQG4AJAM>.

говорится, что 14 % респондентов сталкиваются с проблемами API-интеграции 11–25 % времени (<http://mng.bz/Xa9v>), а по данным Salt Security, 94 % организаций столкнулись с проблемами безопасности API в 2022 году<sup>1</sup>.

Во многих книгах рассматриваются проблемы, упомянутые в предыдущем абзаце, но обычно авторы делают это с узкоспециальной точки зрения: одни фокусируются на архитектуре, другие — на API, третьи — на безопасности. Мне показалось, что стоит написать книгу, которая объединяет все эти вопросы и рассматривает их с практической точки зрения: по сути, книгу, которая поможет обычному разработчику быстро освоить лучшие практики, принципы и паттерны для проектирования и разработки API микросервисов.

За последние годы я работал с различными клиентами, помогая им проектировать микросервисы и внедрять интеграции посредством API. Работа над этими проектами позволила мне получить представление об основных проблемах, с которыми сталкиваются команды разработчиков при работе с микросервисами и API. Как оказалось, обе технологии обманчиво просты. Хорошо спроектированный API прост в навигации и использовании, а хорошо спроектированные микросервисы повышают производительность разработчиков и легко масштабируются. В то же время плохо спроектированные API подвержены ошибкам и сложны в использовании, а плохо спроектированные микросервисы приводят к появлению так называемых распределенных монолитов.

Возникают очевидные вопросы: как спроектировать хорошие API? И как разработать слабосвязанные микросервисы? Книга поможет вам ответить на эти и другие вопросы. Кроме того, в процессе чтения вы сами будете разрабатывать API и сервисы и узнаете, как сделать их безопасными, как их тестировать и разворачивать. Методы, паттерны и принципы, которые я описываю в книге, стали результатом моего многолетнего опыта, и я очень рад поделиться ими. Надеюсь, эта книга будет для вас ценным источником информации на пути вашего развития как разработчика и архитектора программного обеспечения.

---

<sup>1</sup> Salt Security. State of API Security Q3 2022 («Состояние безопасности API в III квартале 2022 года»). — Р. 4. <https://content.salt.security/state-api-report.html>.

# Благодарности

---

Написание этой книги было одним из самых увлекательных событий в моей карьере, но я не решился бы на это без помощи и поддержки своей семьи и коллег. Книга посвящается моей замечательной жене Дживон, без постоянной поддержки и понимания которой я не смог бы завершить работу, и нашей дочери Айви, которая заботится о том, чтобы мне не пришлось заскучать.

Я благодарю людей, которые предложили идеи для этой книги, помогли мне лучше понять инструменты и протоколы, которые я использую в ней, и предоставили отзывы о черновиках. Особая благодарность Дмитрию Дыгалю, Келвину Миксу, Себастьяну Рамиресу Монтаньо, Крису Ричардсону, Джин Янг, Гаджендре Дешпанде, Оскару Исласу, Мехди Меджуи, Бену Хаттону, Андрею Барановскому, Алексу Мистридису, Роупу Хакулинену, Стиву Ардагу-Уолтеру, Катрин Бьеркелунд, Томасу Дину, Марко Антонио Сансу, Винсенту Ванденборну и удивительным разработчикам Ariadne от Mirumee.

С 2020 года я представлял черновики и идеи из книги на различных конференциях, в частности EuroPython, PyCon India, API World, API Specifications Conference, а также на различных подкастах и встречах. Хочу поблагодарить всех, кто посетил мои выступления и дал мне ценную обратную связь. Я также хочу поблагодарить участников моих семинаров на сайте [microapis.io](https://microapis.io) за вдумчивые комментарии к книге.

Спасибо моему редактору по закупкам Энди Уолдрону. Энди проделал блестящую работу, помогая мне превратить мое видение книги в реальную рукопись и сфокусировать внимание на актуальных темах. Он также неустанно поддерживал меня в продвижении издания и помог выйти на более широкую аудиторию.

Книга, которую вы держите в руках, легко читается благодаря неоценимой работе моего литературного редактора Марины Майклс. Она проделала огромную работу, помогая мне улучшить стиль письма и направляя меня в нужное русло.

Хочу поблагодарить моего технического редактора Ника Уоттса, который справедливо указывал на многие недостатки и заставлял меня приводить понятные иллюстрации, и моего технического корректора Эла Кринкера, который усердно проверял все листинги и код в репозитории GitHub для этой книги, убеждаясь, что он корректен и выполняется без ошибок.



Спасибо и остальным членам команды Manning, участвовавшим в создании этой книги, включая Кэндис Гиллхулли, Глорию Лукос, Степана Юрековича, Кристофера Кауфмана, Радмилу Эрцеговац, Михаэлу Батинич, Ану Ромак, Айру Дучич, Мелиссу Айс, Элеонору Гарднер, Брекина Эли, Пола Уэллса, Энди Маринковича, Кэти Теннант, Мишель Митчелл, Сэм Вуд, Пола Спратли, Ника Нейсона и Ребекку Райнхарт. Спасибо также Марьян Бейс за то, что поверила в меня и дала этой книге шанс.

Во время работы над книгой у меня была возможность получить подробные и интересные отзывы от самой замечательной группы рецензентов, включая Алена Ломпо, Бьорна Нойхауса, Брайана Миллера, Клиффорда Тербера, Дэвида Паккуда, Дебмалью Джаша, Гаурава Суда, Джорджа Хейнса, Гленна Лео Суонка, Хартмута Пальма, Икечукву Оконкво, Яна Питера Хервейера, Джоуи Смита, Хуана Хименеса, Джастина Баура, Кшиштофа Камычека, Маниша Джайна, Маркуса Янга, Матийса Аффуртита, Матье Эврена, Майкла Брайта, Михаила Рыбинцева, Михала Рутку, Мигеля Монтальво, Нинослава Черкеза, Пьера-Мишеля Анселя, Рафаэля Айкеля, Роберта Кулаговски, Родни Вайса, Самбасиву Андалури, Сатя Кумара Саху, Симеона Лейзерзона, Стивена К. Макунзву, Стюарта Вудворда, Стути Верму и Уильяма Джейми Сильву.

С тех пор как книга попала в MEAP<sup>1</sup>, я получил множество отзывов и слов поддержки от моих читателей. Мне также посчастливилось пообщаться с читателями, которые активно участвовали в форуме книги на платформе Manning's liveBook. Я сердечно благодарен всем вам.

Эта книга была бы невозможна без неустанной работы тысяч авторов открытого исходного кода, создавших и поддерживающих удивительные библиотеки, которые я использую в ней. Я очень благодарен всем вам и надеюсь, что моя книга поможет сделать вашу удивительную работу более заметной.

Наконец, благодарю вас, читатель, за то, что вы приобрели экземпляр моей книги. Я могу только надеяться, что вы найдете ее полезной и информативной и получите такое же удовольствие от ее чтения, как я — от ее написания. Я люблю слушать своих читателей и буду рад, если вы поделитесь со мной своими мыслями об издании.

<sup>1</sup> Manning Early Access Program — программа издательства Manning по раннему доступу к книгам.

Цель книги — научить вас разрабатывать микросервисы, обеспечивая их интеграцию посредством API. Вы научитесь проектировать микросервисную систему и разрабатывать REST API и GraphQL API для обеспечения взаимодействия между микросервисами. Вы узнаете, как тестировать и валидировать API микросервисов, обеспечивать их безопасность, а также развертывать и эксплуатировать их в облаке.

## КОМУ СТОИТ ПРОЧИТАТЬ ЭТУ КНИГУ

Книга будет полезна разработчикам программного обеспечения, которые работают с микросервисами и API. В ней используется практический подход, и весь материал иллюстрируется полными примерами кода. Поэтому разработчики, уже работающие с API микросервисов, по достоинству оценят оглавление книги.

Примеры кода приведены на языке Python, однако вам не обязательно его знать, чтобы выполнять их. Перед введением нового кода каждая концепция подробно объясняется.

В книге много внимания уделяется стратегиям проектирования, лучшим практикам и процессам разработки, поэтому она также будет полезна техническим директорам (СТО), архитекторам и вице-президентам по разработкам (VP of engineering), которым необходимо решить, стоит ли их компании внедрять микросервисную архитектуру, или тем, кому необходимо сделать выбор между различными стратегиями разработки API и заставить интеграции работать.

## СТРУКТУРА ИЗДАНИЯ

Книга состоит из четырех частей, разделенных на 14 глав.

Часть I знакомит с концепциями микросервисов и API. В ней объясняется, как разработать простой API и спроектировать микросервисную систему.

- В главе 1 вводятся основные понятия книги: микросервисы и API. Из нее вы узнаете, чем микросервисы отличаются от монолитной архитектуры и когда имеет смысл использовать монолит, а не микросервисы. Объясняется, что такое API и как они помогают осуществлять интеграцию микросервисов.
- Глава 2 предлагает пошаговое руководство по реализации простого API с помощью популярного в Python фреймворка FastAPI. Вы научитесь читать спецификацию API и понимать ее требования. Вы также узнаете, как поэтапно создавать API и как тестировать модели валидации данных.
- В главе 3 объясняется, как спроектировать микросервисную систему. В ней описаны три фундаментальных принципа проектирования микросервисов, а также объясняется, как разделить систему на микросервисы, используя декомпозицию по бизнес-функциям и декомпозицию по модулям (поддомам).

Часть II объясняет, как проектировать, документировать и разрабатывать REST API, а также как разработать микросервис.

- В главе 4 рассматриваются принципы проектирования REST API. В ней представлены шесть ограничений архитектуры REST и модель зрелости Ричардсона, а затем объясняется, как мы используем протокол HTTP для разработки хорошо структурированных и понятных REST API.
- В главе 5 рассказывается, как документировать REST API, используя стандарт спецификации OpenAPI. Вы освоите основы синтаксиса схемы JSON (JSON Schema), узнаете, как определять конечные точки, или эндпоинты (endpoints), моделировать данные и рефакторить документацию с помощью многократно используемых схем.
- В главе 6 объясняется, как создавать REST API с помощью двух популярных фреймворков Python: FastAPI и Flask. Вы прочтете о различиях между ними и узнаете, что принципы и модели построения API остаются неизменными и выходят за рамки деталей реализации любого технического стека.
- В главе 7 приводятся фундаментальные принципы и паттерны для построения микросервисов. В ней описана концепция гексагональной архитектуры и объясняется, как обеспечить слабую связанность между уровнями приложения. Вы также узнаете, как реализовать модели баз данных с помощью ORM SQLAlchemy и как управлять миграциями баз данных с помощью Alembic.

В части III объясняется, как проектировать, использовать и разрабатывать GraphQL API.

- В главе 8 рассказывается, как разрабатывать GraphQL API и как использовать Schema Definition Language (SDL). В ней описываются стандартные типы GraphQL, а также объясняется, как определить кастомные типы. Вы узнаете, как создавать связи между типами и писать запросы и мутации.

- В главе 9 мы поговорим о том, как использовать GraphQL API. Вы узнаете, как запустить mock-сервер и изучить документацию GraphQL с помощью GraphQL. Вы научитесь отправлять запросы и мутации на сервер GraphQL и при этом использовать параметры.
- В главе 10 рассказывается о том, как создавать GraphQL API с помощью Python фреймворка Ariadne. Вы научитесь использовать документацию API для автоматической загрузки моделей валидации данных, а также создавать резольверы для кастомных типов, запросов и мутаций.

В части IV объясняется, как тестировать, делать безопасным и разворачивать API микросервисов.

- В главе 11 вы узнаете, как добавить аутентификацию и авторизацию в ваши API с помощью стандартных протоколов, таких как OpenID Connect (OIDC) и Open Authorization (OAuth) 2.1. Мы обсудим, как создавать и валидировать JSON Web Tokens (JWT) и как разработать слой авторизации для ваших API.
- В главе 12 рассказывается о том, как тестировать и валидировать API. Вы узнаете, что такое тестирование на основе свойств (property-based testing, PBT) и как использовать его для проверки API, а также научитесь работать с фреймворками для автотестирования, в частности Dredd и schemathesis.
- В главе 13 объясняется, как упаковать микросервисы в Docker-контейнеры, запускать их локально с помощью Docker Compose и как публиковать Docker-образы в AWS Elastic Container Registry (ECR).
- В главе 14 мы обсудим разворачивание микросервисов в облаке AWS с помощью Kubernetes. Вы узнаете, как создать кластер Kubernetes и управлять им с помощью Elastic Kubernetes Service AWS, как запустить бессерверную базу данных Аулога в защищенной сети, как безопасно развернуть конфиг-файл приложения с помощью шифрования по схеме envelope encryption и как развернуть сервисы с возможностью последующего масштабирования.

На протяжении всех глав мы будем работать над созданием компонентов для вымышленной платформы доставки кофе, которая называется CoffeeMesh. В главе 1 приводится вводная информация о CoffeeMesh, а в главе 3 описывается, как разделить платформу на микросервисы. Поэтому я рекомендую прочитать главы 1 и 3, чтобы лучше понять примеры, приведенные в последующих главах. В остальном все части независимы и все главы самодостаточны. Например, если вы хотите узнать, как проектировать и разрабатывать REST API, можете сразу перейти к части II, а если вас интересует GraphQL API, можете сосредоточиться на части III. Точно так же, если вы хотите научиться добавлять аутентификацию и авторизацию в ваши API, переходите к главе 11 или, если хотите научиться тестировать API, вам в помощь глава 12.

## О КОДЕ

В книге приводится множество примеров исходного кода как в нумерованных листингах, так и отдельными строками. В обоих случаях исходный код оформляется шрифтом фиксированной ширины, как здесь, чтобы отделить его от обычного текста. Иногда код также выделяется **жирным шрифтом** — так оформлен код, который изменился по сравнению с тем, что приводился ранее в главе.

Во многих случаях исходный код был переформатирован; мы добавили переносы строк и изменили отступы, чтобы уместить его на странице. В некоторых случаях и этого было недостаточно, и в листинги добавлялись знаки переноса строк (↵). Кроме того, комментарии в исходном коде удалены из листингов, если код описывается в тексте. Многие листинги сопровождаются выносками — в них подчеркиваются важные моменты.

Почти все главы (за исключением 1, 3 и 4-й) наполнены примерами кода, которые иллюстрируют каждую новую описанную концепцию или паттерн. Большинство примеров написаны на Python, за исключением кода для глав 5, 8 и 9, посвященных проектированию API, — они содержат примеры в формате схем OpenAPI/JSON Schema (см. главу 5) и Schema Definition Language (см. главы 8 и 9). Весь код подробно объясняется, поэтому он должен быть понятен всем читателям, в том числе тем, кто не знает Python.

Полный код примеров, приведенных в книге, можно загрузить из репозитория GitHub, посвященного этой книге, по адресу <https://github.com/abunuwas/microservice-apis>. Для каждой главы в репозитории GitHub создана папка, например `ch02` для главы 2. Если не указано иное, все ссылки на файлы в главе относятся к соответствующей папке на GitHub. Например, в главе 2 файл `orders/app.py` соответствует файлу `ch02/orders/app.py` на GitHub.

В репозитории GitHub приводится финальный код для каждой главы. В некоторых главах постепенно показывается, как создавать функции. В этих случаях код, который представлен на GitHub, соответствует окончательной версии кода для главы.

Примеры кода на Python, приведенные в книге, были протестированы на Python 3.10, хотя любая версия Python, начиная с 3.7, должна работать точно так же. Код и команды, которые я использую в книге, были протестированы на машине Mac, но они должны без проблем работать и в Windows и Linux. Если вы работаете в Windows, рекомендую использовать POSIX-совместимый терминал, например Cygwin.

На протяжении работы над главами для управления зависимостями я задействовал `Pipenv`. В папке каждой главы вы найдете файлы `Pipfile` и `Pipfile.lock`, где

описываются точные версии зависимостей, которые я использовал для выполнения примеров кода. Чтобы избежать проблем при запуске кода, в начале изучения каждой главы вам стоит загрузить эти файлы и установить зависимости из них.

## **ДРУГИЕ ОНЛАЙН-РЕСУРСЫ**

Если вы хотите узнать больше об API микросервисов, загляните в мой блог <https://microapis.io/blog>, где собраны ресурсы, дополняющие уроки этой книги. Там же я поддерживаю актуальный список своих практикумов и семинаров — они также дополняют эту книгу.

## Об авторе

---



**Хосе Антонио Аро Перальта** — консультант по программному обеспечению и архитектуре. Имея более чем десятилетний опыт работы, Хосе помогал крупным и небольшим организациям строить сложные системы, создавать микросервисные системы и обеспечивать интеграцию с помощью API. Он также является основателем компании `microapis.io`, предоставляющей услуги по консультированию и обучению в области программного обеспечения. Будучи авторитетным экспертом в сфере облачных вычислений, DevOps и автоматизации программного обеспечения, Хосе регулярно выступает на международных конференциях и часто организует открытые мастер-классы.

# Иллюстрация на обложке

---

Рисунок на обложке озаглавлен *L'invalidé*, или «Иinvalid», и изображает раненого французского солдата, который проживал в Hôtel national des Invalides – Национальном доме инвалидов. Это изображение взято из коллекции Жака Грассе де Сен-Совера, опубликованной в 1797 году. Каждая иллюстрация в коллекции тщательно прорисована и раскрашена вручную.

В те времена по одежде человека можно было легко определить, где он живет, чем занимается и какое положение в обществе имеет.

Сегодня, когда сложно отличить одну компьютерную книгу от другой, издательство Manning приветствует изобретательность и новаторство компьютерного бизнеса с помощью обложек, отражающих богатое разнообразие региональной жизни многовековой давности, о котором нам напоминают картины Грассе де Сен-Совера.



# *О научных редакторах русскоязычного издания*

---

Александр Петраки — старший инженер-разработчик КРОК, занимается проектированием архитектуры высоконагруженных приложений и выполняет реализацию backend-части на Java, Spring с применением СУБД Mysql, PostgreSQL, Oracle, JanusGraph.

Анатолий Смирнов — технический менеджер. Организует разработку новых систем и модулей к ним, а также руководит проектами по технической поддержке ранее разработанных информационных систем.

## *От издательства*

---

Ваши замечания, предложения, вопросы отправляйте по адресу [comp@piter.com](mailto:comp@piter.com) (издательство «Питер», компьютерная редакция).

Мы будем рады узнать ваше мнение!

На веб-сайте издательства [www.piter.com](http://www.piter.com) вы найдете подробную информацию о наших книгах.

Мы выражаем огромную благодарность компании «КРОК» за помощь в работе над русскоязычным изданием книги и за их вклад в повышение качества переводной литературы.

*Часть I*

*Представляем API  
микросервисов*

Микросервисы — это архитектурный стиль программного обеспечения, в котором компоненты системы проектируются как отдельные и независимо развертываемые приложения. Концепция микросервисов существует с начала 2000-х годов, а с 2010-х активно набирает популярность. Сегодня микросервисы все чаще выбирают для создания современных веб-приложений. Как вы узнаете из главы 1, микросервисы позволяют использовать возможности распределенных приложений, легче масштабировать компоненты и быстрее выпускать релизы приложения.

Однако при всех своих преимуществах микросервисы имеют и недостатки. Они повышают издержки на инфраструктуру, их сложнее мониторить, ими сложнее управлять и сложнее выполнять их отладку. При работе с микросервисами основной задачей становится их правильное проектирование, и в главе 3 вы узнаете несколько принципов и стратегий, которые помогут вам создавать надежные микросервисы.

Микросервисы взаимодействуют через API, и в этой книге вы научитесь проектировать и разрабатывать REST и GraphQL API для своих микросервисов. Глава 2 даст вам представление о создании REST API, а в части II вы узнаете о различных паттернах и принципах создания надежных REST API. Сложнее всего при работе с API обеспечить, чтобы и клиент, и сервер следовали одной спецификации API. В главе 1 вы узнаете о разработке через документирование (*documentation-driven development*, DDD) и о том, как важно начинать путь к API с хорошо документированного проекта.

В части I вы прочтете об основополагающих паттернах и принципах создания микросервисов и их интеграции через API. В остальных частях мы будем опираться на концепции, описанные здесь, и вы узнаете, как создавать надежные API, как их тестировать, защищать и как развертывать ваши микросервисы в облаке. Наше захватывающее путешествие вот-вот начнется!

# Что такое API микросервисов

---

## В этой главе

- ✓ Что такое микросервисы и чем они отличаются от монолитных приложений.
- ✓ Что такое веб-API и как они помогают осуществлять интеграцию между микросервисами.
- ✓ Наиболее важные задачи при разработке и эксплуатации микросервисов.

В этой главе дается определение наиболее важных понятий книги: микросервисов и API. *Микросервисы* — это архитектурный стиль, в котором компоненты системы проектируются как независимо развертываемые сервисы, а *API* — интерфейсы, позволяющие нам взаимодействовать с этими сервисами. Мы рассмотрим важные особенности микросервисной архитектуры и сравним ее с архитектурой монолитных приложений, которые строятся на основе единой кодовой базы и развертываются в виде единой сборки.

Мы обсудим преимущества и недостатки микросервисной архитектуры. В конце главы речь пойдет о проблемах, которые возникают при проектировании, внедрении и эксплуатации микросервисов. Все это я расскажу не для того, чтобы отговорить вас от внедрения микросервисов, а для того, чтобы вы могли принять взвешенное решение, подходят ли микросервисы именно для вас.

## 1.1. ЧТО ТАКОЕ МИКРОСЕРВИСЫ

В этом разделе мы определим, что такое микросервисная архитектура, и проанализируем, чем микросервисы отличаются от монолитных приложений. Рассмотрим преимущества и недостатки каждого архитектурного стиля. Наконец, мы также бегло вспомним исторические события, которые привели к появлению современной микросервисной архитектуры.

### 1.1.1. Определение микросервисов

Итак, что же такое микросервисы? Микросервисы можно определить по-разному, и в зависимости от того, какой аспект архитектуры мы хотим подчеркнуть, авторы дают немного различающиеся, но связанные между собой определения этого термина. Сэм Ньюман, один из самых известных авторов, пишущих о микросервисах, дает лаконичное определение: «Микросервисы — это небольшие, автономные, совместно работающие сервисы»<sup>1</sup>.

Определение подчеркивает тот факт, что микросервисы — это приложения, которые работают независимо друг от друга, но могут взаимодействовать при выполнении своих задач. В определении также подчеркивается, что микросервисы «небольшие». В данном контексте «небольшой» относится не к размеру кодовой базы, а к идее, что микросервисы — это приложения с узкой и четко определенной областью применения, следующие принципу единственной ответственности — делать что-то одно, причем хорошо.

В основополагающей статье Джеймса Льюиса и Мартина Фаулера дается более подробное определение. Они характеризуют микросервисы как архитектурный стиль с «подходом к разработке одного приложения в виде набора небольших сервисов, каждый из которых выполняется отдельно и которые взаимодействуют с помощью упрощенных механизмов, часто с использованием ресурсов HTTP API» (<https://martinfowler.com/articles/microservices.html>). Это определение подчеркивает автономность сервисов, утверждая, что они выполняются в виде отдельных процессов. Льюис и Фаулер также отмечают, что микросервисы имеют узкий функционал, говоря, что они небольшие, и четко описывают, как микросервисы обмениваются информацией с помощью таких протоколов, как HTTP.

#### ОПРЕДЕЛЕНИЕ

Микросервис — это архитектурный стиль, в котором компоненты системы проектируются как независимо развертываемые сервисы. Микросервисы проектируются под четко обозначенные бизнес-функции в определенных модулях (поддоменах) и взаимодействуют друг с другом с помощью простых протоколов, таких как HTTP.

---

<sup>1</sup> Ньюман С. Создание микросервисов. — СПб.: Питер, 2016. — С. 23.

Из приведенных определений видно, что микросервисы можно охарактеризовать как архитектурный стиль, в котором сервисы — это компоненты, выполняющие небольшой и четко определенный набор связанных функций. Как видно на рис. 1.1, это означает, что микросервис проектируется и строится вокруг конкретной бизнес-задачи, например обработки платежей, отправки электронной почты или обработки заказов от покупателей.



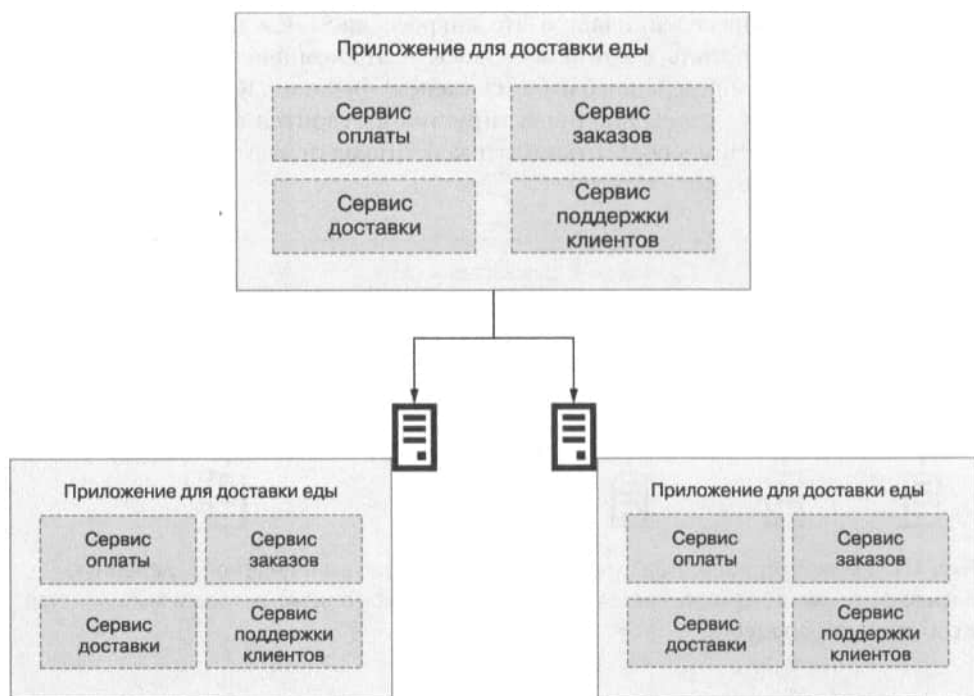
**Рис. 1.1.** В микросервисной архитектуре каждый сервис выполняет определенную бизнес-функцию и разворачивается как независимый компонент, который выполняется в собственном процессе

Микросервисы разворачиваются как независимые процессы, обычно работающие в независимых средах, и предоставляют свой функционал через четко определенные интерфейсы. В книге вы научитесь создавать микросервисы, которые работают через веб-API, хотя возможны и другие типы интерфейсов, например очереди сообщений<sup>1</sup>.

### 1.1.2. Микросервисы против монолитов

Теперь, когда мы знаем, что такое микросервисы, посмотрим, как они соотносятся с монолитной архитектурой приложений. В отличие от микросервисов монолит — это система, в которой вся функциональность разворачивается в виде единой сборки и выполняется в одном процессе. Например, на рис. 1.2 показано приложение для доставки еды с четырьмя сервисами: оплаты, заказов, доставки и поддержки покупателей. Поскольку приложение реализовано как монолит, все функциональные возможности разворачиваются вместе. Мы можем запустить несколько экземпляров монолитного приложения и заставить их работать параллельно для обеспечения избыточности и масштабируемости, но в каждом процессе все равно будет работать целое приложение.

<sup>1</sup> Полный обзор различных интерфейсов, которые можно использовать для обеспечения взаимодействия между микросервисами, приведен в книге: *Ричардсон К. Микросервисы. Паттерны разработки и рефакторинга.* — СПб.: Питер, 2019.



**Рис. 1.2.** В монолитном приложении вся функциональность разворачивается вместе в виде единой сборки на каждом сервере

## ОПРЕДЕЛЕНИЕ

*Монолит* — это архитектурный паттерн, в котором все приложение разворачивается в виде единой сборки.

Монолит стоит выбирать, когда кодовая база невелика и предполагается, что она сильно не разрастется<sup>1</sup>. Монолиты имеют свои преимущества. Во-первых, наличие всей реализации в единой кодовой базе облегчает доступ к данным и бизнес-функциям из различных модулей. А поскольку все выполняется в рамках одного процесса, легко отследить ошибки в приложении: достаточно добавить несколько точек останова в разных частях кода, и вы получите подробную картину того, что происходит, когда что-то идет не так. Кроме того, поскольку весь код находится в рамках одного проекта, вы можете использовать инструменты повышения производительности в своей любимой среде разработки при работе с бизнес-логикой приложения из различных модулей.

<sup>1</sup> Подробный анализ стратегических архитектурных решений, связанных с монолитами и микросервисами, приведен в книге: *Vernon V., Jaskula T. Strategic Monoliths and Microservices* (Addison-Wesley, 2021).



Однако по мере роста и усложнения приложения у данного типа архитектуры проявляются ограничения. Это происходит, когда кодовая база увеличивается до такой степени, что ею становится трудно управлять, а поиск в коде оказывается сложной задачей. Кроме того, возможность повторного использования кода из других модулей в рамках одного проекта часто приводит к сильной связанности (tight coupling) между компонентами. Сильная связанность возникает, когда компонент зависит от деталей реализации другого фрагмента кода.

Чем больше монолит, тем больше времени требуется для его тестирования. Каждую часть монолита необходимо протестировать, и по мере добавления новых функций набор тестов увеличивается. Следовательно, развертывание становится медленнее и разработчикам приходится накапливать изменения в рамках одного релиза, что делает релизы более сложными. Поскольку многие изменения выпускаются вместе, то, если в релизе появляется новая ошибка, часто бывает трудно определить, какое именно изменение ее вызвало, и откатить его назад. А поскольку все приложение работает в рамках одного процесса, при масштабировании ресурсов для одного компонента требуется масштабирование всего приложения. Короче говоря, изменения кода оказываются все более рискованными, а управлять развертыванием становится все сложнее. Как микросервисы могут помочь решить эти проблемы?

Микросервисы помогают нам благодаря жестким границам, разделяющим компоненты. Когда вы реализуете приложение с использованием микросервисов, каждый микросервис работает в отдельном процессе (часто на разных серверах или виртуальных машинах) и может иметь любую модель развертывания. Фактически микросервисы могут быть написаны на совершенно разных языках программирования (но это не значит, что так и должно быть!).

Поскольку у микросервисов кодовая база меньше, чем у монолита, и их логика ограничена и определена в рамках конкретной бизнес-задачи, их легче тестировать и их наборы тестов выполняются быстрее. Микросервисы не зависят от других компонентов платформы на уровне кода (за исключением, возможно, некоторых общих библиотек), их код понятнее, и их легче рефакторить. Это означает, что со временем код может улучшаться и становиться более удобным для поддержки. Следовательно, мы можем вносить небольшие изменения в код и выпускать его чаще. Маленькие релизы более управляемы, и, если обнаружится ошибка, их легче откатить назад. Но я хотел бы подчеркнуть, что микросервисы — это не панацея. Как вы увидите в разделе 1.3, они также имеют ограничения и создают свои проблемы.

Теперь, разобравшись, что такое микросервисы и чем они отличаются от монолитных приложений, сделаем шаг назад и посмотрим, какие события привели к появлению этого типа архитектуры.

### 1.1.3. История появления современных микросервисов

Во многих отношениях микросервисы не повинка<sup>1</sup>. Компании внедряли и разворачивали компоненты в виде независимых приложений задолго до того, как стала популярной концепция микросервисов. Они просто не называли это микросервисами. Вернер Фогельс, технический директор компании Amazon, рассказывает, что Amazon начала экспериментировать с этим типом архитектуры в начале 2000-х годов. К тому времени кодовая база сайта Amazon превратилась в сложную систему без четкой архитектурной схемы, а выпуск новых релизов и масштабирование системы стали серьезными проблемами. Для их решения они начали искать независимые части логики в коде и разделять их на независимо разворачиваемые компоненты, сопровождая их API. В рамках этого процесса в компании также определили данные, принадлежащие этим компонентам, и убедились, что другие части системы не могут получить доступ к этим данным иначе как через API. Они назвали этот новый тип архитектуры *сервис-ориентированной архитектурой* (<https://vimeo.com/29719577>). Компания Netflix также стала пионером в создании архитектурного стиля такого масштаба, и они назвали его «детальной сервис-ориентированной архитектурой» (fine-grained service oriented architecture)<sup>2</sup>.

Для описания этого типа архитектуры термин «*микросервис*» стали активно употреблять в начале 2010-х годов. Например, Джеймс Льюис использовал это понятие на конференции в Кракове в 2012 году в презентации под названием *Micro-Services – Java, the Unix way* («Микросервисы – Java, путь Unix») (<https://vimeo.com/74452550>). В 2014 году концепция была закреплена статьей Мартина Фаулера и Джеймса Льюиса об архитектурных особенностях микросервисов (<https://martinfowler.com/articles/microservices.html>), а также публикацией известной книги Сэма Ньюмана «Создание микросервисов».

Сегодня микросервисы — это широко распространенный архитектурный стиль. Очень многие компании уже используют микросервисы или движутся в направлении их внедрения. Однако микросервисы подходят не всем и, несмотря на множество существенных преимуществ, также могут привести к значительным проблемам, о чем вы узнаете в разделе 1.3.

<sup>1</sup> Более полный анализ истории появления микросервисной архитектуры и ее предшественников см. в книге: *Dragoni N. et al. Microservices: Yesterday, Today and Tomorrow, Present and Ulterior Software Engineering.* — Springer, 2017. — P. 195–216.

<sup>2</sup> *Wang A., Tonse S. Announcing Ribbon: Tying the Netflix Mid-Tier Services Together* («Анонсы ленты: связывая вместе сервисы промежуточного слоя Netflix») // Netflix Technology Blog, 18 января 2013 г. <https://netflixtechblog.com/announcing-ribbon-tying-the-netflix-mid-tier-services-together-a89346910a62>. Прекрасное объяснение различий между сервис-ориентированной архитектурой (SOA) и микросервисной архитектурой см. в книге: *Ричардсон К. Микросервисы.* — С. 40–41.

## 1.2. ЧТО ТАКОЕ ВЕБ-API

В этом разделе мы поговорим о веб-API. Вы узнаете, что это конкретный пример более общего интерфейса прикладного программирования (API). Важно понимать, что API — это просто слой поверх приложения и что существуют различные типы интерфейсов. Начнем с определения API, а затем обсудим, как API позволяют нам интегрировать микросервисы.

### 1.2.1. API

API — это интерфейс, который позволяет нам программно взаимодействовать с приложением. Программные интерфейсы мы можем использовать из нашего кода или терминала — в этом их отличие от графических, в которых пользовательский интерфейс предназначен для взаимодействия с приложением. Существует несколько типов интерфейсов приложений: интерфейсы командной строки (CLI; позволяют использовать приложение из терминала), пользовательские интерфейсы (UI) для настольных компьютеров, пользовательские интерфейсы (UI) для веб-приложений или веб-API. Как показано на рис. 1.3, приложение может иметь один или несколько таких интерфейсов.



**Рис. 1.3.** Приложение может иметь несколько интерфейсов

Для иллюстрации этой идеи вспомните популярную утилиту cURL (client URL). cURL — это интерфейс командной строки для библиотеки libcurl, которая реализует функциональность, позволяющую нам взаимодействовать с URL-адресами, в то время как cURL раскрывает эти возможности через CLI. Например, мы можем использовать cURL для отправки GET-запроса на URL:

```
$ curl -L http://www.google.com
```

Мы также можем использовать cURL с флагом -O, чтобы загрузить содержимое URL в файл:

```
$ curl -O http://www.gnu.org/software/gettext/manual/gettext.html
```

Библиотека `libcurl` скрыта за интерфейсом командной строки `cURL`, и ничто не мешает нам получить к ней прямой доступ через исходный код (если вам интересно, можете взять его с Github: <https://github.com/curl/curl>) и создать для нее дополнительные типы интерфейсов.

## 1.2.2. Веб-API

Теперь, когда мы поняли, что такое API, перейдем к описанию веб-API. Веб-API — это API, который для передачи данных использует HyperText Transfer Protocol (HTTP). HTTP — это протокол связи, лежащий в основе Интернета и позволяющий нам обмениваться по сети различными типами данных (текстом, изображениями, видео и данными в формате JSON<sup>1</sup>). HTTP использует понятие Унифицированного указателя ресурса (Uniform Resource Locator, URL) для определения ресурсов в Интернете, а также предоставляет возможности, которые могут быть использованы при разработке API для улучшения взаимодействия с сервером, например методы запроса (GET, POST, PUT) и HTTP-заголовки. Веб-API реализуются с помощью таких технологий, как SOAP, REST, GraphQL, gRPC и других, которые подробно рассматриваются в приложении А.

## 1.2.3. Как API помогают нам управлять интеграцией микросервисов

Микросервисы взаимодействуют друг с другом через API, поэтому API по сути являются интерфейсами к микросервисам. Эти интерфейсы документируются с использованием стандартных протоколов. Документация API точно указывает нам, что нужно сделать для взаимодействия с микросервисом и каких ответов мы можем от него ожидать. Чем лучше документирован API, тем понятнее разработчикам клиентских приложений, как он работает. В этом смысле документацию API можно представить как контракт между сервисами (рис. 1.4).

Фаулер и Льюис популяризировали идею о том, что наилучшей стратегией интеграции микросервисов является создание *умных эндпоинтов* (*smart endpoints*) и взаимодействие через *глупые каналы* (*dumb pipes*) (<https://martinfowler.com/articles/microservices.html>). В основе этой идеи лежат принципы проектирования Unix-систем, которые устанавливают, что:

- система должна состоять из небольших независимых компонентов, которые выполняют только одну задачу;
- выходные данные для каждого компонента должны быть спроектированы таким образом, чтобы они могли легко стать входными данными для другого компонента.

<sup>1</sup> Данные в формате JSON в общем виде тоже текст. — *Примеч. ред.*



**Рис. 1.4.** Спецификация API представляет собой контракт между сервером и клиентом. Пока клиент и сервер следуют спецификации API, коммуникация будет происходить так, как ожидается

Программы Unix взаимодействуют друг с другом с помощью конвейеров, которые представляют собой простые механизмы передачи сообщений от одного приложения к другому. Чтобы проиллюстрировать этот процесс, вспомните цепочку команд, которую можно запустить с терминала машины на базе Unix (например, компьютера с Mac или Linux):

```
$ history | less
```

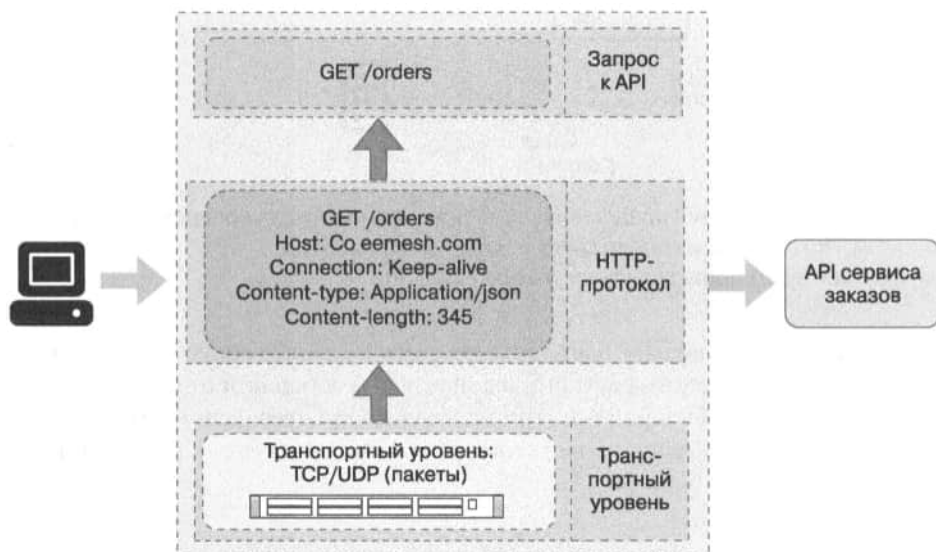
Команда `history` выводит список всех команд, которые вы выполняли из профиля Bash. Список может быть длинным, поэтому можно разбить вывод истории на страницы с помощью команды `less`. Чтобы передать данные от одной команды к другой, используйте символ вертикальной черты (`|`), который дает оболочке команду перехватить вывод `history` и передать его на вход `less`. Мы говорим, что этот тип канала глупый, потому что его единственная задача — передача сообщений от одного процесса к другому.

Как показано на рис. 1.5, веб-API обмениваются данными через HTTP. На транспортном уровне передачи данных ничего не известно о конкретном протоколе API, который мы используем, поэтому он представляет собой наш глупый канал, в то время как сам API содержит всю необходимую логику для обработки данных.

API не должны изменяться, при этом вы можете изменять внутреннюю логику работы сервиса при условии, что она соответствует документации API. Это означает, что покупатели, взаимодействующие с API, должны иметь возможность продолжить обращаться к API точно так же, как и раньше, и получать при этом те же ответы. Это приводит к еще одной важной концепции в микросервисной архитектуре: *заменяемости (replaceability)*<sup>1</sup>. Идея заключается в том, что вы должны иметь возможность полностью заменить код, который скрыт за эндпоинтом, но при этом эндпоинт останется доступным, а значит и взаимодействие между сервисами

<sup>1</sup> Ньюман С. Создание микросервисов. — С. 30.

не нарушится. Теперь, когда вы понимаете, что такое API и как эти интерфейсы помогают осуществлять интеграцию сервисов, рассмотрим наиболее важные проблемы, возникающие при создании микросервисов.



**Рис. 1.5.** Микросервисы взаимодействуют через API, используя транспортный уровень передачи данных, такой как HTTP-over-TCP

### 1.3. ПРОБЛЕМЫ МИКРОСЕРВИСНОЙ АРХИТЕКТУРЫ

Как вы видели в подразделе 1.1.2, микросервисы дают существенные преимущества. Однако их использование также сопряжено со значительными трудностями. В этом разделе мы обсудим наиболее важные проблемы, которые я разделил на пять основных категорий, таких как:

- оптимальная декомпозиция сервиса;
- интеграционные тесты микросервисов;
- обработка недоступности сервиса;
- трассировка распределенных транзакций;
- повышение сложности эксплуатации и накладных расходов на инфраструктуру.

Все проблемы и трудности, которые мы обсуждаем в этом разделе, можно решить с помощью конкретных паттернов и стратегий — часть из них подробно описана в книге. Вы также найдете ссылки на другие ресурсы, где рассматриваются эти вопросы. Главное, чтобы вы понимали, что микросервисы не являются панацеей от всех проблем, с которыми сталкиваются монолитные приложения.

### 1.3.1. Оптимальная декомпозиция сервиса

Одна из самых важных задач при проектировании микросервисов — это декомпозиция системы. Мы должны разбить систему на слабо связанные, но достаточно независимые компоненты с четко заданными границами. Вы можете выявить нецелесообразную связь между сервисами, если всякий раз, когда вы меняете один сервис, приходится менять и другой. Это означает, что либо между сервисами неустоявшиеся правила обмена, либо оба компонента достаточно зависимы друг от друга, чтобы можно было объединить их в один. Неспособность разбить систему на независимые микросервисы может означать то, что Крис Ричардсон, автор книги «Микросервисы. Паттерны разработки и рефакторинга», называет *распределенным монолитом*, — решение, когда вы объединяете все проблемы монолитной и микросервисной архитектур и не получаете при этом преимуществ ни от одной из них. В главе 3 вы узнаете, какие паттерны проектирования и стратегии декомпозиции сервисов могут помочь разбить систему на микросервисы.

### 1.3.2. Интеграционные тесты микросервисов

В подразделе 1.1.2 мы говорили, что микросервисы обычно легче тестировать, и их наборы тестов выполняются быстрее. Однако интеграционные тесты микросервисов могут быть значительно сложнее, особенно в тех случаях, когда для выполнения одной транзакции требуется взаимодействие нескольких микросервисов. Когда все ваше приложение работает в рамках одного процесса, тестировать интеграцию между различными компонентами довольно легко, и для этого в основном требуются хорошо написанные модульные тесты. В контексте микросервисов для тестирования интеграции между несколькими сервисами необходимо иметь возможность запускать их все с параметрами, как в продакшен-окружении.

Для тестирования интеграции микросервисов можно использовать различные стратегии. В первую очередь нужно убедиться, что каждый сервис имеет хорошо документированный и правильно реализованный API. Вы можете проверить реализацию API на соответствие спецификации с помощью таких инструментов, как Dredd и Schemathesis, которые мы рассмотрим в главе 12. Вы также должны убедиться, что клиент использует API именно так, как предписано его документацией. Для этого вы можете написать модульные тесты для клиента, сгенерировав по документации API ответы от сервиса<sup>1</sup>. Но ни один из этих тестов не будет достаточным без полноценного сквозного тестирования (end-to-end, e2e-тестирование), при котором запускаются реальные микросервисы, отправляющие запросы друг другу.

---

<sup>1</sup> Чтобы больше узнать о процессе разработки API и о том, как использовать mock-серверы при разработке клиента, посмотрите мою презентацию API Development Workflows for Successful Integrations («Процессы разработки API для успешных интеграций») // Manning API Conference, 3 августа 2021 года. [https://youtu.be/SUKqmEX\\_uwg](https://youtu.be/SUKqmEX_uwg).

### 1.3.3. Обработка недоступности сервиса

Необходимо убедиться, что приложения устойчивы к недоступности сервисов, тайм-аутам подключений и запросов, ошибочным запросам и т. д. Например, когда мы размещаем заказ через приложение доставки еды, такое как Uber Eats, Delivery Hero или Deliveroo, запускается цепочка запросов между сервисами для обработки и доставки заказа, и любой из этих запросов может завершиться неудачей. Рассмотрим процесс, который происходит, когда пользователь размещает заказ (см. рис. 1.6 для иллюстрации цепочки запросов).

1. Покупатель размещает заказ и оплачивает его. Заказ размещается с помощью сервиса заказов, а для обработки платежа тот взаимодействует с сервисом платежей.
2. Если оплата прошла успешно, сервис заказов делает запрос в сервис кухни для постановки в очередь заказа на приготовление.
3. После приготовления заказа сервис кухни делает запрос в сервис доставки, чтобы запланировать доставку.



**Рис. 1.6.** Микросервисы должны быть устойчивы к таким событиям, как недоступность сервиса, тайм-аут запроса и ошибки обработки, полученные от других сервисов, и либо повторно выполнять запросы, либо выдавать пользователю понятный ответ



Если в столь сложной цепочке запросов один из используемых сервисов не ответит так, как ожидалось, это может вызвать каскадный отказ в приложении, при котором заказ останется или необработанным, или в неопределенном состоянии. Поэтому важно спроектировать микросервисы таким образом, чтобы они могли надежно работать с отказавшими эндпоинтами. Тесты end-to-end должны учитывать эти сценарии и проверять поведение сервисов в таких ситуациях.

### 1.3.4. Трассировка распределенных транзакций

Взаимодействующим сервисам иногда приходится обрабатывать распределенные транзакции. Например, в приложении для доставки еды нужно отслеживать имеющиеся запасы ингредиентов, чтобы ассортимент был актуальным. Когда пользователь делает заказ, нам нужно обновить запас ингредиентов, чтобы актуализировать ассортимент в приложении. В частности, следует обновлять запас ингредиентов сразу после успешной обработки платежа. Как показано на рис. 1.7, успешная обработка заказа включает в себя следующие действия.

1. Обработать платеж.
2. Если оплата прошла успешно, обновить статус заказа, чтобы указать, что он находится в процессе выполнения.
3. Связаться с сервисом кухни, чтобы поставить заказ в очередь на приготовление.
4. Обновить запас ингредиентов, чтобы отразить их наличие на текущий момент.



**Рис. 1.7.** Распределенная транзакция подразумевает взаимодействие между несколькими сервисами. Если какой-либо из них отказывает, то эта ситуация должна быть обработана и пользователю должен быть предоставлен понятный ответ

Все эти операции связаны между собой и должны быть управляемы таким образом, что все либо должны успешно выполняться вместе, либо завершиться отказом. Мы не можем оплатить заказ без корректного обновления его статуса, и мы не должны отправлять его на приготовление, если оплата не прошла. Можно изменить информацию о наличии запаса ингредиентов во время создания заказа, но если впоследствии платеж не пройдет, мы должны убедиться, что откатали это изменение. Если все эти действия происходят в рамках одного процесса, то управление потоком простое, но при использовании микросервисов приходится управлять результатами различных процессов. В таком случае важно обеспечить надежный процесс взаимодействия между сервисами, чтобы мы точно знали, какая ошибка когда возникает, и принимали соответствующие меры.

Если какие-то сервисы совместно обрабатывают определенные запросы, также нужно иметь возможность трассировать путь прохождения запроса через различные сервисы, чтобы выявлять ошибки во время транзакции. Для мониторинга распределенных транзакций необходимо настроить распределенное логирование и трассировку ваших микросервисов. Вы можете узнать больше об этой теме из книги Джейми Ридесела *Software Telemetry: Reliable logging and monitoring* (Manning, 2021).

### **1.3.5. Повышение сложности эксплуатации и накладных расходов на инфраструктуру**

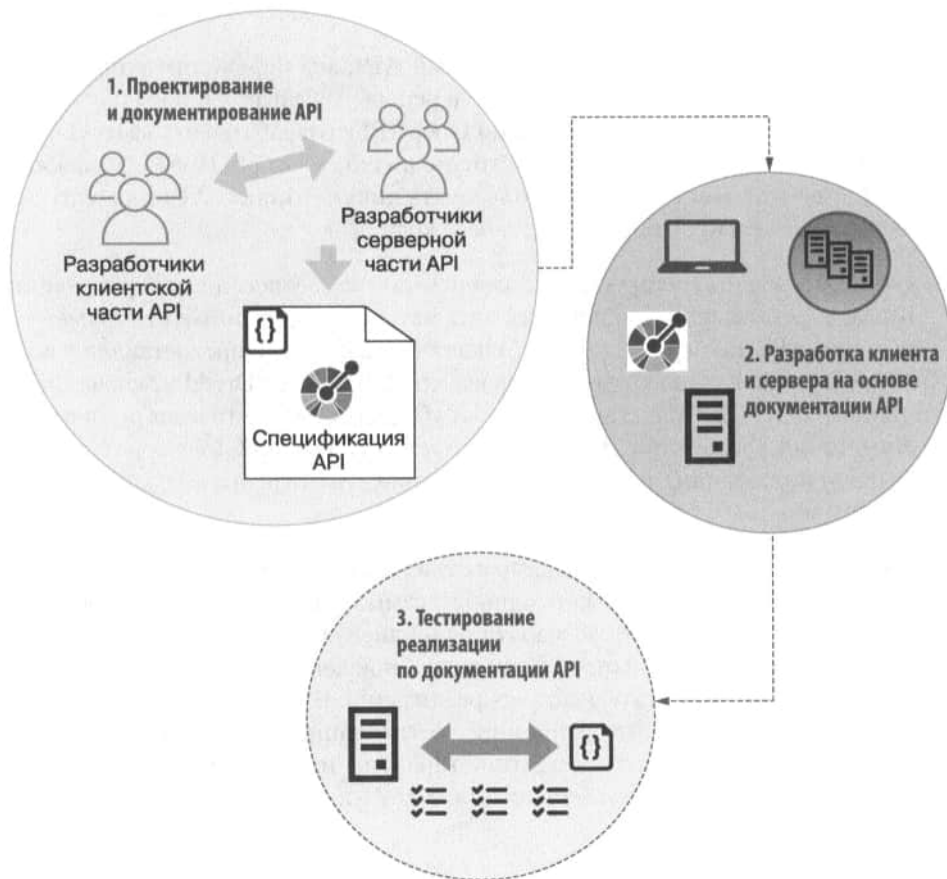
Еще одна важная проблема, возникающая при использовании микросервисов, — это повышенная сложность управления и увеличение накладных расходов для вашей платформы. Когда весь бэкэнд вашего сайта работает в рамках одной сборки приложения, вам нужно развернуть и мониторить только один процесс. Когда у вас дюжина микросервисов, каждый должен быть настроен, развернут и управляем. И это подразумевает не только выделение серверов для развертывания сервисов, но и управление потоками агрегации логов, настройку систем мониторинга, настройку оповещений, механизмов автоматического восстановления и т. д. Как вы узнаете в главе 3, у каждого сервиса есть собственная база данных, а значит, им также требуется несколько инсталляций баз данных со всеми функциями, необходимыми для масштабирования. И вполне обычна ситуация, когда при новом развертывании меняется эндпоинт микросервиса, будь то IP, базовый URL или конкретный путь в полном URL, а это означает, что об изменениях нужно уведомить покупателей.

Когда компания Amazon только начала переходить к микросервисной архитектуре, обнаружилось, что команды разработчиков тратят около 70 % своего времени на управление инфраструктурой (<https://vimeo.com/29719577>, 07:53). Такое вполне вероятно может произойти, если вы с самого начала не внедряете лучшие практики по автоматизации инфраструктуры. И даже если вы это сделаете, вам, скорее всего, придется потратить много времени на разработку собственных утилит для эффективного управления вашими сервисами.

## 1.4. ВВЕДЕНИЕ В РАЗРАБОТКУ ЧЕРЕЗ ДОКУМЕНТИРОВАНИЕ (DDD)

Как пояснялось в подразделе 1.2.3, успех интеграции с помощью API зависит от хорошей документации API, и в этом разделе мы познакомимся с процессом разработки API, при котором документация первична. Как показано на рис. 1.8, разработка через документирование — это подход к разработке API, который состоит из трех этапов.

1. Проектирование и документирование API.
2. Разработка клиентской и серверной части в соответствии с документацией.
3. Тестирование клиента и сервера на соответствие документации.



**Рис. 1.8.** Разработка через документирование состоит из трех этапов: проектирования и документирования, разработки и валидации

Рассмотрим каждый из этих пунктов. Первый шаг включает в себя проектирование и написание спецификации. Мы разрабатываем API для кого-то, поэтому перед этим должны спроектировать API, который отвечает потребностям разработчиков клиентской части. Точно так же, как мы привлекаем пользователей к проектированию пользовательского интерфейса приложения (UI), мы должны привлекать разработчиков клиентских сервисов к проектированию API.

Хороший дизайн API удобен при разработке, а хорошая документация API способствует успешной интеграции. Что такое документация API? Это описание API в соответствии с языком описания интерфейсов (interface description language, IDL), таким как OpenAPI для REST API и Schema Definition Language (SDL) для GraphQL API. Стандартные IDL имеют экосистемы инструментов и фреймворков, которые облегчают разработку, тестирование и визуализацию API, поэтому стоит потратить время на их изучение. В этой книге вы научитесь документировать API с помощью OpenAPI (см. главу 5) и SDL (см. главу 8).

Создав документацию на спроектированный API, мы переходим ко второму этапу — к созданию сервера и клиента на ее основе. В главах 2 и 6 вы научитесь анализировать требования спецификации OpenAPI и разрабатывать на их основе приложение, а в главе 10 мы применим тот же подход к GraphQL API. Разработчики API клиентов могут также использовать документацию API для запуска mock-серверов и тестирования на них своего кода<sup>1</sup>.

На последнем этапе мы тестируем нашу реализацию на соответствие документации API. В главе 12 вы научитесь использовать автоматизированные инструменты тестирования API, такие как Dredd и Schemathesis, которые предоставляют возможность сгенерировать набор тестов для вашего API. Запуск Dredd и Schemathesis в сочетании с модульными тестами позволяет убедиться в том, что ваша реализация API работает так, как должна. Эти тесты следует запускать на CI-сервере (сервере непрерывной интеграции), чтобы случайно не допустить в релиз код, который нарушает контракт документации API.

Благодаря тому, что на первом этапе разработки API выполняется документирование, подход DDD помогает избежать одной из самых распространенных проблем: разногласий между командами разработчиков клиента и сервера по поводу того, как должен работать API. При отсутствии подробной документации API разработчикам часто приходится гадать о деталях реализации. В таких случаях API редко успешно проходит первое интеграционное тестирование. Хотя разработка через документирование не дает стопроцентной гарантии того, что интеграция будет работать, она значительно снижает риск отказа API-интеграции.

---

<sup>1</sup> Чтобы узнать, как разработчики могут использовать документацию API, ознакомьтесь с моим докладом *Leveraging API Documentation to Deliver Reliable API Integrations* («Использование документации API для разработки надежных интеграций») // API Specifications Conference, 28–29 сентября 2021 года. <https://youtu.be/kAWvM-CVcnw>.

## 1.5. ЗНАКОМСТВО С ПРИЛОЖЕНИЕМ COFFEEMESH

Чтобы проиллюстрировать концепции и идеи, которые объясняются в книге, мы создадим компоненты приложения CoffeeMesh. Это вымышленное приложение, через которое покупатели могут заказывать кофе в любом месте и в любое время. Платформа CoffeeMesh состоит из набора микросервисов, реализующих различные возможности, например обработку заказов и планирование доставки. В главе 3 мы спроектируем платформу CoffeeMesh. Чтобы дать вам представление о том, чему вы научитесь в книге, начнем реализовывать API сервиса заказов CoffeeMesh в главе 2. Но прежде, чем мы закончим главу, я хотел бы еще раз кратко сделать обзор того, что вы узнаете из этой книги.

## 1.6. ДЛЯ КОГО ЭТА КНИГА И ЧЕМУ ВЫ НАУЧИТЕСЬ

Чтобы извлечь максимальную пользу из книги, вы должны знать основы веб-разработки. Примеры кода в книге написаны на Python, поэтому неплохо иметь базовое понимание Python, хотя это не обязательно. Вам не нужно уметь разбираться в веб-API или микросервисах, поскольку я подробно расскажу об этих технологиях. Хорошо, если вы знакомы с паттерном Model-View-Controller (MVC) для веб-разработки или с его вариантами, такими как паттерн Model-Template-View (MTV), реализованный в популярном фреймворке Django на Python. Периодически мы будем сравнивать эти паттерны для иллюстрации некоторых концепций. Базовое знакомство с Docker и облачными вычислениями будет полезно для понимания материала глав о развертывании, но я сделаю все возможное, чтобы подробно объяснить каждую концепцию.

В этой книге на практических примерах объясняется, как разрабатывать микросервисы на основе API на Python. Вы узнаете:

- стратегии декомпозиции сервисов для проектирования микросервисной архитектуры;
- как проектировать REST API и документировать их с помощью спецификации OpenAPI;
- как разрабатывать REST API на Python с использованием популярных фреймворков, таких как FastAPI и Flask;
- как проектировать и использовать GraphQL API и разрабатывать их с помощью Python-фреймворка Ariadne;
- как выполнять тестирование API на основе свойств (property-based testing, PBT) и с помощью таких фреймворков, как Dredd и Schemathesis;
- полезные паттерны проектирования для достижения слабой связанности в ваших микросервисах;

- как добавить аутентификацию и авторизацию в ваши API с помощью протоколов Open Authorization (OAuth) и OpenID Connect (OIDC);
- как развернуть микросервисы с помощью Docker и Kubernetes в AWS.

К концу этой книги вы в полной мере оцените преимущества, которые дает веб-приложениям микросервисная архитектура, а также узнаете обо всех возможных проблемах и трудностях ее использования. Вы будете знать, как интегрировать микросервисы с помощью API, как разрабатывать и документировать эти API, используя стандарты и лучшие практики, и будете готовы определить модуль API с четкими границами в приложении. Наконец, вы научитесь тестировать, развертывать и делать безопасными свои микросервисы.

## РЕЗЮМЕ

- Микросервисы — это архитектурный стиль, при котором компоненты системы проектируются и разрабатываются как независимо развертываемые сервисы. Это снижает размеры кодовых баз, упрощает их поддержку и позволяет оптимизировать и масштабировать сервисы независимо друг от друга.
- Монолиты — это архитектурный стиль, при котором целые приложения развертываются в одной сборке и запускаются в одном процессе. Это облегчает развертывание и мониторинг приложения, но усложняет развертывание, когда кодовая база разрастается.
- Приложения могут иметь несколько типов интерфейсов: пользовательские интерфейсы, интерфейсы CLI и API. API — это интерфейс, который позволяет нам программно взаимодействовать с приложением из нашего кода или терминала.
- Веб-API — это API, который работает на веб-сервере и использует протокол HTTP для передачи данных. Мы используем веб-API для предоставления возможностей сервиса через Интернет.
- Микросервисы взаимодействуют друг с другом с помощью умных эндпоинтов и глупых каналов. Глупый канал — это канал, который просто передает данные от одного компонента к другому. Отличным примером такого канала для микросервисов является протокол HTTP, который обменивается данными между клиентом и сервером, ничего не зная о протоколе, используемом API. Поэтому веб-API являются отличной технологией для интеграции между микросервисами.
- Несмотря на свои преимущества, микросервисы влекут за собой некоторые проблемы.
  - *Эффективная декомпозиция сервисов* — необходимо проектировать сервисы с четкими границами вокруг конкретных бизнес-задач; в противном случае мы рискуем построить так называемый распределенный монолит.

- *Необходимость проведения интеграционных тестов* — интеграционное тестирование всех микросервисов оказывается сложной задачей, но можно снизить риск отказов интеграции, обеспечив проверку правильности реализации API.
- *Обработка недоступности сервиса* — совместно работающие сервисы могут сталкиваться с недоступностью сервисов, тайм-аутами запросов и ошибками обработки, поэтому они должны уметь обрабатывать такие ситуации.
- *Трассировка распределенных транзакций* — трассировка ошибок в нескольких сервисах является сложной задачей и требует наличия средств программной телеметрии, которые позволяют централизовать логи, обеспечить видимость API и мониторить запросы в разных сервисах.
- *Повышенная сложность эксплуатации и накладные расходы на инфраструктуру* — каждый микросервис требует создания собственной инфраструктуры, включая серверы, системы мониторинга и оповещения, поэтому вам придется вкладывать дополнительные усилия в автоматизацию инфраструктуры.
- Разработка через документирование — это процесс разработки API, который состоит из трех этапов:
  - 1) проектирования и документирования API;
  - 2) разработки API на основе документации;
  - 3) тестирования API на соответствие документации.

Когда документирование API стоит во главе угла процесса разработки, можно избежать многих распространенных проблем, с которыми сталкиваются разработчики API, и, следовательно, снизить вероятность отказа API-интеграции.

# 2

## Разработка простого API

---

### В этой главе

- ✓ Чтение и понимание требований к спецификации API.
- ✓ Разделение приложения на уровень данных, уровень приложения и уровень интерфейса.
- ✓ Реализация эндпоинтов API с помощью FastAPI.
- ✓ Реализация моделей (схем) валидации данных с помощью pydantic.
- ✓ Тестирование API с помощью Swagger UI.

В этой главе мы реализуем API для сервиса заказов, который является одним из микросервисов сайта CoffeeMesh, проекта, представленного в разделе 1.5. CoffeeMesh — это приложение, в котором можно заказать доставку кофе в любое время, где бы вы ни находились. Сервис заказов позволяет покупателям размещать заказы в CoffeeMesh. По мере реализации API сервиса заказов вы получите первое представление о понятиях и процессах, которые мы более подробно рассмотрим в книге. Код для этой главы доступен в папке `ch02` репозитория GitHub к книге.

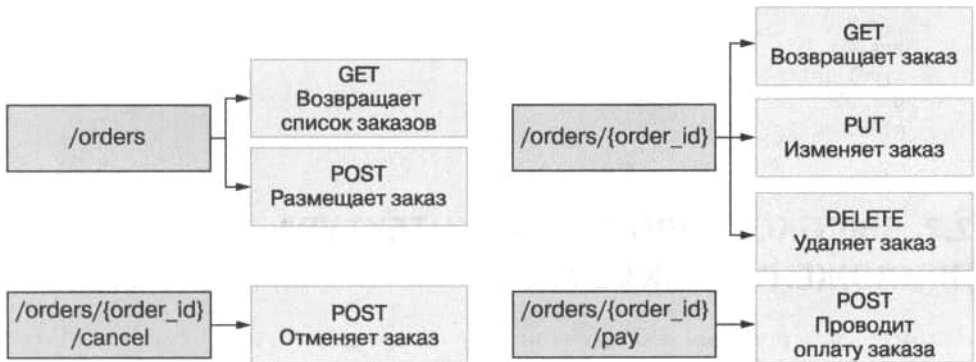


## 2.1. ЗНАКОМИМСЯ СО СПЕЦИФИКАЦИЕЙ API СЕРВИСА ЗАКАЗОВ

Начнем с анализа требований API. Используя API сервиса заказов, мы можем размещать заказы, получать сведения о них, изменять или отменять их. Спецификация API сервиса заказов доступна в файле с именем `ch02/oas.yaml` в репозитории GitHub. OAS расшифровывается как *OpenAPI Specification (спецификация OpenAPI)*, которая является стандартным форматом для документирования REST API. Как документировать API с помощью OpenAPI, мы обсудим в главе 5.

Как видно на рис. 2.1, спецификация API описывает REST API с четырьмя основными путями URL:

- `/orders` — позволяет получать списки заказов (GET) и создавать заказы (POST);
- `/orders/{order_id}` — позволяет получить информацию о конкретном заказе (GET), изменить (PUT) и удалить заказ (DELETE);
- `/orders/{order_id}/cancel` — позволяет отменить заказ (POST);
- `/orders/{order_id}/pay` — позволяет оплатить заказ (POST).



**Рис. 2.1.** API сервиса заказов предоставляет семь эндпоинтов, построенных на четырех URL-путях. Каждый реализует различные функции, такие как размещение и отмена заказа

В дополнение к документированию эндпоинтов спецификация API также содержит модели данных, которые описывают данные, передающиеся между этими эндпоинтами. В OpenAPI мы называем эти модели *схемами* (schemas), и вы можете найти их в разделе `components` спецификации API сервиса заказов (листинг 2.1). В схемах указывается, какие свойства должны быть включены в тело запроса и какие у них типы.

Например, в схеме `OrderItemSchema` указано, что свойства `product` и `size` являются обязательными, а свойство `quantity` — необязательным. Если свойство `quantity` отсутствует в теле запроса, то значение по умолчанию равно 1. Поэтому при реализации API должно быть обеспечено наличие свойств `product` и `size` до того, как мы попытаемся создать заказ.

### Листинг 2.1. Спецификация схемы `OrderItemSchema`

```
# file: oas.yaml

OrderItemSchema:
  type: object
  required:
    - product
    - size
  properties:
    product:
      type: string
    size:
      type: string
      enum:
        - small
        - medium
        - big
    quantity:
      type: integer
      default: 1
      minimum: 1
```

## 2.2. ВЫСОКОУРОВНЕВАЯ АРХИТЕКТУРА ПРИЛОЖЕНИЯ ЗАКАЗОВ

В этом разделе приведен высокоуровневый обзор архитектурной схемы API сервиса заказов. Наша цель — определить уровни приложения и установить четкие границы и разделение задач между всеми уровнями. Мы делим систему на три уровня: уровень API (интерфейса), уровень бизнес-логики и уровень данных (рис. 2.2).

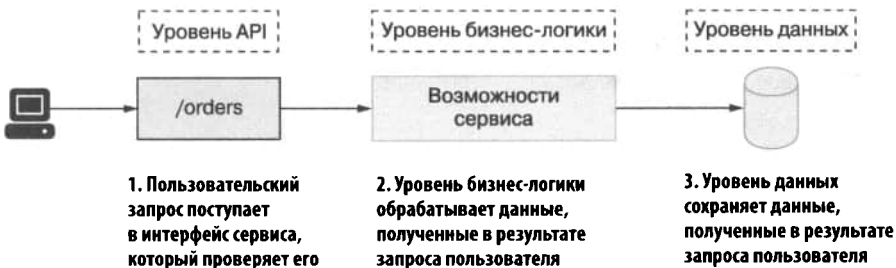
Такой способ структурирования приложения является адаптацией паттерна трехуровневой архитектуры, который разделяет приложение на уровень данных, уровень бизнес-логики и уровень представления. Уровень данных — это часть приложения, которая знает, как сохранить данные, чтобы мы могли получить их позже. Уровень данных реализует модели данных, необходимые для взаимодействия с нашим источником данных (рис. 2.3).

Например, если нашим постоянным хранилищем является база данных SQL, модели на уровне данных будут представлять собой таблицы в базе данных, часто с использованием фреймворка объектно-реляционного отображения (object relational mapper, ORM).



**Рис. 2.2.** Чтобы обеспечить разделение задач между различными компонентами нашего сервиса, мы разделим код на три уровня

Уровень бизнес-логики реализует возможности нашего сервиса. Он управляет взаимодействием между уровнем API и уровнем данных сервиса. Что касается сервиса заказов, эта часть знает, что сделать, чтобы разместить, отменить или оплатить заказ. Уровень бизнес-логики реализует возможности сервиса, а уровень API является переходником поверх логики приложения и предоставляет возможности сервиса его потребителям. Рисунок 2.2 иллюстрирует эту взаимосвязь между уровнями сервиса, а рис. 2.3 показывает, как запрос пользователя обрабатывается каждым уровнем.



**Рис. 2.3.** Когда запрос пользователя поступает в сервис заказов, он проверяется на уровне интерфейса. Затем уровень интерфейса взаимодействует с уровнем бизнес-логики для обработки запроса. После обработки уровень данных сохраняет данные из запроса

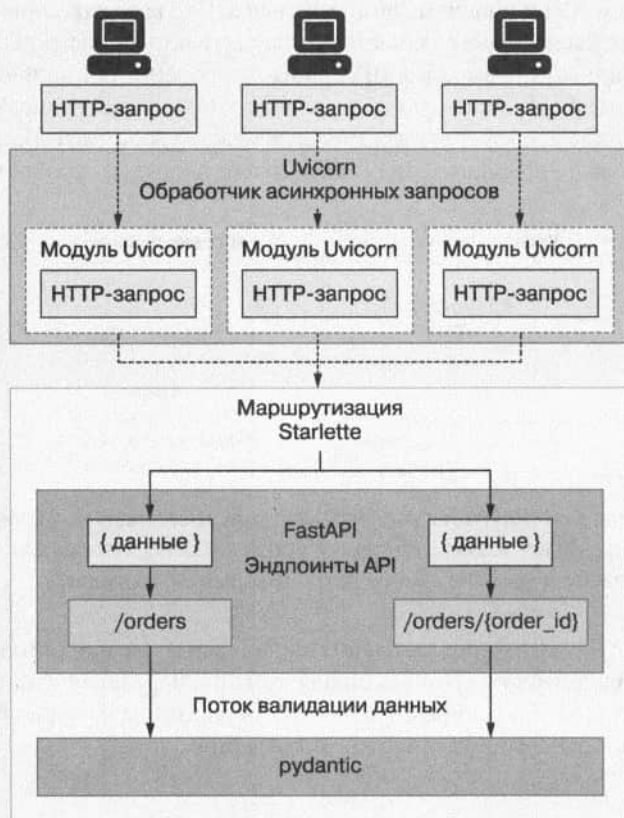
Важнейшая задача уровня API — проверка входящих запросов и возвращение ожидаемых ответов. Он взаимодействует с уровнем бизнес-логики, передавая данные, отправленные пользователем, чтобы ресурсы могли быть обработаны и сохранены на сервере. Уровень API эквивалентен уровню представления.

## 2.3. РЕАЛИЗАЦИЯ ЭНДПОИНТОВ API

В этом разделе вы научитесь реализовывать уровень API сервиса заказов. Я покажу вам, как разбить реализацию API на последовательные шаги. На первом этапе мы создадим минимальную реализацию эндпоинтов с имитированными ответами. В следующих разделах главы улучшим реализацию, добавив проверку данных и динамические ответы. Вы также узнаете о библиотеке FastAPI и о том, как ее можно использовать для разработки веб-API.

## ЧТО ТАКОЕ FASTAPI

FastAPI (<https://github.com/tiangolo/fastapi>) — это фреймворк веб-API, построенный на базе Starlette (<https://github.com/encode/starlette>). Starlette — это высокопроизводительный, легкий веб-фреймворк, асинхронный интерфейс шлюза сервера (asynchronous server gateway interface, ASGI), и с ним мы можем реализовать наши сервисы как набор асинхронных задач для повышения производительности приложений. FastAPI использует библиотеку pydantic (<https://github.com/samuelcolvin/pydantic/>) для валидации данных. На следующем рисунке показано, как все эти технологии сочетаются друг с другом.



Uvicorn (<https://github.com/encode/uvicorn>) — это асинхронный веб-сервер, обычно используемый для запуска приложений Starlette. Uvicorn обрабатывает HTTP-запросы и перенаправляет их в Starlette, а тот определяет, какие функции в вашем приложении следует вызвать при поступлении запроса на сервер. FastAPI построен поверх Starlette и расширяет его возможности функциями валидации данных и документирования API.

Прежде чем приступить к реализации API, необходимо настроить среду для этого проекта. Создайте папку с именем `ch02` и перейдите в нее с помощью команды `cd` в терминале. Для установки наших зависимостей и управления ими мы будем использовать `Pipenv`.

### О ЗАВИСИМОСТЯХ

Если вы хотите быть уверены, что у вас установлены те же зависимости, которые я использовал при написании этой книги, можете взять файлы `ch02/Pipfile` и `ch02/Pipfile.lock` из репозитория GitHub книги и запустить команду `pipenv install`.

`Pipfile` описывает среду, которую мы хотим создать с помощью `Pipenv`. Среди прочего `Pipfile` указывает, какую версию Python нужно использовать для создания среды, и определяет URL-адреса репозитория PyPI для извлечения зависимостей. `Pipenv` также облегчает разделение продакшен-зависимостей и зависимостей разработки с помощью специальных флагов установки для каждого набора. Например, для установки `pytest` мы выполняем `pipenv install pytest -- dev`. `Pipenv` также предоставляет команды, позволяющие легко управлять нашими виртуальными средами, например `pipenv shell` для активации виртуальной среды или `pipenv --rm` для ее удаления.

`Pipenv` — инструмент управления зависимостями для Python, который гарантирует, что одни и те же версии наших зависимостей будут установлены в разных средах. Другими словами, `Pipenv` позволяет создавать среды детерминированным способом, для чего используется файл `Pipfile.lock` с описанием точных версий пакетов, которые были установлены (листинг 2.2).

**Листинг 2.2.** Создание виртуальной среды и установка зависимостей с помощью `pipenv`

```
$ pipenv --three      ← Создаем виртуальную среду, используя pipenv,
                        и прописываем использование Python 3
$ pipenv install fastapi uvicorn ← Устанавливаем FastAPI и Uvicorn
$ pipenv shell        ← Активизируем
                        виртуальную среду
```

Теперь, когда наши зависимости установлены, создадим API. Сначала скопируйте спецификацию API в разделе `ch02/oas.yaml` репозитория GitHub в папку `ch02`, которую создали ранее. Затем создайте подпапку `orders`, в которой будет храниться наша реализация API. В папке `orders` создайте файл `app.py`. Создайте еще одну

подпапку с названием `orders/api` и в ней — файл `orders/api/api.py`. На данном этапе структура проекта должна выглядеть следующим образом:

```

├── Pipfile
├── Pipfile.lock
├── oas.yaml
├── orders
│   ├── api
│   │   └── api.py
│   └── app.py

```

В листинге 2.3 показано, как создать экземпляр приложения FastAPI в файле `orders/app.py`. Экземпляр класса `FastAPI` — это объект, соответствующий API, который мы реализуем. Он предоставляет *декораторы* (функции, добавляющие дополнительные возможности функции или классу), которые позволяют нам регистрировать наши функции представления<sup>1</sup>.

### Листинг 2.3. Создание экземпляра приложения FastAPI

```
# file: orders/app.py
```

```
from fastapi import FastAPI
```

```
app = FastAPI(debug=True)
```

```
from orders.api import api
```

Создаем экземпляр класс `FastAPI`. Этот объект представляет наше приложение API

Импортируем модуль `api`, чтобы наши функции представления были зарегистрированы во время загрузки

В листинге 2.4 показана минимальная реализация эндпоинтов нашего API. Код находится в файле `orders/api/api.py`. Мы объявляем статический объект `order` и возвращаем одни и те же данные во всех эндпоинтах, кроме `DELETE /orders/{order_id}`, который выдает пустой ответ. Позже мы изменим реализацию, чтобы использовать динамический список заказов. Декораторы FastAPI преобразуют данные, которые мы возвращаем в каждой функции, в HTTP-ответ; они также сопоставляют наши функции с определенным URL на нашем сервере. По умолчанию FastAPI включает в наши ответы статус-коды 200 (OK), но мы можем переопределить это поведение, используя параметр `status_code` в декораторах маршрутов, как делаем в эндпоинтах `POST /orders` и `DELETE /orders/{order_id}`.

### Листинг 2.4. Минимальная реализация API сервиса заказов

```
# file: orders/api/api.py
```

```
from datetime import datetime
```

```
from uuid import UUID
```

<sup>1</sup> Классическое объяснение паттерна Декоратор см. в книге: *Гамма Э. и др.* Паттерны объектно-ориентированного проектирования. — СПб.: Питер, 2020. — С. 209–221. Более питоническое введение в декораторы см. в книге: *Рамальо Л.* Python. К вершинам мастерства. — 2022.

```

from starlette.responses import Response
from starlette import status

from orders.app import app

order = {
    'id': 'ff0f1355-e821-4178-9567-550dec27a373',
    'status': "delivered",
    'created': datetime.utcnow(),
    'order': [
        {
            'product': 'cappuccino',
            'size': 'medium',
            'quantity': 1
        }
    ]
}

@app.get('/orders')
def get_orders():
    return {'orders': [orders]}

@app.post('/orders', status_code=status.HTTP_201_CREATED)
def create_order():
    return order

@app.get('/orders/{order_id}')
def get_order(order_id: UUID):
    return order

@app.put('/orders/{order_id}')
def update_order(order_id: UUID):
    return order

@app.delete('/orders/{order_id}', status_code=status.HTTP_204_NO_CONTENT)
def delete_order(order_id: UUID):
    return Response(status_code=HTTPStatus.NO_CONTENT.value)

@app.post('/orders/{order_id}/cancel')
def cancel_order(order_id: UUID):
    return order

@app.post('/orders/{order_id}/pay')
def pay_order(order_id: UUID):
    return order

```

Определяем объект `order`, который будет возвращаться в наших ответах

Регистрируем эндпоинт GET для пути `/orders`

Указываем, что статус-код ответа — 201 (Created)

Определяем в фигурных скобках параметры URL, такие как `order_id`

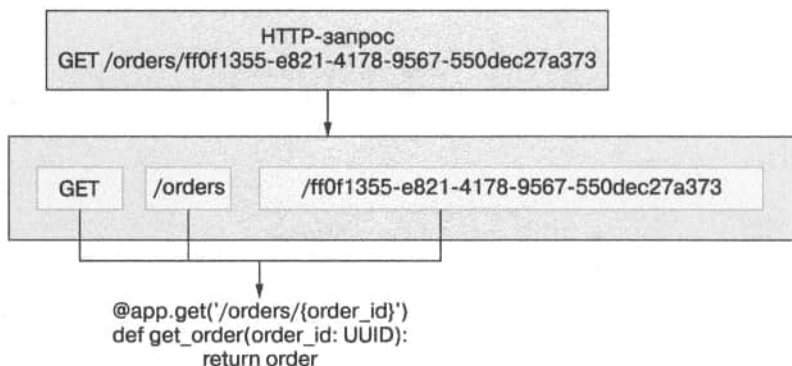
Захватываем параметр URL в качестве аргумента функции

Используем `HTTPStatus.NO_CONTENT.value`, чтобы вернуть пустой ответ

FastAPI предоставляет декораторы, названные в честь HTTP-методов, такие как `get()` и `post()`. Мы используем их для регистрации эндпоинтов API. Декораторы FastAPI принимают по крайней мере один аргумент, являющийся URL-адресом, который мы хотим зарегистрировать.

Наши функции представления могут принимать любое количество параметров. Если имя параметра соответствует имени параметра URL-адреса, FastAPI передает его из URL в нашу функцию представления при ее вызове. Например, как показано на рис. 2.4, URL `/orders/{order_id}` определяет параметр пути с именем `order_id`, и, соответственно, наши функции представления, зарегистрированные для этого URL-адреса, принимают аргумент с именем `order_id`. Если пользователь переходит по URL `/orders/53e80ed2-b9d6-4c3b-b549-258aaaf9533`, наши функции представления будут вызваны с параметром `order_id`, установленным равным `53e80ed2-b9d6-4c3b-b549-258aaaf9533`.

FastAPI позволяет нам задать тип и формат параметра URL-адреса с помощью подсказок типа. В листинге 2.4 мы указываем, что тип `order_id` — это *универсально уникальный идентификатор* (universally unique identifier, UUID). FastAPI аннулирует все вызовы, в которых `order_id` не соответствует этому формату.



**Рис. 2.4.** FastAPI знает, как сопоставить запрос с нужной функцией, и передает все соответствующие параметры из запроса в функцию. На этом рисунке GET-запрос к эндпоинту `/orders/{order_id}` с параметром `order_id`, равным `ff0f1355-e821-4178-9567-550dec27a373`, передается функции `get_order()`

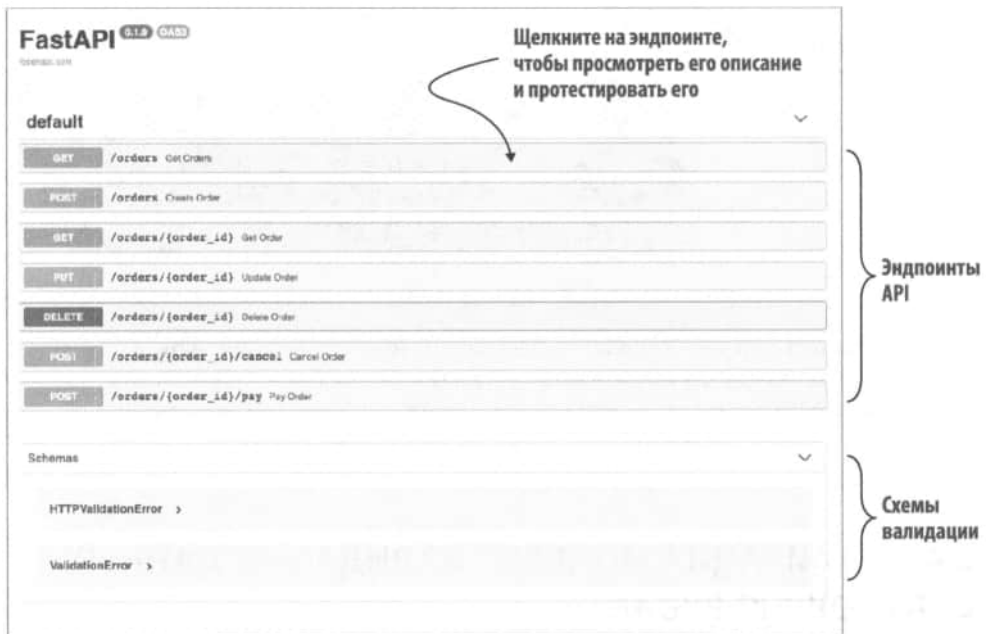
По умолчанию ответы FastAPI содержат статус-код 200 (OK), но мы можем изменить это поведение, установив параметр `status_code` в декораторах эндпоинтов. В листинге 2.4 мы определили `status_code` равным 201 (Created) в эндпоинте `POST /orders` и 204 (No Content) — в эндпоинте `DELETE /orders/{order_id}`. Подробнее статус-коды объясняются в разделе 4.6.

Теперь вы можете запустить приложение, чтобы получить представление о том, как выглядит API. Выполните следующую команду из каталога `orders` верхнего уровня:

```
$ uvicorn orders.app:app --reload
```



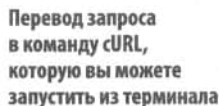
Она загружает сервер в режиме *горячей перезагрузки*. Этот режим позволяет перезагружать сервер всякий раз, когда вы вносите изменения в код приложения. Перейдите по адресу <http://127.0.0.1:8000/docs> в браузере, и вы увидите диалоговое отображение документации API, сгенерированной FastAPI на основе кода (рис. 2.5). Такую визуализацию предоставляет фреймворк Swagger UI, и это один из самых популярных инструментов визуализации REST API. Другой популярный инструмент — генератор документации Redoc, который также поддерживается FastAPI под URL <http://127.0.0.1:8000/redoc>.



**Рис. 2.5.** Страница Swagger UI, динамически генерируемая FastAPI на основе нашего кода. Мы можем использовать ее для тестирования реализации наших эндпоинтов

Если вы щелкнете на любом из эндпоинтов, представленных в пользовательском интерфейсе Swagger UI, то получите дополнительную информацию о нем. Вы также увидите кнопку Try it Out (Попробовать), которая дает возможность протестировать эндпоинт прямо из этого пользовательского интерфейса. Нажмите ее, затем нажмите Execute (Выполнить), и вы получите ответ, который мы включили в наши эндпоинты (рис. 2.6).

Теперь, когда у нас есть каркас нашего API, перейдем к реализации валидаторов для входящей полезной нагрузки и исходящих ответов.



Эндпоинт  
запроса

### Полезная нагрузка ответа на запрос

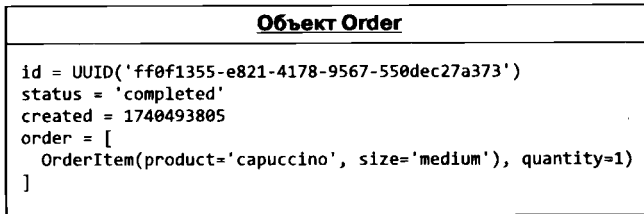
**Рис. 2.6.** Для тестирования эндпоинта щелкните на нем, чтобы развернуть. В правом верхнем углу описания вы увидите кнопку Try it Out (Попробовать). Нажмите ее, а затем кнопку Execute (Выполнить). Так вы отправите запрос к серверу и сможете увидеть ответ

## 2.4. РЕАЛИЗАЦИЯ МОДЕЛЕЙ ВАЛИДАЦИИ ДАННЫХ С ПОМОЩЬЮ PYDANTIC

Теперь, когда мы реализовали основной макет для путей URL нашего API, добавим проверку входящей полезной нагрузки и способ маршалинга исходящих ответов. Валидация и маршалинг данных — важнейшие операции в API, и для успешной интеграции с помощью API нам нужно выполнить их правильно. В следующих разделах вы узнаете, как добавить надежные возможности валидации и маршалинга данных в ваши API. FastAPI использует для валидации данных библиотеку `pydantic`, поэтому в этом разделе мы начнем с создания `pydantic`-моделей.

## ОПРЕДЕЛЕНИЕ

**Маршалинг** — процесс преобразования структуры данных в памяти в формат, пригодный для хранения или передачи по сети. В контексте веб-API маршалинг относится к процессу преобразования объекта в структуру данных, которая может быть сериализована в выбранный тип содержимого, например XML или JSON, с явным отображением атрибутов объекта (рис. 2.7).

**Объект Python (несериализуемый напрямую)**

Маршалинг

Демаршалинг

**Словарь Python (может быть сериализован)**

```
order = {
    'id': 'ff0f1355-e821-4178-9567-550dec27a373',
    'status': 'completed',
    'created': 1740493805,
    'order': [
        {
            'product': 'cappuccino',
            'size': 'medium',
            'quantity': 1
        }
    ]
}
```

Сериализация

Десериализация

**JSON-документ (сериализованные данные)**

```
{
  "id": "ff0f1355-e821-4178-9567-550dec27a373",
  "status": "completed",
  "created": 1740493805,
  "order": [{
    "product": "cappuccino",
    "size": "medium",
    "quantity": 1
  }]
}
```

**Рис. 2.7.** Чтобы создать полезную нагрузку HTTP-ответа из объекта Python, мы сначала маршализуем объект в сериализуемую структуру данных с явным отображением атрибутов между объектом и новой структурой. Десериализация полезной нагрузки возвращает нам объект, идентичный тому, который мы сериализовали

Спецификация API сервиса заказов содержит три схемы: `CreateOrderSchema`, `GetOrderSchema` и `OrderItemSchema`. Проанализируем их, чтобы убедиться, что мы понимаем, как нам нужно реализовать наши модели валидации (листинг 2.5).

**Листинг 2.5.** Спецификация схем API сервиса заказов

```
# file: oas.yaml
```

```
components:
```

```
  schemas:
```

```
    OrderItemSchema:
```

```
      type: object
```

```
      required:
```

- product
- size

```
      properties:
```

```
        product:
```

```
          type: string
```

```
        size:
```

```
          type: string
```

```
          enum:
```

- small
- medium
- big

```
        quantity:
```

```
          type: integer
```

```
          default: 1
```

```
          minimum: 1
```

Каждая схема имеет тип, который в данном случае является объектом

Перечисляем обязательные свойства по ключевому слову `required`

Перечисляем свойства объекта по ключевому слову `properties`

Ограничиваем значения свойства с помощью перечисления

Атрибуты могут иметь значение по умолчанию

Мы также можем указать минимальное значение для свойства

```
    CreateOrderSchema:
```

```
      type: object
```

```
      required:
```

- order

```
      properties:
```

```
        order:
```

```
          type: array
```

```
          items:
```

```
            $ref: '#/components/schemas/OrderItemSchema'
```

Указываем тип элементов в массиве, используя ключевое слово `items`

Используем указатель JSON для ссылки на другую схему в том же документе

```
    GetOrderSchema:
```

```
      type: object
```

```
      required:
```

- order
- id
- created
- status

```
      properties:
```

```
        id:
```

```
          type: string
```

```
          format: uuid
```

```
        created:
```

```
          type: string
```

```
          format: date-time
```

```
        status:
```

```
          type: string
```

```
enum:
  - created
  - progress
  - cancelled
  - dispatched
  - delivered
order:
  type: array
  items:
    $ref: '#/components/schemas/OrderItemSchema'
```

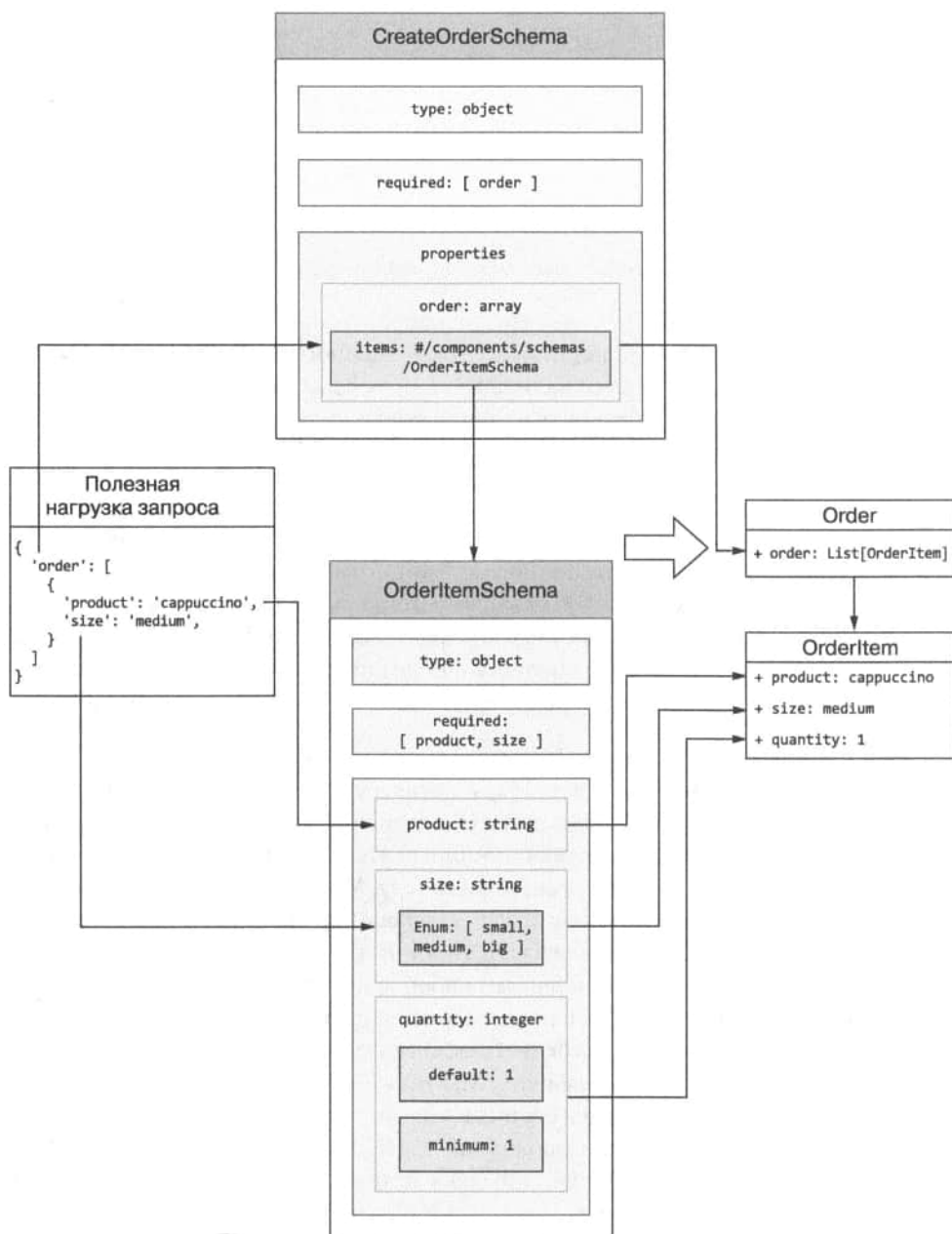
Мы используем `GetOrderSchema`, когда возвращаем детали заказа с сервера, и `CreateOrderSchema` для валидации заказа, размещенного покупателем. На рис. 2.8 показано, как работает процесс валидации данных для `CreateOrderSchema`. Как видите, `CreateOrderSchema` требует наличия только одного свойства в полезной нагрузке: свойства `order`, представляющего собой массив объектов, спецификация которых определяется `OrderItemSchema`.

`OrderItemSchema` имеет два обязательных свойства, `product` и `size`, и одно необязательное свойство, `quantity`, которое по умолчанию имеет значение 1. Это означает, что при обработке полезной нагрузки запроса мы должны проверить, что в ней есть свойства `product` и `size` и что они имеют правильный тип. На рис. 2.8 показано, что происходит, когда свойства `quantity` в полезной нагрузке нет. В этом случае мы устанавливаем для него на сервере приложения значение по умолчанию 1.

Теперь, разобравшись в схемах API, можем приступить к их реализации. Создайте новый файл с именем `orders/api/schemas.py`. В нем будут храниться наши модели pydantic. В листинге 2.6 показано, как мы реализуем `CreateOrderSchema`, `GetOrderSchema` и `OrderItemSchema` с помощью этой библиотеки. Код в листинге 2.6 находится в модуле `orders/api/schemas.py`. Мы определяем каждую схему как класс, который наследуется от класса `BaseModel` из pydantic, и указываем тип каждого атрибута, используя подсказки типов Python<sup>1</sup>. Для атрибутов, которые могут принимать только ограниченный набор значений, мы определяем класс-перечисление. В данном случае мы делаем это для свойств `size` и `status`. Устанавливаем тип свойства `quantity` из `OrderItemSchema` как `conint` от pydantic, который обеспечивает целочисленные значения. Мы также указываем, что `quantity` является необязательным свойством и его значения должны быть равны или больше 1, и присваиваем ему значение по умолчанию 1. Наконец, используем тип `conlist` из pydantic, чтобы определить свойство `order` из `CreateOrderSchema` как список по крайней мере с одним элементом.

---

<sup>1</sup> Подсказки типов (type hints) – система типов, которая позволяет разработчикам указывать типы переменных, не обеспечивая их проверку. Впервые предложена в 2015 году основателем языка Python Гвидо ван Россумом. Описана в PEP 484 (<https://peps.python.org/pep-0484/>). – *Примеч. ред.*



**Рис. 2.8.** Процесс валидации данных для полезной нагрузки запроса по модели **CreateOrderSchema**. На диаграмме показано, как каждое свойство полезной нагрузки запроса проверяется на соответствие свойствам, определенным в схеме, и как мы создаем объект на основе полученной валидации

**Листинг 2.6.** Реализация моделей валидации с помощью pydantic

```
# file: orders/api/schemas.py
```

```
from enum import Enum
from typing import List
from uuid import UUID
```

```
from pydantic import BaseModel, Field, conlist, conint
```

```
class Size(Enum):
    small = 'small'
    medium = 'medium'
    big = 'big'
```

← Объявляем схему  
перечисления

```
class Status(Enum):
    created = 'created'
    progress = 'progress'
    cancelled = 'cancelled'
    dispatched = 'dispatched'
    delivered = 'delivered'
```

```
class OrderItemSchema(BaseModel):
    product: str
    size: Size
    quantity: Optional[conint(ge=1, strict=True)] = 1
```

← Каждая модель pydantic  
наследуется от базовой модели

← Используем подсказки типа Python,  
чтобы указать тип атрибута

← Ограничиваем значения свойства, устанавливая  
для его типа значение перечисления

```
class CreateOrderSchema(BaseModel):
    order: conlist(OrderItemSchema, min_items=1)
```

← Указываем минимальное  
значение количества  
и задаем его по умолчанию

```
class GetOrderSchema(CreateOrderSchema):
    id: UUID
    created: datetime
    status: Status
```

← Используем тип conlist  
от pydantic для определения  
списка, содержащего  
хотя бы один элемент

```
class GetOrdersSchema(BaseModel):
    orders: List[GetOrderSchema]
```

Теперь, когда наши модели валидации реализованы, свяжем их с API для валидации и маршалинга полезной нагрузки.

## 2.5. ВАЛИДАЦИЯ ПОЛЕЗНОЙ НАГРУЗКИ ЗАПРОСА С ПОМОЩЬЮ PYDANTIC

В этом разделе мы используем модели, созданные в разделе 2.4, для валидации полезной нагрузки запроса. Как мы получаем доступ к полезной нагрузке запроса в функциях представления? Мы перехватываем ее, объявляя параметром функции представления, а для валидации устанавливаем ее тип в соответствующую модель pydantic (листинг 2.7).

**Листинг 2.7.** Подключение моделей валидации к эндпоинтам API

```
# file: orders/api/api.py

from uuid import UUID

from starlette.responses import Response
from starlette import status

from orders.app import app
from orders.api.schemas import CreateOrderSchema

...

@app.post('/orders', status_code=status.HTTP_201_CREATED)
def create_order(order_details: CreateOrderSchema):
    return order

@app.get('/orders/{order_id}')
def get_order(order_id: UUID):
    return order

@app.put('/orders/{order_id}')
def update_order(order_id: UUID, order_details: CreateOrderSchema):
    return order

...
```

Импортируем модели pydantic, чтобы использовать их для валидации

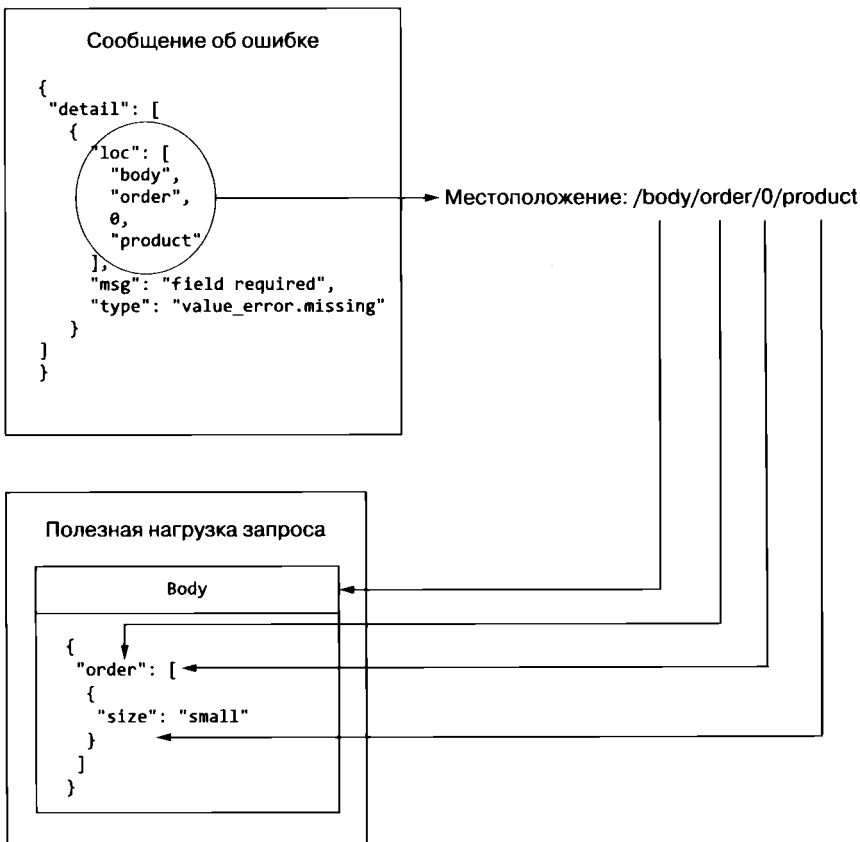
Перехватываем полезную нагрузку, объявляя ее параметром в нашей функции, и используем подсказки типов для ее валидации

Если ваше приложение остается запущенным, в режиме горячей перезагрузки изменения загружаются сервером автоматически, поэтому нужно просто обновить браузер, чтобы обновился и пользовательский интерфейс. Если вы выполните запрос POST URL-адреса `/orders`, то увидите пример полезной нагрузки, ожидаемой сервером. Теперь, если вы попытаетесь отредактировать полезную нагрузку, чтобы удалить любое из обязательных полей, например `product`, и отправите ее на сервер, вы получите следующее сообщение об ошибке:

```
{
  "detail": [
    {
      "loc": [
        "body",
        "order",
        0,
        "product"
      ],
      "msg": "field required",
      "type": "value_error.missing"
    }
  ]
}
```



FastAPI генерирует сообщение об ошибке, указывающее, где в полезной нагрузке запроса обнаружена ошибка. В этом сообщении используется указатель JSON для идентификации проблемы. Указатель JSON — это синтаксис, который позволяет представить путь к определенному значению в документе JSON. Если вы впервые сталкиваетесь с указателями JSON, думайте о них как о другом варианте представления синтаксиса словаря и индексной записи в Python. Например, сообщение об ошибке `"loc: /body/order/0/product"` эквивалентно следующей нотации в Python: `loc['body'] ['order'] [0] ['product']`. На рис. 2.9 показано, как интерпретировать указатель JSON из сообщения об ошибке, чтобы определить источник проблемы в полезной нагрузке.



**Рис. 2.9.** Когда запрос завершается неудачей из-за неправильно сформированной полезной нагрузки, мы получаем ответ с сообщением об ошибке. В нем используется указатель JSON, уведомляющий нас, где находится ошибка. В данном случае в сообщении говорится, что в полезной нагрузке отсутствует свойство `/body/order/0/product`

Вы также можете изменить полезную нагрузку так, чтобы она содержала недопустимое значение для свойства `size`:

```
{
  "order": [
    {
      "product": "string",
      "size": "somethingelse"
    }
  ]
}
```

В этом случае вы также получите информативное сообщение о такой ошибке: "значение не является допустимым элементом перечисления; допустимые значения: 'small', 'medium', 'big'". Что произойдет, если мы сделаем опечатку в полезной нагрузке? Представьте, что клиент отправил на сервер следующее:

```
{
  "order": [
    {
      "product": "string",
      "size": "small",
      "quantit": 5
    }
  ]
}
```

В этом случае FastAPI предполагает, что свойство `quantity` отсутствует и клиент хочет установить его значение равным 1. Это может привести к путанице между клиентом и сервером, и в таких случаях аннулирование полезной нагрузки с недопустимыми свойствами помогает сделать интеграцию с помощью API более надежной. В главе 6 вы научитесь справляться с такими ситуациями.

Крайний случай с необязательными свойствами, такими как `quantity` в `OrderItemSchema`, заключается в том, что rудantic предполагает, что значения этих свойств могут быть `null`, и поэтому будет принимать полезную нагрузку с `quantity`, равным `null`. Например, если мы отправим следующую полезную нагрузку в эндпоинт `POST /orders`, наш сервер примет ее:

```
{
  "order": [
    {
      "product": "string",
      "size": "small",
      "quantity": null
    }
  ]
}
```

С точки зрения API-интеграции необязательное — это не совсем то же самое, что допускающее значение `null`: свойство может быть необязательным, потому что

имеет значение по умолчанию, но это не значит, что оно может быть равным `null`. Чтобы обеспечить правильное поведение `pydantic`, следует включить дополнительное правило валидации, которое не позволит пользователям установить `quantity` равным `null`. Воспользуемся декоратором `validator()` от `pydantic` для определения дополнительных правил валидации для наших моделей (листинг 2.8).

**Листинг 2.8.** Включение дополнительных правил валидации для моделей `pydantic`

```
# file: orders/api/schemas.py

from datetime import datetime
from enum import Enum
from typing import List, Optional
from uuid import UUID

from pydantic import BaseModel, conint, validator

...

class OrderItemSchema(BaseModel):
    product: str
    size: Size
    quantity: Optional[conint(ge=1, strict=True)] = 1

    @validator('quantity')
    def quantity_non_nullable(cls, value):
        assert value is not None, 'quantity may not be None'
        return value

...
```

Теперь, когда мы знаем, как тестировать реализацию нашего API с помощью Swagger UI, посмотрим, как использовать `pydantic` для валидации и сериализации ответов API.

## 2.6. МАРШАЛИНГ И ВАЛИДАЦИЯ ПОЛЕЗНОЙ НАГРУЗКИ ОТВЕТА С ПОМОЩЬЮ PYDANTIC

В этом разделе мы будем использовать модели `pydantic`, созданные в разделе 2.4, для маршалинга и валидации полезной нагрузки ответов нашего API. Неправильно сформированная полезная нагрузка становится одной из наиболее распространенных причин сбоев в API-интеграции, поэтому этот шаг крайне важен для обеспечения надежности API. Например, схемой для полезной нагрузки ответа эндпоинта `POST /orders` является `GetOrderSchema`, и она требует наличия полей `id`, `created`, `status` и `order`. Клиенты API ожидают наличия всех этих полей в ответе и будут выдавать ошибки, если какое-либо из них отсутствует или имеет неправильный тип либо формат.

**ПРИМЕЧАНИЕ**

Неправильно сформированные ответы являются распространенным источником ошибок интеграции с помощью API. Вы можете избежать этой проблемы, проверяя полезную нагрузку ответа до того, как она будет отправлена с сервера. В FastAPI это легко сделать, задав параметр `response_model` в декораторе маршрута.

В листинге 2.9 показано, как модели `pydantic` используются для валидации ответов от эндпоинтов `GET /orders` и `POST /orders`. Мы установили параметр `response_model` на `pydantic`-модель в декораторах маршрута FastAPI. Это сделано для валидации ответов от всех эндпоинтов, кроме `DELETE /orders/{order_id}`, который возвращает пустой ответ. Не ленитесь посмотреть полный код в репозитории GitHub для понимания этой реализации.

**Листинг 2.9.** Подключение моделей валидации для ответов в эндпоинтах API

```
# file: orders/api/api.py

from uuid import UUID

from starlette.responses import Response
from starlette import status

from orders.app import app
from orders.api.schemas import (
    GetOrderSchema,
    CreateOrderSchema,
    GetOrdersSchema,
)

...

@app.get('/orders', response_model=GetOrdersSchema)
def get_orders():
    return [
        order
    ]

@app.post(
    '/orders',
    status_code=status.HTTP_201_CREATED,
    response_model=GetOrderSchema,
)

def create_order(order_details: CreateOrderSchema):
    return order
```

Теперь, когда у нас есть модели ответов, FastAPI будет выдавать ошибку, если требуемое свойство отсутствует в полезной нагрузке ответа. Он также удалит все свойства, которые не являются частью схемы, и попытается привести каждое свойство к нужному типу. Посмотрим на это поведение в действии.

В адресной строке браузера введите `http://127.0.0.1:8000/docs`, чтобы загрузить Swagger UI для API. Затем перейдите к эндпоинту `GET /orders` и отправьте запрос. Вы получите заказ, который мы жестко закодировали в начале файла `orders/api/api.py`. Внесем некоторые изменения в эту полезную нагрузку, чтобы посмотреть, как FastAPI их обрабатывает. Для начала добавим дополнительное свойство `updated`:

```
# orders/api/api.py
...

order = {
    'id': 'ff0f1355-e821-4178-9567-550dec27a373',
    'status': 'delivered',
    'created': datetime.utcnow(),
    'updated': datetime.utcnow(),
    'order': [
        {
            'product': 'cappuccino',
            'size': 'medium',
            'quantity': 1
        }
    ]
}

...
```

Если мы снова выполним запрос `GET /orders`, то получим тот же ответ, что и раньше, но без свойства `updated`, поскольку оно не является частью модели `GetOrderSchema`:

```
[
  {
    "order": [
      {
        "product": "cappuccino",
        "size": "medium",
        "quantity": 1
      }
    ],
    "id": "ff0f1355-e821-4178-9567-550dec27a373",
    "created": datetime.utcnow(),
    "status": "delivered"
  }
]
```

Теперь удалим свойство `created` из полезной нагрузки `order` и снова выполним запрос `GET /orders`:

```
# orders/api/api.py
...

order = {
    'id': 'ff0f1355-e821-4178-9567-550dec27a373',
```

```

    'status': "delivered",
    'updated': datetime.utcnow(),
    'order': [
        {
            'product': 'cappuccino',
            'size': 'medium',
            'quantity': 1
        }
    ]
}

```

На этот раз FastAPI выдает ошибку сервера, сообщая нам, что необходимое свойство `created` в полезной нагрузке отсутствует:

```

pydantic.error_wrappers.ValidationError: 1 validation error for GetOrderSchema
response -> 0 -> created
    field required (type=value_error.missing)

```

Теперь зададим в качестве значения свойства `created` случайную строку и выполним еще один запрос `GET /orders`:

```

# orders/api/api.py
...

order = {
    'id': 'ff0f1355-e821-4178-9567-550dec27a373',
    'status': "delivered",
    'created': 'asdf',
    'updated': 1740493905,
    'order': [
        {
            'product': 'cappuccino',
            'size': 'medium',
            'quantity': 1
        }
    ]
}

...

```

В этом случае FastAPI выдает полезную ошибку:

```

pydantic.error_wrappers.ValidationError:
    1 validation error for GetOrderSchema
response -> 0 -> created
    value is not a valid integer (type=type_error.integer)

```

Наши ответы корректно проверяются и маршализуются. Добавим простой механизм управления состоянием приложения, чтобы размещать заказы и изменять их состояние через API.

## 2.7. ДОБАВЛЕНИЕ В API СОХРАНЕННОГО В ПАМЯТИ СПИСКА ЗАКАЗОВ

До сих пор наш API возвращал один и тот же объект. Давайте изменим это, добавив простой набор заказов в память для управления состоянием приложения. Чтобы упростить реализацию, представим набор заказов в виде списка на языке Python. Мы будем управлять им с помощью функций представления на уровне API. В главе 7 вы познакомитесь с полезными паттернами, позволяющими добавить в приложение надежный контроллер и уровень хранения данных.

В листинге 2.10 показаны изменения в файле `orders/api/api.py`, необходимые для управления сохраненным в оперативной памяти списком заказов в наших функциях представления. Мы создаем коллекцию заказов в виде списка Python и присваиваем его переменной `ORDERS`. Для простоты храним информацию о каждом заказе в словаре и обновляем, изменяя свойства в словаре.

**Листинг 2.10.** Управление состоянием приложения с помощью списка в оперативной памяти

```
# file: orders/api/api.py
```

```
import time
import uuid
```

```
from datetime import datetime
from uuid import UUID
```

```
from fastapi import HTTPException
from starlette.responses import Response
from starlette import status
```

```
from orders.app import app
from orders.api.schemas import GetOrderSchema, CreateOrderSchema
```

```
ORDERS = []
```

← Представляем наш список заказов  
в оперативной памяти в виде списка Python

```
@app.get('/orders', response_model=GetOrdersSchema)
```

```
def get_orders():
    return ORDERS
```

← Чтобы вернуть список заказов,  
мы просто возвращаем список ORDERS

```
@app.post(
    '/orders',
    status_code=status.HTTP_201_CREATED,
    response_model=GetOrderSchema,
)
```

Преобразуем  
каждый заказ  
в словарь

```
def create_order(order_details: CreateOrderSchema):
    order = order_details.dict()
    order['id'] = uuid.uuid4()
    order['created'] = datetime.utcnow()
```

Дополняем объект order  
атрибутами для запуска  
на сервере, такими как ID

```

order['status'] = 'created'
ORDERS.append(order)  ← Чтобы создать заказ, добавляем его в список
return order          ← После добавления заказа в список возвращаем его

```

```

@app.get('/orders/{order_id}', response_model=GetOrderSchema)
def get_order(order_id: UUID):
    for order in ORDERS:
        if order['id'] == order_id:
            return order
    raise HTTPException(
        status_code=404, detail=f'Order with ID {order_id} not found'
    )

```

← Чтобы найти заказ по идентификатору, просматриваем список заказов и проверяем их идентификаторы

← Если заказ не найден, вызываем HTTPException с кодом status\_code, равным 404, чтобы вернуть ответ 404

```

@app.put('/orders/{order_id}', response_model=GetOrderSchema)
def update_order(order_id: UUID, order_details: CreateOrderSchema):
    for order in ORDERS:
        if order['id'] == order_id:
            order.update(order_details.dict())
            return order
    raise HTTPException(
        status_code=404, detail=f'Order with ID {order_id} not found'
    )

```

```

@app.delete(
    '/orders/{order_id}',
    status_code=status.HTTP_204_NO_CONTENT,
    response_class=Response,
)
def delete_order(order_id: UUID):
    for index, order in enumerate(ORDERS):
        if order['id'] == order_id:
            ORDERS.pop(index)
            return Response(status_code=HTTPStatus.NO_CONTENT.value)
    raise HTTPException(
        status_code=404, detail=f'Order with ID {order_id} not found'
    )

```

← Получаем заказы из списка, используя метод list.pop()

```

@app.post('/orders/{order_id}/cancel', response_model=GetOrderSchema)
def cancel_order(order_id: UUID):
    for order in ORDERS:
        if order['id'] == order_id:
            order['status'] = 'cancelled'
            return order
    raise HTTPException(
        status_code=404, detail=f'Order with ID {order_id} not found'
    )

```

```

@app.post('/orders/{order_id}/pay', response_model=GetOrderSchema)
def pay_order(order_id: UUID):
    for order in ORDERS:
        if order['id'] == order_id:
            order['status'] = 'progress'
            return order

```



```
raise HTTPException(  
    status_code=404, detail=f'Order with ID {order_id} not found'  
)
```

Если вы поэкспериментируете с эндпоинтом `POST /orders`, то сможете создавать новые заказы, а используя их идентификаторы, сможете обновлять их взаимодействием с эндпоинтом `PUT /orders/{order_id}`. В каждом эндпоинте по пути `/orders/{order_id}` мы проверяем, существует ли заказ, запрошенный клиентом API, и если нет, то возвращаем ответ 404 (Not Found) с информативным сообщением.

Теперь мы можем использовать наш API для создания заказов, их обновления, оплаты, отмены и получения подробной информации о них. Вы реализовали полностью рабочий веб-API для микросервисного приложения! Параллельно познакомились с кучей новых библиотек для создания веб-API и увидели, как внедрить надежную проверку данных в свои API. Вы также научились собирать все это вместе и успешно запускать. Надеюсь, эта глава вызвала у вас интерес к проектированию и разработке микросервисов, предоставляющих веб-API. В следующих главах мы углубимся в эти темы, и вы научитесь создавать и предоставлять надежные и безопасные API-интеграции микросервисов.

## РЕЗЮМЕ

- Для разделения микросервисов на модульные уровни мы используем адаптацию паттерна трехуровневой архитектуры и получаем:
  - уровень данных, который знает, как взаимодействовать с источником данных;
  - уровень бизнес-логики, реализующий возможности сервиса;
  - интерфейс, или уровень представления, который раскрывает возможности сервиса через API.
- FastAPI — это популярный фреймворк для разработки веб-API. Он обладает высокой производительностью и богатой экосистемой библиотек, облегчающих разработку API.
- FastAPI использует `pydantic`, популярную библиотеку валидации данных для Python. `Pydantic` предоставляет подсказки типов для создания правил валидации, что позволяет реализовать чистые и простые для понимания модели.
- FastAPI динамически генерирует пользовательский интерфейс Swagger на основе нашего кода. `Swagger UI` — это популярный интерфейс интерактивной визуализации для API. Используя `Swagger UI`, можно легко проверить правильность своей реализации.

# 3

## Проектирование микросервисов

---

### В этой главе

- ✓ Принципы проектирования микросервисов.
- ✓ Декомпозиция сервисов по бизнес-функциям.
- ✓ Декомпозиция сервисов по поддоменам.

Когда мы разрабатываем систему микросервисов, сразу сталкиваемся с такими вопросами: как разбить систему на микросервисы, как определить, где заканчивается один сервис и начинается другой? Другими словами, как определить границы между микросервисами? В этой главе вы получите ответы на эти вопросы и научитесь оценивать качество архитектуры микросервисов, опираясь на принципы проектирования.

Процесс разбиения системы на микросервисы называется *декомпозицией сервисов*. Декомпозиция — это важнейший шаг в проектировании микросервисов, поскольку она помогает создать приложения с конкретными границами, четко определенными областями применения и обязанностями. Хорошо продуманная архитектура микросервисов позволит снизить риск возникновения так называемого распределенного монолита. В этой главе вы изучите две стратегии декомпозиции сервисов: по бизнес-функциям и поддоменам (модулям). Вы увидите, как работают эти методы, и на практическом примере научитесь их применять. Прежде чем мы углубимся в стратегии декомпозиции сервисов, представлю проект, над которым мы будем работать в книге: CoffeeMesh.

## 3.1. ПРЕДСТАВЛЯЕМ COFFEEMESH

CoffeeMesh — вымышленная компания, которая позволяет покупателям заказывать кофе, а также другие напитки и выпечку. У CoffeeMesh одна задача — готовить и доставлять лучший в мире кофе своим покупателям независимо от того, где они находятся и когда делают заказ. У компании обширная сеть подразделений, которая охватывает несколько стран. Приготовление кофе полностью автоматизировано, а доставка осуществляется парком беспилотных дронов, работающих в режиме 24/7.

Когда покупатель делает заказ на сайте CoffeeMesh, заказанная продукция изготавливается по требованию. Алгоритм определяет, какой филиал является наиболее подходящим местом для подготовки каждого заказа, основываясь на имеющихся запасах, количестве имеющихся в работе заказов и расстоянии до покупателя. Как только товар готов, он сразу же отправляется заказчику. Миссия CoffeeMesh заключается в том, чтобы покупатель получал свой заказ свежим и горячим.

Теперь, когда у нас есть пример для работы, посмотрим, как спроектировать архитектуру микросервисов для платформы CoffeeMesh. Прежде чем мы научимся применять стратегии декомпозиции сервисов, обсудим три принципа, которыми будем руководствоваться при проектировании.

## 3.2. ПРИНЦИПЫ ПРОЕКТИРОВАНИЯ МИКРОСЕРВИСОВ

Какой микросервис считается хорошо спроектированным? Как мы установили в главе 1, микросервисы разрабатываются вокруг четко определенных бизнес-задач, имеют четко определенные границы приложения и взаимодействуют друг с другом через облегченные протоколы. Что это означает на практике? В этом разделе мы рассмотрим три принципа проектирования, которые помогут проверить, правильно ли спроектированы наши микросервисы.

- Принцип «База данных для каждого сервиса».
- Принцип слабой связанности.
- Принцип единственной ответственности (SRP).

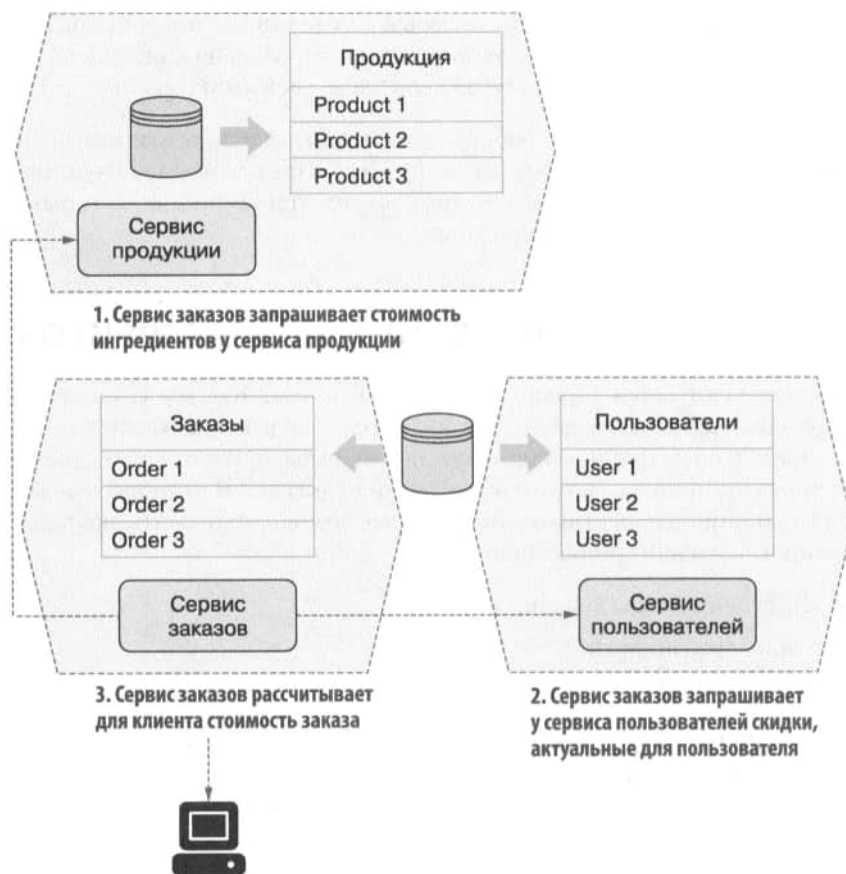
Следование этим принципам поможет вам избежать создания распределенного монолита. В следующих разделах мы оцениваем наш архитектурный проект по этим принципам, и они помогают нам выявить ошибки в проекте.

### 3.2.1. Принцип «База данных для каждого сервиса»

Принцип «База данных для каждого сервиса» гласит, что каждый микросервис владеет определенным набором данных и никакой другой сервис не должен иметь к ним доступа, кроме как через API. Но пусть название принципа не вводит вас в заблуждение: оно не означает, что все микросервисы должны быть подключены

к совершенно разным базам данных. Это могут быть разные таблицы в базе данных SQL или разные коллекции в базе данных NoSQL. Смысл принципа в том, чтобы гарантировать, что данные, принадлежащие одному сервису, не будут напрямую доступны другому сервису.

На рис. 3.1 показано, как микросервисы обмениваются данными. Сервис заказов рассчитывает стоимость заказа покупателя. Для этого сервису нужна стоимость каждого блюда в заказе, которая доступна в базе данных *Products*. Ему также необходимо знать, есть ли у пользователя действующая скидка, которую можно проверить в базе данных *Users*. Однако вместо того, чтобы обращаться к обеим базам напрямую, сервис заказов запрашивает эти данные у сервисов продукции и пользователей.



**Рис. 3.1.** Каждый микросервис имеет собственную базу данных, а доступ к данным другого сервиса осуществляется через API

Почему этот принцип важен? Инкапсуляция доступа к данным в рамках сервиса дает нам возможность проектировать модели данных с оптимальным доступом

к сервису. Это также позволяет нам вносить изменения в базу данных, не нарушая кода другого сервиса. Если бы сервис заказов на рис. 3.1 имел прямой доступ к базе данных *Products*, то при изменении ее схемы пришлось бы обновлять как сервис продукции, так и сервис заказов. Мы связали бы код сервиса заказов с базой данных *Products*, а значит, нарушили бы принцип слабой связанности, который обсудим в следующем подразделе.

### 3.2.2. Принцип слабой связанности

Принцип слабой связанности предполагает, что мы будем проектировать сервисы с четким разделением задач. Слабо связанные сервисы не зависят от деталей реализации другого сервиса. Что это означает на практике? Принцип имеет два практических значения.

- Каждый сервис должен работать независимо от остальных. Если у нас есть сервис, который не может выполнить ни одного запроса без вызова другого сервиса, то нет четкого разделения проблемных областей между ними обоими и они связаны друг с другом.
- Каждый сервис можно обновлять, не затрагивая другие сервисы. Если изменения в каком-либо сервисе требуют обновления других сервисов, мы имеем сильную связанность между ними, поэтому их необходимо перепроектировать.

На рис. 3.2 показан сервис прогнозирования продаж, который знает, как рассчитывать прогноз на основе данных за прошедший период. Здесь также показан сервис прошлых данных, который владеет сведениями о продажах. Чтобы составить прогноз, сервис прогнозирования продаж выполняет вызов API сервиса прошлых данных для получения информации за прошедший период. Этот сервис прогнозирования продаж не может обработать ни один запрос без обращения к сервису прошлых данных, поэтому между обоими существует сильная связанность. Решение состоит в том, чтобы перепроектировать оба сервиса так, чтобы они не зависели друг от друга, или объединить их.



**Рис. 3.2.** Когда сервис не может обработать запрос без вызова другого сервиса, мы говорим, что оба сервиса сильно связаны

### 3.2.3. Принцип единственной ответственности

SRP<sup>1</sup> утверждает, что следует проектировать компоненты с небольшим количеством ответственностей, а в идеале — только с одной. В отношении архитектуры микросервисов это означает, что мы должны стремиться проектировать сервисы вокруг одной бизнес-функции или поддомена. В следующих разделах вы узнаете, как распределить сервисы по бизнес-функциям и поддоменам. Если будете следовать любому из этих методов, то сможете проектировать микросервисы в соответствии с SRP.

## 3.3. ДЕКОМПОЗИЦИЯ СЕРВИСОВ ПО БИЗНЕС-ФУНКЦИЯМ

При декомпозиции по бизнес-функциям мы рассматриваем виды деятельности компании и то, как она организуется для их осуществления. Затем мы разрабатываем микросервисы, которые отражают эту организационную структуру. Например, если в компании есть команда управления покупателями, мы создаем сервис управления покупателями; если в компании есть команда управления претензиями, мы создаем сервис управления претензиями; для команды кухни мы создаем соответствующий сервис кухни и т. д. Для предприятий, которые занимаются изготовлением продуктов, у нас может быть микросервис для каждого продукта. Например, компания, производящая корм для домашних животных, может иметь команду, занимающуюся кормом для собак, другую команду, занимающуюся кормом для кошек, третью — кормом для черепах и т. д. При таком сценарии мы создаем микросервисы для каждой из этих команд.

Как видно на рис. 3.3, декомпозиция по бизнес-функциям обычно приводит к архитектуре, которая сопоставляет каждую бизнес-команду с микросервисом. Посмотрим, как применить этот подход к платформе CoffeeMesh.



**Рис. 3.3.** Используя декомпозицию сервисов по бизнес-функциям, мы отражаем структуру бизнеса в нашей архитектуре микросервисов

<sup>1</sup> О принципе единственной ответственности можно узнать в книге: *Мартин Р.* Чистая архитектура. — СПб.: Питер, 2022. — С. 78.

### 3.3.1. Анализ бизнес-структуры CoffeeMesh

Чтобы применить декомпозицию по бизнес-функциям, необходимо проанализировать структуру и организацию бизнеса. Проведем такой анализ для CoffeeMesh. Через сайт CoffeeMesh покупатели могут заказывать различные виды товаров из каталога, управляемого командой по продукции, которая отвечает за создание новых товаров. Доступность продукции и ингредиентов зависит от запасов в CoffeeMesh на момент заказа, за чем следит команда инвентаризации.

Отдел продаж занимается улучшением опыта заказа продукции через сайт CoffeeMesh. Их цель — максимизировать продажи и сделать все возможное, чтобы покупатели были довольны своим опытом и хотели вернуться. Команда по платежам следит за тем, чтобы компания была прибыльной, и заботится о финансовой инфраструктуре, необходимой для обработки платежей покупателей и возврата их денег при отмене заказа.

Как только пользователь размещает заказ, кухня получает детали, чтобы приступить к приготовлению. Работа кухни полностью автоматизирована, а специальная команда инженеров и поваров, называемая командой кухни, следит за работой кухни, чтобы не допустить сбоев в процессе производства. Когда заказ готов к доставке, беспилотник забирает его и доставляет заказчику. Специальная команда инженеров, называемая командой доставки, контролирует этот процесс, чтобы обеспечить безупречный процесс доставки.

На этом завершим анализ организационной структуры CoffeeMesh и приступим к разработке соответствующей архитектуры микросервисов.

### 3.3.2. Декомпозиция микросервисов по функциям

Для декомпозиции сервисов по бизнес-функциям мы сопоставляем каждую бизнес-команду с микросервисом. На основе анализа, проведенного в предыдущем подразделе, мы можем сопоставить следующие бизнес-команды с микросервисами.

- *Команда по продукции сопоставляется с сервисом продукции* — этот сервис владеет данными каталога продукции CoffeeMesh. Разработчики используют этот сервис для ведения каталога CoffeeMesh, добавляя новые товары или обновляя существующие через интерфейс сервиса.
- *Команда по ингредиентам сопоставляется с сервисом ингредиентов* — он владеет данными о запасах ингредиентов CoffeeMesh. Разработчики используют этот сервис для синхронизации базы данных ингредиентов со складами CoffeeMesh.
- *Команда по продажам сопоставляется с сервисом продаж* — этот сервис проводит покупателей по пути размещения заказов и отслеживает их выполнение. Команда владеет данными о заказах покупателей и управляет жизненным

циклом каждого заказа. Разработчики собирают данные из этого сервиса для анализа и улучшения взаимодействия с покупателями.

- *Команда по платежам сопоставляется с сервисом платежей* — он реализует платежную систему и владеет данными о реквизитах пользователей и истории платежей. Команда использует этот сервис для поддержания актуальности счетов компании и обеспечения корректной оплаты.
- *Команда кухни сопоставляется с сервисом кухни* — этот сервис отправляет заказы в автоматизированную кухонную систему и следит за их выполнением. Команда владеет данными о заказах, выполненных на кухне, собирает данные из этого сервиса для мониторинга производительности кухонной системы.
- *Команда по доставке сопоставляется с сервисом доставки* — он организует доставку заказа покупателю после того, как тот приготовлен на кухне. Этот сервис умеет преобразовать местоположение пользователя в координаты и рассчитать наилучший маршрут к месту назначения. Он владеет данными о каждой доставке, выполненной CoffeeMesh. Команда по доставке собирает данные из этого сервиса для мониторинга производительности автоматизированной системы доставки.

В этой архитектуре микросервисов мы назвали каждый сервис в честь бизнес-структуры, которую он представляет. Здесь мы сделали это для удобства, но не обязательно делать именно так. Например, сервис платежей можно переименовать в денежный сервис, поскольку все взаимодействия пользователей с этим сервисом будут связаны с их деньгами.

Декомпозиция по бизнес-функциям дает нам архитектуру, в которой каждый сервис соответствует бизнес-команде. Согласуется ли этот результат с принципами проектирования микросервисов, которые мы изучали в разделе 3.2? Давайте посмотрим.

Из предыдущего анализа ясно, что каждый сервис владеет собственными данными: сервис продукции — данными о товарах, сервис ингредиентов — данными об ингредиентах и т. д. SRP также работает, поскольку каждый сервис ограничен одной сферой деятельности: сервис платежей только обрабатывает платежи, сервис доставки только управляет доставками и т. д.

Однако, как видно на рис. 3.4, это решение не удовлетворяет принципу слабой связанности. Чтобы поддерживать актуальность каталога CoffeeMesh, сервис продукции должен определить наличие каждого товара, которое зависит от доступного запаса ингредиентов. Поскольку данные о запасах ингредиентов принадлежат сервису ингредиентов, сервису продукции необходимо выполнить вызов API к нему для каждого товара.

Таким образом, сервисы продукции и ингредиентов будут связаны между собой, и поэтому обе бизнес-функции лучше реализовать в рамках одного сервиса.





**Рис. 3.4.** Чтобы определить, доступен ли товар, сервис продукции проверяет запас ингредиентов в соответствующем сервисе

На рис. 3.5 показана окончательная схема архитектуры микросервисов CoffeeMesh с использованием стратегии декомпозиции по бизнес-функциям.



**Рис. 3.5.** Когда мы декомпозируем сервисы по бизнес-функциям, мы сопоставляем каждую команду с сервисом

Теперь, когда мы знаем, как распределить сервисы по бизнес-функциям, посмотрим, как работает декомпозиция по поддоменам.

### 3.4. ДЕКОМПОЗИЦИЯ СЕРВИСОВ ПО ПОДДОМЕНАМ

Декомпозиция по поддоменам — это стратегия, которая вытекает из *предметно-ориентированного проектирования* (domain-driven design, DDD) — подхода к разработке, который фокусируется на моделировании процессов и потоков бизнеса с помощью программного обеспечения на языке, используемом бизнес-пользователями. Применительно к проектированию системы микросервисов DDD помогает определить основные обязанности и границы каждого сервиса.

#### 3.4.1. Что такое предметно-ориентированное проектирование

DDD — это подход, который фокусируется на моделировании процессов и потоков бизнес-пользователей. Методы DDD были лучше всего описаны Эриком Эвансом в книге *Domain-Driven Design*<sup>1</sup> (Addison-Wesley, 2003), которую еще называют «большой синей книгой». DDD означает подход к разработке ПО, который пытается как можно точнее отразить идеи и язык, используемые предприятиями — конечными пользователями программного обеспечения — для обозначения своих процессов и потоков. Чтобы достичь этого соответствия, разработчикам предлагается создать строгий, основанный на моделях язык, которым они смогут поделиться с конечными пользователями. В таком языке, который называется *единым*, не должно быть двусмысленных значений.

Чтобы создать единый язык, мы должны определить главную бизнес-область, которая соответствует основной деятельности, выполняемой организацией в процессе создания ценности. Для логистической компании это может быть перевозка товаров по всему миру. Для компании, занимающейся электронной торговлей, — продажа товаров. Для платформы социальных сетей — предоставление пользователю релевантного контента. В отношении приложения для знакомств — подбор пользователей. Для CoffeeMesh основной задачей является максимально быстрая доставка отличного кофе покупателям независимо от их местонахождения.

Смыслового ядра<sup>2</sup> (core domain) часто недостаточно для охвата всех областей деятельности в бизнесе, поэтому DDD также различает вспомогательные и общие подобласти (поддомены). *Вспомогательная подобласть* представляет собой сферу бизнеса, которая не связана напрямую с созданием ценности, но является основополагающей для ее поддержки. Для логистической компании это может быть поддержка пользователей, отправляющих свою продукцию, аренда оборудования, управление партнерскими отношениями с другими предприятиями и т. д. Для

<sup>1</sup> Эванс Э. Предметно-ориентированное проектирование (DDD): структуризация сложных программных систем.

<sup>2</sup> Часть домена (предметной области), имеющая первостепенное значение для выполнения главной задачи. — *Примеч. ред.*

компании, занимающейся электронной коммерцией, это может быть маркетинг, поддержка покупателей, складское хранение и т. д.

Смысловое ядро дает определение *проблемного пространства* — задачи, которую вы пытаетесь решить с помощью программного обеспечения. Решение представляет собой модель в виде системы абстракций, которая описывает область и решает задачу. В идеале существует только одна универсальная модель, которая обеспечивает *пространство решения* задачи, с четко определенным единым языком. Однако на практике большинство задач настолько сложны, что требуют совместной работы различных моделей с их собственными едиными языками. Процесс определения таких моделей мы называем *стратегическим проектированием*.

### 3.4.2. Применение стратегического анализа в CoffeeMesh

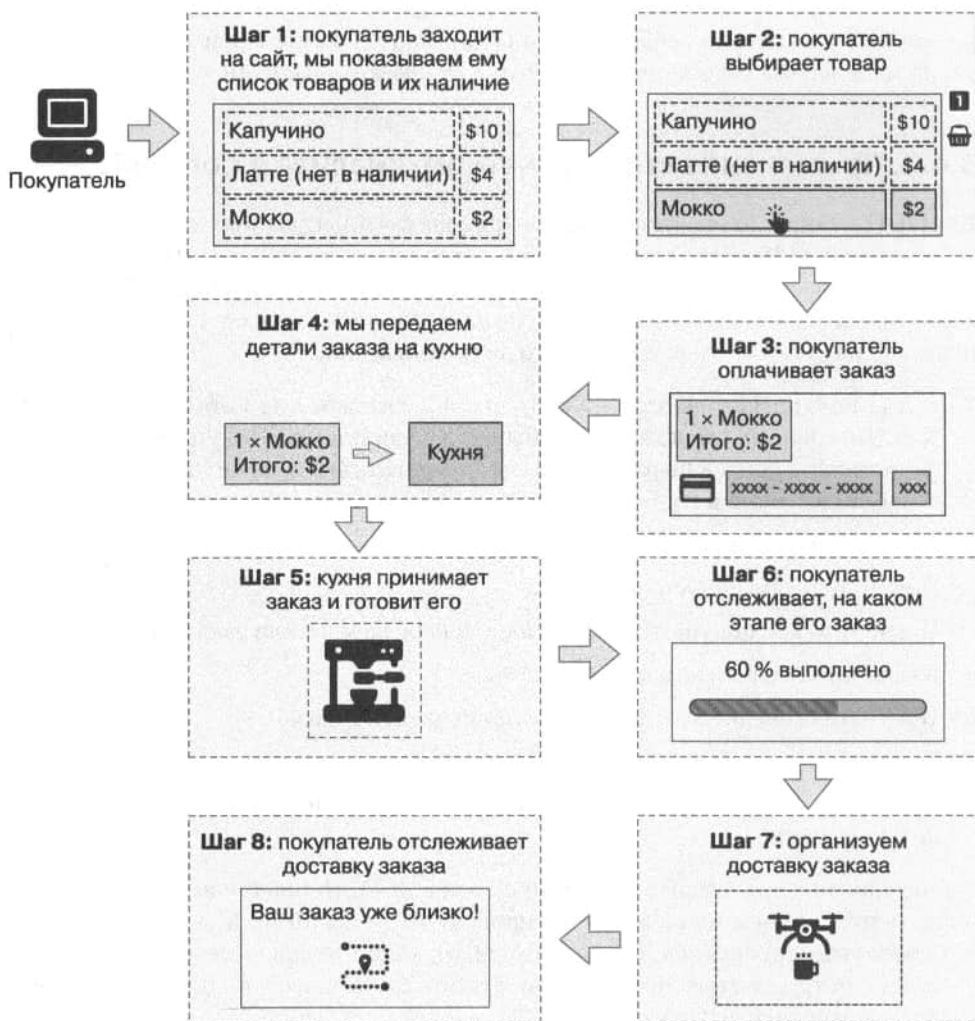
Как DDD работает на практике? Как мы применим его для декомпозиции CoffeeMesh на поддомены? Чтобы разбить систему на поддомены, нужно подумать, какие операции должна выполнить система в процессе достижения цели. В CoffeeMesh мы хотим смоделировать процесс принятия заказа и доставки его покупателю. Как показано на рис. 3.6, мы разбиваем его на восемь этапов.

1. Когда покупатель заходит на сайт, мы показываем ему каталог продукции. Каждый товар отмечен как доступный или недоступный. Покупатель может отфильтровать список по наличию и отсортировать его по цене (от самой низкой до самой высокой и наоборот).
2. Покупатель выбирает товары.
3. Покупатель оплачивает свой заказ.
4. После того как покупатель заплатил, мы передаем детали заказа на кухню.
5. Кухня принимает заказ и готовит его.
6. Покупатель следит за ходом выполнения своего заказа.
7. Как только заказ будет готов, мы организуем его доставку.
8. Покупатель отслеживает маршрут беспилотника до тех пор, пока его заказ не будет доставлен.

Сопоставим каждый шаг с поддоменом (рис. 3.7). Первый шаг соответствует поддомену, связанному с каталогом продукции CoffeeMesh. Мы можем назвать его *поддоменом продукции*. Он сообщает нам, какие товары имеются в наличии, а какие — нет. Для этого поддомен продукции отслеживает количество каждого товара и ингредиента на складе.

Второй шаг соответствует поддомену, который позволяет пользователям выбирать товары. Этот поддомен управляет жизненным циклом каждого заказа, и мы называем его *поддоменом заказов*. Он владеет данными о заказах пользователей и предоставляет интерфейс, который позволяет нам управлять заказами и проверять

их статус. Он скрывает сложность системы, чтобы пользователю не приходилось разбираться в различных эндпоинтах и знать, что с ними делать. Поддомен заказов также заботится о второй части четвертого шага: передаче деталей заказа на кухню после успешной обработки платежа. Он также отвечает требованиям шестого шага: позволяет пользователю проверить состояние своего заказа. Как распределитель заказов, этот поддомен также работает с поддоменом доставки для организации доставки.



**Рис. 3.6.** Чтобы сделать заказ, покупатель заходит на сайт CoffeeMesh, выбирает товары из каталога и оплачивает заказ. После оплаты мы передаем данные о заказе на кухню, которая готовит его, а покупатель может следить за его выполнением. Наконец, мы организуем доставку заказа



**Рис. 3.7.** Мы привязываем к поддомену каждый шаг в процессе размещения и доставки заказа. Например, за поддержание актуальности каталога продукции отвечает поддомен продукции, а за процесс принятия заказа — поддомен заказов

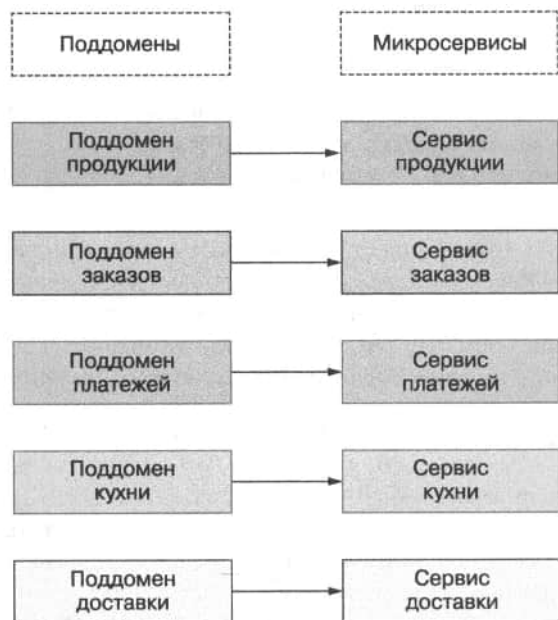
Третий шаг соответствует поддомену, который может обрабатывать платежи пользователей. Мы будем называть его *поддоменом платежей*. Он содержит специализированную логику для обработки платежей, включая проверку карт, интеграцию со сторонними платежными провайдерами, обработку различных способов оплаты и т. д. Поддомен платежей владеет данными, относящимися к платежам пользователей.

Пятый шаг соответствует поддомену, который работает с кухней для управления выполнением заказов покупателей. Мы называем его *поддоменом кухни*. Кухня полностью автоматизирована, и рассматриваемый поддомен взаимодействует с системой кухни для планирования выполнения заказов и отслеживания этого процесса. Как только заказ готов, поддомен кухни уведомляет об этом поддомен заказов, который затем организует доставку. Поддомен кухни владеет данными, связанными с производством заказов, и предоставляет интерфейс, который позволяет нам передавать заказы на кухню и отслеживать их выполнение. Поддомен заказов взаимодействует с поддоменом кухни для обновления статуса заказа в соответствии с требованиями шестого шага.

Седьмой шаг соответствует поддомену, который взаимодействует с автоматизированной системой доставки. Мы называем его *поддоменом доставки*. Он содержит специализированную логику для определения геолокации покупателя и расчета наиболее оптимального маршрута. Он управляет парком дронов-курьеров и оптимизирует доставку, а также владеет данными по всем доставкам. Поддомен заказов взаимодействует с поддоменом доставки для обновления маршрута заказа в соответствии с требованиями восьмого шага.

Используя стратегический анализ, мы получаем декомпозицию CoffeeMesh на пять поддоменов, которые можно сопоставить с микросервисами, поскольку каждый из них включает в себя четко определенную область с собственными данными. Результатом стратегического анализа DDD являются микросервисы, которые удовлетворяют принципам проектирования, перечисленным в разделе 3.2: все поддомены могут выполнять свои основные задачи, не полагаясь на другие микросервисы, поэтому мы говорим, что они слабо связаны; каждый сервис владеет собственными данными, что соответствует принципу «База данных для каждого сервиса»; наконец, каждый сервис выполняет задачи в пределах узко определенного поддомена, что соответствует SRP.

В результате стратегического анализа мы получаем следующую архитектуру микросервисов (рис. 3.8).



**Рис. 3.8.** В результате стратегического анализа DDD мы разбили платформу CoffeeMesh на пять поддоменов, которые могут быть непосредственно сопоставлены с микросервисами

- Поддомен продукции связан с сервисом продукции — управляет каталогом товаров CoffeeMesh.
- Поддомен заказов сопоставляется с сервисом заказов — управляет заказами покупателей.
- Поддомен платежей сопоставляется с сервисом платежей — управляет платежами покупателей.
- Поддомен кухни сопоставляется с сервисом кухни — управляет выполнением заказов на кухне.
- Поддомен доставки сопоставляется с сервисом доставки — управляет доставкой товаров покупателям.

В следующем разделе мы сравним результаты стратегического анализа DDD с результатами декомпозиции сервисов по бизнес-функциям и оценим преимущества и проблемы каждого подхода.

## **3.5. ДЕКОМПОЗИЦИЯ ПО БИЗНЕС-ФУНКЦИЯМ В СРАВНЕНИИ С ДЕКОМПОЗИЦИЕЙ ПО ПОДДОМЕНАМ**

Какую стратегию декомпозиции сервисов нам лучше использовать: по бизнес-функциям или поддоменам? В то время как декомпозиция по бизнес-функциям фокусируется на структуре и организации бизнеса, декомпозиция по поддоменам анализирует бизнес-процессы и потоки. Таким образом, оба подхода позволяют по-разному взглянуть на бизнес, и если вы можете выделить время, то лучшей стратегией будет применение обоих подходов.

Иногда можно объединить результаты подходов. Например, на платформе CoffeeMesh мы дадим возможность покупателям писать отзывы о каждом товаре, а приложение будет использовать эту информацию, чтобы рекомендовать новые товары другим людям. Для этого можно создать отдельную команду. С технической точки зрения отзывы могут стать просто еще одной таблицей в базе данных Products. Однако для облегчения взаимодействия с бизнесом удобнее создать сервис отзывов. Он мог бы передавать новые отзывы в систему рекомендаций, а сервис заказов использовал бы интерфейс сервиса отзывов для выдачи рекомендаций новым пользователям.

Преимущество декомпозиции по бизнес-функциям в том, что архитектура платформы согласуется с существующей организационной структурой бизнес-компании. Такое соответствие может облегчить сотрудничество между бизнес- и техническими командами. Но существующая организационная структура компании не обязательно будет наиболее эффективной. Она может оказаться устаревшей и отражать старые бизнес-процессы. В этом случае неэффективность бизнеса будет отражена в архитектуре микросервисов, и это большой недостаток данного подхода.

Когда мы применили декомпозицию по бизнес-функциям в подразделе 3.3.2, мы получили нежелательное разделение между сервисами продукции и ингредиентов. Проанализировав зависимости между обоими сервисами, мы пришли к выводу, что обе возможности должны входить в один сервис. Однако в реальных ситуациях этого дополнительного анализа не бывает и полученная архитектура не является оптимальной. На основании анализа, проведенного в разделах 3.3 и 3.4, можно сказать, что декомпозиция по поддоменам позволяет моделировать бизнес-процессы и потоки с учетом особенностей архитектуры, и если требуется выбрать только один подход, то лучше остановиться на этом варианте.

Теперь, когда мы знаем, как спроектировать наши микросервисы, пришло время начать работу над их интерфейсами. В следующих главах вы научитесь разрабатывать интерфейсы REST и GraphQL для микросервисов.

## РЕЗЮМЕ

- Процесс разбиения системы на микросервисы называется декомпозицией сервисов. Она определяет границы между сервисами, и необходимо правильно организовать этот процесс, чтобы случайно не создать распределенный монолит.
- При декомпозиции по бизнес-функциям анализируется структура бизнеса и разрабатываются микросервисы для каждой команды в организации. Этот подход подразумевает согласование структуры бизнеса и архитектуры системы, но при этом возможно отражение в платформе неэффективности бизнеса.
- При декомпозиции по поддоменам применяется подход DDD, когда процессы и потоки бизнеса моделируются через поддомены. Используя этот подход, мы разрабатываем микросервис для каждого поддомена, что приводит к более надежному техническому проекту.
- Чтобы оценить качество архитектуры микросервисов, следует ориентироваться на три принципа проектирования.
  - *Принцип «База данных для каждого сервиса»* — каждый микросервис владеет своими данными, а доступ к ним осуществляется через API сервиса.
  - *Принцип слабой связанности* — можно обновить сервис, не затрагивая другие сервисы, и каждый сервис должен иметь возможность работать без постоянного вызова других сервисов.
  - *Принцип единственной ответственности* — нужно разрабатывать каждый сервис на основе конкретной бизнес-функции или поддомена.



## *Часть II*

# *Проектирование и разработка REST API*

В части I вы узнали, что такое API микросервисов и как разбить систему на микросервисы. Теперь пришло время обсудить, как построить микросервис и заставить сервисы взаимодействовать.

Сервисы общаются друг с другом через API, и в части II вы научитесь проектировать и разрабатывать REST API. Representational state transfer, REST — самая популярная технология для создания API, и в главе 4 вы узнаете все фундаментальные принципы проектирования REST API. Мы будем придерживаться практического подхода: в главе 1 мы представили CoffeeMesh, приложение для доставки кофе, а в главе 6 вы научитесь создавать API сервисов заказов и кухни для CoffeeMesh с помощью популярных фреймворков Python FastAPI и Flask.

В главе 1 мы обсудили разработку на основе документации и подчеркнули важность документации API. REST API документируют, используя стандарт OpenAPI, и в главе 5 мы шаг за шагом рассмотрим, как это делать. Наконец, в главе 7 вы узнаете все необходимое для разработки микросервисов. Вы научитесь реализовывать уровень данных с помощью SQLAlchemy и управлять миграциями, используя Alembic. Мы поговорим, как структурировать приложение с помощью гексагональной архитектуры, а также рассмотрим многие другие полезные паттерны и принципы, позволяющие инкапсулировать код и поддерживать слабую связанность между уровнями.

К концу части II вы сможете разрабатывать отличные REST API, писать превосходную документацию по API и создавать удобные в обслуживании реализации сервисов.

# Принципы проектирования REST API

## В этой главе

- ✓ Принципы проектирования REST API.
- ✓ Как модель зрелости Ричардсона помогает нам понять преимущества лучших принципов проектирования REST.
- ✓ Концепция ресурса и проектирование эндпоинтов для REST API.
- ✓ Использование методов HTTP и статус-кодов HTTP для создания информативных REST API.
- ✓ Разработка хорошей полезной нагрузки и параметров URL-запросов для REST API.

REST описывает архитектурный стиль для приложений, взаимодействующих по сети. Первоначально эта концепция включала в себя список ограничений для проектирования распределенных и масштабируемых веб-приложений. Со временем появились подробные протоколы и спецификации, которые дают четко определенные рекомендации по проектированию REST API. Сегодня REST, безусловно, является самым популярным выбором для разработки веб-API<sup>1</sup>. В этой главе мы разберем принципы проектирования REST и научимся применять их, разрабатывая API сервиса заказов для платформы CoffeeMesh.

<sup>1</sup> В отчете State of the API Report за 2022 год, подготовленном компанией Postman, было установлено, что большинство участников опроса (89 %) используют REST (<https://www.postman.com/state-of-api/api-technologies/#api-technologies>).

Вы узнаете, что такое ресурс и в чем его значение для разработки REST API. Вы также научитесь использовать возможности протокола HTTP, такие как методы HTTP и статус-коды, для создания высокоинформативных API. В конце главы рассматриваются лучшие практики проектирования полезной нагрузки API и параметров URL.

## 4.1. ЧТО ТАКОЕ REST

REST — термин, введенный Роем Филдингом в его докторской диссертации *Architectural Styles and the Design of Network-based Software Architectures* («Архитектурные стили и проектирование сетевых архитектур программного обеспечения») (University of California, Irvine, 2000). Он описывает архитектурный стиль для слабосвязанных и хорошо масштабируемых приложений, взаимодействующих по сети. Сюда же относится возможность передачи представления состояния ресурса. Ресурс является фундаментальным понятием в приложениях REST.

### ОПРЕДЕЛЕНИЕ

REST — это архитектурный стиль для создания слабосвязанных и хорошо масштабируемых API. REST API структурированы вокруг ресурсов, сущностей, которыми можно управлять с помощью API.

*Ресурс* — это объект, на который можно сослаться с помощью уникальной гипертекстовой ссылки (URL). Существует два типа ресурсов: коллекции и одиночные элементы (*singleton*). *Одиночный элемент* представляет собой одну сущность, в то время как *коллекции* — это списки сущностей<sup>1</sup>. Что это означает на практике? Это означает, что мы используем разные URL-адреса для каждого типа ресурсов. Например, сервис заказов CoffeeMesh управляет заказами, и через его API мы можем получить доступ к конкретному заказу по URL-пути `/orders/{order_id}`, а коллекция заказов доступна по URL-пути `/orders`. Поэтому `/orders/{order_id}` является одиночным эндпоинтом, а `/orders` — эндпоинтом коллекции.

Одни ресурсы могут быть вложены в другой, например полезная нагрузка для заказа с несколькими позициями, перечисленными во вложенном массиве (листинг 4.1).

### Листинг 4.1. Пример полезной нагрузки с вложенными ресурсами

```
{
  "id": "924721eb-a1a1-4f13-b384-37e89c0e0875",
  "status": "progress",
  "created": "2023-09-01",
```

<sup>1</sup> Подробный обзор ресурсов и их моделирования в REST API см. в отличной статье: *Subramaniam P. REST API Design-Resource Modeling* («Проектирование REST API — моделирование ресурсов») (<https://www.thoughtworks.com/en-gb/insights/blog/rest-api-design-resource-modeling>).

```
"order": [
  {
    "product": "cappuccino",
    "size": "small",
    "quantity": 1
  },
  {
    "product": "croissant",
    "size": "medium",
    "quantity": 2
  }
]
```

Мы можем создавать вложенные эндпоинты для представления вложенных ресурсов — они позволяют получить доступ к конкретным сведениям о ресурсе. Например, можно предоставить эндпоинт `GET /orders/{order_id}/status`, который дает информацию о статусе заказа без остальных подробностей о нем. Использование вложенных эндпоинтов — популярная стратегия оптимизации. В таком случае ресурсы представлены большой полезной нагрузкой и можно избежать дорогостоящей передачи данных, когда нас интересует только одно свойство.

Ресурсно-ориентированная природа REST API иногда может казаться ограничивающей. Часто встает вопрос о том, как моделировать действия через эндпоинты, сохраняя при этом API соответствующими REST (то есть RESTful). Например, как представить действие отмены заказа? Один из лучших эвристических подходов состоит в том, чтобы представлять действия как вложенные ресурсы. Например, у нас может быть эндпоинт `POST /orders/{order_id}/cancel` для отмены заказов. В этом случае мы моделируем отмену заказа как создание события отмены.

Проектирование понятных эндпоинтов — это первый шаг к созданию REST API, которые легко поддерживать и использовать. В этом нам помогут паттерны, описанные в данном разделе, а в остальных разделах главы вы узнаете о дополнительных паттернах и принципах проектирования чистых API. Далее поговорим о шести архитектурных ограничениях приложений REST API.

## 4.2. АРХИТЕКТУРНЫЕ ОГРАНИЧЕНИЯ ПРИЛОЖЕНИЙ REST

Архитектурные ограничения приложений REST были перечислены Филдингом. Они определяют, как сервер должен обрабатывать запрос клиента и отвечать на него. Прежде чем мы углубимся в детали, приведу краткий обзор каждого ограничения.

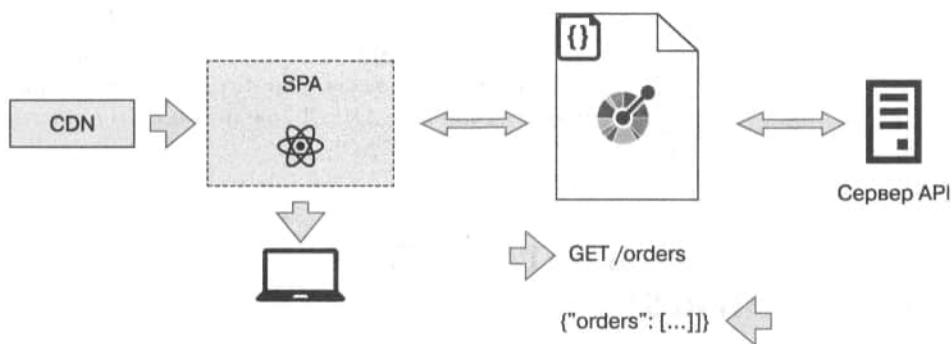
- *Клиент-серверная архитектура* — пользовательский интерфейс (UI) должен быть отделен от серверной части.

- *Отсутствие записи состояния клиента* — сервер не должен управлять состояниями между запросами.
- *Кэшируемость* — запросы, которые всегда возвращают один и тот же ответ, должны быть кэшируемыми.
- *Многоуровневость системы* — архитектура API может быть многослойной, но такая сложность должна быть скрыта от пользователя.
- *Предоставление кода по запросу* — сервер может внедрять код в пользовательский интерфейс по требованию.
- *Единство интерфейса* — API должен предоставлять согласованный интерфейс для доступа к ресурсам и управления ими.

Обсудим каждое из этих ограничений более подробно.

### 4.2.1. Разделение задач: принцип клиент-серверной архитектуры

REST опирается на принцип разделения задач и, следовательно, требует, чтобы пользовательские интерфейсы были отделены от хранилища данных и серверной логики. Это позволяет компонентам на стороне сервера развиваться независимо от элементов пользовательского интерфейса. Как видно на рис. 4.1, архитектурный паттерн «клиент — сервер» часто реализуется в виде построения пользовательского интерфейса как отдельного приложения, например одностраничного (SPA).

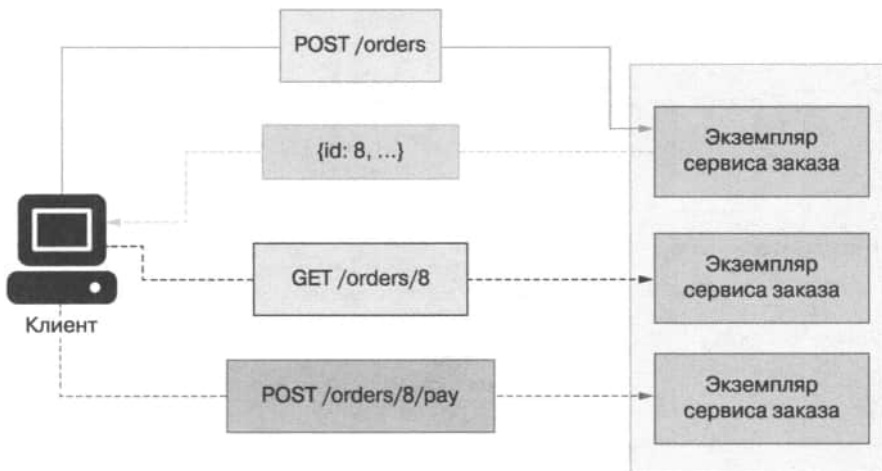


**Рис. 4.1.** Принцип клиент-серверной архитектуры REST гласит, что реализация сервера должна быть отделена от реализации клиента<sup>1</sup>

<sup>1</sup> CDN — это Content-Delivery Network (сеть доставки контента), географически распределенная сетевая инфраструктура, обеспечивающая быструю доставку контента пользователям веб-сервисов и сайтов. — *Примеч. ред.*

### 4.2.2. Добавляем масштабирование: принцип отсутствия записи состояния

В REST каждый запрос к серверу должен содержать всю информацию, необходимую для его выполнения. В частности, сервер не должен сохранять состояние от одного запроса к другому. Как показано на рис. 4.2, если убрать управление состоянием из компонентов сервера, можно облегчить горизонтальное масштабирование серверной части. Это позволяет развернуть несколько экземпляров сервера, и, поскольку ни один из них не управляет состоянием клиента API, тот может взаимодействовать с любым экземпляром.

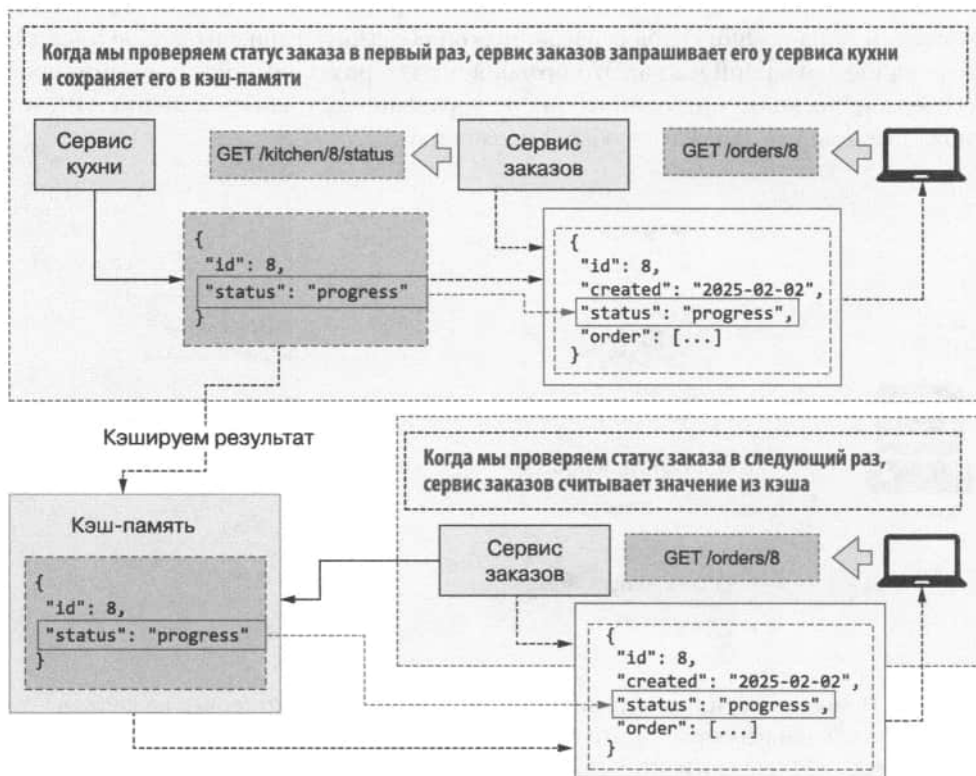


**Рис. 4.2.** Принцип отсутствия записи состояния в REST гласит, что сервер не должен управлять состоянием клиента. Это позволяет нам развертывать несколько экземпляров API сервера и отвечать API клиента с помощью любого из них

### 4.2.3. Оптимизация производительности: принцип кэшируемости

Когда это уместно, ответы сервера должны кэшироваться. Кэширование улучшает производительность API, поскольку позволяет не выполнять все вычисления, необходимые для получения ответа, снова и снова. Для кэширования подходят GET-запросы, поскольку они возвращают данные, уже сохраненные на сервере. Как показано на рис. 4.3, кэшируя GET-запросы, можно не получать данные каждый раз, когда пользователь запрашивает одну и ту же информацию. Чем больше времени требуется на сборку ответа на GET-запрос, тем больше пользы от кэширования.

Рисунок 4.3 иллюстрирует преимущества кэширования. Как вы узнали в главе 3, покупатели могут отслеживать ход выполнения своих заказов. Для этого сервис заказов взаимодействует с сервисом кухни. Чтобы сэкономить время, когда покупатель в следующий раз будет проверять состояние заказа, мы ненадолго кэшируем его значение.



**Рис. 4.3.** Принцип кэшируемости REST гласит, что ответы должны кэшироваться, что помогает повысить производительность сервера API. В данном примере мы ненадолго кэшируем статус заказа, чтобы избежать многократных запросов к сервису кухни

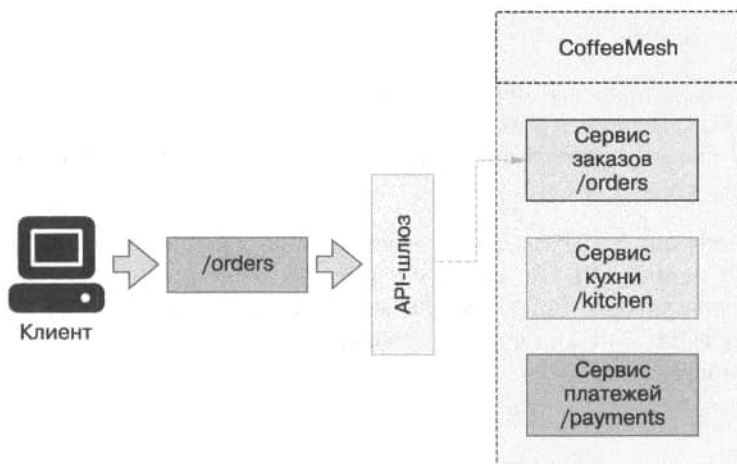
#### 4.2.4. Упрощение для клиента: принцип многоуровневой системы

В архитектуре REST у клиентов должна быть уникальная точка входа в ваш API и не должно быть возможности определить, подключены ли они непосредственно к конечному серверу или к промежуточному уровню типа балансировщика



нагрузки. Вы можете развернуть на разных серверах как различные компоненты серверного приложения, так и один и тот же компонент для обеспечения избыточности и масштабируемости. Эта сложность должна быть скрыта от пользователя путем создания единого эндпоинта, который инкапсулирует доступ к вашим сервисам.

Как видно на рис. 4.4, общим решением этой задачи является паттерн API-шлюз, который представляет собой компонент, служащий точкой входа для всех микросервисов. API-шлюз знает адреса серверов каждого сервиса и как сопоставить каждый запрос с соответствующим сервисом<sup>1</sup>.



**Рис. 4.4.** Принцип многоуровневой системы REST гласит, что сложность серверной части должна быть скрыта от клиента. Распространенным решением этой задачи является паттерн API-шлюз, который служит точкой входа для всех сервисов платформы

### 4.2.5. Расширяемые интерфейсы: принцип «код по запросу»

Серверы могут расширять функциональность клиентского приложения, отправляя исполняемый код непосредственно из серверной части, например файлы JavaScript, необходимые для запуска пользовательского интерфейса. Это ограничение опционально и применяется только к приложениям, в которых серверная часть обслуживает клиентский интерфейс.

<sup>1</sup> Подробнее об этом паттерне см. в книге: *Ричардсон К. Микросервисы.* – С. 307–329. <https://livebook.manning.com/book/microservices-patterns/chapter-8/point-8620-53-297-0>.

### 4.2.6. Сохранение согласованности: принцип единства интерфейса

Приложения REST должны предоставлять своим пользователям единообразный и согласованный интерфейс. Требуется, чтобы интерфейс был документирован, а спецификация API строго соблюдалась сервером и клиентом. Отдельные ресурсы идентифицируются с помощью унифицированного идентификатора ресурса (URI)<sup>1</sup>, и каждый URI должен быть уникальным и всегда возвращать один и тот же ресурс. Например, URI `/orders/8` представляет заказ с идентификатором 8, и запрос GET по этому URI всегда возвращает состояние заказа с идентификатором 8. Если заказ удален из системы, идентификатор не должен повторно использоваться для представления другого заказа.

Ресурсы необходимо представлять с применением выбранного метода сериализации, и этот подход должен использоваться последовательно во всем API. Сегодня REST API в качестве формата сериализации обычно используют JSON, хотя возможны и другие форматы, например XML.

Архитектурные ограничения REST дают нам твердую основу для проектирования надежных и масштабируемых API. Но, как мы увидим в следующих разделах, есть еще несколько факторов, которые придется учитывать при разработке API. Далее поговорим о том, как сделать API доступным для обнаружения, добавив в описание ресурса связанные гипермедиассылки.

## 4.3. ГИПЕРМЕДИА КАК ДВИГАТЕЛЬ СОСТОЯНИЯ ПРИЛОЖЕНИЙ

Теперь, зная наиболее важные ограничения проектирования REST API, рассмотрим еще одну важную концепцию REST: гипермедиа как двигатель состояния приложения (*hypermedia as the engine of application state*, HATEOAS). HATEOAS — это парадигма в проектировании REST API, которая акцентирует внимание на легкости обнаружения (*discoverability*). HATEOAS упрощает применение API, дополняя ответы информацией, необходимой пользователям для взаимодействия с ресурсом. В этом разделе я объясню, как работает HATEOAS, и приведу преимущества и недостатки этого подхода.

Что именно представляет собой HATEOAS? В статье *REST APIs Must Be Hypertext-Driven* («REST API должны быть гипертекстовыми») (<http://mng.bz/p6y5>), написанной в 2008 году, Филдинг предположил, что REST API должны включать в свои ответы связанные ссылки, позволяющие клиентам перемещаться по API.

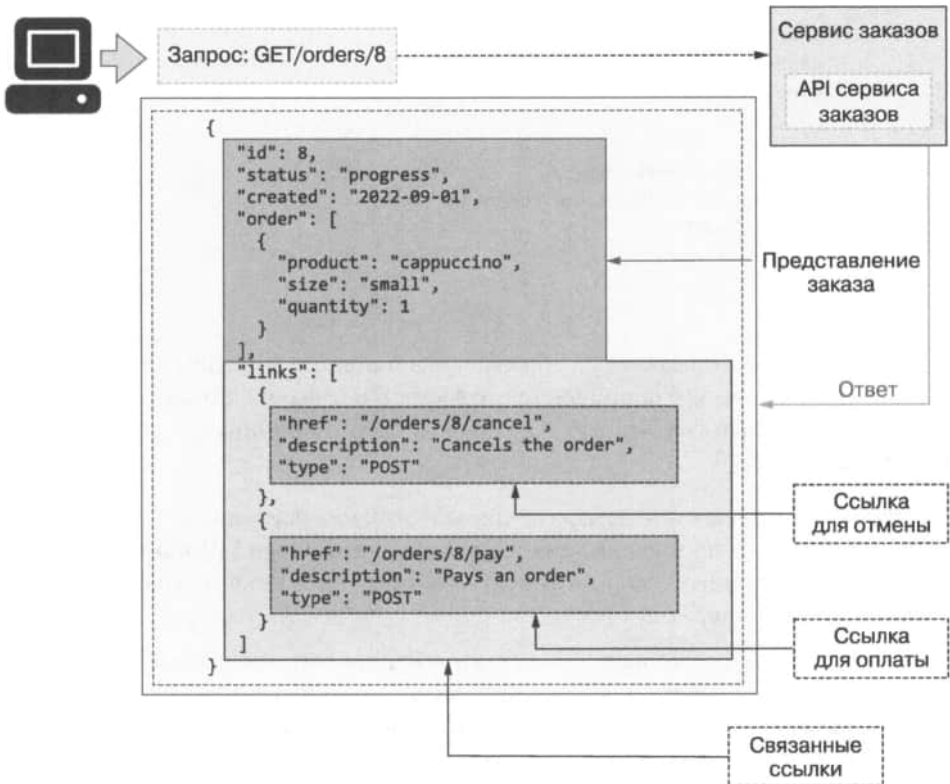
---

<sup>1</sup> Последняя спецификация URI приведена в документе RFC 7320: URI Design and Ownership М. Ноттингема (июль 2004 года, <https://tools.ietf.org/html/rfc7320>).

**ОПРЕДЕЛЕНИЕ**

*Гипермедиа как двигатель состояния приложения (HATEOAS)* — это парадигма проектирования REST, которая подчеркивает идею легкости обнаружения. Каждый раз, когда клиент запрашивает ресурс у сервера, ответ должен включать список связанных ссылок на ресурс. Например, если клиент запрашивает детали заказа, в ответе должны быть ссылки на отмену и оплату заказа.

Например, когда клиент запрашивает детали заказа, API выдает в том числе набор ссылок, связанных с ним (рис. 4.5, листинг 4.2). С их помощью мы можем отменить заказ или оплатить его.



**Рис. 4.5.** В парадигме HATEOAS API отправляет представление запрашиваемого ресурса с другими ссылками, связанными с этим ресурсом

**Листинг 4.2.** Представление заказа, включающего ссылки на гипермедиа

```
{
  "id": 8,
  "status": "progress",
  "created": "2023-09-01",
```

```

"order": [
  {
    "product": "cappuccino",
    "size": "small",
    "quantity": 1
  },
  {
    "product": "croissant",
    "size": "medium",
    "quantity": 2
  }
],
"links": [
  {
    "href": "/orders/8/cancel",
    "description": "Cancels the order",
    "type": "POST"
  },
  {
    "href": "/orders/8/pay",
    "description": "Pays for the order",
    "type": "POST"
  }
]
}

```

Предоставление связанных ссылок упрощает навигацию по API и их использование, поскольку каждый ресурс поставляется со всеми URL, необходимыми для работы с ним. Однако на практике многие API по нескольким причинам не реализуются таким образом.

- Информация, предоставляемая гиперссылками, уже доступна в документации API. На самом деле информация в спецификации OpenAPI намного богаче и структурированнее, чем та, которую вы можете предоставить со связанными ссылками для конкретных ресурсов.
- Не всегда ясно, какие именно ссылки должны быть возвращены. Разные пользователи имеют разные уровни прав, которые позволяют им выполнять разные действия и получать доступ к разным ресурсам. Например, внешние пользователи могут использовать эндпоинт `POST /orders` в API CoffeeMesh для размещения заказа, а также эндпоинт `GET /orders/{order_id}` для получения информации о заказе. Однако они не могут задействовать эндпоинт `DELETE /orders/{order_id}` для удаления заказа, поскольку он доступен только внутренним пользователям платформы CoffeeMesh. Если цель HATEOAS в том, чтобы сделать старт навигации по API из единой точки входа, нет смысла возвращать эндпоинт `DELETE /orders/{order_id}` внешним пользователям, поскольку они не могут его использовать. Тогда необходимо возвращать разные списки связанных ссылок разным пользователям в соответствии с их разрешениями. Однако такой уровень гибкости вносит до-

полнительную сложность в проекты и реализации API и объединяет уровень авторизации с уровнем API.

- В зависимости от состояния ресурса могут быть недоступны определенные действия и ресурсы. Например, вы можете вызвать эндпоинт `POST /orders/1234/cancel` для активного заказа, но не для отмененного. Такой уровень неоднозначности затрудняет определение и реализацию надежных интерфейсов, соответствующих принципам HATEOAS.
- Наконец, в некоторых API список связанных ссылок может быть большим, и, следовательно, это будет увеличивать полезную нагрузку ответа, что влияет на производительность API и надежность соединения для небольших устройств с низким уровнем сетевого подключения.

Работая над своими API, вы сами решаете, следовать ли принципам HATEOAS. В некоторых случаях предоставление списков связанных ресурсов несет определенную пользу. Например, в вики-приложении раздел связанных ресурсов можно использовать для перечисления содержимого, относящегося к конкретной статье, ссылок на ту же статью на других языках и ссылок на действия, которые можно выполнить со статьей. В целом можно найти баланс между тем, что ваша API-документация уже предоставляет клиенту, и тем, что вы можете предложить в своих ответах, чтобы облегчить взаимодействие между клиентом и API. Если ваш API ориентирован на широкую публику, то клиентам не мешают связанные ссылки. Однако если речь идет о небольшом внутреннем API, включать относительные ссылки, скорее всего, не нужно.

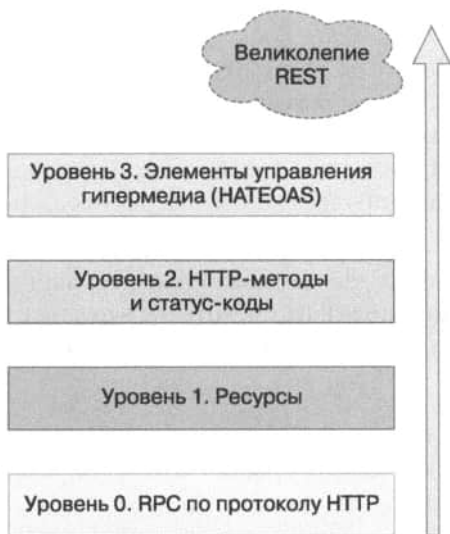
Теперь, зная, как сделать API доступными для обнаружения и когда это стоит делать, перейдем к знакомству с моделью зрелости Ричардсона, которая поможет понять, в какой степени API соответствуют принципам проектирования REST.

## 4.4. АНАЛИЗ ЗРЕЛОСТИ API С ПОМОЩЬЮ МОДЕЛИ РИЧАРДСОНА

В этом разделе мы рассмотрим ментальную модель, разработанную Леонардом Ричардсоном. С ее помощью можно определять степень соответствия API принципам REST<sup>1</sup>. Модель зрелости Ричардсона выделяет четыре уровня (от 0 до 3) зрелости API. Каждый уровень вводит дополнительные элементы хорошего дизайна REST API (рис. 4.6). Обсудим каждый уровень подробно.

---

<sup>1</sup> Леонард Ричардсон представил модель зрелости в своем выступлении *Justice Will Take Us Millions of Intricate Moves* («Справедливость требует от нас миллионов сложных шагов») на QCon в Сан-Франциско в 2008 году (<https://www.crummy.com/writing/speaking/2008-QCon/>).



**Рис. 4.6.** Модель зрелости Ричардсона выделяет четыре уровня зрелости API, где высший уровень представляет собой проект API, который соответствует лучшим практикам и стандартам REST, а низший уровень — тип API, в котором не применяется ни один из принципов REST

#### 4.4.1. Уровень 0. Веб-API в стиле RPC

На уровне 0 HTTP используется в качестве транспортной системы для взаимодействия с сервером. Понятие API в этом случае ближе к понятию *удаленного вызова процедур* (*remote procedure call*, RPC; см. приложение А). Все запросы к серверу выполняются в одном и том же эндпоинте и с использованием одного и того же метода HTTP, обычно GET или POST. Детали запроса клиента передаются в полезной нагрузке HTTP. Например, чтобы разместить заказ на сайте CoffeeMesh, клиент может отправить POST-запрос в общий эндпоинт `/api` со следующей полезной нагрузкой:

```
{
  "action": "placeOrder",
  "order": [
    {
      "product": "mocha",
      "size": "medium",
      "quantity": 2
    }
  ]
}
```

Сервер неизменно отвечает статус-кодом 200 и сопутствующей полезной нагрузкой, информирующей нас о результатах обработки запроса. Аналогично, чтобы получить детали заказа, клиент может сделать следующий POST-запрос в общем эндпоинте `/api` (при условии, что ID заказа равен 8):

```
{
  "action": "getOrder",
  "order": [
    {
      "id": 8
    }
  ]
}
```

#### 4.4.2. Уровень 1. Введение понятия ресурса

Уровень 1 вводит понятие URL-адреса ресурса. Вместо общего эндпоинта `/api` сервер предоставляет URL-адреса, которые соответствуют ресурсам. Например, URL `/orders` связан с коллекцией заказов, а URL `/orders/{order_id}` — с одним заказом. Чтобы разместить заказ, клиент отправляет POST-запрос к эндпоинту `/orders` с такой же полезной нагрузкой, как и на уровне 0:

```
{
  "action": "placeOrder",
  "order": [
    {
      "product": "mocha",
      "size": "medium",
      "quantity": 2
    }
  ]
}
```

На этот раз при запросе деталей последнего заказа клиент сделает POST-запрос на URI<sup>1</sup>, представляющий этот заказ: `/orders/8`. На этом уровне API не проводит различий между HTTP-методами для представления различных действий.

---

<sup>1</sup> Автор использует термин URL для обозначения адреса неединичного ресурса и URI для обозначения единичного ресурса, например заказа. Термин «унифицированный указатель ресурсов» (URL) относится к подмножеству «унифицированный идентификатор ресурсов» (URI). Такой идентификатор, кроме идентификации ресурса, позволяет определять его местоположение. Понятие URI включает в себя следующие разновидности: URL, URN, URC и Data URI. — *Примеч. ред.*

### 4.4.3. Уровень 2. Использование методов HTTP и статус-кодов

Уровень 2 знакомит с понятием методов HTTP и статус-кодов. На этом уровне методы HTTP используются для обозначения конкретных действий. Например, чтобы разместить заказ, клиент отправляет POST-запрос к эндпоинту `/orders` со следующей полезной нагрузкой:

```
{
  "order": [
    {
      "product": "mocha",
      "size": "medium",
      "quantity": 2
    }
  ]
}
```

В этом случае HTTP-метод POST указывает на операцию, которую мы хотим выполнить, а полезная нагрузка включает только детали заказа, который мы планируем разместить. Аналогично, чтобы получить детали заказа, мы отправляем запрос GET на URI заказа: `/orders/{order_id}`. Используем HTTP-метод GET, чтобы сообщить серверу, что хотим получить подробную информацию о ресурсе, указанном в URI.

Если предыдущие уровни включают один и тот же статус-код (обычно 200), то на уровне 2 статус-коды HTTP используются в зависимости от назначения, для сообщения о результатах обработки запроса клиента. Например, когда мы создаем ресурс с помощью POST-запроса, мы получаем статус-код 201, а запрос на несуществующий ресурс получает код 404. Более подробно о статус-кодах HTTP мы поговорим в разделе 4.6.

### 4.4.4. Уровень 3. Возможность обнаружения API

На уровне 3 вводится понятие легкости обнаружения. Здесь применяются принципы HATEOAS и ответы дополняются ссылками, представляющими действия, которые можно выполнять с ресурсом. Например, GET-запрос к эндпоинту `/orders/{order_id}` возвращает представление заказа и включает список связанных с ним ссылок (листинг 4.3).

**Листинг 4.3.** Представление заказа, включая ссылки на гипермедиа

```
{
  "id": 8
  "status": "progress",
  "created": "2023-09-01",
```



```
"order": [
  {
    "product": "cappuccino",
    "size": "small",
    "quantity": 1
  },
  {
    "product": "croissant",
    "size": "medium",
    "quantity": 2
  }
],
"links": [
  {
    "href": "/orders/8/cancel",
    "description": "Cancels the order",
    "type": "POST"
  },
  {
    "href": "/orders/8/pay",
    "description": "Pays for the order",
    "type": "GET"
  }
]
}
```

В модели зрелости Ричардсона уровень 3 представляет собой последний шаг к тому, что он называет «великолепием REST».

Как модель зрелости Ричардсона влияет на разработку наших API? Опираясь на нее, можно понять, как наши проекты API вписываются в общие принципы REST. Эта модель не предназначена для измерения степени того, насколько API «соответствует» принципам REST, или для иной оценки проекта API; вместо этого она позволяет проанализировать, насколько хорошо мы используем протокол HTTP для создания API, которые легко понять и использовать.

Теперь, изучив основные принципы проектирования REST API, приступим к проектированию API сервиса заказов! Начнем с проектирования эндпоинтов API, научившись использовать методы HTTP.

## 4.5. СТРУКТУРИРОВАННЫЕ URL-АДРЕСА РЕСУРСОВ С МЕТОДАМИ HTTP

Как мы только что обсудили, использование методов HTTP и статус-кодов по модели Ричардсона соответствует зрелому проекту API. В этом разделе мы научимся правильно использовать методы HTTP, применив их к API сервиса заказов для приложения CoffeeMesh.

HTTP-методы — это специальные ключевые слова. Если правильно прописать HTTP-методы, API будут более структурированными, понятными и простыми в использовании.

## ОПРЕДЕЛЕНИЕ

*Методы HTTP-запросов* — это ключевые слова, используемые в HTTP-запросах для указания типа действия, которое требуется выполнить. Например, метод GET позволяет получить подробную информацию о ресурсе, а метод POST создает новый ресурс. Наиболее важными методами HTTP для REST API являются GET, POST, PUT, PATCH и DELETE.

По моему опыту, многие путаются в назначении методов HTTP. Давайте проясним эту путаницу, изучив семантику каждого метода.

- GET — возвращает информацию о запрашиваемом ресурсе.
- POST — создает новый ресурс.
- PUT — выполняет полное обновление путем замены ресурса.
- PATCH — обновляет определенные свойства ресурса.
- DELETE — удаляет ресурс.

### СЕМАНТИКА МЕТОДА PUT

Согласно спецификации HTTP, PUT может быть идемпотентным, поэтому можно использовать его для создания ресурса, если его не существует. Однако спецификация также гласит: «[а] сервис, который выбирает правильный URI от имени клиента после получения запроса на изменение состояния, должен быть реализован с использованием метода POST, а не PUT». Это означает, что, когда сервер отвечает за генерацию URI нового ресурса, мы должны создавать ресурсы с помощью POST, а PUT можно использовать только для обновления<sup>1</sup>.

Методы HTTP позволяют моделировать основные операции, которые допускаются выполнять над ресурсом: создание (POST), чтение (GET), изменение (PUT и PATCH) и удаление (DELETE). Эти операции обозначаются аббревиатурой CRUD, которая пришла из области баз данных<sup>2</sup>, но очень популярна в мире API. Вы часто будете слышать о CRUD API, которые представляют собой API, разработанные для выполнения перечисленных операций с ресурсами.

<sup>1</sup> См. статью: *Fielding R. HyperText Transfer Protocol (HTTP/1.1): Semantics and Content* («Протокол передачи гипертекста (HTTP/1.1): семантика и содержание») (RFC 7231, июнь 2014 года, <https://tools.ietf.org/html/rfc7231#section-4.3.4>).

<sup>2</sup> Аббревиатура CRUD была введена Джеймсом Мартином в его известной книге *Managing the Data-Base Environment* (Prentice-Hall, 1983, p. 381).

### PUT И PATCH: В ЧЕМ РАЗНИЦА И КОГДА ИХ ИСПОЛЬЗОВАТЬ?

Мы можем использовать как PUT, так и PATCH для выполнения обновлений. В чем же разница между ними? В то время как PUT требует от клиента API отправки совершенно нового представления ресурса, PATCH позволяет отправлять только изменившиеся свойства.

Например, заказ имеет следующее представление:

```
{
  "id": "96247264-7d42-4a95-b073-44cedf5fc07d",
  "status": "progress",
  "created": "2023-09-01",
  "order": [
    {
      "product": "cappuccino",
      "size": "small",
      "quantity": 1
    },
    {
      "product": "croissant",
      "size": "medium",
      "quantity": 2
    }
  ]
}
```

Теперь предположим, что пользователь хочет внести небольшую правку в этот заказ и изменить размер круассанов со среднего на маленький. Хотя он хочет изменить одно конкретное поле, при использовании PUT он должен отправить всю полезную нагрузку обратно на сервер. Однако с помощью PATCH можно отправить только те поля, которые следует обновить на сервере. Запросы PATCH более оптимальны, поскольку полезная нагрузка, отправляемая на сервер, меньше. Однако, как видно из следующего примера, запросы PATCH также имеют более сложную структуру, и иногда их труднее обрабатывать в серверной части:

```
{
  "op": "replace",
  "path": "order/1/size",
  "value": "medium"
}
```

Это соответствует рекомендациям спецификации JSON Patch<sup>1</sup>: запрос JSON Patch должен содержать тип операции, которую мы хотим выполнить, а также целевой атрибут и его желаемое значение.

Реализация эндпоинтов PATCH хорошо подходит для публичных API, но во внутренних API часто реализуются только эндпоинты PUT для обновлений, поскольку с ними проще обращаться. В API сервиса заказов мы будем задавать обновления как PUT-запросы.

<sup>1</sup> Bryan P., Nottingham M. JavaScript Object Notation (JSON) Patch. <https://www.rfc-editor.org/rfc/rfc6902>.

Для определения эндпоинтов API сервиса заказов CoffeeMesh мы используем методы HTTP в сочетании с URL-адресами, поэтому сначала определим URL ресурсов. В разделе 4.1 мы научились различать два типа URL ресурсов в REST: одиночные, которые представляют один ресурс, и коллекции, которые представляют список ресурсов. В API сервиса заказов у нас есть оба вида:

- `/orders` — представляет собой список заказов;
- `/orders/{orders_id}` — представляет один заказ. Фигурные скобки вокруг `{order_id}` указывают, что это параметр URL-адреса и он должен быть заменен идентификатором заказа.

Как показано на рис. 4.7, мы используем одиночный URL `/orders/{order_id}` для выполнения действий над заказом, таких как его обновление, и коллекцию URL `/orders` для размещения и отображения списка предыдущих заказов. Методы HTTP помогают нам моделировать эти операции:

- `POST /orders` для размещения заказов, поскольку `POST` предназначен для создания новых ресурсов;
- `GET /orders` для получения списка заказов, поскольку `GET` используется для получения информации;
- `GET /orders/{order_id}` для получения сведений о конкретном заказе;
- `PUT /orders/{order_id}` для обновления заказа, поскольку `PUT` предназначен для обновления ресурса;
- `DELETE /orders/{order_id}` для удаления заказа;
- `POST /orders/{order_id}/cancel` для отмены заказа, так как `POST` применяется для создания объекта отмены;
- `POST /orders/{order_id}/pay` для оплаты заказа, поскольку мы используем `POST` для создания платежа.



**Рис. 4.7.** Мы комбинируем HTTP-методы с URL-путями для создания эндпоинтов нашего API. Используя семантику HTTP-методов, мы передаем задачу каждого эндпоинта. Например, метод `POST` предназначен для создания новых ресурсов, поэтому мы используем его в эндпоинте `POST /orders` для размещения заказов

Теперь, когда мы знаем, как разрабатывать эндпоинты API, сочетая URL-адреса с методами HTTP, посмотрим, как использовать семантику статус-кодов HTTP для получения информативных ответов.

## 4.6. ИСПОЛЬЗОВАНИЕ СТАТУС-КОДОВ HTTP ДЛЯ СОЗДАНИЯ ИНФОРМАТИВНЫХ HTTP-ОТВЕТОВ

В этом разделе объясняется, как статус-коды HTTP используются в ответах REST API. Сначала мы уточним, что такое статус-коды HTTP и как мы классифицируем их по группам, а затем поговорим, как применять их для моделирования ответов API.

### 4.6.1. Что такое статус-коды HTTP

Мы используем статус-коды, чтобы сигнализировать о результате обработки запроса на сервере. Если все верно задано, статус-коды HTTP помогают предоставлять информативные ответы потребителям API. Они делятся на следующие пять групп:

- 1xx — сигнализирует о том, что выполняется операция;
- 2xx — означает, что запрос был успешно обработан;
- 3xx — сигнализирует о том, что ресурс был перемещен в новое место;
- 4xx — означает ошибку в составлении запроса клиентским приложением;
- 5xx — сигнализирует о том, что произошла ошибка при обработке запроса на стороне сервера.

#### ПРИМЕЧАНИЕ

Статус-коды HTTP-ответа используются для указания результата обработки HTTP-запроса. Например, статус-код 200 означает, что запрос был успешно обработан, а код 500 говорит о том, что при обработке запроса произошла внутренняя ошибка сервера. Статус-коды HTTP сопровождаются полем Reason Phrase, в котором поясняется смысл кода. Например, для статус-кода 404 это фраза Not Found («Не найдено»). Ознакомиться с полным списком статус-кодов и узнать о них больше можно на сайте <https://developer.mozilla.org/en-US/docs/Web/HTTP/Status>.

Полный список статус-кодов HTTP очень длинный, и перечисление их всех вряд ли поможет понять, как с ними работать. Вместо этого разберем наиболее часто используемые коды и посмотрим, как применять их в проектах API.

Имея дело со статус-кодами HTTP, полезно различать успешные и неуспешные ответы. Успешный ответ означает, что запрос был обработан, а неуспешный — что

при обработке запроса что-то пошло не так. Для каждого эндпоинта, определенного в разделе 4.5, используем следующие успешные статус-коды HTTP:

- `POST /orders: 201 (Created)` — сигнализирует о том, что ресурс был создан;
- `GET /orders: 200 (OK)` — означает, что запрос был успешно обработан;
- `GET /orders/{order_id}: 200 (OK)` — сигнализирует, что запрос был успешно обработан;
- `PUT /orders/{order_id}: 200 (OK)` — сигнализирует, что ресурс был успешно обновлен;
- `DELETE /orders/{order_id}: 204 (No Content)` — означает, что запрос был успешно обработан, но содержимое ответа не доставлено. В отличие от всех других методов запрос `DELETE` не требует ответа с полезной нагрузкой, поскольку мы лишь даем указание серверу удалить ресурс. Таким образом, код 204 (`No Content`) является хорошим выбором для этого типа HTTP-запроса;
- `POST /orders/{order_id}/cancel: 200 (OK)` — хотя это эндпоинт `POST`, мы используем статус-код 200 (`OK`), поскольку на самом деле не создаем ресурс, и клиенту достаточно знать, что отмена была успешно обработана;
- `POST /orders/{order_id}/pay: 200 (OK)` — хотя это эндпоинт `POST`, мы используем статус-код 200 (`OK`), поскольку на самом деле не создаем ресурс, и клиенту достаточно знать, что платеж был успешно обработан.

С успешными ответами все понятно, но как насчет ответов с ошибками? С какими ошибками мы можем столкнуться на сервере при обработке запросов и какие статус-коды HTTP им соответствуют? Выделяют две группы ошибок:

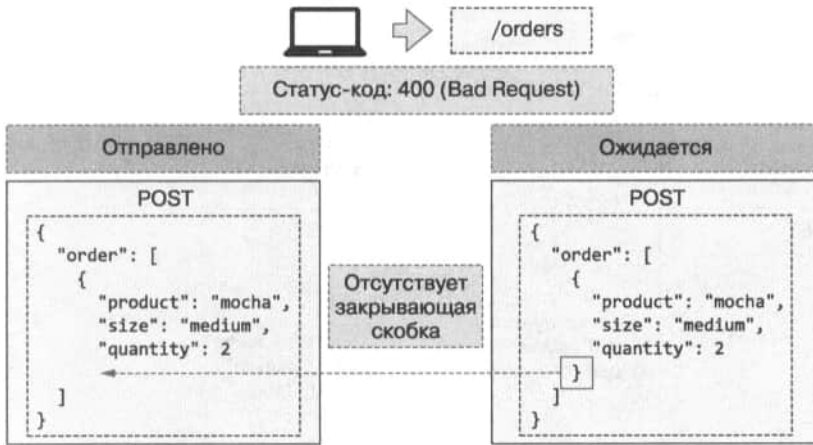
- ошибки, допущенные пользователем при отправке запроса, например, из-за неправильно сформированной полезной нагрузки или из-за того, что запрос был отправлен в несуществующий эндпоинт. Мы указываем на этот тип ошибок с помощью статус-кода HTTP из группы 4xx;
- ошибки, неожиданно возникающие на сервере при обработке запроса, как правило, из-за ошибки в нашем коде. Здесь мы используем статус-коды HTTP из группы 5xx.

Поговорим о каждом из этих типов ошибок более подробно.

#### 4.6.2. Использование статус-кодов HTTP для сообщения об ошибках клиента в запросе

Клиент API может совершать различные ошибки при отправке запроса к API. Чаще всего это отправка на сервер некорректной полезной нагрузки. Различают два типа такой полезной нагрузки: полезная нагрузка с недопустимым синтаксисом и необработываемые объекты.

Полезная нагрузка с *недопустимым синтаксисом* — это информация, которую сервер не может ни парсировать, ни понять. Типичным примером такой полезной нагрузки является неправильно сформированный JSON. Как показано на рис. 4.8, для устранения ошибок такого рода используется статус-код 400 (Bad Request).



**Рис. 4.8.** Пример использования статус-кода 400 (Bad Request)

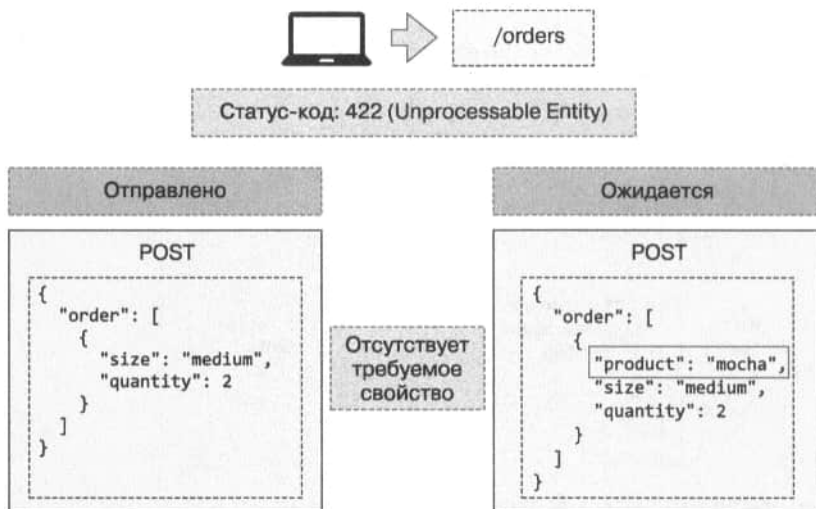
*Необрабатываемые объекты* — это синтаксически допустимая полезная нагрузка, в которой отсутствует необходимый параметр, содержатся недопустимые параметры или параметру присваивается неправильное значение или тип. Предположим, для размещения заказа наш API ожидает POST-запрос по URL-адресу `/orders` с такой полезной нагрузкой:

```
{
  "order": [
    {
      "product": "mocha",
      "size": "medium",
      "quantity": 2
    }
  ]
}
```

То есть мы ожидаем, что пользователь пришлет нам список элементов, где каждый элемент представляет собой пункт заказа и описывается следующими свойствами:

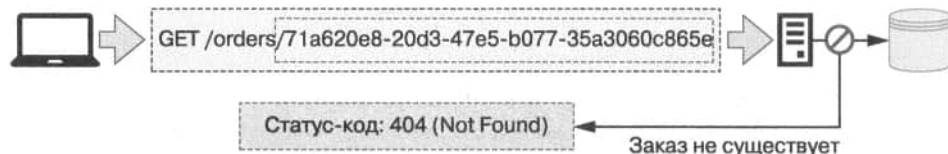
- **product** — идентифицирует продукт, заказанный пользователем;
- **size** — указывает размер (объем, если касается напитков), который применяется к заказанному товару;
- **quantity** — сообщает нам, сколько единиц одного и того же товара и какого размера товар пользователь хочет заказать.

Как показано на рис. 4.9, клиент API может отправить полезную нагрузку, в которой отсутствует одно из необходимых свойств, например `product`. Мы устраняем этот тип ошибки с помощью статус-кода 422 (Unprocessable Entity), сигнализирующего о том, что с запросом что-то не так и он не может быть обработан.



**Рис. 4.9.** Пример использования статус-кода 400 (Bad Request)

Другая распространенная ошибка — когда клиент API запрашивает несуществующий ресурс. Например, мы знаем, что endpoint `GET /orders/{order_id}` предоставляет подробную информацию о заказе. Если клиент использует этот endpoint с несуществующим идентификатором заказа, мы должны ответить статус-кодом HTTP, сигнализирующим о том, что заказ не существует. Как показано на рис. 4.10, мы отвечаем на эту ошибку статус-кодом 404 (Not Found), который означает, что запрошенный ресурс недоступен или не может быть найден.

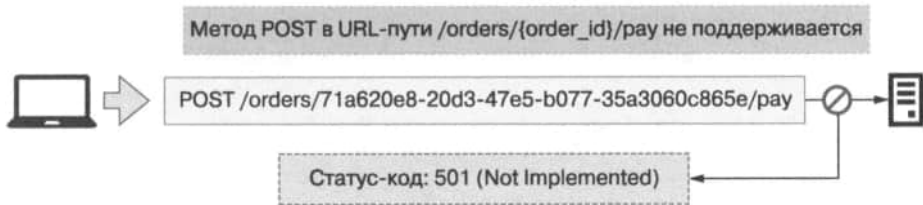


**Рис. 4.10.** Пример использования статус-кода 404 (Not Found)

Случается, что клиенты API отправляют запрос с использованием метода HTTP, который не поддерживается. Например, если пользователь отправил запрос PUT

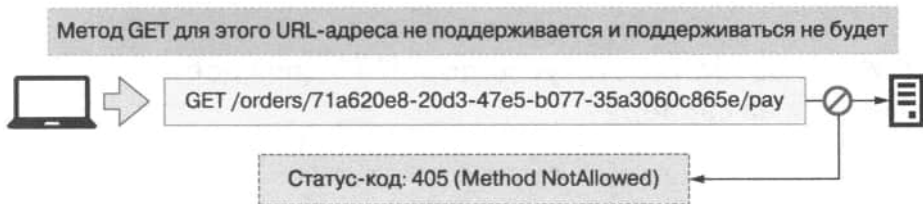


к эндпоинту `/orders`, мы должны сообщить ему, что метод `PUT` на этом URL-адресе не поддерживается. Есть два статус-кода, которые можно использовать в данном случае. Можно вернуть `501 (Not Implemented)`, если метод еще не реализован, но будет доступен в будущем (то есть у нас есть план по его реализации) (рис. 4.11).



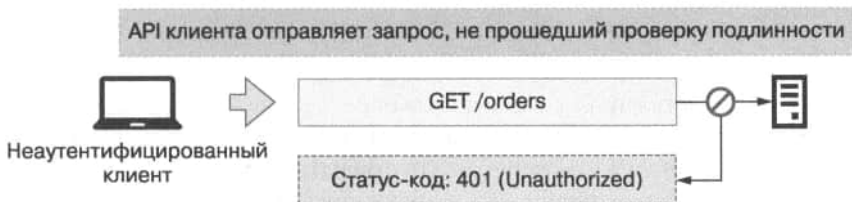
**Рис. 4.11.** Пример использования статус-кода 501 (Not Implemented)

Если запрошенный метод HTTP недоступен и у нас нет плана по его реализации, мы отвечаем статус-кодом `405 (Method Not Allowed)` (рис. 4.12).



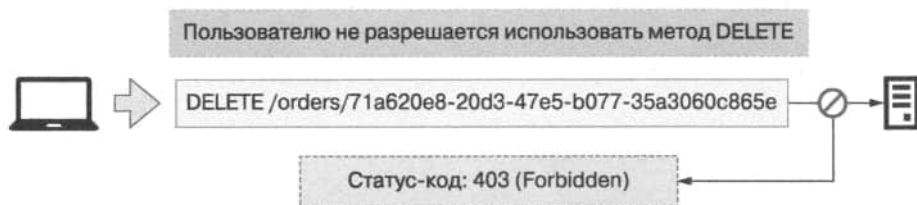
**Рис. 4.12.** Пример использования статус-кода 405 (Method Not Allowed)

Две распространенные ошибки в запросах API связаны с аутентификацией и авторизацией. Первая происходит, когда клиент посылает неаутентифицированный запрос к защищенному эндпоинту. В этом случае мы должны сообщить ему, что сначала он должен пройти процесс аутентификации. Как вы можете видеть на рис. 4.13, мы решаем эту ситуацию с помощью статус-кода `401 (Unauthorized)`, который сигнализирует о том, что пользователь не прошел аутентификацию.



**Рис. 4.13.** Пример использования статус-кода 401 (Unauthorized)

Вторая ошибка возникает, когда пользователь правильно аутентифицирован и пытается использовать endpoint или ресурс, для доступа к которому у него нет прав. Например, пытается получить доступ к деталям чужого заказа. Как показано на рис. 4.14, в таком случае подойдет статус-код 403 (Forbidden), который сигнализирует о том, что у пользователя нет прав на доступ к запрашиваемому ресурсу или выполнение запрашиваемой операции.

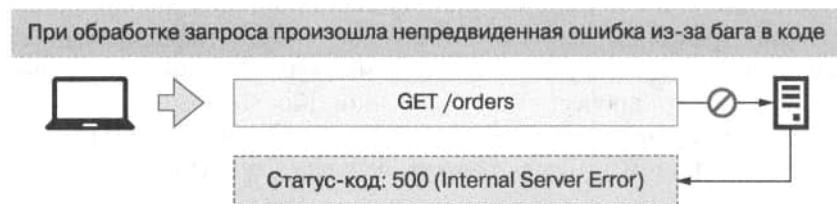


**Рис. 4.14.** Пример использования статус-код 403 (Forbidden)

Теперь, когда мы знаем, как применять статус-коды HTTP для сообщения об ошибках пользователей, перейдем к статус-кодам ошибок сервера.

### 4.6.3. Использование статус-кодов HTTP для сообщения об ошибках на сервере

Вторая группа ошибок — ошибки, возникающие на сервере. Они возможны из-за багов в коде или ограничений в инфраструктуре. Наиболее распространенный вариант — неожиданный сбой приложения из-за ошибки. В таких ситуациях мы отвечаем статус-кодом 500 (Internal Server Error) (рис. 4.15).



**Рис. 4.15.** Пример использования статус-кода 500 (Internal Server Error)

Схожий тип ошибки — когда приложение не может обслуживать запросы. Обычно с этой ситуацией можно справиться с помощью прокси-сервера или API-шлюза (см. подраздел 4.2.4). Наш API может перестать отвечать на запросы, если сервер перегружен или находится на техническом обслуживании, и мы должны сообщить об этом пользователю, отправив информативный статус-код. Возможны два сценария.

- Когда сервер не может принимать новые соединения, мы должны ответить статус-кодом 503 (Service Unavailable), который означает, что сервер перегружен или находится на техническом обслуживании и поэтому не может обслуживать дополнительные запросы (рис. 4.16).

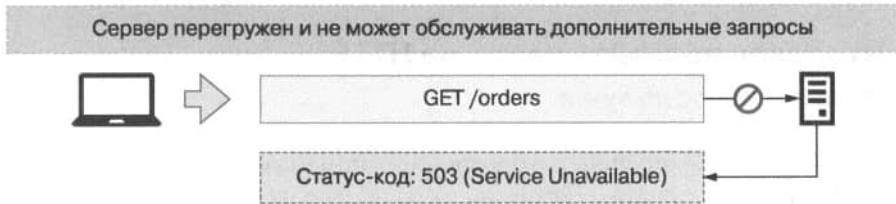


Рис. 4.16. Пример использования статус-кода 503 (Service Unavailable)

- Когда серверу требуется слишком много времени для ответа на запрос, мы отвечаем статус-кодом 504 (Gateway Timeout) (рис. 4.17).

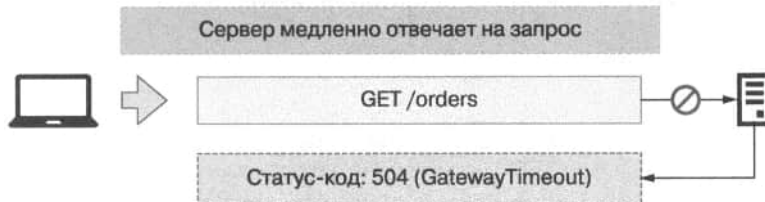


Рис. 4.17. Пример использования статус-кода 504 (Gateway Timeout)

На этом мы завершаем обзор статус-кодов HTTP, которые чаще всего применяются при разработке веб-API. Правильно используя статус-коды, вы обеспечите хорошее взаимодействие разработчиков с вашими клиентами API, но есть еще одна вещь, которую мы должны безупречно спроектировать: полезная нагрузка API. В следующем разделе мы обратимся к этой важной теме.

## 4.7. ПРОЕКТИРОВАНИЕ ПОЛЕЗНОЙ НАГРУЗКИ API

В этом разделе приводятся рекомендации по разработке удобной для пользователя полезной нагрузки HTTP-запросов и ответов. *Полезная нагрузка (payload)* — это данные, которыми клиент и сервер обмениваются посредством HTTP-запроса. Мы отправляем полезную нагрузку на сервер, когда хотим создать или обновить ресурс, а сервер отправляет нам ее, когда мы запрашиваем данные. Удобство использования API во многом зависит от качества полезной нагрузки. Непродуманная

структура полезной нагрузки запроса делают API сложным в применении и ухудшают опыт пользователя. Таким образом, важно потратить немного времени на проработку структуры полезной нагрузки запроса, и в этом разделе я перечислю паттерны и лучшие практики, которые помогут вам в решении этой задачи<sup>1</sup>.

### 4.7.1. Что такое полезная нагрузка HTTP и когда мы ее используем

HTTP-запрос — это сообщение, которое клиент приложения отправляет веб-серверу, а HTTP-ответ — это ответ сервера на запрос. *HTTP-запрос* включает в себя URL-адрес, метод HTTP, набор заголовков и по желанию тело или полезную нагрузку. В HTTP-заголовках приводятся метаданные о содержимом запроса, например формат кодировки. Аналогично *HTTP-ответ* включает в себя статус-код, набор заголовков и по желанию полезную нагрузку, которую можно предоставлять с помощью различных методов сериализации данных, таких как XML и JSON. В REST API данные обычно представляются в виде документа JSON.

#### ОПРЕДЕЛЕНИЕ

*Тело HTTP-сообщения, или полезная нагрузка*, — это сообщение, содержащее данные, которыми обмениваются в запросе HTTP. Как запросы, так и ответы HTTP могут содержать тело сообщения. Оно кодируется в одном из типов медиа, поддерживаемых HTTP, в частности XML или JSON. Заголовок Content-Type HTTP-запроса сообщает медиатип сообщения. В REST API тело сообщения обычно в формате JSON.

В HTTP-запросы включают полезную нагрузку, когда нужно отправить данные на сервер. Например, запрос POST обычно отправляет данные для создания ресурса. Согласно спецификации HTTP полезную нагрузку стоит включать во все методы HTTP, но не рекомендуется — в запросы GET (<http://mng.bz/O69K>) и DELETE (<http://mng.bz/YKeo>).

Формулировка спецификации HTTP намеренно расплывчата в отношении того, могут ли запросы DELETE и GET включать полезную нагрузку. В них не запрещено использовать полезную нагрузку, но она не имеет определенной семантики. Известный пример — Elasticsearch, который позволяет клиентам отправлять документы в теле GET-запроса (<http://mng.bz/G14M>).

<sup>1</sup> Помимо изучения лучших практик, вам будет полезно прочитать об антипаттернах. В моей статье *How Bad Models Ruin an API (or Why Design-First is the Way to Go)* («Как плохие модели разрушают API (или почему самый первый прототип — это путь к успеху)») приводится обзор распространенных антипаттернов, которых следует избегать (<https://www.microapis.io/blog/how-bad-models-ruin-an-api>).

А как насчет HTTP-ответов? Они могут включать полезную нагрузку в зависимости от статус-кода. Согласно спецификации HTTP, не должны содержать полезную нагрузку ответы со статус-кодом из группы 1xx, а также с кодами 204 (No Content) и 304 (Not Modified). Все остальные ответы — могут. В контексте REST API наиболее важной полезной нагрузкой является та, что включена в ответы об ошибках 4xx и 5xx, а также в ответы 2xx об успешном завершении, кроме статус-кода 204. В следующем разделе вы научитесь разрабатывать корректную полезную нагрузку для всех этих ответов.

### 4.7.2. Модели проектирования полезной нагрузки HTTP

Здесь мы сосредоточимся на разработке полезной нагрузки ответов, поскольку они более разнообразны. Как мы узнали в подразделе 4.6.1, есть два вида ответов: об ошибке и успешном завершении. Полезная нагрузка ответов об ошибках должна включать ключевое слово "error" с подробным описанием, почему клиент получает сообщение об ошибке. Например, ответ 404, который генерируется, когда запрошенный ресурс не может быть найден на сервере, может содержать следующее сообщение об ошибке:

```
{  
  "error": "Resource not found"  
}
```

"error" является общепринятым ключевым словом для сообщений об ошибках, но можно использовать и другие, например "detail" и "message". Большинство фреймворков для веб-разработки обрабатывают ошибки HTTP и имеют шаблоны по умолчанию для ответов об ошибках. Например, FastAPI использует "detail", поэтому мы добавим это ключевое слово в спецификацию API сервиса заказов.

В случае успешных ответов мы выделяем три сценария: когда создаем ресурс, обновляем его и получаем подробную информацию о нем. Посмотрим, как разрабатывать ответы для каждого из этих сценариев.

#### Полезная нагрузка ответов для POST-запросов

Для создания ресурсов мы используем POST-запросы. В API сервиса заказов CoffeeMesh мы размещаем заказы через эндпоинт POST /orders. Чтобы оформить заказ, мы отправляем список желаемых товаров на сервер, который присваивает уникальный идентификатор заказу, и, следовательно, ID заказа должен быть возвращен в полезной нагрузке ответа. Сервер также устанавливает время, когда заказ был принят, и его начальный статус. Свойства, определенные сервером, называются *свойствами на стороне сервера* или *свойствами только для чтения*, и их нужно включать в полезную нагрузку ответа. Хорошая практика — возвращение полного

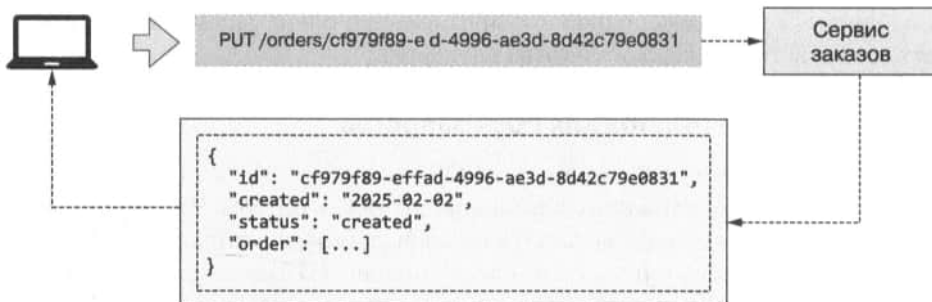
представления ресурса в ответе на POST-запрос (рис. 4.18). Такая полезная нагрузка подтверждает, что ресурс был создан правильно.



**Рис. 4.18.** Когда клиент API отправляет POST-запрос на создание нового ресурса, сервер отвечает полным представлением только что созданного ресурса с его идентификатором и любыми другими свойствами, установленными сервером

### Полезная нагрузка ответов для PUT- и PATCH-запросов

Чтобы обновить ресурс, используем PUT- или PATCH-запрос. В разделе 4.5 мы делали запросы PUT/PATCH на URI одиночного ресурса, такого как эндпоинт PUT /orders/{order\_id} API сервиса заказов CoffeeMesh. В этом случае хорошая практика — возвращать полное представление ресурса, по которому клиент может проверить правильность выполнения обновления (рис. 4.19).

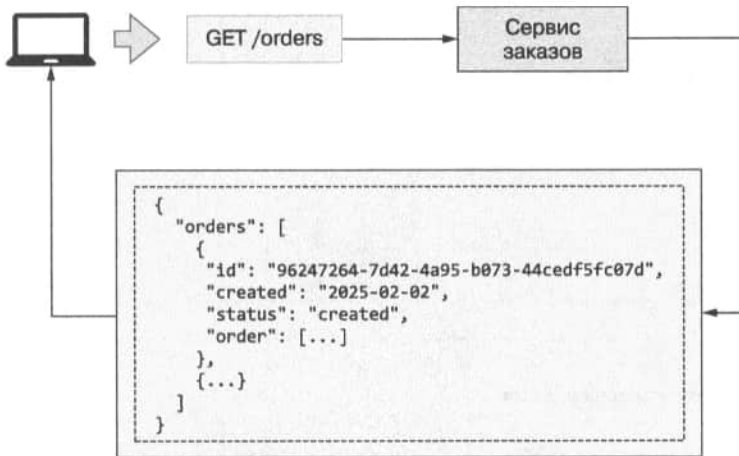


**Рис. 4.19.** Когда клиент API отправляет запрос PUT для обновления ресурса, сервер отвечает полным представлением ресурса

## Полезная нагрузка ответов для GET-запросов

Ресурсы с сервера мы получаем с помощью GET-запросов. Как мы выяснили в разделе 4.5, API сервиса заказов CoffeeMesh предоставляет два эндпоинта GET: `/orders` и `/orders/{orders_id}`. Посмотрим, какие есть варианты полезной нагрузки ответов для этих эндпоинтов.

Функция GET `/orders` возвращает список заказов. При проектировании содержимого списка можно включить полное или частичное представление каждого заказа. Как показано на рис. 4.20, в первом случае мы предоставляем клиенту API всю необходимую информацию в одном запросе. Однако это может снизить производительность API: когда элементы в списке имеют большой размер, увеличивается полезная нагрузка ответа.



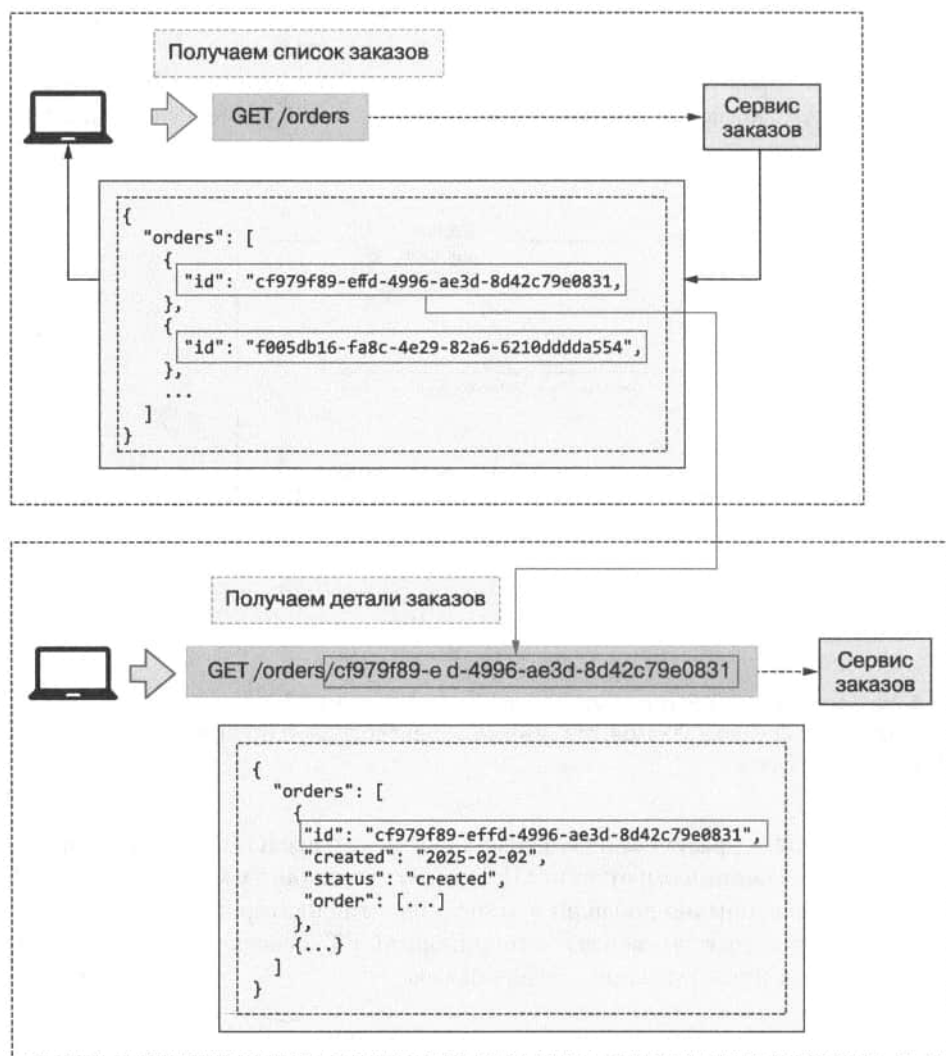
**Рис. 4.20.** Когда клиент API отправляет запрос к эндпоинту GET `/orders`, сервер отвечает списком заказов, где каждый объект содержит полную информацию о заказе

Вторая стратегия подразумевает включение частичного представления каждого заказа (рис. 4.21). Например, в ответ на GET-запрос к эндпоинту коллекции, такому как GET `/orders`, обычно добавляют только идентификатор каждого элемента. В этой ситуации клиент должен вызвать эндпоинт GET `/orders/{order_id}`, чтобы получить полное представление каждого заказа.

Какой подход лучше? Зависит от ситуации. В публичных API чаще встречается подход с отправкой полного представления ресурса. Однако, если вы работаете над внутренним API и полная информация о каждом элементе не требуется, можете сократить полезную нагрузку, добавив в нее только те свойства, которые

нужны клиенту. Полезная нагрузка небольшого размера быстрее обрабатывается, что приводит к улучшению пользовательского опыта. Наконец, одиночные энд-поинты, такие как `GET /orders/{order_id}`, всегда должны возвращать полное представление ресурса.

Теперь, зная, как разрабатывать полезную нагрузку API, обратим внимание на параметры запроса URL.



**Рис. 4.21.** Когда клиент API делает GET-запрос по URL-пути `/orders`, сервер отвечает списком идентификаторов заказов. Клиент использует эти идентификаторы для запроса деталей каждого заказа в эндпоинте `GET /orders/{order_id}`



## 4.8. ПРОЕКТИРОВАНИЕ ПАРАМЕТРОВ ЗАПРОСА URL

Рассмотрим параметры запроса URL: что это, как, зачем и когда их следует использовать. Некоторые эндпоинты, такие как GET /orders API сервиса заказов, возвращают список ресурсов. Когда эндпоинт возвращает список ресурсов, рекомендуется разрешить пользователям фильтровать результаты и запрашивать их постранично. Например, при использовании GET /orders можно ограничить результаты только пятью последними заказами или перечислить лишь отмененные заказы. Это позволяют сделать параметры запроса URL. Они должны быть заданы как необязательные, и при необходимости сервер может присваивать им значения по умолчанию.

### ОПРЕДЕЛЕНИЕ

*Параметры запроса URL (URL query parameters)* — это параметры «ключ — значение» в URL-адресе. Параметры указываются после знака вопроса (?) и обычно предназначены для фильтрации результатов эндпоинта. Можно объединить несколько параметров запроса, разделив их амперсандами (&).

Например, если мы хотим выполнить запрос GET /orders и отфильтровать результаты по отмененным заказам, то можем написать следующее:

```
GET /orders?cancelled=true
```

Можно связать несколько параметров запроса в одном URL, разделяя их амперсандами. Добавим параметр запроса с именем limit к эндпоинту GET /orders, чтобы ограничить количество результатов. Чтобы отфильтровать эндпоинт GET /orders по отмененным заказам и ограничить количество результатов до пяти, отправляем следующий API-запрос:

```
GET /orders?cancelled=true&limit=5
```

Также распространенной практикой является предоставление клиентам API возможности постраничного просмотра результатов. Суть пагинации заключается в разбишке результата на страницы и предоставлении одной из них за раз. Чаще всего используют комбинацию параметров page и per\_page: page представляет собой набор данных, а per\_page указывает, сколько элементов следует включить в каждый набор. По значению per\_page сервер определяет, сколько наборов данных мы получим. Объединим оба параметра в запросе API:

```
GET /orders?page=1&per_page=10
```

На этом мы завершаем знакомство с лучшими практиками и принципами проектирования REST API. Теперь у вас есть все необходимые ресурсы для разработки высокоинформативных и структурированных REST API, которые легко понять и использовать. В следующей главе вы научитесь документировать свои проекты API с помощью стандарта OpenAPI.

## РЕЗЮМЕ

- REST определяет следующие принципы проектирования API.
  - *Клиент-серверная архитектура* — клиентский и серверный код должны быть разделены.
  - *Отсутствие записи состояния клиента* — сервер не должен сохранять состояние между запросами.
  - *Кэшируемость* — запросы должны быть кэшированы, если это возможно.
  - *Многоуровневость системы* — архитектурная сложность серверной части не должна быть видна конечным пользователям.
  - *Предоставление кода по запросу* — клиентские приложения могут загружать исполняемый код с сервера.
  - *Единство интерфейса* — API должен обеспечивать единообразный и согласованный интерфейс.
- Гипермедиа как механизм состояния приложений (hypermedia as the engine of application state, HATEOAS) — это парадигма, которая утверждает, что REST API должны включать в свои ответы связанные ссылки. HATEOAS упрощает навигацию по API и их использование.
- В хорошем проекте REST API используются возможности протокола HTTP, такие как методы HTTP и статус-коды, для создания структурированных и высокоинформативных API.
- Наиболее важными методами HTTP для REST API являются:
  - GET — для получения ресурсов с сервера;
  - POST — для создания новых ресурсов;
  - PUT и PATCH — для обновления ресурсов;
  - DELETE — для удаления ресурсов.
- Мы обмениваемся данными с сервером API, используя полезную загрузку внутри запроса. Полезная нагрузка находится в теле HTTP-запроса или ответа. Клиенты отправляют полезную нагрузку запроса через HTTP-методы POST, PUT и PATCH. Ответы сервера всегда содержат полезную нагрузку, за исключением случаев со статус-кодами 204, 304 или кодами из группы 1xx.
- Параметры запроса URL — это пары «ключ — значение» в URL-адресе, и мы используем их для фильтрации, страничного просмотра и сортировки результатов эндпоинта GET.

# Документирование *REST API* с помощью *OpenAPI*

---

## В этой главе

- ✓ Использование JSON Schema для создания моделей проверки JSON-документов.
- ✓ Описание REST API с использованием стандарта документации OpenAPI.
- ✓ Моделирование полезной нагрузки для запросов и ответов API.
- ✓ Создание повторно используемых схем в спецификациях OpenAPI.

В этой главе вы научитесь документировать API с помощью OpenAPI: самого популярного стандарта для описания RESTful API, с богатой экосистемой инструментов для тестирования, валидации и визуализации API. Большинство языков программирования включают в себя библиотеки, поддерживающие спецификации OpenAPI, и в главе 6 вы научитесь использовать библиотеки из экосистемы Python, совместимые с OpenAPI.

OpenAPI использует JSON Schema для описания структуры и моделей API, поэтому начнем с обзора того, как работает этот стандарт. JSON Schema — это спецификация для определения структуры документа JSON, в том числе типов и форматов значений в документе.

После знакомства с JSON Schema мы рассмотрим, как строится документ OpenAPI, каковы его свойства и как его использовать для предоставления информативных спецификаций API нашим потребителям. Эндпоинты API составляют ядро

спецификации, поэтому мы уделим им особое внимание. Процесс определения эндпоинтов и схем для полезной нагрузки запросов и ответов API рассматривается шаг за шагом. В примерах этой главы мы будем работать с API сервиса заказов CoffeeMesh. Как мы уже говорили в главе 1, CoffeeMesh — это вымышленная платформа для доставки кофе, а сервис заказов — это компонент, который позволяет клиентам размещать свои заказы и управлять ими. Полная спецификация API сервиса заказов доступна в файле `ch05/oas.yaml` в репозитории GitHub для этой книги.

## 5.1. ИСПОЛЬЗОВАНИЕ JSON SCHEMA ДЛЯ МОДЕЛИРОВАНИЯ ДАННЫХ

Рассмотрим стандарт спецификации JSON Schema. OpenAPI использует расширенное подмножество спецификации JSON Schema для определения структуры документов JSON, а также типов и форматов их свойств. Это полезно для документирования интерфейсов, использующих для представления данных формат JSON, и для проверки корректности данных, которыми обмениваются. Спецификация JSON Schema находится в стадии активной разработки, последняя версия — 2020-12<sup>1,2</sup>.

### ОПРЕДЕЛЕНИЕ

*JSON Schema* — это стандарт спецификации для определения структуры документа JSON, а также типов и форматов его свойств. OpenAPI использует JSON Schema для описания свойств API.

Спецификация JSON Schema обычно определяет объект с некоторыми атрибутами или свойствами. Объект JSON Schema представлен ассоциативным массивом пар «ключ — значение». Спецификация JSON Schema чаще всего выглядит следующим образом:

```
{
  "status": {
    "type": "string"
  }
}
```

Каждое свойство в спецификации JSON Schema поставляется в виде ключа, значения которого являются дескрипторами свойства

Минимальным дескриптором, необходимым для свойства, является тип. В данном случае мы указываем, что свойство `status` является строкой

<sup>1</sup> Wright A., Andrews H., Hutton B. JSON Schema: A Media Type for Describing JSON Documents («JSON Schema: Медиа-тип для описания документов JSON») (December 8, 2020), <https://datatracker.ietf.org/doc/html/draft-bhutton-json-schema-00>. Вы можете следить за развитием JSON Schema и вносить свой вклад в ее улучшение через ее репозиторий на GitHub: <https://github.com/json-schema-org/json-schema-spec>. См. также веб-сайт проекта: <https://json-schema.org/>.

<sup>2</sup> Десятого июня 2022 года к этой версии выпущен патч, не изменяющий функциональности. Новые идентификаторы документов IETF имеют форму `draft-bhutton-*-01`. — *Примеч. ред.*

В этом примере мы определяем схему объекта с одним атрибутом `status`, тип которого — `string`.

JSON Schema позволяет быть очень точными в отношении типов и форматов данных, которые сервер и клиент должны ожидать от полезной нагрузки. Это важно для интеграции между API-провайдером и API-потребителем, поскольку мы понимаем, как парсировать полезную нагрузку и преобразовывать ее в нужные типы данных в нашей среде выполнения.

JSON Schema поддерживает следующие основные типы данных:

- `string` (строка) для символьных значений;
- `number` (число) для целочисленных и десятичных значений;
- `object` (объект) для ассоциативных массивов (например, словарей в Python);
- `array` (массив) для коллекций других типов данных (то есть списков в Python);
- `boolean` (булево значение) для значений `true` или `false`;
- `null` для неинициализированных данных.

Чтобы определить объект с помощью JSON Schema, мы объявляем его тип как `object` и перечисляем свойства и их типы. В листинге 5.1 показано, как мы определяем объект с именем `order`, который является одной из основных моделей API сервиса заказов.

#### Листинг 5.1. Определение схемы объекта с помощью JSON Schema

```
{
  "order": {
    "type": "object",
    "properties": {
      "product": {
        "type": "string"
      },
      "size": {
        "type": "string"
      },
      "quantity": {
        "type": "integer"
      }
    }
  }
}
```

Мы можем объявить  
схему как объект

Описываем свойства  
объекта с помощью  
ключевого слова `properties`

Поскольку `order` является объектом, он имеет свойства, определенные в атрибуте `properties`. У каждого свойства собственный тип. JSON-документ, соответствующий спецификации из листинга 5.1, выглядит следующим образом:

```
{
  "order": {
    "product": "coffee",
```

```

    "size": "big",
    "quantity": 1
  }
}

```

Как видите, в этом документе приводится каждое свойство, описанное в спецификации, и каждое из них имеет ожидаемый тип.

Свойство также может представлять собой массив элементов. Например, в листинге 5.2 объект `order` — это массив объектов. Для определения элементов в массиве используется ключевое слово `items`.

### Листинг 5.2. Определение массива объектов с помощью JSON Schema

```

{
  "order": {
    "type": "array",
    "items": {
      "type": "object",
      "properties": {
        "product": {
          "type": "string"
        },
        "size": {
          "type": "string"
        },
        "quantity": {
          "type": "integer"
        }
      }
    }
  }
}

```

← Определяем элементы массива с помощью ключевого слова `items`

Здесь свойство `order` является массивом. Для массивов нужно указывать дополнительное свойство в схеме — `items`, определяющее тип каждого элемента в массиве. В данном случае элементами массива являются объекты, представляющие детали заказа.

Объект может иметь любое количество вложенных объектов. Однако, когда вложенных объектов слишком много, отступы становятся большими, что затрудняет чтение спецификации. Чтобы избежать этой проблемы, JSON Schema позволяет определять каждый объект отдельно и использовать JSON-указатели для ссылок на них. *Указатель JSON* (JSON pointer) — это специальный синтаксис, который позволяет указывать на другое определение объекта в рамках той же спецификации.

Как видно из листинга 5.3, мы можем извлечь определение каждого элемента в массиве `order` в виде модели `OrderItemSchema` и использовать указатель JSON для ссылки на нее с помощью специального ключевого слова `$ref`.

**Листинг 5.3.** Использование указателей JSON для ссылок на другие схемы

```
{
  "OrderItemSchema": {
    "type": "object",
    "properties": {
      "product": {
        "type": "string"
      },
      "size": {
        "type": "string"
      },
      "quantity": {
        "type": "integer"
      }
    }
  },
  "Order": {
    "status": {
      "type": "string"
    },
    "order": {
      "type": "array",
      "items": {
        "$ref": '#/OrderItemSchema'
      }
    }
  }
}
```

Мы можем указать тип элементов массива с помощью указателя JSON

В указателях JSON используется специальное ключевое слово `$ref` и синтаксис `JSONPath` для ссылки на другое определение в схеме. В синтаксисе `JSONPath` корень документа обозначается символом хештега (`#`), а связь вложенных свойств — слешами (`/`). Например, если нужно создать указатель на свойство `size` модели `OrderItemSchema`, используем следующий синтаксис: `'#/OrderItemSchema/size'`.

**ОПРЕДЕЛЕНИЕ**

*Указатель JSON* — это специальный синтаксис в JSON Schema, который позволяет указывать на другое определение в той же спецификации. Для объявления указателя JSON предназначено специальное ключевое слово `$ref`. Чтобы построить путь к другой схеме, мы используем синтаксис `JSONPath`. Например, чтобы указать на схему `OrderItemSchema`, определенную на верхнем уровне документа, пишем так: `{"$ref": "#/OrderItemSchema"}`.

Мы можем реорганизовать нашу спецификацию с помощью указателей JSON, извлекая общие объекты схемы в повторно используемые модели, и ссылаться на них с помощью указателей JSON. Это поможет избежать дублирования и сохранить спецификацию чистой и лаконичной.

Помимо типа, JSON Schema позволяет указать формат свойства. Можно разработать собственные пользовательские форматы или использовать кастомные форматы JSON Schema. Например, для свойства, представляющего дату, мы можем использовать `date` — встроенный формат, поддерживаемый JSON Schema, который обозначает дату ISO<sup>1</sup> (например, 2025-05-21). Вот пример:

```
{
  "created": {
    "type": "string",
    "format": "date"
  }
}
```

В этом разделе мы работали с примерами в формате JSON. Однако документы JSON Schema не обязательно должны быть написаны в JSON-формате. На самом деле чаще всего они пишутся в формате YAML, поскольку он более читабелен и прост для понимания. Спецификация OpenAPI также обычно предоставляется в формате YAML, и далее в главе мы будем использовать именно его для разработки спецификации API сервиса заказов.

## 5.2. СТРУКТУРА СПЕЦИФИКАЦИИ OPENAPI

В этом разделе мы познакомимся со стандартом OpenAPI и научимся структурировать спецификацию API. Последняя версия OpenAPI — 3.1, однако на момент написания книги она все еще имеет мало поддержки в текущей экосистеме, поэтому будем документировать API, используя OpenAPI 3.0. Разница между этими версиями невелика, и почти все, что вы узнаете о OpenAPI 3.0, применимо и к 3.1.<sup>2</sup>

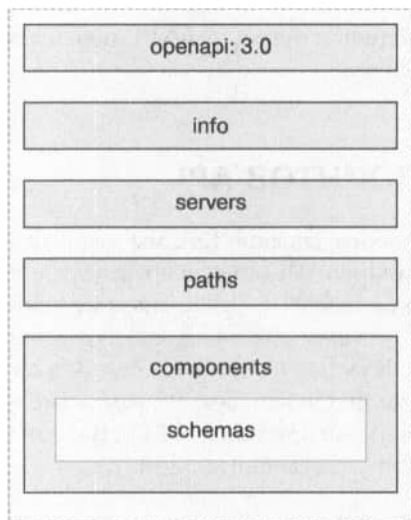
OpenAPI — это стандартный формат спецификации для документирования RESTful API (рис. 5.1). Он позволяет подробно описать каждый элемент API, включая его эндпоинты, формат полезной нагрузки запросов и ответов, схемы безопасности и т. д. OpenAPI был создан в 2010 году под названием Swagger как формат спецификации с открытым исходным кодом для описания веб-RESTful API. Со временем популярность этого формата росла, и в 2015 году Linux Foundation и консорциум крупных компаний спонсировали создание инициативы OpenAPI — проекта, направленного на совершенствование протоколов и стандартов для создания RESTful API. Сегодня OpenAPI — самый популярный формат спецификаций, используемых

<sup>1</sup> Дата ISO — это строковый формат, который может содержать дату, время, миллисекунды и часовой пояс и понятен для человека. Официальный формат даты схемы JSON является подмножеством стандарта ISO 8601.

<sup>2</sup> Для понимания различий между OpenAPI 3.0 и 3.1 ознакомьтесь с руководством OpenAPI по переходу с 3.0 на 3.1: <https://www.openapis.org/blog/2021/02/16/migrating-from-openapi-3-0-to-3-1-0>.



для документирования RESTful API<sup>1</sup>, который опирается на богатую экосистему инструментов для визуализации, тестирования и валидации API.



**Рис. 5.1.** Спецификация OpenAPI содержит пять разделов. Например, раздел `paths` описывает эндпоинты API, а раздел `components` содержит схемы повторного использования, на которые ссылаются во всем документе

Спецификация OpenAPI содержит все, что потребителю API необходимо знать, чтобы иметь возможность взаимодействовать с API. Как показано на рис. 5.1, OpenAPI состоит из пяти разделов:

- `openapi` — указывает версию OpenAPI, которая использовалась для создания спецификации;
- `info` — содержит общую информацию, такую как название и версия API;
- `servers` — содержит список URL-адресов, по которым доступен API. Можно перечислить несколько URL для различных сред, например для производственной и среды обкатки;
- `paths` — описывает эндпоинты, открытые API, включая ожидаемую полезную нагрузку, допустимые параметры и формат ответов. Это самая важная часть спецификации, поскольку она представляет API, и именно этот раздел будут искать потребители, чтобы узнать, как интегрироваться с интерфейсом;
- `components` — определяет повторно используемые элементы, на которые есть ссылки в спецификации. Сюда относятся схемы, параметры, схемы безопасно-

<sup>1</sup> Согласно отчету State of the API («Состояние API») за 2022 год, подготовленному компанией Postman (<https://www.postman.com/state-of-api/api-technologies/#api-technologies>).

сти, тела запросов и ответов<sup>1</sup>. *Схема* — это определение ожидаемых атрибутов и типов в объектах запроса и ответа. Схемы OpenAPI определяются с использованием синтаксиса JSON Schema.

Теперь, разобравшись, как структурировать спецификацию OpenAPI, перейдем к документированию эндпоинтов API сервиса заказов.

### 5.3. ДОКУМЕНТИРОВАНИЕ ЭНДПОИНТОВ API

В этом разделе мы объявим эндпоинты API сервиса заказов. Как мы уже говорили в разделе 5.2, раздел `paths` спецификации OpenAPI описывает интерфейс API. В нем перечислены URL-адреса, предоставляемые API, с указанием методов HTTP, которые они реализуют, типов запросов, которые они ожидают, и ответов, которые они возвращают, включая статус-коды. Каждый путь — это объект, а его атрибутами являются HTTP-методы, которые он поддерживает. В этом разделе мы сосредоточимся на документировании URL-адресов и методов HTTP. В главе 4 мы установили, что API сервиса заказов содержит следующие эндпоинты:

- `POST /orders` — размещение заказа. Требуется полезная нагрузка с деталями заказа;
- `GET /orders` — возвращение списка заказов. Принимает параметры запроса URL, которые позволяют фильтровать результаты;
- `GET /orders/{order_id}` — вывод информации о конкретном заказе;
- `PUT /orders/{order_id}` — обновление сведений о заказе. Поскольку это эндпоинт PUT, требуется полное представление заказа;
- `DELETE /orders/{order_id}` — удаление заказа;
- `POST /orders/{order_id}/pay` — оплата заказа;
- `POST /orders/{order_id}/cancel` — отмена заказа.

В листинге 5.4 показаны определения эндпоинтов API сервиса заказов. Мы объявляем URL-адреса и HTTP-методы, реализуемые каждым URL, и добавляем идентификатор операции к каждому эндпоинту, чтобы иметь возможность ссылаться на них в других разделах документа.

Теперь, когда у нас есть эндпоинты, нужно заполнить детали. Для эндпоинта `GET /orders` следует описать параметры, которые он принимает, а для эндпоинтов `POST` и `PUT` — полезную нагрузку запроса. Нам также нужно описать ответы для каждого эндпоинта. В следующих разделах мы разберемся, как

---

<sup>1</sup> Полный список повторно используемых элементов, которые могут быть определены в разделе компонентов спецификации API, см. на сайте <https://swagger.io/docs/specification/components/>.

создавать спецификации для различных элементов API, начиная с параметров запроса URL.

**Листинг 5.4.** Высокоуровневое определение эндпоинтов API сервиса заказов

```
paths:
  /orders:  ← Объявляем путь URL
    get:    ← Поддерживаемый метод HTTP по URL-адресу /orders
      operationId: getOrders
    post: # creates a new order
      operationId: createOrder

  /orders/{order_id}:
    get:
      operationId: getOrder
    put:
      operationId: updateOrder
    delete:
      operationId: deleteOrder

  /orders/{order_id}/pay:
    post:
      operationId: payOrder

  /orders/{order_id}/cancel:
    post:
      operationId: cancelOrder
```

## 5.4. ДОКУМЕНТИРОВАНИЕ ПАРАМЕТРОВ ЗАПРОСА URL

Как вы узнали в главе 4, параметры запроса URL позволяют фильтровать и сортировать результаты эндпоинта GET. В этом разделе мы разберем, как определять параметры запроса URL с помощью OpenAPI. Эндпоинт GET `/orders` позволяет фильтровать заказы с помощью таких параметров:

- `cancelled` — определяет, был ли отменен заказ. Это значение будет булевым;
- `limit` — указывает максимальное количество заказов, которые должны быть возвращены пользователю. Значением этого параметра будет число.

Как `cancelled`, так и `limit` можно комбинировать в одном запросе для фильтрации результатов:

```
GET /orders?cancelled=true&limit=5
```

Этот запрос запрашивает у сервера список из пяти отмененных заказов. В листинге 5.5 показана спецификация параметров запроса эндпоинта GET `/orders`. Для определения параметра требуется указать его имя (`name`) — значение, используемое

для ссылки на него в фактическом URL. Мы также указываем, к какому типу относится этот параметр. OpenAPI 3.1 различает четыре типа параметров: пути, запроса, HTTP-заголовок и cookie. Параметры заголовка передаются в поле заголовка HTTP, а параметры cookie — в полезную нагрузку cookie. Параметры пути являются частью URL-адреса и обычно используются для идентификации ресурса. Например, в `/orders/{order_id}` параметр пути — `order_id`. Он идентифицирует конкретный заказ. Параметры запроса — это необязательные параметры, которые позволяют фильтровать и сортировать результаты эндпоинта. Мы определяем тип параметра с помощью ключевого слова `schema` (булево значение для `cancelled` и число для `limit`) и, когда это уместно, также указываем его формат<sup>1</sup>.

### Листинг 5.5. Спецификация параметров запроса эндпоинта GET /orders

```
paths:
  /orders:
    get:
      parameters:
        - name: cancelled
          in: query
          required: false
          schema:
            type: boolean
        - name: limit
          in: query
          required: false
          schema:
            type: integer
```

Описываем параметры запроса URL в свойстве `parameters`

Имя параметра

Используем дескриптор `in`, чтобы указать, что параметр приводится в URL-пути

Указываем, является ли параметр обязательным

Определяем тип параметра в разделе `schema`

Итак, мы разобрались, как описывать параметры запроса URL. Далее займемся более сложной задачей — документированием полезной нагрузки запроса.

## 5.5. ДОКУМЕНТИРОВАНИЕ ПОЛЕЗНОЙ НАГРУЗКИ ЗАПРОСОВ

В главе 4 мы обсудили, что запрос представляет собой данные, которые клиент отправляет на сервер посредством POST- или PUT-запроса. В этом разделе мы научимся документировать полезную нагрузку запросов эндпоинтов API сервиса заказов. Начнем с метода POST `/orders`. В разделе 5.1 мы выяснили, что полезная нагрузка для эндпоинта POST `/orders` выглядит следующим образом:

```
{
  "order": [
    {
      "product": "cappuccino",
```

<sup>1</sup> Чтобы узнать больше о типах данных и форматах, доступных в OpenAPI 3.1, см: <http://spec.openapis.org/oas/v3.1.0#data-types>.

```

        "size": "big",
        "quantity": 1
    }
}

```

У нас есть атрибут `order`, который представляет собой массив элементов. Каждый элемент определяется тремя атрибутами и ограничениями:

- `product` — тип товара, который заказывает пользователь;
- `size` — размер товара. Это может быть один из трех следующих вариантов: `small`, `medium` и `big`;
- `quantity` — количество товаров. Это может быть любое целое число, равное или больше 1.

В листинге 5.6 показано, как определяется схема для этой полезной нагрузки. Полезную нагрузку запроса мы задаем в параметре `content` свойства `requestBody` метода. Ее можно задавать в различных форматах. В данном случае допускаются данные только в формате JSON, который имеет определение медиатипа `application/json`. Схема для нашей полезной нагрузки представляет собой объект с одним свойством: `order`, тип которого — `array`. Элементы в массиве — это объекты с тремя свойствами: свойство `product` типа `string`, свойство `size` типа `string` и свойство `quantity` типа `integer`. Кроме того, мы определяем перечисление для свойства `size`, которое ограничивает допустимые значения `small`, `medium` и `big`. Наконец, мы также задаем значение по умолчанию `1` для свойства `quantity`, поскольку это единственное необязательное поле в полезной нагрузке. Всякий раз, когда пользователь отправляет запрос с элементом без свойства `quantity`, мы предполагаем, что он хочет заказать только одну единицу этого товара.

#### Листинг 5.6. Спецификация эндпоинта POST /orders

paths:

```

/orders:
  post:
    operationId: createOrder
    requestBody:
      required: true
      content:
        application/json:
          schema:
            type: object
            properties:
              order:
                type: array
                items:
                  type: object
                  properties:
                    product:
                      type: string

```

Описываем полезную нагрузку запроса в разделе `requestBody`

Указываем тип содержимого полезной нагрузки

Определяем схему полезной нагрузки

Указываем, требуется ли полезная нагрузка

```

size:
  type: string
  enum:
    - small
    - medium
    - big
quantity:
  type: integer
  required: false
  default: 1
required:
  - product
  - size

```

Мы можем ограничить значения свойства, используя перечисление

Указываем значение по умолчанию

Встраивание схем полезной нагрузки в определения эндпоинтов, как в листинге 5.6, может затруднить чтение и понимание спецификации. В следующем разделе мы разберемся, как рефакторить схемы полезной нагрузки для повторного использования и удобства чтения.

## 5.6. РЕФАКТОРИНГ ОПРЕДЕЛЕНИЙ СХЕМ ВО ИЗБЕЖАНИЕ ПОВТОРОВ

В этом разделе мы изучим стратегии рефакторинга схем, позволяющие сохранить спецификацию API чистой и удобочитаемой. Определение эндпоинта POST /orders в листинге 5.6 длинное и содержит несколько уровней отступов. В результате его трудно читать, а это значит, что в дальнейшем его будет трудно расширять и поддерживать. Мы можем перенести схему полезной нагрузки в другой раздел спецификации API — раздел components (листинг 5.7). Как было сказано выше, этот раздел используется для объявления схем, ссылки на которые содержатся в спецификации. Каждая схема представляет собой объект, где ключ — это имя схемы, а значения — свойства, которые ее описывают.

**Листинг 5.7.** Спецификация эндпоинта POST /orders с использованием указателя JSON

```

paths:
  /orders:
    post:
      operationId: createOrder
      requestBody:
        required: true
        content:
          application/json:
            schema:
              $ref: '#/components/schemas/CreateOrderSchema'

```

Используем указатель JSON для ссылки на схему, определенную в другом месте документа

Определения схем находятся в разделе components

```

components:
  schemas:

```

```
CreateOrderSchema:
  type: object
  properties:
    order:
      type: array
      items:
        type: object
        properties:
          product:
            type: string
          size:
            type: string
            enum:
              - small
              - medium
              - big
          quantity:
            type: integer
            required: false
            default: 1
        required:
          - product
          - size
```

← Каждая схема — это объект, где ключом является имя, а значениями — свойства, которые его описывают

Перемещение схемы для полезной нагрузки запроса `POST /orders` в раздел `components` API делает документ более читабельным. Это позволяет нам сохранить раздел `paths` документа чистым и сосредоточиться на деталях более высокого уровня эндпоинта. Нам просто нужно сослаться на схему `CreateOrderSchema` с помощью указателя JSON:

```
#/components/schemas/CreateOrderSchema
```

Спецификация уже хорошо выглядит, но может стать еще лучше. `CreateOrderSchema` слишком длинная и содержит несколько уровней вложенных определений. Если `CreateOrderSchema` будет усложняться, то со временем ее станет трудно читать и поддерживать. Мы можем сделать ее более читабельной, изменив определение элемента заказа в массиве (листинг 5.8). Эта стратегия позволит нам повторно использовать схему для элемента заказа в других частях API.

#### Листинг 5.8. Определения схем для `OrderItemSchema` и `Order`

```
components:
  schemas:
    OrderItemSchema:
      type: object
      properties:
        product:
          type: string
        size:
          type: string
          enum:
            - small
```

← Вводим `OrderItemSchema`

```

    - medium
    - big
  quantity:
    type: integer
    default: 1
CreateOrderSchema:
  type: object
  properties:
    order:
      type: array
      items:
        $ref: '#/OrderItemSchema'

```

Используем указатель JSON, чтобы указать на OrderItemSchema

Наши схемы отлично выглядят! Схема `CreateOrderSchema` предназначена для создания заказа или его обновления, поэтому мы можем повторно использовать ее в эндпоинте `PUT /orders/{order_id}` (листинг 5.9). Как вы узнали в главе 4, URL-путь `/orders/{order_id}` представляет собой единичный ресурс, и поэтому URL содержит параметр `path`, который является идентификатором заказа. В OpenAPI параметры пути представлены в фигурных скобках. Мы указываем, что параметр `order_id` — это строка с форматом `UUID` (длинная случайная строка)<sup>1</sup>. Определяем параметр URL `path` непосредственно под URL `path`, чтобы убедиться, что он применяется ко всем HTTP-методам.

#### Листинг 5.9. Спецификация для эндпоинта `PUT /orders/{order_id}`

```

paths:
  /orders:
    get:
    ...
  /orders/{order_id}:
    parameters:
      - in: path
        name: order_id
        required: true
        schema:
          type: string
          format: uuid
    put:
      operationId: updateOrder
      requestBody:
        required: true
        content:
          application/json:
            schema:
              $ref: '#/components/schemas/CreateOrderSchema'

```

Объявляем URL-адрес ресурса заказа

Определяем параметр URL `path`

Параметр `order_id` является частью URL-пути

Имя параметра

Параметр `order_id` является обязательным

Указываем формат параметра (UUID)

Определяем HTTP-метод, используемый для текущего URL-пути

Документируем тело запроса эндпоинта PUT

Разобравшись, как определять схемы для полезной нагрузки запросов, перейдем к ответам.

<sup>1</sup> Leach P., Mealling M., Salz R. A Universally Unique Identifier (UUID) URN Namespace // RFC 4122. <https://datatracker.ietf.org/doc/html/rfc4122>.



## 5.7. ДОКУМЕНТИРОВАНИЕ ОТВЕТОВ API

В этом разделе мы будем документировать ответы API. Начнем с определения полезной нагрузки для эндпоинта GET /orders/{order\_id}. Ответ этого эндпоинта выглядит следующим образом:

```
{
  "id": "924721eb-a1a1-4f13-b384-37e89c0e0875",
  "status": "progress",
  "created": "2022-05-01",
  "order": [
    {
      "product": "cappuccino",
      "size": "small",
      "quantity": 1
    },
    {
      "product": "croissant",
      "size": "medium",
      "quantity": 2
    }
  ]
}
```

Здесь представлены товары, заказанные пользователем, время размещения заказа и статус заказа. Эта полезная нагрузка похожа на полезную нагрузку запроса, которую мы определили в разделе 5.6 для эндпоинтов POST и PUT, поэтому мы можем повторно использовать предыдущие схемы (листинг 5.10).

### Листинг 5.10. Определение схемы GetOrderSchema

```
components:
  schemas:
    OrderItemSchema:
      ...
    GetOrderSchema:
      type: object
      properties:
        status:
          type: string
          enum:
            - created
            - paid
            - progress
            - cancelled
            - dispatched
            - delivered
        created:
          type: string
          format: date-time
```

Определяем схему GetOrderSchema

Ограничиваем значения свойства status перечислением

Строка с форматом даты и времени

```

order:
  type: array
  items:
    $ref: '#/components/schemas/OrderItemSchema'

```

Ссылаемся на схему OrderItemSchema с помощью указателя JSON

В листинге 5.10 мы используем указатель JSON, чтобы сослаться на `GetOrderSchema`. Альтернатива повторному использованию существующих схем — применение наследования. В OpenAPI мы можем наследовать и расширять схему с помощью *композиции модели* — стратегии, позволяющей объединять свойства различных схем в одном определении объекта. Через специальное ключевое слово `allOf` мы указываем, что объект требует всех свойств в перечисленных схемах.

## ОПРЕДЕЛЕНИЕ

*Композиция модели* — это стратегия в JSON Schema, которая позволяет объединять свойства различных схем в один объект. Она полезна, когда схема содержит свойства, которые уже были определены в другом месте, то есть мы можем избежать повторений.

В листинге 5.11 показано альтернативное определение `GetOrderSchema` с использованием ключевого слова `allOf`. В этом случае `GetOrderSchema` является композицией двух других схем: `CreateOrderSchema` и анонимной схемы с двумя ключами — `status` и `created`.

### Листинг 5.11. Альтернативная реализация `GetOrderSchema` с использованием ключевого слова `allOf`

```

components:
  schemas:
    OrderItemSchema:
      ...
    GetOrderSchema:
      allOf:
        - $ref: '#/components/schemas/CreateOrderSchema'
        - type: object
          properties:
            status:
              type: string
              enum:
                - created
                - paid
                - progress
                - cancelled
                - dispatched
                - delivered
            created:
              type: string
              format: date-time

```

Используем ключевое слово `allOf` для наследования свойств из других схем

Используем указатель JSON для ссылки на другую схему

Определяем новый объект для включения свойств, специфичных для `GetOrderSchema`

Композиция моделей приводит к более чистой и лаконичной спецификации, но работает только в том случае, если схемы строго совместимы. Если мы решим расширить `CreateOrderSchema` новыми свойствами, то эта схема может перестать быть переносимой в модель `GetOrderSchema`. Поэтому иногда лучше искать общие элементы среди различных схем и реорганизовывать их определения в отдельные схемы.

Итак, у нас есть схема для полезной нагрузки ответа эндпоинта `GET /orders/{order_id}` и мы можем завершить спецификацию эндпоинта. Определяем ответы как объекты, в которых ключом является статус-код ответа, например 200. Мы так же описываем тип содержимого ответа и его схему, `GetOrderSchema` (листинг 5.12).

### Листинг 5.12. Спецификация для эндпоинта `GET /orders/{order_id}`

```
paths:
  /orders:
    get:
      ...

  /orders/{order_id}:
    parameters:
      - in: path
        name: order_id
        required: true
        schema:
          type: string
          format: uuid
    put:
      ...
    get:
      summary: Returns the details of a specific order
      operationId: getOrder
      responses:
        '200':
          description: OK
          content:
            application/json:
              schema:
                $ref: '#/components/schemas/GetOrderSchema'
```

Определяем эндпоинт  
GET URL-пути `/orders/{order_id}`

Приводим краткое  
описание этого  
эндпоинта

Определяем ответы эндпоинта

Каждый ответ представляет собой объект,  
где ключом является статус-код

Краткое описание ответа

Описываем типы содержимого ответа

Используем указатель  
JSON для ссылки  
на `GetOrderSchema`

Как вы можете видеть, мы определяем схемы ответов в разделе `responses` эндпоинта. В данном случае мы предоставляем спецификацию только для успешного ответа 200 (ОК), но можем документировать и другие статус-коды, например ответы об ошибках. Далее рассмотрим, как создавать общие ответы, которые можно повторно использовать в эндпоинтах.

## 5.8. СОЗДАНИЕ ОБЩИХ ОТВЕТОВ

Как мы уже говорили в главе 4, ответы на ошибки носят более общий характер, поэтому мы можем использовать раздел `components` спецификации API для предоставления общих определений этих ответов, а затем повторно применять их в наших эндпоинтах.

Общие ответы определяются в заголовке ответов в разделе `components`. В листинге 5.13 приводится общее определение для ответа 404 с именем `NotFound`. Как и для любого другого ответа, мы также документируем полезную нагрузку содержимого, которая в данном случае определяется схемой `Error`.

**Листинг 5.13.** Определение ответа с общим статус-кодом 404

```

components:
  responses:
    NotFound:
      description: The specified resource was not found.
      content:
        application/json:
          schema:
            $ref: '#/components/schemas/Error'
  schemas:
    OrderItemSchema:
    ...
    Error:
      type: object
      properties:
        detail:
          type: string
      required:
        - detail
  
```

Эту спецификацию ответа 404 можно повторно использовать в спецификации всех эндпоинтов в URL-пути `/orders/{order_id}`, поскольку все они специально разработаны для того, чтобы указывать на определенный ресурс.

### ПРИМЕЧАНИЕ

Вы можете задаться вопросом: если некоторые ответы являются общими для всех эндпоинтов URL-адреса, почему мы не можем определить ответы непосредственно под URL и избежать повторений? Суть в том, что на данный момент это невозможно. Ключевое слово `responses` не допускается указывать непосредственно в URL-адресе, поэтому мы должны документировать все ответы каждого эндпоинта отдельно. В репозитории OpenAPI GitHub есть запрос на включение общих ответов непосредственно под URL-адресом, но он не был реализован (<https://mng.bz/097p>).

Мы можем использовать общий ответ 404 из листинга 5.13 в эндпоинте GET /orders/{order\_id} (листинг 5.14).

**Листинг 5.14.** Используем схему ответа 404 в разделе GET /orders/{order\_id}

paths:

...

/orders/{order\_id}:

parameters:

- in: path

name: order\_id

required: true

schema:

type: string

"format": uuid

get:

summary: Returns the details of a specific order

operationId: getOrder

responses:

'200':

description: OK

content:

application/json:

schema:

\$ref: '#/components/schemas/GetOrderSchema'

'404': ← Определяем ответ 404

\$ref: '#/components/responses/NotFound'

← Ссылаемся на найденный  
ответ, используя указатель JSON

Спецификация API сервиса заказов в репозитории GitHub для этой книги также содержит общее определение для 422 ответов и расширенное определение компонента Error, учитывающее различную полезную нагрузку ошибок, которые мы получаем от FastAPI.

Мы почти закончили. Остался только эндпоинт GET /orders, который возвращает список заказов. В полезной нагрузке эндпоинта повторно используется GetOrderSchema для определения элементов в массиве orders (листинг 5.15).

**Листинг 5.15.** Спецификация для эндпоинта GET /orders

paths:

/orders:

get:

operationId: getOrders

responses:

'200':

description: A JSON array of orders

content:

application/json:

← Определяем новый метод GET  
URL-пути /orders

```

schema:
  type: object
  properties:
    orders:
      type: array ← Это массив
      items:
        $ref: '#/components/schemas/GetOrderSchema' ←
      required:
        - order
                                Каждый элемент массива
                                определяется GetOrderSchema

post:
  ...

/orders/{order_id}:
  parameters:
    ...

```

Теперь эндпоинты нашего API полностью документированы! В определениях эндпоинтов вы можете использовать гораздо больше элементов, например `tags` и `externalDocs`. Эти атрибуты не являются необходимыми, но могут помочь обеспечить более четкую структуру вашего API или облегчить группировку эндпоинтов. Например, вы можете использовать теги для создания групп эндпоинтов, которые логически связаны друг с другом или имеют общие функции.

Прежде чем мы закончим эту главу, нужно рассмотреть еще одну тему: документирование схемы аутентификации нашего API.

## 5.9. ОПРЕДЕЛЕНИЕ СХЕМЫ АУТЕНТИФИКАЦИИ API

Если API защищен, его спецификация должна описывать, как пользователи будут аутентифицировать и авторизировать свои запросы. В этом разделе объясняется, как документировать схемы безопасности API. Определения безопасности API хранятся в разделе `components` спецификации под заголовком `securitySchemes`.

С помощью OpenAPI мы можем описать различные схемы безопасности, такие как аутентификация на основе HTTP, аутентификация на основе ключа, Open Authorization 2 (OAuth2) и OpenID Connect<sup>1</sup>. В главе 11 мы будем реализовывать аутентификацию и авторизацию с помощью протоколов OpenID Connect и OAuth2, поэтому добавим определения для этих схем. В листинге 5.16 показаны изменения, которые необходимо внести в спецификацию API для документирования схем безопасности.

<sup>1</sup> Полный перечень всех схем безопасности, доступных в OpenAPI, приведен на сайте <https://swagger.io/docs/specification/authentication/>.

**Листинг 5.16.** Документирование схемы безопасности API

```

components:
  responses:
    ...

  schemas:
    ...

  securitySchemes:
    openId:
      type: openIdConnect
      openIdConnectUrl: https://coffeemesh-dev.eu.auth0.com/.well-known/openid-configuration
    oauth2:
      type: oauth2
      flows:
        clientCredentials:
          tokenUrl: https://coffeemesh-dev.eu.auth0.com/oauth/token
          scopes: {}
    bearerAuth:
      type: http
      scheme: bearer
      bearerFormat: JWT
    ...

security:
  - oauth2:
      - getOrders
      - createOrder
      - getOrder
      - updateOrder
      - deleteOrder
      - payOrder
      - cancelOrder
  - bearerAuth:
      - getOrders
      - createOrder
      - getOrder
      - updateOrder
      - deleteOrder
      - payOrder
      - cancelOrder

```

Схемы безопасности в заголовке securitySchemes раздела компонентов API

Предоставляем имя для схемы безопасности (может быть любым)

Тип схемы безопасности

URL-адрес, описывающий конфигурацию OpenID Connect в нашем бэкенде

Название другой схемы безопасности

Тип схемы безопасности

Потоки авторизации, доступные в рамках этой схемы безопасности

Описание потока учетных данных клиента

Доступные области при запросе токена авторизации

URL-адрес, по которому пользователи могут запрашивать токены авторизации

Токен на предъявителя имеет формат веб-токена JSON (JWT)

Мы описываем три схемы безопасности: одну — для OpenID Connect, другую — для OAuth2 и еще одну — для авторизации на предъявителя. Задействуем OpenID Connect для авторизации доступа пользователей через внешнее приложение, а для прямой интеграции с API предложим схему клиентских учетных данных OAuth. Как работает каждый протокол и каждая схема авторизации, я подробно расскажу в главе 11.

Для OpenID Connect в свойстве `openIdConnectUrl` мы должны предоставить URL-адрес конфигурации, который описывает, как работает наша внутренняя

аутентификация. Для OAuth2 следует описать доступные схемы авторизации, а также URL, который клиенты будут использовать для получения токенов авторизации, и доступные области определения. Авторизация на предъявителя сообщает пользователям, что они должны включить JSON Web Token (JWT) в заголовок *Authorization* для авторизации своих запросов.

На этом мы завершаем наше путешествие по документированию REST API с помощью OpenAPI. И какое путешествие! Вы узнали, как использовать JSON Schema, как работает OpenAPI, как структурировать спецификацию API, разбить процесс документирования API на небольшие шаги и создать полную спецификацию API. Теперь, когда вам придется работать над API, у вас будут все возможности для документирования его проекта с помощью этих стандартных технологий.

## РЕЗЮМЕ

- JSON Schema — это спецификация для определения типов и форматов свойств документа JSON. JSON Schema полезна для определения моделей проверки данных независимо от языка.
- OpenAPI — стандартный формат документации REST API, который использует JSON Schema для описания свойств API. Вместе с OpenAPI вы получаете целую экосистему инструментов и фреймворков, созданных на основе этого стандарта, что упрощает создание API-интеграций.
- Указатель JSON позволяет ссылаться на схему с помощью ключевого слова *\$ref*. Благодаря указателям JSON можно создавать повторно используемые определения схем, чтобы добавлять их в разных частях спецификации API, сохраняя ее чистоту и простоту понимания.
- Спецификация OpenAPI содержит следующие разделы:
  - *openapi* — указывает версию OpenAPI, используемую для документирования API;
  - *info* — содержит информацию об API, например его название и версию;
  - *servers* — документирует URL-адреса, под которыми доступен API;
  - *paths* — описывает эндпоинты API, включая схемы для запросов и ответов API и любые соответствующие URL-адреса или параметры запроса;
  - *components* — описывает повторно используемые компоненты API, такие как схемы полезной нагрузки, типовые ответы и схемы аутентификации.



# Разработка REST API на Python

---

## В этой главе

- ✓ Добавление параметров URL-запроса к эндпоинту с помощью FastAPI.
- ✓ Запрет наличия неизвестных свойств в полезной нагрузке с использованием pydantic и marshmallow.
- ✓ Реализация REST API с использованием flask-smorest.
- ✓ Определение схем проверки и параметров URL-запроса с помощью marshmallow.

В предыдущих главах вы научились проектировать и документировать REST API. В этой главе мы создадим API для сервиса заказов и сервиса кухни нашей воображаемой платформы CoffeeMesh. Сервис заказов является основным входом в CoffeeMesh для клиентов платформы. Через него они могут размещать заказы, оплачивать их, обновлять и отслеживать. Сервис кухни занимается планированием заказов для производства и следит за их выполнением. В ходе работы над этими примерами мы изучим лучшие практики реализации REST API.

В главе 2 мы проработали часть API сервиса заказов. В первых разделах главы мы продолжим работу с этим API и реализуем его остальные функции с помощью FastAPI, высокопроизводительного API-фреймворка для Python и популярного инструмента для создания REST API. Вы узнаете, как добавлять параметры запроса URL к эндпоинтам с помощью FastAPI. Как вы видели в главе 2, FastAPI

использует `pydantic` для проверки данных, и в этой главе мы воспользуемся им, чтобы запретить неизвестные поля в полезной нагрузке. Мы познакомимся с паттерном толерантного читателя (`tolerant reader`) и сопоставим его преимущества с риском сбоев в интеграции с помощью API из-за таких ошибок, как опечатки.

После завершения реализации API сервиса заказов перейдем к API сервиса кухни. Для работы с ним воспользуемся `flask-smorest` — популярным фреймворком, построенным поверх `Flask` и `marshmallow`. Мы разберем, как реализовывать API, следуя шаблонам приложений `Flask`, и определим схемы валидации с помощью `marshmallow`.

К концу этой главы вы будете знать, как реализовать REST API с помощью `FastAPI` и `Flask`, двух самых популярных библиотек в экосистеме `Python`. Вы увидите, как принципы реализации REST API выходят за рамки деталей реализации каждого фреймворка и могут применяться независимо от используемой технологии. Код для этой главы доступен в папке `ch06` в репозитории, прилагаемом к этой книге. Папка `ch06` содержит две подпапки: для API сервиса заказов (`ch06/orders`) и API сервиса кухни (`ch06/kitchen`). С учетом сказанного давайте наконец приступим к работе!

## 6.1. ОБЗОР API СЕРВИСА ЗАКАЗОВ

Сначала вспомним минимальную реализацию API, которую рассмотрели в главе 2. Полную спецификацию API сервиса заказов можно найти в `ch06/orders/oas.yaml` в репозитории GitHub для книги. Прежде чем перейти непосредственно к реализации, проанализируем спецификацию и посмотрим, что осталось нереализованным.

В главе 2 мы добавили эндпоинты API и создали `pydantic`-схемы для проверки полезной нагрузки запросов и ответов. Мы намеренно пропустили реализацию уровня бизнес-логики приложения, поскольку это сложная задача, которую мы рассмотрим в главе 7.

Напомню, какие у нас эндпоинты API сервиса заказов:

- `/orders` — позволяет извлекать списки заказов (GET) и размещать заказы (POST);
- `/orders/{order_id}` — получать информацию о конкретном заказе (GET), обновлять (PUT) и удалять заказ (DELETE);
- `/orders/{order_id}/cancel` — отменить заказ (POST);
- `/orders/{order_id}/pay` — оплатить заказ (POST).

POST `/orders` и PUT `/orders/{order_id}` требуют полезной нагрузки запроса, определяющей свойства заказа, и в главе 2 мы реализовали схемы для этой полезной нагрузки. Чего нет в реализации, так это параметров запроса URL для эндпоинта GET `/orders`. Кроме того, схемы `pydantic`, которые мы добавили в главе 2, не делают недействительной полезную нагрузку с недопустимыми свойствами в ней. Как мы увидим в разделе 6.3, в одних случаях это хорошо, но в других может привести

к проблемам интеграции. Далее вы научитесь настраивать схемы таким образом, чтобы сделать недействительной полезную нагрузку с недопустимыми свойствами.

Если вы хотите следовать примерам в этой главе, создайте папку с именем `ch06` и скопируйте в нее код из `ch02` как для `ch06/orders`. Не забудьте установить зависимости и активировать виртуальную среду:

```
$ mkdir ch06
$ cp -r ch02 ch06/orders
$ cd ch06/orders
$ pipenv install --dev && pipenv shell
```

Вы можете запустить веб-сервер, выполнив следующую команду:

```
$ uvicorn orders.app:app --reload
```

### НАПОМИНАНИЕ О FASTAPI + UVICORN

Мы реализуем API сервиса заказов с помощью FastAPI — популярного фреймворка Python. Он построен поверх Starlette, асинхронной реализации веб-сервера. Для демонстрации нашего приложения FastAPI мы используем Uvicorn — еще одну реализацию асинхронного сервера, которая эффективно обрабатывает входящие запросы.

Флаг `--reload` заставляет Uvicorn следить за изменениями в ваших файлах, так что при каждом обновлении приложение перезагружается. Это сэкономит вам время на перезагрузку сервера каждый раз, когда вы вносите изменения в код. С этим разобрались, давайте завершим реализацию API сервиса заказов!

## 6.2. ПАРАМЕТРЫ URL-ЗАПРОСА ДЛЯ API СЕРВИСА ЗАКАЗОВ

В этом разделе мы улучшим эндпоинт `GET /orders` API сервиса заказов, добавив параметры запроса URL. Мы также реализуем схемы проверки параметров. В главе 4 вы узнали, что параметры запроса URL позволяют фильтровать результаты эндпоинта `GET`. В главе 5 выяснили, что эндпоинт `GET /orders` принимает параметры запроса URL для фильтрации заказов по отмене, а также для ограничения списка заказов, возвращаемых эндпоинтом (листинг 6.1).

### Листинг 6.1. Спецификация параметров запроса URL `GET /orders`

```
# file: orders/oas.yaml

paths:
  /orders:
    get:
      parameters:
        - name: cancelled
          in: query
          required: false
```

```

    schema:
        type: boolean
- name: limit
  in: query
  required: false
  schema:
    type: integer

```

Нам необходимо реализовать два параметра запроса URL: `cancelled` (булево значение) и `limit` (целое число). Ни один из них не является обязательным, поэтому пользователи должны иметь возможность вызывать эндпоинт `GET /orders` даже при их отсутствии. Давайте посмотрим, как это сделать.

Реализовать параметры запроса URL для эндпоинта с помощью FastAPI несложно. Все, что нужно сделать, — включить их в сигнатуру функции эндпоинта и использовать подсказки типов, чтобы добавить для них правила проверки. Поскольку параметры запроса являются необязательными, мы пометим их как таковые, используя тип `Optional`, и установим для них значения по умолчанию `None` (листинг 6.2).

#### Листинг 6.2. Реализация параметров запроса URL для `GET /orders`

```
# file: orders/orders/api/api.py
```

```

import uuid
from datetime import datetime
from typing import Optional
from uuid import UUID

```

```
...
```

```

@app.get('/orders', response_model=GetOrdersSchema)
def get_orders(cancelled: Optional[bool] = None, limit: Optional[int] = None):
    ...

```

Включаем параметры запроса  
URL в сигнатуру функции

Теперь, когда у нас есть параметры запроса, доступные в эндпоинте `GET /orders`, как обрабатывать их внутри функции? Поскольку параметры запроса являются необязательными, сначала мы проверим, были ли они установлены. Это можно сделать, проверив, содержат ли они значение, отличное от `None`. В листинге 6.3 показано, как можно обрабатывать параметры запроса URL в теле функции эндпоинта `GET /orders`. Изучите рис. 6.1, чтобы понять процесс принятия решения о фильтрации списка заказов на основе параметров запроса.

#### Листинг 6.3. Реализация параметров URL-запроса для `GET /orders`

```
# file: orders/orders/api/api.py
```

```

@app.get('/orders', response_model=GetOrdersSchema)
def get_orders(cancelled: Optional[bool] = None, limit: Optional[int] = None):
    if cancelled is None and limit is None:
        return {'orders': orders}

```

Если параметры не были заданы,  
мы немедленно выполняем return

```
query_set = [order for order in orders]
```

Если какой-либо из параметров был  
задан, фильтруем список в `query_set`

```

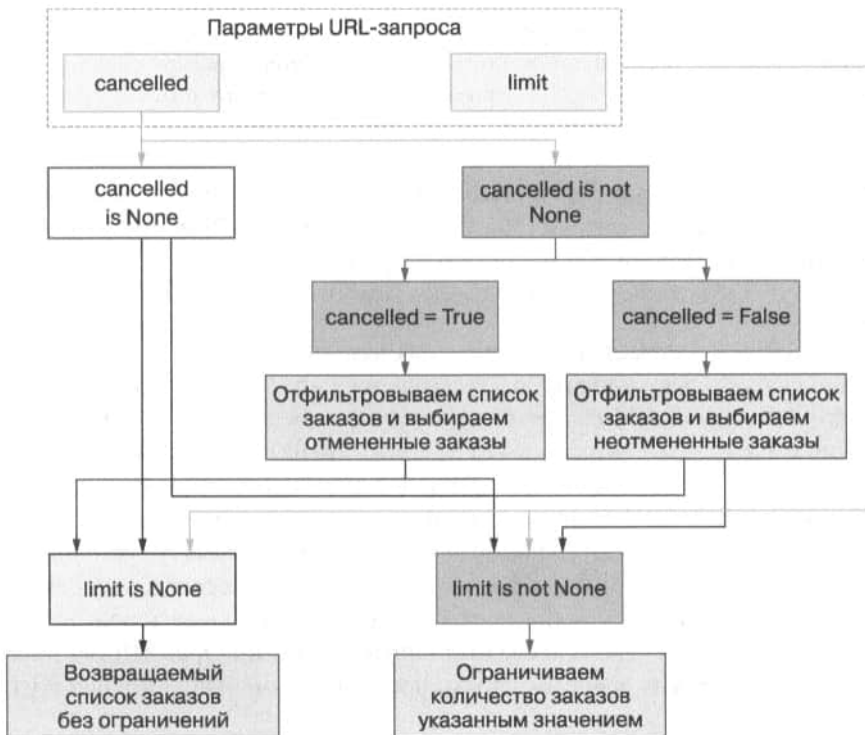
if cancelled is not None:
    if cancelled:
        query_set = [
            order
            for order in query_set
            if order['status'] == 'cancelled'
        ]
    else:
        query_set = [
            order
            for order in query_set
            if order['status'] != 'cancelled'
        ]
if limit is not None and len(query_set) > limit:
    return {'orders': query_set[:limit]}

return {'orders': query_set}

```

Проверяем, установлен ли  
параметр cancelled

Если установлено ограничение  
и его значение меньше длины  
query\_set, возвращаем  
подмножество query\_set



**Рис. 6.1.** Процесс принятия решения о фильтрации заказов на основе параметров запроса. Если параметру `cancelled` присвоено значение `True` или `False`, мы используем его для фильтрации списка заказов. После этого шага проверяем, установлен ли параметр `limit`. Если ограничение установлено, возвращаем только соответствующее количество заказов из списка

Теперь, когда мы знаем, как добавлять параметры запроса URL к эндпоинтам, посмотрим, как улучшить схемы проверки.

## 6.3. ВАЛИДАЦИЯ ПОЛЕЗНОЙ НАГРУЗКИ С НЕИЗВЕСТНЫМИ ПОЛЯМИ

До сих пор наши модели `pydantic` были терпимы к полезной нагрузке запросов. Если клиент API отправляет полезную нагрузку с полями, которые не были объявлены в схемах, она будет принята. Иногда это неплохо, но возможны ситуации, когда это чревато ошибками интеграции. Далее мы посмотрим, как настроить `pydantic`, чтобы запретить наличие неизвестных полей. Неизвестные поля — это поля, которые не были определены в схеме.

### НАПОМИНАНИЕ О PYDANTIC

Как мы видели в главе 2, `FastAPI` использует `pydantic` с целью определения моделей валидации для наших API. `Pydantic` — популярная библиотека проверки данных для Python с современным интерфейсом, которая позволяет определять правила проверки данных с помощью подсказок типов.

В главе 2 мы реализовали определения схем API сервиса заказов в соответствии с паттерном толерантного читателя (<https://martinfowler.com/bliki/TolerantReader.html>), который построен на принципах закона Постела, рекомендующего быть консервативными в том, что вы делаете, и либеральными в том, что вы принимаете от других<sup>1</sup>.

В области веб-API это означает, что мы должны строго проверять ту полезную нагрузку, которую отправляем клиенту, и в то же время допускать наличие неизвестных полей в той полезной нагрузке, которую мы получаем от клиентов API. `JSON Schema` по умолчанию следует этому шаблону, и, если это не объявлено явно, объект `JSON Schema` принимает любые свойства. Чтобы запретить необъявленные свойства с использованием `JSON Schema`, мы устанавливаем для `additionalProperties` значение `false`. Если мы используем композицию моделей, лучше всего установить `unevaluatedProperties` равным `false`, поскольку `additionalProperties` вызывает конфликты между различными моделями<sup>2</sup>. `OpenAPI 3.1` позволяет использовать как `additionalProperties`, так и `unevaluatedProperties`, но `OpenAPI 3.0` принимает только `additionalProperties`. Поскольку мы документируем наши API,

<sup>1</sup> *Postel J. Ed.* Transmission Control Protocol // RFC 761. — P. 13. <https://tools.ietf.org/html/rfc761>.

<sup>2</sup> Чтобы понять, почему `additionalProperties` не работает при использовании композиции моделей, ознакомьтесь с отличным постом на эту тему в репозитории `JSON Schema` на GitHub: <https://github.com/json-schema-org/json-schema-spec/issues/556>.

используя OpenAPI 3.0.3, то запретим необъявленные свойства с помощью `additionalProperties`:

```
# file: orders/oas.yaml
```

```
GetOrderSchema:
  additionalProperties: false
  type: object
  required:
    - order
    - id
    - created
    - status
  properties:
    id:
      type: string
      format: uuid
  ...
```

Посмотрите спецификацию API сервиса заказов в файле `ch06/orders/oas.yaml` в репозитории, чтобы увидеть дополнительные примеры `additionalProperties`.

Паттерн толерантного читателя полезен, когда API не полностью консолидирован или часто меняется и когда нужно иметь возможность вносить в него изменения, не нарушая интеграции с существующими клиентами. Однако в других случаях (см. раздел 2.5) этот паттерн может повлечь новые ошибки или привести к неожиданным проблемам интеграции.

Например, `OrderItemSchema` имеет три свойства: `product`, `size` и `quantity`. `product` и `size` — обязательные свойства, а `quantity` — необязательное, и если оно не определено, сервер присваивает ему значение 1 по умолчанию. В некоторых сценариях это может привести к путанице. Представьте, что клиент отправляет полезную нагрузку с опечаткой в представлении свойства `quantity`, например такую:

```
{
  "order": [
    {
      "product": "capuccino",
      "size": "small",
      "quantit": 5
    }
  ]
}
```

Используя реализацию паттерна толерантного читателя, мы игнорируем поле `quantit`, предполагаем, что свойство `quantity` отсутствует, и по умолчанию устанавливаем его значение равным 1. Эта ситуация может запутать клиента, который намеревался установить другое значение для `quantity`.

**КЛИЕНТ API ДОЛЖЕН ПРОТЕСТИРОВАТЬ СВОЙ КОД!**

Вы можете возразить, что прежде, чем обращаться к серверу, клиент должен был протестировать свой код и убедиться, что тот работает правильно. И вы будете правы. Но в реальной жизни код часто остается непроверенным или неправильно протестированным, и дополнительная валидация на сервере пойдет только на пользу. Если мы проверим полезную нагрузку на наличие недопустимых свойств, эта ошибка будет обнаружена и мы уведомим клиента.

Как можно добиться этого с помощью pydantic? Чтобы запретить неизвестные атрибуты, нужно определить класс `Config` в наших моделях и установить для свойства `extra` значение `forbid` (запретить) (листинг 6.4).

**Листинг 6.4.** Запрет дополнительных свойств в моделях

```
# file: orders/orders/api/schemas.py

from datetime import datetime
from enum import Enum
from typing import List, Optional
from uuid import UUID

from pydantic import BaseModel, Extra, conint, conlist, validator

...

class OrderItemSchema(BaseModel):
    product: str
    size: Size
    quantity: int = Optional[conint(ge=1, strict=True)] = 1

    class Config:
        extra = Extra.forbid

class CreateOrderSchema(BaseModel):
    order: List[OrderItemSchema]

    class Config:
        extra = Extra.forbid

class GetOrderSchema(CreateOrderSchema):
    id: UUID
    created: datetime
    status: StatusEnum
```

Используем `Config`, чтобы запретить свойства, которые не были определены в схеме

Протестируем новую функциональность. Выполните следующую команду для запуска сервера:

```
$ uvicorn orders.app:app --reload
```



Как вы видели в главе 2, FastAPI генерирует из кода пользовательский интерфейс Swagger, который можно использовать для тестирования эндпоинтов. Воспользуемся этим интерфейсом для тестирования наших новых правил валидации с помощью следующей полезной нагрузки:

```
{
  "order": [
    {
      "product": "string",
      "size": "small",
      "quantity": 5
    }
  ]
}
```

### ОПРЕДЕЛЕНИЕ

Swagger UI — это популярный стиль для представления интерактивных визуализаций REST API. Обеспечивает удобный интерфейс, который помогает понять реализацию API. Другим популярным UI для интерфейсов REST является Redoc (<https://github.com/Redocly/redoc>).

Для перехода к пользовательскому интерфейсу Swagger посетите <http://127.0.0.1:8000/docs> и следуйте инструкциям на рис. 6.2, чтобы узнать, как выполнить тест для эндпоинта POST /orders.

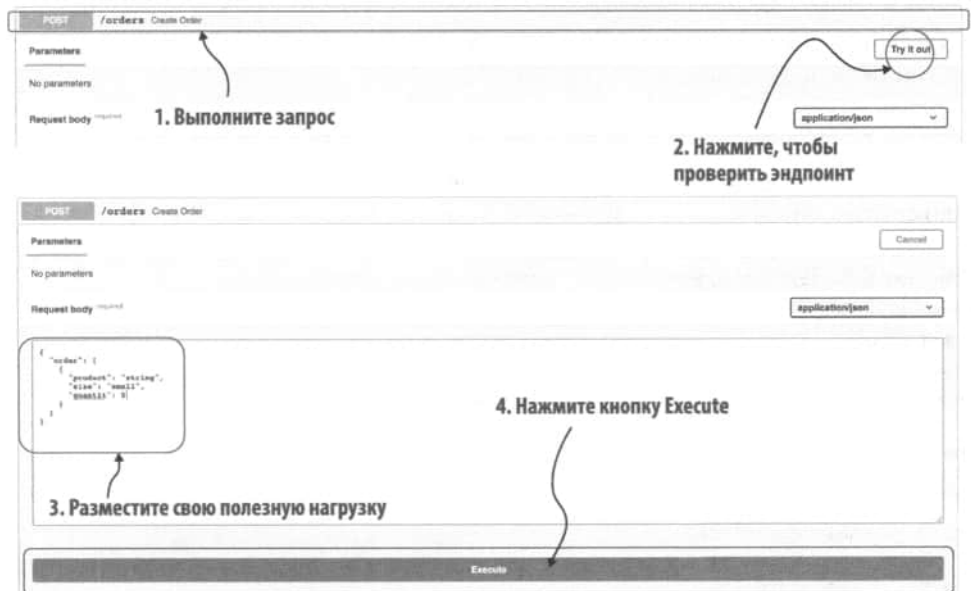


Рис. 6.2. Тестирование API с помощью Swagger UI

После выполнения этого теста вы увидите, что теперь FastAPI аннулирует полезную нагрузку и возвращает ответ с кодом 422, содержащий следующую полезную информацию: *Extra fields not permitted* (Дополнительные поля не разрешены).

## 6.4. ПЕРЕОПРЕДЕЛЕНИЕ ДИНАМИЧЕСКИ ГЕНЕРИРУЕМОЙ СПЕЦИФИКАЦИИ FASTAPI

До сих пор мы полагались на динамически генерируемую спецификацию API FastAPI для тестирования, визуализации и документирования API сервиса заказов. Такая спецификация отлично подходит для того, чтобы понять, как реализован API. Однако наш код может содержать ошибки реализации, которые способны привести к неточной документации. Кроме того, фреймворки разработки API имеют ограничения, когда дело доходит до создания документации по API, и в них, как правило, не предусмотрена поддержка некоторых функций OpenAPI. Например, часто отсутствующей функцией является документирование ссылок OpenAPI, которые мы добавим в нашу спецификацию API в главе 12.

Чтобы понять, как должен работать API, нужно посмотреть на его проектный документ. Он находится в `orders/oas.yaml` и поэтому является спецификацией, которую мы хотим показать при развертывании API. В этом разделе вы научитесь переопределять динамически генерируемую спецификацию от FastAPI с помощью нашего документа о дизайне API.

Чтобы загрузить документ спецификации API, нам понадобится `PuYAML`, который можно установить с помощью следующей команды:

```
$ pipenv install pyyaml
```

В файле `orders/app.py` мы загружаем спецификацию API и перезаписываем свойство `openapi` объекта нашего приложения (листинг 6.5).

**Листинг 6.5.** Переопределение динамически генерируемой спецификации API FastAPI

```
# file: orders/orders/app.py
```

```
from pathlib import Path
```

```
import yaml
```

```
from fastapi import FastAPI
```

```
app = FastAPI(debug=True)
```

```
oas_doc = yaml.safe_load(
    (Path(__file__).parent / '../oas.yaml').read_text()
)
```

Загружаем спецификацию API  
с помощью `PuYAML`

```
app.openapi = lambda: oas_doc
```

```
from orders.api import api
```

← Переопределяем свойство `openapi` в `FastAPI`, чтобы оно возвращало нашу спецификацию API

Чтобы иметь возможность тестировать API с помощью Swagger UI, нужно добавить в спецификацию URL-адрес `localhost`. Откройте файл `orders/oas.yaml` и добавьте этот адрес в раздел `servers`:

```
# file: orders/oas.yaml
```

```
servers:
```

- url: http://localhost:8000  
description: URL for local development and testing
- url: https://coffeemesh.com  
description: main production server
- url: https://coffeemesh-staging.com  
description: staging server for testing purposes only

По умолчанию FastAPI предоставляет пользовательский интерфейс Swagger UI по URL-адресу `/docs`, а спецификацию OpenAPI — по адресу `/openapi.json`. Это хорошо, когда есть только один API, но CoffeeMesh имеет несколько API микросервисов; следовательно, нам потребуется несколько путей для доступа к документации каждого API. Разместим Swagger UI API сервиса заказов в каталоге `/docs/orders`, а его спецификацию OpenAPI — в каталоге `/openapi/orders.json`. Можем переопределить эти пути непосредственно в инициализаторе объекта приложения FastAPI:

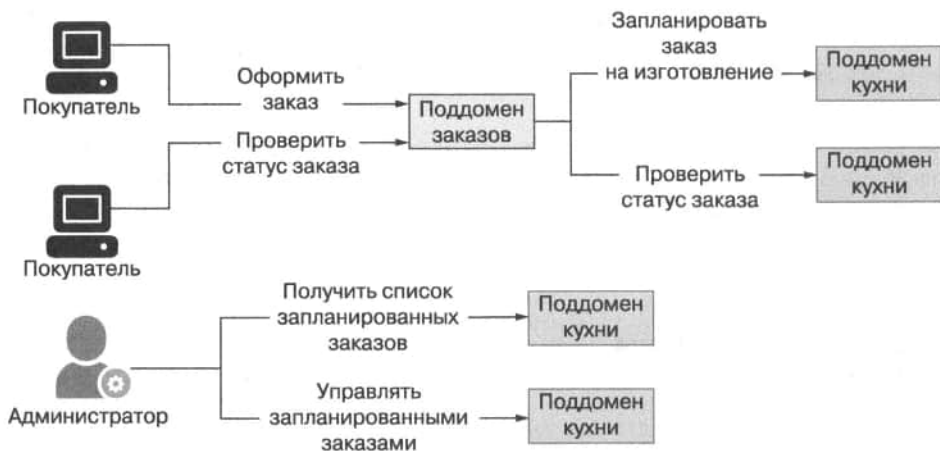
```
# file: orders/app.py
```

```
app = FastAPI(  
    debug=True, openapi_url='/openapi/orders.json', docs_url='/docs/orders'  
)
```

На этом мы завершаем наше путешествие по созданию API сервиса заказов с помощью FastAPI. Теперь пора переходить к созданию API сервиса кухни, для которого мы будем использовать новый стек: Flask + marshmallow.

## 6.5. ОБЗОР API СЕРВИСА КУХНИ

В этом разделе мы проанализируем требования к реализации API сервиса кухни. Как показано на рис. 6.3, сервис кухни управляет изготовлением заказов клиентов. Клиенты взаимодействуют с сервисом кухни через сервис заказов, когда размещают заказ или проверяют его статус. Персонал CoffeeMesh также может использовать сервис кухни для проверки количества запланированных заказов и управления ими.



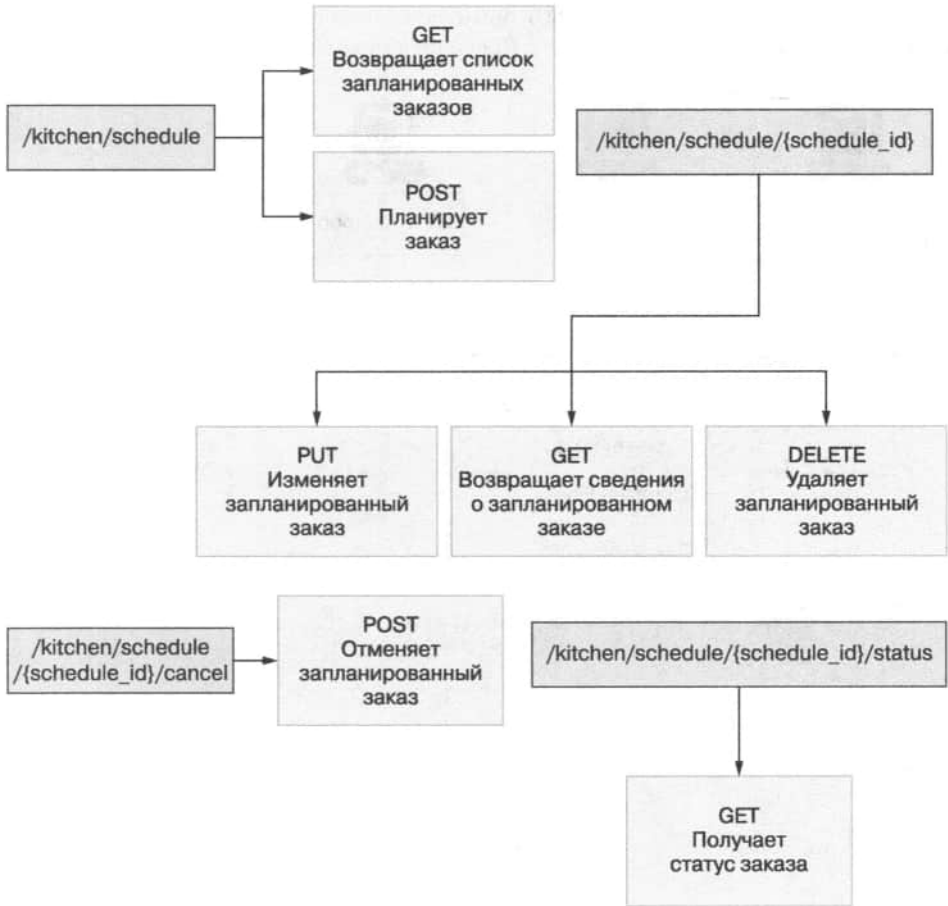
**Рис. 6.3.** Сервис кухни получает заказы в производство и отслеживает их выполнение. Сотрудники CoffeeMesh пользуются услугами кухни для управления запланированными заказами

Спецификация API сервиса кухни представлена в файле `ch06/kitchen/oas.yaml` в репозитории, прилагаемом к книге. API содержит четыре пути URL (рис. 6.4):

- `/kitchen/schedules` — позволяет запланировать заказ на изготовление (POST) и получить список запланированных заказов (GET);
- `/kitchen/schedules/{schedule_id}` — получить информацию о запланированном заказе (GET), обновить информацию о нем (PUT) и удалить его из наших записей (DELETE);
- `/kitchen/schedules/{schedule_id}/status` — прочитать статус запланированного заказа;
- `/kitchen/schedules/{schedule_id}/cancel` — отменить запланированный заказ.

API сервиса кухни содержит три схемы: `OrderItemSchema`, `ScheduleOrderSchema` и `GetScheduledOrderSchema`. Схема `ScheduleOrderSchema` представляет полезную нагрузку, необходимую для планирования заказа на изготовление, а схема `GetScheduledOrderSchema` содержит детали заказа, который был запланирован. Как и в API сервиса заказов, схема `OrderItemSchema` представляет детали каждого товара в заказе.

Как и в главе 2, мы сделаем реализацию простой и сосредоточимся только на уровне API. Сымитируем уровень бизнес-логики с помощью сохраненного в оперативной памяти представления расписаний, управляемого сервисом. В главе 7 мы изучим паттерны, которые помогут нам реализовать уровень бизнес-логики.

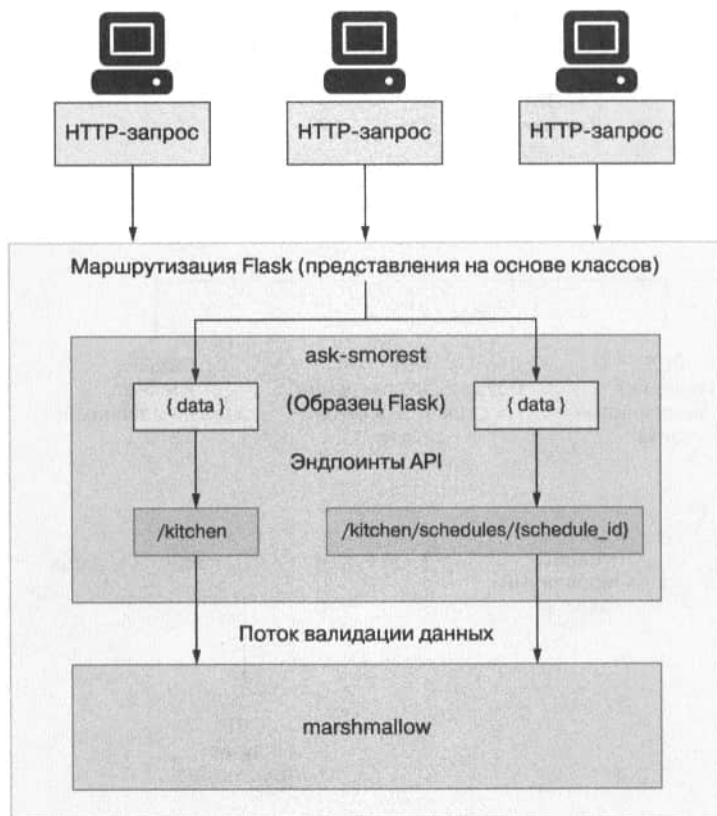


**Рис. 6.4.** API сервиса кухни имеет четыре URL-пути

## 6.6. ВВОДИМ В РАБОТУ FLASK-SMOREST

В этом разделе представлен фреймворк, который мы будем использовать для создания API сервиса кухни: flask-smorest (<https://github.com/marshmallow-code/flask-smorest>). Это фреймворк REST API, построенный на основе Flask и marshmallow. Flask — это популярный фреймворк для создания веб-приложений, а marshmallow — библиотека проверки данных, которая отвечает за перенос сложных структур данных в собственные объекты Python и обратно. Flask-smorest строится поверх обоих фреймворков, и это означает, что мы реализуем наши схемы API с помощью marshmallow, а эндпоинты API реализуем по образцу типичного приложения Flask (рис. 6.5). Как вы увидите, принципы и паттерны, которые мы использовали при создании API сервиса

заказов с помощью FastAPI, могут быть применены независимо от фреймворка, и мы применим тот же подход к созданию API сервиса кухни с помощью flask-smorest.



**Рис. 6.5.** Архитектура приложения, построенного с помощью flask-smorest. Flask-smorest реализует типичную схему Flask, которая позволяет создавать и настраивать эндпоинты нашего API точно так же, как мы делали бы это в стандартном приложении Flask

Создание API с помощью flask-smorest напоминает создание API с помощью FastAPI, но с двумя основными отличиями.

- *FastAPI* использует для валидации данных *pydantic*, а *flask-smorest* — *marshmallow*. Это означает, что в FastAPI для создания правил проверки данных мы используем родные подсказки типа Python, а в *marshmallow* — классы полей.
- *Flask* позволяет реализовывать эндпоинты API с помощью представлений на основе классов. Это означает, что мы можем использовать класс для представления пути URL и реализовать его HTTP-методы как методы класса. Представления на основе классов помогают писать более структурированный код и инкапсулировать специфическое поведение каждого пути URL

внутри класса. В отличие от этого, FastAPI позволяет определять эндпоинты только с помощью функций. Обратите внимание, что Starlette разрешает реализовывать маршруты на основе классов, так что это ограничение FastAPI в будущем может исчезнуть.

С этим разобрались, приступим к реализации API сервиса кухни!

## 6.7. ИНИЦИАЛИЗАЦИЯ ВЕБ-ПРИЛОЖЕНИЯ ДЛЯ API

В этом разделе мы настроим среду для начала работы над API сервиса кухни. Мы также создадим точку входа для приложения и добавим базовую конфигурацию для веб-сервера. Вы узнаете, как создать проект с flask-smorest и внедрять объекты конфигурации в приложения Flask.

Flask-smorest построен на основе фреймворка Flask, поэтому мы создадим веб-приложение по образцу типичного приложения Flask. Создайте папку `ch06/kitchen` для реализации API сервиса кухни. В нее скопируйте спецификацию API, которая доступна в файле `ch06/kitchen/oas.yaml`. С помощью команды `cd` перейдите в папку `ch06/kitchen` и выполните следующие команды для установки зависимостей, которые нам понадобятся для продолжения реализации:

```
$ pipenv install flask-smorest
```

### ПРИМЕЧАНИЕ

Если вы хотите убедиться, что устанавливаете ту же версию зависимостей, которую я использовал при написании этой главы, скопируйте файлы `ch06/kitchen/Pipfile` и `ch06/kitchen/Pipfile.lock` из репозитория GitHub на свою локальную машину и выполните команду `pipenv install`.

Введите следующую команду, чтобы активировать среду:

```
$ pipenv shell
```

Теперь, когда у нас есть необходимые библиотеки, создадим файл `kitchen/app.py`. Он будет содержать экземпляр объекта приложения Flask, который представляет наш веб-сервер. Мы также создадим экземпляр API объекта flask-smorest, который будет представлять наш API (листинг 6.6).

### Листинг 6.6. Инициализация объекта приложения Flask и объекта Api

```
# file: kitchen/app.py

from flask import Flask
from flask_smorest import Api

app = Flask(__name__)
kitchen_api = Api(app)
```

Создаем экземпляр объекта приложения Flask

Создаем экземпляр API объекта flask-smorest

Для работы flask-smorest требуются некоторые параметры конфигурации. Например, нужно указать версию OpenAPI, которую мы используем, название и версию нашего API. Мы передаем эту конфигурацию через объект приложения Flask. Flask предлагает различные стратегии для ввода конфигурации, но удобнее всего загрузить конфигурацию из класса. Создадим файл kitchen/config.py для наших параметров конфигурации. В нем мы добавим класс BaseConfig, который будет содержать общую конфигурацию для API (листинг 6.7).

### Листинг 6.7. Конфигурация для API сервиса заказов

```
# file: kitchen/config.py

class BaseConfig:
    API_TITLE = 'Kitchen API'
    API_VERSION = 'v1'
    OPENAPI_VERSION = '3.0.3'
    OPENAPI_JSON_PATH = 'openapi/kitchen.json'
    OPENAPI_URL_PREFIX = '/'
    OPENAPI_REDOC_PATH = '/redoc'
    OPENAPI_REDOC_URL =
        'https://cdn.jsdelivr.net/npm/redoc@next/bundles/redoc.standalone.js'
    OPENAPI_SWAGGER_UI_PATH = '/docs/kitchen'
    OPENAPI_SWAGGER_UI_URL = 'https://cdn.jsdelivr.net/npm/swagger-ui-dist/'
```

Название нашего API

Версия нашего API

Версия OpenAPI, которую мы используем

Путь к динамически сгенерированной JSON-спецификации

Префикс URL-пути для файла спецификации OpenAPI

Путь к пользовательскому интерфейсу Redoc API

Путь к скрипту, который будет использоваться для рендеринга пользовательского интерфейса Redoc

Путь к пользовательскому интерфейсу Swagger нашего API

Путь к скрипту, который будет использоваться для рендеринга пользовательского интерфейса Swagger

Теперь, когда конфигурация готова, можем передать ее объекту приложения Flask (листинг 6.8).

### Листинг 6.8. Загрузка конфигурации

```
# file: kitchen/app.py

from flask import Flask
from flask_smorest import Api

from config import BaseConfig

app = Flask(__name__)
app.config.from_object(BaseConfig)

kitchen_api = Api(app)
```

Импортируем класс BaseConfig, который определили ранее

Используем метод from\_object для загрузки конфигурации из класса

Теперь, когда точка входа для нашего приложения готова и настроена, перейдем к реализации эндпоинтов для API сервиса кухни.



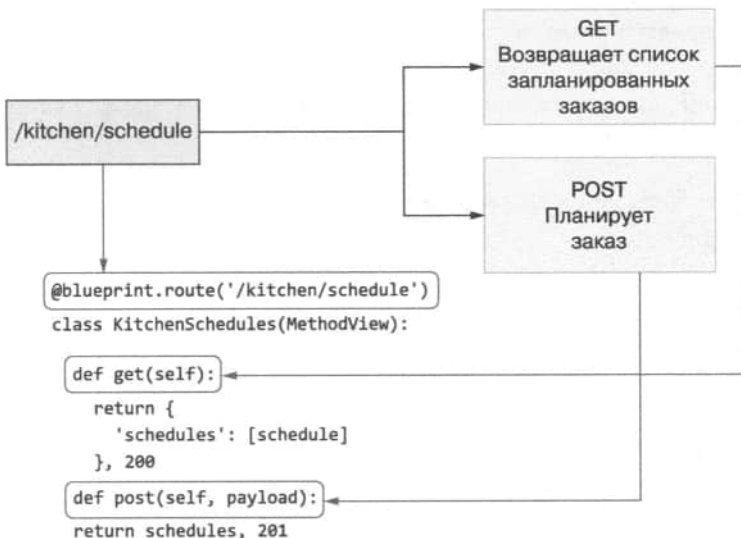
## 6.8. РЕАЛИЗАЦИЯ ЭНДПОИНТОВ API

В этом разделе мы реализуем эндпоинты API сервиса кухни с помощью flask-smorest. Поскольку flask-smorest построен поверх Flask, мы создаем эндпоинты для нашего API точно так же, как и для любого другого приложения Flask. Во Flask мы регистрируем эндпоинты, используя декоратор маршрутов Flask:

```
@app.route('/orders')
def process_order():
    pass
```

Декоратор маршрутов подходит для простых случаев, но для более сложных моделей приложений мы используем эскизы (blueprints) Flask. Они позволяют предоставить конкретную конфигурацию для группы URL-адресов. Для реализации эндпоинтов API сервиса кухни воспользуемся классом `Blueprint` от flask-smorest, который является подклассом `Blueprint` от Flask, поэтому дает нам функциональность эскизов Flask и расширяет ее за счет дополнительных возможностей и конфигурации, а именно генерирует документацию API и предоставляет модели для проверки полезной нагрузки.

Мы можем использовать декораторы маршрутов `Blueprint` для создания эндпоинта или пути URL. Как видно из рис. 6.6, функции подходят для URL-адресов, которые предоставляют только один метод HTTP. Когда URL ведет к нескольким HTTP-методам, удобнее использовать маршруты на основе классов, которые мы реализуем с помощью класса `MethodView` из Flask.



**Рис. 6.6.** Когда URL-адрес предоставляет доступ к нескольким HTTP-методам, удобнее реализовать его в виде представления на основе класса, методы которого соответствуют доступным HTTP-методам

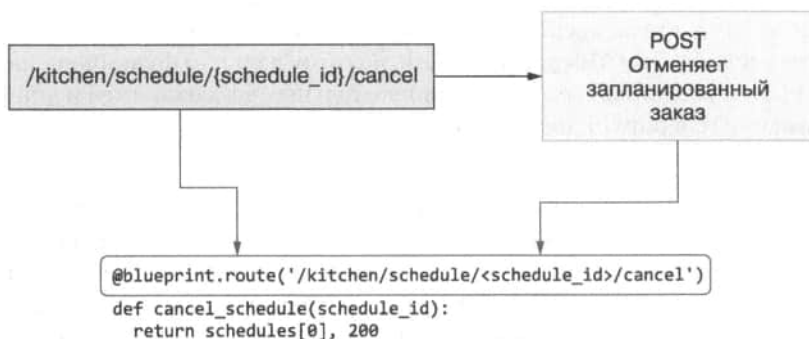
Как показано на рис. 6.7, с помощью `MethodView` мы представляем путь URL как класс, а соответствующие методы HTTP реализуем как методы этого класса.

Например, если у нас есть URL-путь `/kitchen`, который ведет к эндпоинтам GET и POST, мы можем реализовать следующее представление на основе класса:

```
class Kitchen(MethodView):
```

```
    def get(self):
        pass
```

```
    def post(self):
        pass
```



**Рис. 6.7.** Когда URL-адрес предоставляет доступ только к одному HTTP-методу, удобнее реализовать его в виде представления, основанного на функциях

В листинге 6.9 показано, как мы реализуем эндпоинты для API сервиса кухни, используя представления на основе классов и функций.

#### Листинг 6.9. Реализация эндпоинтов API сервиса кухни

```
# file: kitchen/api/api.py
```

```
import uuid
```

```
from datetime import datetime
```

```
from flask.views import MethodView
```

```
from flask_smorest import Blueprint
```

Создаем экземпляр класса  
Blueprint от flask-smorest

```
blueprint = Blueprint('kitchen', __name__, description='Kitchen API')
```

```
schedules = [{
```

```
    'id': str(uuid.uuid4()),
```

```
    'scheduled': datetime.now(),
```

```
    'status': 'pending',
```

Объявляем жестко  
заданное расписание

```

'order': [
    {
        'product': 'capuccino',
        'quantity': 1,
        'size': 'big'
    }
]
}]

Используем декоратор маршрутов
для регистрации класса или
функции в качестве URL-пути

@blueprint.route('/kitchen/schedules')
class KitchenSchedules(MethodView):

    Реализуем URL-путь к /kitchen/schedules
    как представление на основе классов

    def get(self):
        Каждое представление метода в представлении
        на основе классов названо соответственно
        HTTP-методу, который оно реализует
        return {
            'schedules': schedules
        }, 200

        Возвращаем как полезную
        нагрузку, так и статус-код

    def post(self, payload):
        return schedules[0], 201

    В угловых скобках
    определяем
    параметры URL

@blueprint.route('/kitchen/schedules/<schedule_id>')
class KitchenSchedule(MethodView):

    Включаем параметр URL-пути
    в сигнатуру функции

    def get(self, schedule_id):
        return schedules[0], 200

    def put(self, payload, schedule_id):
        return schedules[0], 200

    def delete(self, schedule_id):
        return '', 204

@blueprint.route(
    '/kitchen/schedules/<schedule_id>/cancel', methods=['POST']
)
    Реализуем URL-путь /kitchen/schedules/<schedule_id>/cancel
    как представление, основанное на функциях
    def cancel_schedule(schedule_id):
        return schedules[0], 200

@blueprint.route('/kitchen/schedules/<schedule_id>/status', methods=[GET])
    def get_schedule_status(schedule_id):
        return schedules[0], 200

```

Код из листинга 6.9 находится в файле `kitchen/api/api.py`. Сначала мы создаем экземпляр `Blueprint` от `flask-smorest`. Объект `Blueprint` позволяет нам установить эндпоинты и добавить к ним проверку данных.

Чтобы создать экземпляр `Blueprint`, мы должны передать два обязательных позиционных аргумента: название самого `Blueprint` и имя модуля, в котором реализованы маршруты. В данном случае мы передаем имя модуля с помощью атрибута `__name__`, который преобразуется в имя файла.

Как только `Blueprint` создан, прописываем в нем пути URL с помощью декоратора `route()`. Мы используем маршруты на основе классов для `/kitchen/schedules` и `/kitchen/schedules/{schedule_id}`, поскольку они предоставляют более одного HTTP-метода, и маршруты на основе функций для путей `/kitchen/schedules/{schedule_id}/cancel` и `/kitchen/schedules/{schedule_id}/status`, поскольку они предоставляют только один HTTP-метод. Возвращаем `mock`-объект `schedule` в каждом эндпоинте. Далее, в разделе 6.12, мы преобразуем его в динамическую коллекцию расписаний в памяти. Возвращаемое значение каждой функции — кортеж, где первый элемент — это полезная нагрузка, а второй — статус-код ответа.

Теперь, когда мы создали схему, можем прописать ее с помощью нашего объекта API в файле `kitchen/app.py` (листинг 6.10).

#### Листинг 6.10. Регистрация схемы с помощью объекта API

```
# file: kitchen/app.py

from flask import Flask
from flask_smorest import Api

from api.api import blueprint ← Импортируем схему,
from config import BaseConfig    которую определили ранее

app = Flask(__name__)
app.config.from_object(BaseConfig)

kitchen_api = Api(app)

kitchen_api.register_blueprint(blueprint) ← Регистрируем эскиз с помощью
                                           объекта API сервиса кухни
```

С помощью команды `cd` перейдите в каталог `ch06/kitchen`, а затем запустите приложение следующей командой:

```
$ flask run --reload
```

Как и в `Uvicorn`, флаг `--reload` запускает сервер с наблюдением за вашими файлами, так что он перезапускается, когда вы вносите изменения в код.

Если вы откроете URL `http://127.0.0.1:5000/docs`, то увидите интерактивный пользовательский интерфейс `Swagger`, динамически созданный на основе эндпоинтов, которые мы реализовали ранее. Вы также можете увидеть спецификацию `OpenAPI`, динамически сгенерированную `flask-smorest` по адресу `http://127.0.0.1:5000/openapi.json`. На данном этапе нашей реализации невозможно взаимодействовать с эндпоинтами через пользовательский интерфейс `Swagger`. Поскольку у нас еще нет моделей `marshmallow`, `flask-smorest` не умеет сериализовать данные и поэтому не возвращает полезную нагрузку. Однако все еще можно вызвать API с помощью `cURL` и посмотреть ответы. Если вы запустите `curl http://127.0.0.1:5000/`

kitchen/schedules, то получите mock-объект, который мы определили в модуле kitchen/api/api.py.

Все уже неплохо, но пришло время разнообразить реализацию, добавив модели marshmallow. Переходите к следующему разделу, чтобы узнать, как это сделать!

## 6.9. РЕАЛИЗАЦИЯ МОДЕЛЕЙ ПРОВЕРКИ ПОЛЕЗНОЙ НАГРУЗКИ С ПОМОЩЬЮ MARSHMALLOW

Flask-smorest использует модели marshmallow для проверки полезной нагрузки запросов и ответов. В этом разделе мы научимся работать с ними, реализуя схемы API сервиса кухни. Модели marshmallow помогут flask-smorest проверить нашу полезную нагрузку и сериализовать данные.

Как вы можете видеть в спецификации API сервиса кухни в файле ch06/kitchen/oas.yaml, API включает три схемы: ScheduleOrderSchema, которая содержит сведения, необходимые для планирования заказа; GetScheduledOrderSchema, которая представляет сведения о запланированном заказе; и OrderItemSchema, которая представляет коллекцию товаров в заказе. В листинге 6.11 показано, как реализовать эти схемы в виде моделей marshmallow в файле kitchen/api/schemas.py.

### Листинг 6.11. Определения схем для API сервиса заказов

# file: kitchen/api/schemas.py

```
from marshmallow import Schema, fields, validate, EXCLUDE
```

```
class OrderItemSchema(Schema):
    class Meta:
        unknown = EXCLUDE

    product = fields.String(required=True)
    size = fields.String(
        required=True, validate=validate.OneOf(['small', 'medium', 'big'])
    )
    quantity = fields.Integer(
        validate=validate.Range(1, min_inclusive=True), required=True
    )
```

Используем метакласс для запрета неизвестных свойств

```
class ScheduleOrderSchema(Schema):
    class Meta:
        unknown = EXCLUDE

    order = fields.List(fields.Nested(OrderItemSchema), required=True)
```

```
class GetScheduledOrderSchema(ScheduleOrderSchema):
    id = fields.UUID(required=True)
    scheduled = fields.DateTime(required=True)
```

Применяем наследование классов для повторного использования определений существующей схемы

```

    status = fields.String(
        required=True,
        validate=validate.OneOf(
            ["pending", "progress", "cancelled", "finished"]
        ),
    )
)

class GetScheduledOrdersSchema(Schema):
    class Meta:
        unknown = EXCLUDE

    schedules = fields.List(
        fields.Nested(GetScheduledOrderSchema), required=True
    )

class ScheduleStatusSchema(Schema):
    class Meta:
        unknown = EXCLUDE

    status = fields.String(
        required=True,
        validate=validate.OneOf(
            ["pending", "progress", "cancelled", "finished"]
        ),
    )
)

```

Чтобы создать модели `marshmallow`, мы создаем подклассы класса `Schema`. Определяем свойства моделей с помощью таких классов полей, как `String` и `Integer`. `Marshmallow` использует эти определения свойств для проверки полезной нагрузки на соответствие модели. Чтобы настроить поведение моделей `marshmallow`, воспользуемся классом `Meta` — установим для атрибута `unknown` значение `EXCLUDE`, которое предписывает `marshmallow` аннулировать полезную нагрузку с неизвестными свойствами.

Теперь, когда наши модели проверки готовы, можем связать их с нашими представлениями. В листинге 6.12 показано, как использовать модели для проверки полезной нагрузки запроса и ответа на наших эндпоинтах. Чтобы добавить проверку полезной нагрузки запроса в представление, мы вводим декоратор `arguments()` из `Blueprint` в сочетании с моделью `marshmallow`, а для проверки полезной нагрузки ответа — декоратор `response()`.

#### Листинг 6.12. Добавление валидации к эндпоинтам API

```

# file: kitchen/api/api.py

import uuid
from datetime import datetime

from flask.views import MethodView
from flask_smorest import Blueprint

```

```
from api.schemas import (
    GetScheduledOrderSchema,
    ScheduleOrderSchema,
    GetScheduledOrdersSchema,
    ScheduleStatusSchema,
)
```

← Импортируем наши модели marshmallow

```
blueprint = Blueprint('kitchen', __name__, description='Kitchen API')
```

```
...
```

```
@blueprint.route('/kitchen/schedules')
class KitchenSchedules(MethodView):
```

Используем декоратор response() для полезной нагрузки ответа

```
    @blueprint.response(status_code=200, schema=GetScheduledOrdersSchema)
    def get(self):
        return {'schedules': schedules}
```

```
    @blueprint.arguments(ScheduleOrderSchema)
    @blueprint.response(status_code=201, schema=GetScheduledOrderSchema)
```

← Используем декоратор arguments() для полезной нагрузки запроса

```
    def post(self, payload):
        return schedules[0]
```

← Присваиваем параметру status\_code значение желаемого статус-кода

```
@blueprint.route('/kitchen/schedules/<schedule_id>')
class KitchenSchedule(MethodView):
```

```
    @blueprint.response(status_code=200, schema=GetScheduledOrderSchema)
    def get(self, schedule_id):
        return schedules[0]
```

```
    @blueprint.arguments(ScheduleOrderSchema)
    @blueprint.response(status_code=200, schema=GetScheduledOrderSchema)
    def put(self, payload, schedule_id):
        return schedules[0]
```

```
    @blueprint.response(status_code=204)
    def delete(self, schedule_id):
        return
```

```
    @blueprint.response(status_code=200, schema=GetScheduledOrderSchema)
    @blueprint.route(
        '/kitchen/schedules/<schedule_id>/cancel', methods=['POST']
    )
    def cancel_schedule(schedule_id):
        return schedules[0]
```

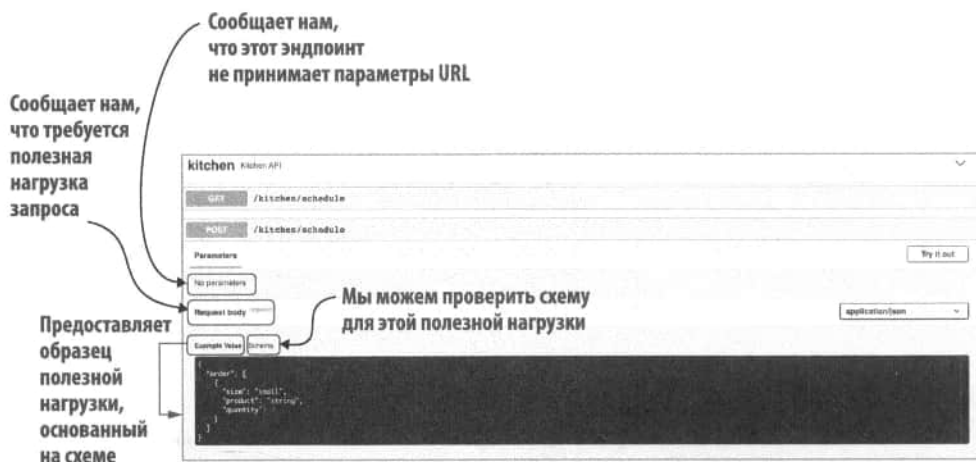
```
    @blueprint.response(status_code=200, schema=ScheduleStatusSchema)
    @blueprint.route(
        '/kitchen/schedules/<schedule_id>/status', methods=['GET']
    )
    def get_schedule_status(schedule_id):
        return schedules[0]
```

Оформив методы и функции с помощью декоратора `response()`, мы больше не должны возвращать кортеж из полезной нагрузки и статус-кода. `Flask-smorest` позаботится о добавлении статус-кода за нас. По умолчанию `flask-smorest` добавляет к нашим ответам статус-код 200. Если мы хотим изменить это, нам просто нужно указать в декораторе желаемый статус-код с помощью параметра `status_code`.

В то время как декоратор `arguments()` проверяет и десериализует полезную нагрузку запроса, декоратор `response()` не выполняет проверку и только сериализует полезную нагрузку. В разделе 6.11 мы подробнее обсудим эту возможность и посмотрим, как обеспечить проверку данных перед сериализацией.

Чтобы ознакомиться с последствиями новых изменений в реализации, снова посетите URL <http://127.0.0.1:5000/docs>. Если вы запускаете сервер с флагом `--reload`, изменения будут автоматически перезагружены. В ином случае остановите сервер и запустите его снова. Как видно на рис. 6.8, `flask-smorest` теперь распознает схемы валидации, которые должны использоваться в API, поэтому они представлены в пользовательском интерфейсе Swagger. Если вы теперь поэкспериментируете с UI, например выбрав endpoint GET `/kitchen/schedules`, то сможете увидеть полезную нагрузку ответа.

API хорошо выглядит, и мы почти закончили его реализацию. Следующим шагом будет добавление параметров URL-запроса в endpoint GET `/kitchen/schedules`.



**Рис. 6.8.** Пользовательский интерфейс Swagger показывает схему запроса полезной нагрузки эндпоинта POST `/kitchen/schedules` и приводит ее образец



## 6.10. ПРОВЕРКА ПАРАМЕТРОВ ЗАПРОСА URL

Как показано в листинге 6.13, эндпоинт GET /kitchen/schedules принимает три параметра URL-запроса:

- **progress** (булево значение) — указывает, находится ли заказ в процессе выполнения;
- **limit** (целое число) — ограничивает количество результатов, возвращаемых эндпоинтом;
- **since** (дата/время) — фильтрует результаты по времени заказа. Дата в формате «дата/время» — это дата ISO со следующей структурой: YYYY-MM-DDTHH:mm:ssZ. Пример такого формата — 2021-08-31T01:01:01Z. Для получения дополнительной информации об этом формате см.: <https://tools.ietf.org/html/rfc3339#section-5.6>.

**Листинг 6.13.** Спецификация параметров URL-запроса GET /kitchen/schedules

```
# file: kitchen/oas.yaml
```

```
paths:
  /kitchen/schedules:
    get:
      summary: Returns a list of orders scheduled for production
      parameters:
        - name: progress
          in: query
          description: >-
            Whether the order is in progress or not.
            In progress means it's in production in the kitchen.
          required: false
          schema:
            type: boolean
        - name: limit
          in: query
          required: false
          schema:
            type: integer
        - name: since
          in: query
          required: false
          schema:
            type: string
            format: 'date-time'
```

Как реализовать параметры запроса URL в flask-smorest? Нужно создать новую модель marshmallow для их представления. Определим параметры URL-запросов для API сервиса кухни с помощью marshmallow (листинг 6.14). Можете добавить модель для этих параметров в файл kitchen/api/schemas.py вместе с другими моделями marshmallow.

**Листинг 6.14.** Параметры URL-запроса в marshmallow

```
# file: kitchen/api/schemas.py

from marshmallow import Schema, fields, validate, EXCLUDE

...

class GetKitchenScheduleParameters(Schema):
    class Meta:
        unknown = EXCLUDE

    progress = fields.Boolean()
    limit = fields.Integer()
    since = fields.DateTime()
```

Определяем поля  
параметров URL-запроса

Мы прописываем схему для параметров URL-запроса с помощью декоратора `arguments()` Blueprint (листинг 6.15). Указываем, что свойства, определенные в схеме, ожидаются в URL, поэтому устанавливаем параметр `location` равным `query`.

**Листинг 6.15.** Добавление параметров URL-запроса в GET /kitchen/schedules

```
# file: kitchen/api/api.py

import uuid
from datetime import datetime

from flask.views import MethodView
from flask_smorest import Blueprint

from api.schemas import (
    GetScheduledOrderSchema, ScheduleOrderSchema, GetScheduledOrdersSchema,
    ScheduleStatusSchema, GetKitchenScheduleParameters
)

blueprint = Blueprint('kitchen', __name__, description='Kitchen API')

...

@blueprint.route('/kitchen/schedules')
class KitchenSchedules(MethodView):

    @blueprint.arguments(GetKitchenScheduleParameters, location='query')
    @blueprint.response(status_code=200, schema=GetScheduledOrdersSchema)
    def get(self, parameters):
        return schedules
```

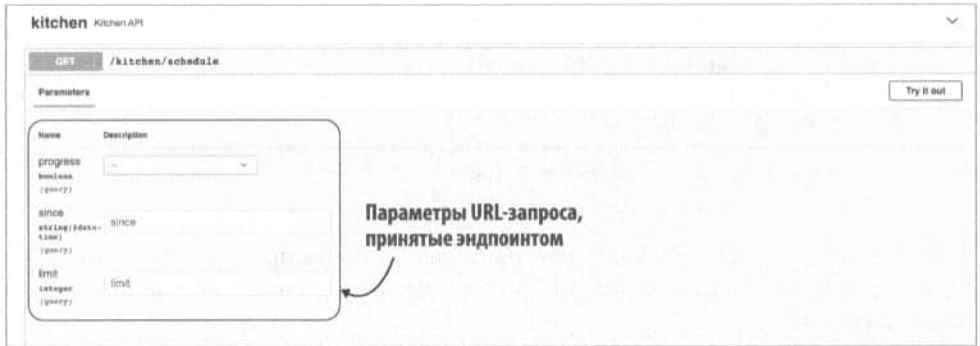
Импортируем модель marshmallow  
для параметров URL-запроса

Прописываем модель с помощью  
декоратора arguments() и устанавливаем  
параметру location значение query

Используем параметр URL-запроса  
в сигнатуре функции

Если вы перезагрузите пользовательский интерфейс Swagger, то увидите, что энд-поинт GET /kitchen/schedules теперь принимает три необязательных параметра URL-запроса (рис. 6.9). Мы должны передавать эти параметры нашему уровню бизнес-логики, который будет использовать их для фильтрации списка результатов.

Параметры URL-запроса представлены в виде словаря. Если пользователь не задал никаких параметров запроса, словарь будет пустым и, следовательно, будет иметь значение `False`. Поскольку параметры URL-запроса необязательны, мы проверяем их наличие с помощью метода `get()` словаря. Если параметр не задан, то `get()` возвращает `None`; таким образом, при любом другом значении мы понимаем, что параметр задан. К реализации уровня бизнес-логики мы приступим только в главе 7, но сейчас можем использовать параметры запроса, чтобы отфильтровать список, соответствующий расписанию, в памяти (листинг 6.16).



**Рис. 6.9.** Пользовательский интерфейс Swagger показывает параметры URL-запроса эндпоинта GET /kitchen/schedules и предлагает поля формы, которые можно заполнить, чтобы поэкспериментировать с различными значениями

#### Листинг 6.16. Использование фильтров в GET /kitchen/schedules

```
# file: kitchen/api/api.py
```

```
...
```

```
@blueprint.route('/kitchen/schedules')
```

```
class KitchenSchedules(MethodView):
```

```
    @blueprint.arguments(GetKitchenScheduleParameters, location='query')
```

```
    @blueprint.response(status_code=200, schema=GetScheduledOrdersSchema)
```

```
    def get(self, parameters):
```

```
        if not parameters:
```

```
            return {'schedules': schedules}
```

← Если параметр не задан,  
возвращаем полное расписание

```
        query_set = [schedule for schedule in schedules]
```

```
        in_progress = parameters.get('progress')
```

```
        if in_progress is not None:
```

```
            if in_progress:
```

```
                query_set = [
```

```
                    schedule for schedule in schedules
```

```
                    if schedule['status'] == 'progress'
```

```
                ]
```

← Если пользователь задал  
какие-либо параметры  
URL-запроса, используем их  
для фильтрации расписания

← Проверяем наличие каждого  
параметра URL-запроса  
с помощью метода `get()` словаря

```

else:
    query_set = [
        schedule for schedule in schedules
        if schedule['status'] != 'progress'
    ]

since = parameters.get('since')
if since is not None:
    query_set = [
        schedule for schedule in schedules
        if schedule['scheduled'] >= since
    ]

limit = parameters.get('limit')
if limit is not None and len(query_set) > limit:
    query_set = query_set[:limit]

return {'schedules': query_set}
...

```

Если установлено ограничение и его значение меньше длины query\_set, возвращаем подмножество query\_set

Возвращаем отфильтрованный список, соответствующий расписанию

Теперь, когда мы знаем, как обрабатывать параметры запросов URL с помощью flask-smorest, нам нужно рассмотреть еще одну тему — проверку данных перед сериализацией.

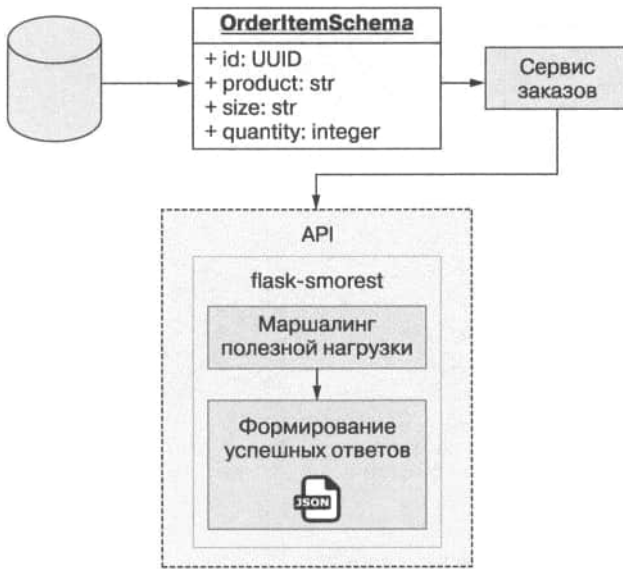
## 6.11. ПРОВЕРКА ДАННЫХ ПЕРЕД СЕРИАЛИЗАЦИЕЙ ОТВЕТА

Итак, у нас есть схемы для проверки полезной нагрузки запроса и мы связали их с нашими маршрутами. Сейчас нам нужно убедиться, что полезная нагрузка ответа тоже проверена. В этом разделе мы будем использовать модели marshmallow для проверки полезной нагрузки ответа, но вы можете применять тот же подход для проверки любого типа данных, например объектов конфигурации.

Когда мы отправляем полезную нагрузку в ответе, flask-smorest сериализует ее с помощью marshmallow. Однако он не проверяет, правильно ли она сформирована<sup>1</sup> (рис. 6.10). В отличие от marshmallow, FastAPI проверяет данные перед тем, как сериализовать их для ответа (рис. 6.11).

То, что marshmallow не выполняет проверку перед сериализацией, не так уж и плохо. На самом деле можно даже сказать, что это желательное поведение, поскольку так задача сериализации отделяется от задачи проверки полезной нагрузки.

<sup>1</sup> До версии 3.0.0 marshmallow выполнял такую проверку перед сериализацией (см. журнал изменений: <https://github.com/marshmallow-code/marshmallow/blob/dev/CHANGELOG.rst#300-2019-08-18>).



**Рис. 6.10.** Обработка полезной нагрузки с помощью фреймворка flask-smorest.

Предполагается, что полезная нагрузка ответа поступает из «доверенного источника» и, следовательно, не проверяется перед маршалингом

Marshmallow может не выполнять проверку перед сериализацией по двум причинам (<https://mng.bz/9Vwx>):

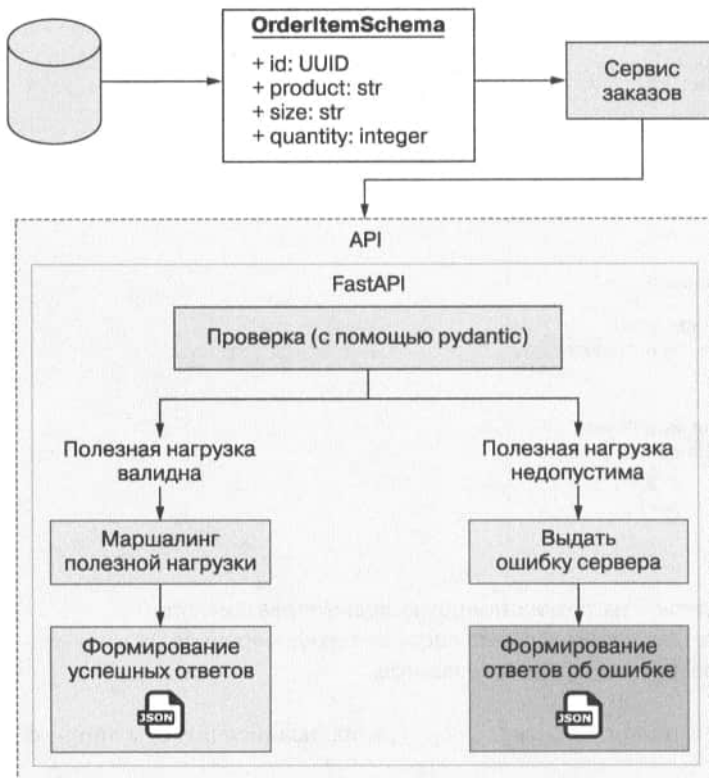
- это повышает производительность, поскольку проверка выполняется медленно;
- данные, поступающие с сервера, считаются надежными, поэтому их не нужно проверять.

Однако если вы достаточно долго работали с API и вообще с сайтами, то знаете, что доверять можно очень немногому, даже в собственной системе.

### ПОДХОД С НУЛЕВЫМ ДОВЕРИЕМ ДЛЯ НАДЕЖНЫХ API

API-интеграции терпят неудачу, если сервер отправляет неправильную полезную нагрузку клиенту или, наоборот, клиент отправляет неправильно сформированную полезную нагрузку на сервер. Когда это возможно, при разработке систем лучше использовать подход «нулевого доверия» и проверять все данные независимо от их происхождения.

Данные, которые мы отправляем с помощью API сервиса кухни, поступают из базы данных. В главе 7 мы изучим паттерны и методы, позволяющие убедиться, что наша база данных содержит нужные данные в требуемом формате. Однако даже при самых строгих мерах безопасности всегда существует вероятность того, что в базу попадут неправильно сформированные данные. Поскольку мы не хотим ухудшить пользовательский опыт, лучше всегда проверять данные перед сериализацией.



**Рис. 6.11.** Обработка полезной нагрузки данных с помощью фреймворка FastAPI. Перед отправкой ответа FastAPI проверяет, соответствует ли полезная нагрузка указанной схеме

К счастью, проверить данные с помощью `marshmallow` очень просто. Нужно лишь получить экземпляр схемы, которую требуется проверить, и использовать ее метод `validate()` для передачи данных. Если `validate()` обнаруживает ошибки, он не вызывает исключения, а возвращает словарь с ошибками или пустой словарь (когда ошибок нет). Чтобы понять, как это работает, откройте оболочку Python, набрав `python` в терминале, и запустите следующий код:

```
>>> from api.schemas import GetScheduledOrderSchema
>>> GetScheduledOrderSchema().validate({'id': 'asdf'})
{'order': ['Missing data for required field.'], 'scheduled': ['Missing
➔ data for required field.'], 'status': ['Missing data for required
➔ field.'], 'id': ['Not a valid UUID.']}
```

После импорта схемы в первой строке мы передаем во второй строке неверное представление расписания, содержащее только поле `id`. В третьей строке `marshmallow`

услужливо сообщает, что поля `order`, `scheduled` и `status` отсутствуют, а поле `id` содержит недопустимый UUID. Мы можем использовать эту информацию, чтобы выдать сообщение об ошибке на сервере, как показано в листинге 6.17.

### Листинг 6.17. Проверка данных перед сериализацией

```
# file: kitchen/api/api.py

import copy
import uuid
from datetime import datetime

from flask.views import MethodView
from flask_smorest import Blueprint
from marshmallow import ValidationError  ← Импортируем класс ValidationError
                                        из marshmallow

...

@blueprint.route('/kitchen/schedules')
class KitchenSchedules(MethodView):

    @blueprint.arguments(GetKitchenScheduleParameters, location='query')
    @blueprint.response(status_code=200, schema=GetScheduledOrdersSchema)
    def get(self, parameters):
        for schedule in schedules:
            schedule = copy.deepcopy(schedule)
            schedule['scheduled'] = schedule['scheduled'].isoformat()
            errors = GetScheduledOrderSchema().validate(schedule)  ← Заносим ошибки проверки
                                                                    в переменную errors
            if errors:
                raise ValidationError(errors)  ← Если функция validate() обнаруживает
                                                ошибки, выбрасываем исключение
                                                ValidationError
        ...
        return {'schedules': query_set}

...
```

Перед сборкой и возвратом набора запросов мы проверяем расписание в представлении метода GET `/kitchen/schedules`, проходя по списку с расписаниями для валидации по одному за раз. Перед валидацией создаем глубокую копию расписания, чтобы преобразовать его объект `datetime` в строку даты ISO, поскольку именно такой формат ожидается методом валидации. Получив ошибку валидации, вызываем исключение `ValidationError` от `marshmallow`, которое автоматически преобразует сообщение об ошибке в соответствующий HTTP-ответ.

Имейте в виду, что в `marshmallow` возможны некоторые проблемы с валидацией, особенно если ваши модели содержат сложные конфигурации для определения того, какие поля должны быть сериализованы, а какие — нет (для получения дополнительной информации см. <https://github.com/marshmallow-code/marshmallow/issues/682>). Учитывайте также, что валидация — медленный процесс, поэтому, если вы обрабатываете объемную полезную нагрузку, стоит использовать для валидации

другой инструмент, проверять только часть данных или вообще пропустить этот процесс. Тем не менее, когда это возможно, данные лучше проверять.

На этом реализация функциональности API сервиса кухни завершена. Тем не менее API по-прежнему возвращает одно и то же расписание во всех эндпоинтах. Прежде чем завершить эту главу, добавим минимальную реализацию списка расписаний в памяти, чтобы сделать наш API динамичным. Это позволит нам проверить, что все эндпоинты функционируют так, как задумано.

## 6.12. РЕАЛИЗАЦИЯ СПИСКА РАСПИСАНИЙ В ПАМЯТИ

В этом разделе мы реализуем простое представление расписаний в памяти, чтобы получать динамические результаты из API. К концу раздела мы сможем планировать заказы, обновлять их и отменять через API. Поскольку расписания представлены в памяти в виде списка, при перезапуске сервера мы теряем информацию из нашего предыдущего сеанса. В следующей главе мы решим эту проблему.

Итак, наша коллекция расписаний представлена списком Python, и мы будем просто добавлять в него элементы и удалять их на уровне API. В листинге 6.18 показано, какие изменения необходимо внести в `kitchen/api/api.py`, чтобы это стало возможным.

### Листинг 6.18. Реализация расписаний в памяти

```
# file: kitchen/api/api.py
```

```
import copy
import uuid
from datetime import datetime
```

```
from flask import abort
```

```
...
```

```
schedules = []
```

← Инициализируем расписания  
в виде пустого списка

```
def validate_schedule(schedule):
    schedule = copy.deepcopy(schedule)
    schedule['scheduled'] = schedule['scheduled'].isoformat()
    errors = GetScheduledOrderSchema().validate(schedule)
    if errors:
        raise ValidationError(errors)
```

← Преобразуем наш код проверки  
данных в функцию

```
@blueprint.route('/kitchen/schedules')
class KitchenSchedules(MethodView):
```

```
    @blueprint.arguments(GetKitchenScheduleParameters, location='query')
    @blueprint.response(GetScheduledOrdersSchema)
    def get(self, parameters):
        ...
```



```

@blueprint.arguments(ScheduleOrderSchema)
@blueprint.response(status_code=201, schema=GetScheduledOrderSchema,)
def post(self, payload):
    payload['id'] = str(uuid.uuid4())
    payload['scheduled'] = datetime.utcnow()
    payload['status'] = 'pending'
    schedules.append(payload)
    validate_schedule(payload)
    return payload

```

← Задаем атрибуты расписания на стороне сервера, такие как идентификатор

```

@blueprint.route('/kitchen/schedules/<schedule_id>')
class KitchenSchedule(MethodView):

```

```

    @blueprint.response(status_code=200, schema=GetScheduledOrderSchema)
    def get(self, schedule_id):
        for schedule in schedules:
            if schedule['id'] == schedule_id:
                validate_schedule(schedule)
                return schedule
        abort(404, description=f'Resource with ID {schedule_id} not found')

```

← Если расписание не найдено, возвращаем ответ 404

```

    @blueprint.arguments(ScheduleOrderSchema)
    @blueprint.response(status_code=200, schema=GetScheduledOrderSchema)
    def put(self, payload, schedule_id):
        for schedule in schedules:
            if schedule['id'] == schedule_id:
                schedule.update(payload)
                validate_schedule(schedule)
                return schedule
        abort(404, description=f'Resource with ID {schedule_id} not found')

```

← Когда пользователь обновляет расписание, обновляем соответствующее свойство содержимым полезной нагрузки

```

    @blueprint.response(status_code=204)
    def delete(self, schedule_id):
        for index, schedule in enumerate(schedules):
            if schedule['id'] == schedule_id:
                schedules.pop(index)
                return
        abort(404, description=f'Resource with ID {schedule_id} not found')

```

← Удаляем расписание из списка и возвращаем пустой ответ

```

    @blueprint.response(status_code=200, schema=GetScheduledOrderSchema)
    @blueprint.route(
        '/kitchen/schedules/<schedule_id>/cancel', methods=['POST']
    )
    def cancel_schedule(schedule_id):
        for schedule in schedules:
            if schedule['id'] == schedule_id:
                schedule['status'] = 'cancelled'
                validate_schedule(schedule)
                return schedule
        abort(404, description=f'Resource with ID {schedule_id} not found')

```

← Устанавливаем для расписания статус «отменено»

```

    @blueprint.response(status_code=200, schema=ScheduleStatusSchema)
    @blueprint.route(
        '/kitchen/schedules/<schedule_id>/status', methods=['GET']
    )

```

```
def get_schedule_status(schedule_id):
    for schedule in schedules:
        if schedule['id'] == schedule_id:
            validate_schedule(schedule)
            return {'status': schedule['status']}
    abort(404, description=f'Resource with ID {schedule_id} not found')
```

Мы инициализируем пустой список и присваиваем его переменной с именем `schedules`. Мы также преобразуем наш код проверки данных в независимую функцию с именем `validate_schedule()`, чтобы затем повторно использовать его в других методах представления или функциях. Когда полезная нагрузка расписания поступает в метод `post()` класса `KitchenSchedules`, мы устанавливаем атрибуты на стороне сервера: идентификатор, запланированное время и статус. В одиночных эндпоинтах мы ищем запрошенное расписание, перебирая список расписаний и проверяя их идентификаторы. Если запрашиваемое расписание не найдено, возвращаем ответ 404.

Если вы перезагрузите пользовательский интерфейс Swagger и протестируете эндпоинты, то увидите, что теперь можете добавлять расписания, обновлять их, отменять, фильтровать и удалять, а также получать из них подробную информацию.

## 6.13. ПЕРЕОПРЕДЕЛЕНИЕ ДИНАМИЧЕСКИ ГЕНЕРИРУЕМОЙ СПЕЦИФИКАЦИИ API FLASK-SMOREST

Как вы узнали в разделе 6.4, спецификации API, динамически генерируемые из кода, подходят для тестирования и визуализации реализации, но перед публикацией API нужно убедиться, что мы предоставляем его проектную документацию. Для этого мы переопределим динамически генерируемую документацию API в `flask-smorest` (листинг 6.19). Сначала нужно установить PyYAML, который мы будем использовать для загрузки проектной документации API:

```
$ pipenv install pyyaml
```

Мы переопределяем свойство `spec` объекта API с помощью кастомного объекта `APISpec`. Переопределим также метод `to_dict()` объекта `APISpec`, чтобы он возвращал нашу проектную документацию API.

**Листинг 6.19.** Переопределение динамически генерируемой спецификации API `flask-smorest`

```
# file: kitchen/app.py

from pathlib import Path

import yaml
from apispec import APISpec
```

```
from flask import Flask
from flask_smorest import Api

from api.api import blueprint
from config import BaseConfig

app = Flask(__name__)

app.config.from_object(BaseConfig)

kitchen_api = Api(app)

kitchen_api.register_blueprint(blueprint)

api_spec = yaml.safe_load((Path(__file__).parent/ "oas.yaml").read_text())
spec = APISpec(
    title=api_spec["info"]["title"],
    version=api_spec["info"]["version"],
    openapi_version=api_spec["openapi"],
)
spec.to_dict = lambda: api_spec
kitchen_api.spec = spec
```

На этом мы завершаем наше путешествие по созданию REST API с использованием Python. В следующей главе разберем реализацию остальной части сервиса с помощью лучших практик и полезных паттернов проектирования. Дальше будет еще веселее!

## РЕЗЮМЕ

- Вы можете создавать REST API на Python с помощью таких фреймворков, как FastAPI и flask-smorest. У них отличные экосистемы инструментов и библиотек, облегчающих создание API.
- FastAPI — это современный API-фреймворк, который упрощает создание высокопроизводительных и надежных REST API. FastAPI построен на базе Starlette и pydantic. Starlette — это высокопроизводительный асинхронный серверный фреймворк, а pydantic — библиотека проверки данных, которая использует подсказки типов для создания правил проверки.
- Flask-smorest построен поверх Flask и работает как схема Flask. Flask — один из самых популярных фреймворков Python, и через flask-smorest вы можете использовать его богатую экосистему библиотек, чтобы упростить создание API.
- FastAPI задействует pydantic для валидации данных. Pydantic — это современный фреймворк, который использует подсказки типов для определения правил проверки, что позволяет получить более чистый и легко читаемый код. По умолчанию FastAPI проверяет полезную нагрузку как запроса, так и ответа.

- Flask-smorest позволяет проверять данные с помощью `marshmallow`. Это фреймворк, который для определения правил валидации использует поля классов. По умолчанию `flask-smorest` не проверяет полезную нагрузку ответов, но вы можете делать это, используя метод `validate()` моделей `marshmallow`.
- С помощью `flask-smorest` вы можете использовать `MethodView` Flask для создания представлений на основе классов, которые соответствуют путям URL. В представлении, основанном на классе, вы реализуете методы HTTP как методы класса, например `get()` и `post()`.
- Паттерн толерантного читателя работает по принципам закона Постела, который рекомендует быть терпимыми к ошибкам в HTTP-запросах и проверять полезную нагрузку ответа. При разработке API необходимо соблюдать баланс между использованием преимуществ паттерна толерантного читателя и риском неудачи интеграции из-за таких ошибок, как опечатки.

# Паттерны реализации сервисов

---

## В этой главе

- ✓ Как гексагональная архитектура помогает разрабатывать слабосвязанные сервисы.
- ✓ Создание уровня бизнес-логики для микросервиса и реализация моделей баз данных с помощью SQLAlchemy.
- ✓ Использование паттерна Репозиторий для отделения уровня данных от уровня бизнес-логики.
- ✓ Использование паттерна Единица работы для обеспечения атомарности всех транзакций и применение принципа инверсии зависимостей для создания программного обеспечения, устойчивого к изменениям.
- ✓ Применение принципа инверсии управления и паттерна Внедрение зависимостей для разделения компонентов, зависящих друг от друга.

В этой главе вы узнаете, как реализовать уровень бизнес-логики микросервиса. В предыдущих главах при разработке и внедрении REST API мы использовали представление в памяти ресурсов, управляемых сервисом. Этот подход позволил нам сохранить простоту реализации и сосредоточиться на уровне API сервиса.

Теперь мы завершим реализацию сервиса заказов, добавив уровень бизнес-логики и уровень данных. Уровень бизнес-логики будет реализовывать такие возможности сервиса заказов, как прием заказов, обработка платежей или планирование заказов

на изготовление. Для выполнения перечисленных задач сервис заказов должен взаимодействовать с другими сервисами, и мы изучим полезные паттерны для управления этими интеграциями.

Уровень данных будет отвечать за возможности сервиса по управлению данными. Сервис заказов владеет и управляет данными о заказах, поэтому мы реализуем постоянное решение для хранения данных и интерфейс к нему. При этом, представляя собой средство информирования пользователей о жизненном цикле заказа, сервис заказов также должен получать данные от других сервисов, например для отслеживания статуса заказа во время изготовления и доставки. Мы также рассмотрим полезные паттерны для управления доступом к этим сервисам.

Чтобы понять принципы реализации сервиса, мы также рассмотрим элементы архитектурного решения, необходимые для обеспечения слабой связанности всех частей нашего микросервиса. Слабая связанность позволит нам гарантировать, что мы сможем изменить реализацию конкретного компонента без необходимости вносить изменения в другие зависящие от него компоненты. Это также сделает нашу кодовую базу в целом более читабельной, удобной для сопровождения и тестирования. Код для этой главы доступен в каталоге `ch07` в репозитории к книге.

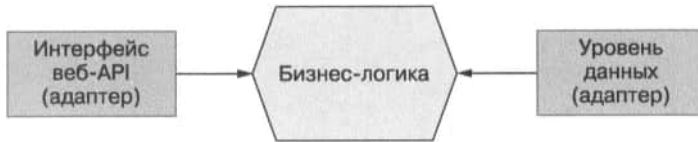
## 7.1. ГЕКСАГОНАЛЬНЫЕ АРХИТЕКТУРЫ ДЛЯ МИКРОСЕРВИСОВ

В этой главе мы рассмотрим гексагональную архитектуру и то, как применять ее при проектировании сервиса заказов. В главе 2 мы представили паттерн трехуровневой архитектуры, позволившей организовать компоненты приложения модульным и слабосвязанным способом. Здесь мы продолжим эту идею, используя уже гексагональную архитектуру.

В 2005 году Алистер Кокберн представил *концепцию гексагональной (шестиугольной) архитектуры*, также называемой *архитектурой портов и адаптеров*, как способ помочь разработчикам программного обеспечения разделить свой код на слабосвязанные компоненты<sup>1</sup>. Как показано на рис. 7.1, идея гексагональной архитектуры заключается в том, что в любом приложении есть основная часть логики, которая реализует возможности сервиса, а к ней мы «присоединяем» *адаптеры*,

<sup>1</sup> Cockburn A. Hexagonal Architecture («Гексагональная архитектура»). <https://alistair.cockburn.us/hexagonal-architecture/>. Возможно, вам интересно, почему именно шестиугольная, а не пяти- или семиугольная. Как отмечает Алистер, это «шестиугольник не потому, что важно число шесть», а потому, что он помогает визуально подчеркнуть идею связи основного приложения с внешними компонентами через порты (стороны шестиугольника) и позволяет нам представить две основные стороны приложения: общедоступную (веб-компоненты, API и т. д.) и внутреннюю (базы данных, сторонние интеграции и т. д.).

которые помогают ядру взаимодействовать с внешними компонентами. Например, веб-API — это адаптер, который помогает ядру взаимодействовать с веб-клиентами через Интернет. То же самое касается базы данных, которая является просто внешним компонентом, помогающим сервису сохранять данные. Мы можем при желании заменить базу данных, а сервис будет все тем же. Следовательно, база данных также является адаптером.



**Рис. 7.1.** В гексагональной архитектуре мы выделяем базовый уровень в приложении — уровень бизнес-логики, который реализует возможности сервиса. Другие компоненты, такие как интерфейс веб-API или база данных, считаются адаптерами, зависящими от уровня бизнес-логики

Как это помогает нам создавать слабосвязанные сервисы? Гексагональная архитектура требует, чтобы мы строго разделяли основную логику сервиса и логику для адаптеров. Другими словами, логика, реализующая наш уровень веб-API, не должна пересекаться с реализацией основной бизнес-логики. То же самое касается и базы данных: независимо от выбранной технологии и ее особенностей она не должна мешать основной бизнес-логике. Как этого достичь? Путем создания портов между основным уровнем бизнес-логики и адаптерами. *Порты* — это интерфейсы, которые не зависят от технологии и соединяют уровень бизнес-логики с адаптерами. Позже в этой главе вы познакомитесь с некоторыми паттернами проектирования, которые помогут нам спроектировать эти порты или интерфейсы.

При разработке взаимосвязи между основной бизнес-логикой и адаптерами мы применяем принцип инверсии зависимостей, который гласит следующее (рис. 7.2).

- Высокоуровневые модули не должны зависеть от низкоуровневых деталей. Вместо этого они должны зависеть от абстракций, таких как интерфейсы. Например, при сохранении данных это стоит делать через интерфейс, который не требует понимания конкретных деталей реализации базы данных. Будь то база данных SQL, NoSQL или кэш-память, интерфейс должен быть одинаковым.
- Абстракции не должны зависеть от деталей. Наоборот, детали должны зависеть от абстракций<sup>1</sup>. Например, при разработке интерфейса между уровнем бизнес-логики и уровнем данных мы должны убедиться, что интерфейс не меняется в зависимости от деталей реализации базы данных. Вместо этого мы вносим изменения в уровень данных, чтобы заставить его работать с интерфейсом. Другими словами, уровень данных зависит от интерфейса, а не наоборот.

<sup>1</sup> Мартин Р. К. Гибкая разработка программ на Java и C++. Принципы, паттерны и методики.

## ОПРЕДЕЛЕНИЕ

Согласно *принципу инверсии зависимостей*, нужно проектировать программное обеспечение на основе интерфейсов и убедиться в том, что мы не создаем зависимостей между низкоуровневыми деталями наших компонентов.



**Рис. 7.2.** Мы применяем принцип инверсии зависимостей, чтобы определить, какие компоненты управляют изменениями. В гексагональной архитектуре это означает, что наши адаптеры будут зависеть от интерфейса, предоставляемого базовым уровнем бизнес-логики

Инверсия зависимостей часто идет рука об руку вместе с инверсией управления и внедрением зависимостей. Это взаимосвязанные, но разные понятия. Как мы увидим в разделе 7.5, принцип инверсии управления заключается в том, чтобы предоставлять зависимости кода через контекст выполнения (также называемый инверсией контейнера управления). Чтобы предоставить такие зависимости, мы можем использовать паттерн внедрения зависимостей, который будет рассмотрен в разделе 7.5.

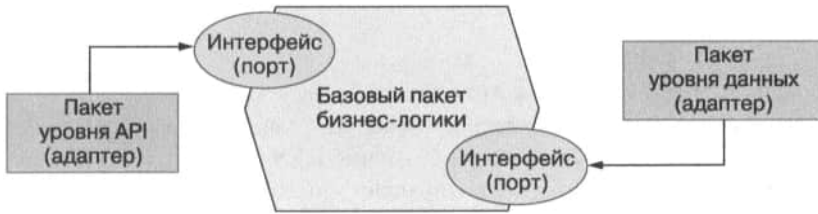
Что это означает на практике? То, что мы должны сделать адаптеры зависимыми от интерфейса, предоставляемого основной бизнес-логикой. То есть нормально, если наш уровень API знает об интерфейсе основной бизнес-логики, но ненормально, если наша бизнес-логика знает конкретные детали нашего уровня API или низкоуровневые детали протокола HTTP. То же самое касается базы данных: наш уровень данных должен знать, как работает приложение и как приспособить его к выбранной технологии хранения данных, но основной уровень бизнес-логики не должен знать ничего конкретного о базе данных. Наш уровень бизнес-логики будет предоставлять интерфейс, а все остальные компоненты мы реализуем на его основе.

Что именно мы инвертируем с помощью принципа инверсии зависимостей? Этот принцип переворачивает наше представление о программном обеспечении. Вместо более традиционного подхода, при котором сначала создаются низкоуровневые детали ПО, а затем поверх них строятся интерфейсы, принцип инверсии зависимостей заставляет нас сначала думать об интерфейсах, а затем создавать на их основе низкоуровневые детали<sup>1</sup>.

<sup>1</sup> Принцип инверсии зависимостей отлично описан в книге: Фримен Э., Робсон Э., Сьерра К., Бейтс Б. Head First. Паттерны проектирования. — СПб.: Питер, 2021. — С. 171.



Как показано на рис. 7.3, для сервиса заказов у нас есть основной пакет, реализующий возможности сервиса. Он позволяет обрабатывать и оплачивать заказ, планировать его изготовление или отслеживать статус. Этот основной пакет предоставляет интерфейсы для других компонентов приложения. Еще один пакет реализует уровень веб-API, и наши модули API будут использовать функции и классы из интерфейса уровня бизнес-логики для обслуживания запросов наших пользователей. Следующий пакет реализует уровень данных, который знает, как взаимодействовать с базой данных и возвращать бизнес-объекты для основного уровня бизнес-логики.



**Рис. 7.3.** Сервис заказов состоит из трех пакетов: основной бизнес-логики, которая реализует возможности сервиса, уровня API, который позволяет клиентам взаимодействовать с сервисом по протоколу HTTP, и уровня данных, который отвечает за взаимодействие сервиса с базой данных. Основная бизнес-логика предоставляет интерфейсы, на основе которых реализуются уровень API и уровень данных

Теперь, когда мы знаем, как будем структурировать приложение, пора приступить к его реализации. В следующем разделе мы настроим среду для начала работы с сервисом.

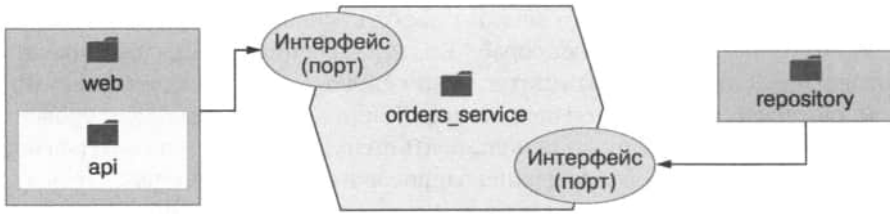
## 7.2. НАСТРОЙКА СРЕДЫ И СТРУКТУРЫ ПРОЕКТА

В этом разделе мы создадим среду для работы над сервисом заказов и наметим общую структуру проекта. Как и в предыдущих главах, для управления зависимостями будем использовать `Pipenv`. Выполните следующие команды, чтобы настроить среду `Pipenv` и активировать ее:

```
$ pipenv --three
$ pipenv shell
```

Мы установим зависимости, как нам нужно, в следующих разделах. Или, если хотите, скопируйте файлы `Pipfile` и `Pipfile.lock` из репозитория GitHub в папку `ch07` и запустите `pipenv install`.

Наша реализация сервиса будет храниться в папке `orders`, поэтому создайте ее. Чтобы усилить разделение задач между основным уровнем бизнес-логики и адаптерами API и базы данных, реализуем каждый из них в разных каталогах (рис. 7.4). Уровень бизнес-логики будет находиться в разделе `orders/orders_service`.

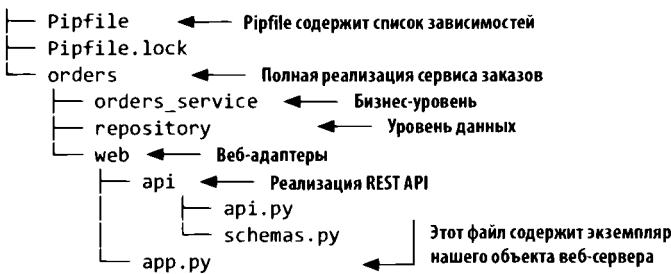


**Рис. 7.4.** Чтобы усилить разделение задач, реализуем каждый уровень приложения в разных каталогах: `orders_service` для основного уровня бизнес-логики, `repository` для уровня данных и `web/api` для уровня API

Поскольку уровень API является веб-компонентом, он будет находиться в разделе `orders/web`, который содержит веб-адаптеры для сервиса заказов. В данном случае мы включаем только один тип веб-адаптера, а именно REST API, но ничто не мешает вам добавить веб-адаптер, который возвращает динамически отображаемый контент с сервера, как это делается в более традиционных приложениях Django.

Уровень данных будет храниться в разделе `orders/repository` (листинг 7.1). «Репозиторий» может показаться не очень подходящим названием для уровня данных, но я выбрал это название, потому что для взаимодействия с данными мы будем реализовывать паттерн Репозиторий. Это станет понятнее в разделе 7.4. В главах 2 и 6 мы рассмотрели реализацию уровня API, поэтому скопируйте файлы из раздела `ch07/order/web` на GitHub в свой локальный каталог. Обратите внимание, что реализация API была адаптирована для этой главы.

### Листинг 7.1. Высокоуровневая структура сервиса заказов



Поскольку структура папок теперь другая, путь к нашему объекту приложения FastAPI также изменил расположение, поэтому команда для запуска сервера API теперь выглядит следующим образом:

```
$ uvicorn orders.web.app:app --reload
```

В связи с новой структурой папок изменились также некоторые пути импорта и расположение файлов. Полный список изменений см. в папке `ch07` в репозитории GitHub для книги.

Теперь, когда наш проект настроен и готов к работе, пора приступить к его реализации. Переходите к следующему разделу, чтобы узнать, как добавить модели баз данных в сервис!

## 7.3. РЕАЛИЗАЦИЯ МОДЕЛЕЙ БАЗ ДАННЫХ

В предыдущем разделе мы узнали, как разделить проект на три различных уровня: основной уровень бизнес-логики, уровень API и уровень данных. Такая структура усиливает разделение задач между каждым уровнем, как это рекомендуется в модели гексагональной архитектуры, которую мы изучали в разделе 7.1. Теперь, когда мы знаем, как будем структурировать код, пришло время сосредоточиться на реализации.

В этом разделе мы определим модели базы данных для сервиса заказов, то есть продумаем таблицы базы данных и их поля. Начнем нашу реализацию с базы данных, поскольку это облегчит дальнейшее обсуждение. В реальных условиях вы могли бы начать с уровня бизнес-логики, имитируя уровень данных и переходя туда и обратно между каждым уровнем, пока не закончите с реализацией. Имейте в виду, что линейный подход, который мы используем в этой главе, не отражает реальный процесс разработки, а предназначен для иллюстрации концепций, которые требуется объяснить.

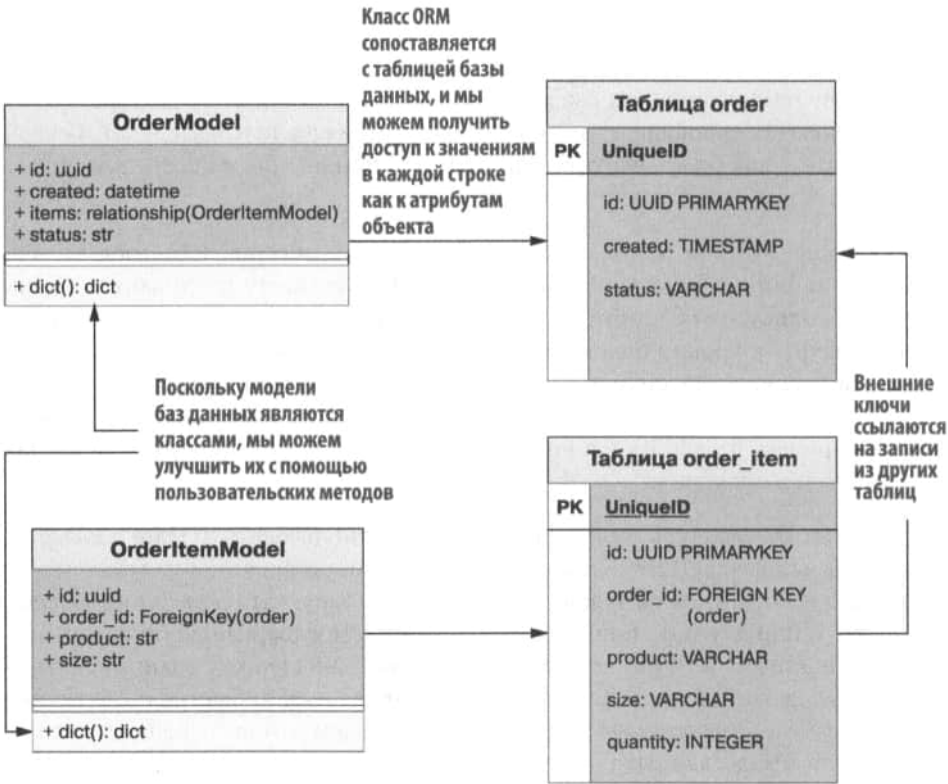
Чтобы упростить задачу, в этой главе мы будем использовать SQLite в качестве движка базы данных. SQLite — это основанная на файлах реляционная система баз данных. Для работы с ней не нужно устанавливать и запускать сервер, как в случае с PostgreSQL или MySQL, и не требуется никакой предварительной настройки. Основная библиотека Python по умолчанию имеет поддержку взаимодействия с SQLite, что делает ее подходящим выбором, когда нужно быстро создать прототипы и поэкспериментировать, прежде чем мы будем готовы перейти к использованию системы баз данных в продакшене.

Мы не станем вручную управлять нашим соединением с базой данных и запросами. То есть нам не придется писать собственные SQL-запросы для взаимодействия с базой данных. Вместо этого мы воспользуемся SQLAlchemy — на сегодняшний день это самая популярная технология ORM (object relational mapper) в экосистеме Python. ORM — это фреймворк, реализующий паттерн преобразования данных, который позволяет нам передавать данные между объектами и базой данных.

### ОПРЕДЕЛЕНИЕ

Средство преобразования данных (data mapper) — это объектная оболочка для таблиц и строк базы данных. Она инкапсулирует операции с базой данных в виде методов класса и позволяет нам получать доступ к полям данных через атрибуты класса.

Как показано на рис. 7.5, использование ORM облегчает управление данными, поскольку мы получаем интерфейс класса, соответствующего таблице в базе данных. Это позволяет нам использовать преимущества объектно-ориентированного программирования, в том числе добавлять кастомные методы и свойства к нашим моделям баз данных, которые расширяют их функциональность и инкапсулируют поведение.



**Рис. 7.5.** Используя ORM, мы можем реализовать наши модели данных в виде классов, которые сопоставляются с таблицами базы данных. Поскольку модели являются классами, мы можем расширить их с помощью кастомных методов, чтобы добавить новую функциональность

Со временем модели наших баз данных будут меняться, и нам нужно иметь возможность отслеживать эти изменения. Изменение схемы базы данных называется *миграцией*. По мере развития базы данных будет накапливаться все больше и больше миграций. Нам необходимо отслеживать их, поскольку они позволяют дублировать схему базы данных в различных средах и внедрять изменения базы данных в рабочую среду. Для управления этой сложной задачей мы воспользуемся Alembic. Это библиотека миграции схем, которая легко интегрируется с SQLAlchemy.

Начнем с установки обеих библиотек, выполнив следующую команду:

```
$ pipenv install sqlalchemy alembic
```

Прежде чем мы начнем работать над моделями баз данных, настроим Alembic. (Для получения дополнительной информации ознакомьтесь с моим видеоуроком по настройке Alembic с помощью SQLAlchemy: [https://youtu.be/nt5sSr1A\\_qw](https://youtu.be/nt5sSr1A_qw).) Выполните следующую команду для создания папки `migrations`, которая будет содержать историю всех миграций в базе данных:

```
$ alembic init migrations
```

В результате этой команды создается папка `migrations`, куда входит файл конфигурации `env.py` и каталог `versions/`, который будет содержать файлы миграции. Команда установки также создает файл конфигурации `alembic.ini`. Чтобы заставить Alembic работать с базой данных SQLite, откройте этот файл, найдите строку с объявлением для переменной `sqlalchemy.url` и замените ее следующим содержимым:

```
sqlalchemy.url = sqlite:///orders.db
```

Это позволит вам повторить настройку базы данных в новых средах.

## ЗАФИКСИРУЙТЕ ФАЙЛЫ, СОЗДАННЫЕ ALEMBIC

Папка `migrations` содержит всю информацию, необходимую для управления изменениями схемы базы данных, поэтому вам следует зафиксировать (`commit`) эту папку, а также `alembic.ini`. Это позволит вам воспроизвести настройки базы данных в новых средах.

Кроме того, откройте файл `migrations/env.py` и найдите такие строки<sup>1</sup>:

```
# from myapp import mymodel
# target_metadata = mymodel.Base.metadata
target_metadata = None
```

Замените их следующим содержимым:

```
from orders.repository.models import Base
target_metadata = Base.metadata
```

Установив параметру `target_metadata` значение `Base` нашей модели `metadata`, мы даем возможность Alembic загружать наши модели SQLAlchemy и генерировать из них таблицы базы данных. Далее мы реализуем наши модели базы данных. Но сначала ненадолго остановимся и подумаем, сколько моделей нам понадобится и какими свойствами должна обладать каждая модель. Основным объектом сервиса заказов является заказ. Пользователи размещают, оплачивают, обновляют или отменяют заказы. Заказы имеют жизненный цикл, и мы будем отслеживать

---

<sup>1</sup> Форма и формат этого файла могут со временем измениться, но на момент написания книги это были строки 18–20.

его с помощью свойства `status`. Нам понадобится следующий список свойств для определения нашей модели заказа.

- *ID* — уникальный идентификатор заказа. Определим для него формат универсального уникального идентификатора (Universally Unique Identifier, UUID). UUID хорошо работают в распределенных системах и помогают скрыть от пользователей информацию о количестве заказов, имеющихся в базе данных.
- *Creation date* — дата размещения заказа.
- *Items* — список товаров, включенных в заказ, и количество каждого товара. Поскольку заказ может иметь любое количество товаров, мы будем использовать другую модель для товаров и создадим отношения «один ко многим» между заказом и товарами.
- *Status* — статус заказа во всей системе. Заказ может иметь следующие статусы:
  - *Created* — размещен;
  - *Paid* — успешно оплачен;
  - *Progress* — готовится на кухне;
  - *Cancelled* — был отменен;
  - *Dispatched* — доставляется пользователю;
  - *Delivered* — был доставлен пользователю.
- *Schedule ID* — идентификатор заказа в сервисе кухни. Создается сервисом кухни после планирования заказа на изготовление, и мы будем использовать его для отслеживания его выполнения на кухне.
- *Delivery ID* — идентификатор заказа в сервисе доставки. Создается сервисом доставки после планирования отправки, и мы будем использовать его для отслеживания хода доставки.

Когда пользователи размещают заказ, они добавляют к нему любое количество товаров. Каждый элемент содержит информацию о товаре, выбранном пользователем, его объеме и количестве. Между заказами и товарами существует связь «один ко многим», поэтому мы реализуем модель для товаров и свяжем их с помощью внешнего ключа. Модель товара будет иметь следующий список атрибутов.

- *ID* — уникальный идентификатор элемента в формате UUID.
- *Order ID* — внешний ключ, соответствующий идентификатору заказа, к которому относится товар. Позволяет связывать элементы и соответствующие заказы.
- *Product* — товар, выбранный пользователем.
- *Size* — объем товара.
- *Quantity* — количество товаров одного вида, которое пользователь хочет приобрести.

Наши модели SQLAlchemy будут храниться в папке `orders/repository`, которую мы создали для инкапсуляции уровня данных, в файле `orders/repository/models.py`. Мы будем использовать эти классы для взаимодействия с базой данных и полагаться на SQLAlchemy в преобразовании этих моделей в соответствующие таблицы базы данных. В листинге 7.2 показано, как определяются модели базы данных для сервиса заказов. Сначала мы создаем декларативную базовую модель с помощью функции SQLAlchemy `declarative_base()`. Декларативная базовая модель — это класс, который может сопоставлять классы ORM с таблицами и столбцами базы данных, и все наши модели баз данных должны наследоваться от него. Мы сопоставляем атрибуты класса с определенными столбцами базы данных, устанавливая их как экземпляры класса столбцов SQLAlchemy.

### Листинг 7.2. Модели SQLAlchemy для сервиса заказов

# file: orders/repository/models.py

```
import uuid
from datetime import datetime
```

```
from sqlalchemy import Column, Integer, String, ForeignKey, DateTime
from sqlalchemy.ext.declarative import declarative_base
from sqlalchemy.orm import relationship
```

```
Base = declarative_base()
```

Создаем нашу декларативную базовую модель

```
def generate_uuid():
    return str(uuid.uuid4())
```

Кастомная функция для создания случайных UUID для наших моделей

```
class OrderModel(Base):
    __tablename__ = 'order'
```

Все модели должны наследоваться от Base

Имя таблицы, которая сопоставляется с этой моделью

Каждое свойство класса сопоставляется со столбцом базы данных с помощью класса Column

```
    id = Column(String, primary_key=True, default=generate_uuid)
    items = relationship('OrderItemModel', backref='order')
    status = Column(String, nullable=False, default='created')
    created = Column(DateTime, default=datetime.utcnow)
    schedule_id = Column(String)
    delivery_id = Column(String)
```

Используем relationship() для создания отношения «один ко многим» с моделью OrderItemModel

```
    def dict(self):
        return {
```

Кастомный метод для отображения наших объектов в виде словарей Python

```
            'id': self.id,
            'items': [item.dict() for item in self.items],
            'status': self.status,
            'created': self.created,
            'schedule_id': self.schedule_id,
            'delivery_id': self.delivery_id,
```

Вызываем dict() для каждого элемента, чтобы получить его словарное представление

```
        }
```

```
class OrderItemModel(Base):
    __tablename__ = 'order_item'
```

```

id = Column(String, primary_key=True, default=generate_uuid)
order_id = Column(Integer, ForeignKey('order.id'))
product = Column(String, nullable=False)
size = Column(String, nullable=False)
quantity = Column(Integer, nullable=False)

def dict(self):
    return {
        'id': self.id,
        'product': self.product,
        'size': self.size,
        'quantity': self.quantity
    }

```

Чтобы сопоставить атрибут с другой моделью, мы используем функцию `SQLAlchemy relationship()`. В листинге 7.2 она применяется для создания отношения «один ко многим» между атрибутом `items` модели `OrderModel` и моделью `OrderItemModel`. Это означает, что мы можем получить доступ к списку элементов в заказе через атрибут `items` модели `OrderModel`. Каждый элемент также сопоставляется с заказом, к которому он относится, через свойство `order_id`, которое определено как внешний ключ. Более того, аргумент `backref` в `relationship()` позволяет нам получить доступ к полному объекту заказа из элемента непосредственно через свойство `order`.

Мы хотим, чтобы наши идентификаторы хранились в формате UUID, поэтому создаем функцию, которую SQLAlchemy может использовать для генерации значения. Если позже мы перейдем на движок базы данных со встроенной поддержкой генерации значений UUID, то предоставим базе данных генерировать идентификаторы. Каждая модель базы данных дополнена методом `dict()`, который позволяет нам вывести свойства записи в формате словаря.

Поскольку мы будем использовать этот метод для перевода моделей базы данных в бизнес-объекты, то метод `dict()` возвращает только свойства, относящиеся к уровню бизнес-логики.

Чтобы применить модели к базе данных, выполните следующую команду из каталога `ch07`:

```
$ PYTHONPATH=`pwd` alembic revision --autogenerate -m "Initial migration"
```

Она создаст файл миграции в каталоге `migrations/versions`. С помощью команды `pwd` мы установили переменную среды `PYTHONPATH` в текущем каталоге, чтобы Python искал наши модели относительно этого каталога. Вам следует зафиксировать файлы миграции и сохранить их в системе контроля версий (например, в репозитории Git), поскольку они позволят повторно создавать базу данных для различных сред. Загляните в эти файлы, чтобы понять, какие операции с базой данных будет выполнять SQLAlchemy для применения миграций.



Чтобы применить миграции и создать схемы для этих моделей в базе данных, выполните следующую команду:

```
$ PYTHONPATH=`pwd` alembic upgrade heads
```

Она позволит создать пужные схемы в нашей базе данных.

Теперь, когда модели баз данных реализованы и база данных содержит положенные схемы, пора переходить к следующему шагу.

## 7.4. РЕАЛИЗАЦИЯ ПАТТЕРНА РЕПОЗИТОРИЙ ДЛЯ ДОСТУПА К ДАННЫМ

В предыдущем разделе вы научились разрабатывать модели базы данных для сервиса заказов и управлять изменениями схемы базы данных с помощью миграций. Подготовив модели базы данных, мы можем взаимодействовать с базой данных для создания заказов и управления ими. Теперь нам нужно решить, как сделать данные доступными для уровня бизнес-логики. В этом разделе мы сначала обсудим различные стратегии связи уровня бизнес-логики с уровнем данных, а затем рассмотрим паттерн Репозиторий.

### 7.4.1. Паттерн Репозиторий: что это и чем он полезен

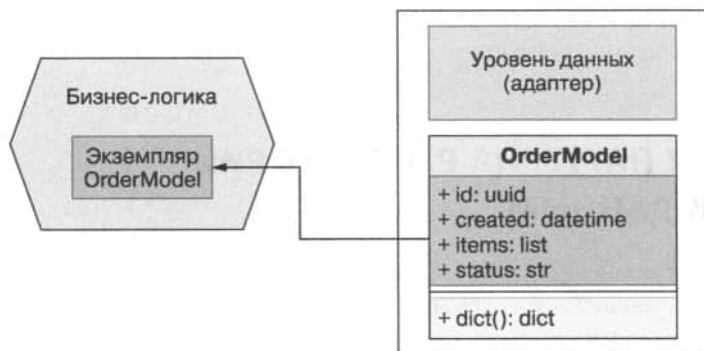
Здесь мы обсудим разные стратегии взаимодействия с базой данных с уровня бизнес-логики и рассмотрим паттерн Репозиторий как стратегию, которая помогает отделить уровень бизнес-логики от деталей реализации базы данных. Как показано на рис. 7.6, общая стратегия для обеспечения взаимодействия между уровнем бизнес-логики и базой данных заключается в использовании моделей базы данных непосредственно на уровне бизнес-логики. Наши модели баз данных уже содержат данные о заказах, поэтому мы можем дополнить их методами, реализующими бизнес-функции. Это называется паттерном *активной записи*, который является популярным способом доступа к данным в базе данных. При использовании этого паттерна в объекте совмещаются как данные, так и логика предметной области<sup>1</sup>. Он подходит, когда у нас есть однозначное соответствие между возможностями сервиса и операциями с базой данных или когда нам не пужна совместная работа нескольких доменов.

Этот паттерн хорош для простых случаев, однако он привязывает реализацию уровня бизнес-логики к базе данных и выбранному фреймворку ORM. Что произойдет, если в дальнейшем мы захотим изменить фреймворк или перейти на другую технологию хранения данных, не связанную с SQL? Тогда придется внести

---

<sup>1</sup> Фаулер М., Фоммел М., Райс Д. Шаблоны корпоративных приложений.

изменения в наш уровень бизнес-логики. Это нарушает принципы, прописанные в разделе 7.1. Помните, что база данных — это адаптер, который сервис заказов использует для сохранения данных, и детали реализации базы не должны просачиваться в бизнес-логику. Вместо этого доступ к данным будет инкапсулирован нашим уровнем взаимодействия с базой данных.



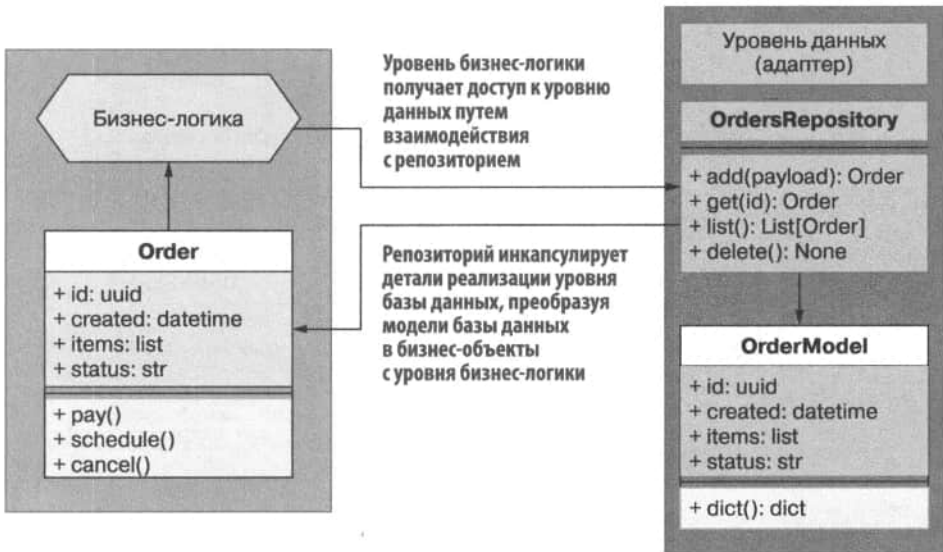
**Рис. 7.6.** Распространенный подход к обеспечению взаимодействия между уровнем данных и уровнем бизнес-логики — использование моделей баз данных непосредственно на уровне бизнес-логики

Чтобы отделить уровень бизнес-логики от уровня данных, воспользуемся паттерном Репозиторий. Он предоставляет интерфейс для коллекции данных в памяти. Это означает, что мы можем получать, добавлять или удалять заказы из списка, а Репозиторий позаботится о переводе этих операций в команды, специфичные для базы данных. Использование паттерна Репозиторий означает, что уровень данных предоставляет уровню бизнес-логики согласованный интерфейс для взаимодействия с базой данных, независимо от того, какую технологию хранения данных мы применяем. Используем ли мы базу данных SQL, например PostgreSQL, базу данных NoSQL, например MongoDB, или другое, интерфейс паттерна Репозиторий останется неизменным и будет инкапсулировать любые специфические операции, необходимые для взаимодействия с базой данных. На рис. 7.7 показано, как паттерн помогает инвертировать зависимость между уровнем данных и уровнем бизнес-логики.

## ОПРЕДЕЛЕНИЕ

Паттерн Репозиторий — это паттерн разработки программного обеспечения, который предоставляет интерфейс коллекции в памяти для нашего хранилища данных. Помогает отделить компоненты от низкоуровневых деталей реализации базы данных.

Репозиторий берет на себя управление взаимодействием с базой данных и обеспечивает согласованный интерфейс для компонентов независимо от используемой технологии базы данных. Это позволяет нам менять систему баз данных, не изменяя основную бизнес-логику.



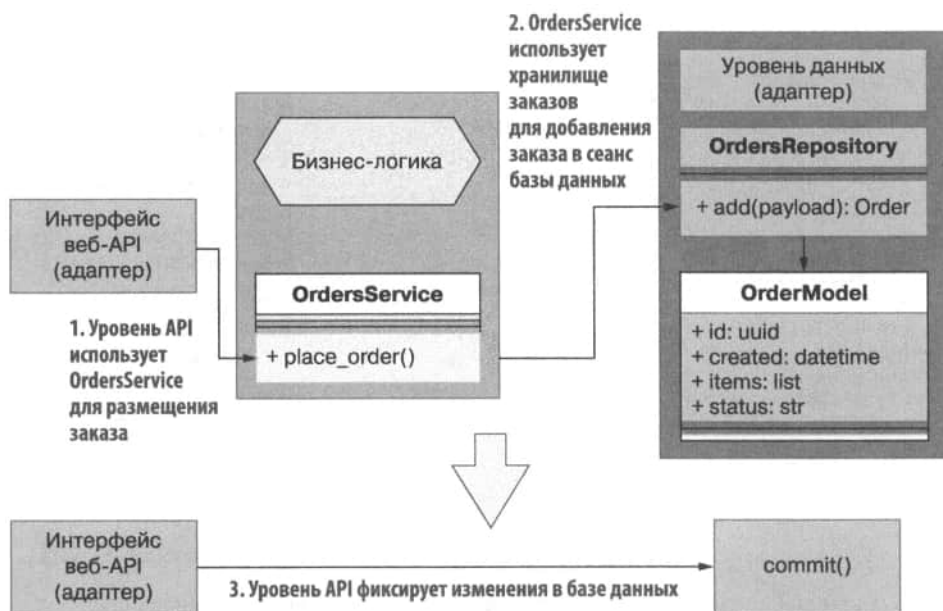
**Рис. 7.7.** Паттерн Репозиторий инкапсулирует детали реализации уровня данных, предоставляя интерфейс коллекции данных в памяти уровню бизнес-логики, и преобразует модели базы данных в бизнес-объекты

Теперь, когда мы знаем, как можно применять паттерн Репозиторий, чтобы позволить уровню бизнес-логики взаимодействовать с базой данных, отделяя при этом его реализацию от низкоуровневых деталей базы данных, воспользуемся его возможностями.

## 7.4.2. Реализация паттерна Репозиторий

Для реализации паттерна мы можем использовать различные подходы, но обязательно придерживаться следующего правила: ни одна из операций, выполняемых репозиторием, не должна быть зафиксирована (committed) репозиторием. Что это означает? Что когда мы добавляем объект заказа в репозиторий, репозиторий добавит заказ в сеанс базы данных, но не будет фиксировать изменения. Вместо этого фиксация изменений будет возложена на потребителя сервиса `OrdersService` (то есть на уровень API). Этот процесс показан на рис. 7.8.

Почему мы не можем фиксировать изменения в базе данных в репозитории? Во-первых, потому, что репозиторий работает точно так же, как представление списка данных в памяти и в нем не поддерживается концепция сеансов базы данных и транзакций как таковая. Во-вторых, потому, что репозиторий не является подходящим местом для выполнения транзакций базы данных. Вместо этого окружение, в котором вызывается репозиторий, обеспечивает правильный контекст для выполнения транзакций базы данных.



**Рис. 7.8.** Когда используется паттерн Репозиторий, на уровне API для размещения заказа применяется функция `place_order()`. При размещении заказа сервис `OrdersService` взаимодействует с репозиторием заказов, добавляя заказ в базу данных. Наконец, уровень API должен зафиксировать изменения, чтобы сохранить их в базе данных

Во многих случаях приложения выполняют несколько операций, в которых участвуют один или несколько репозиторий, а также обращаются к другим сервисам. Например, на рис. 7.9 показано, сколько операций связано с обработкой платежа.

1. Уровень API получает запрос от пользователя и использует метод `pay_order()` сервиса `OrdersService` для его обработки.
2. `OrdersService` обращается к сервису платежей для обработки платежа.
3. Если оплата прошла успешно, `OrdersService` согласовывает заказ с сервисом кухни.
4. `OrdersService` обновляет состояние заказа в базе данных, используя репозиторий заказов.
5. Если все предыдущие операции успешно выполнены, уровень API фиксирует транзакцию в базе данных; в противном случае он откатывает изменения.

Эти шаги могут выполняться синхронно, один за другим, или асинхронно, без определенного порядка, но в любом случае все шаги могут быть одновременно или успешными, или неудачными. Как компонент контекста выполнения, уровень API несет ответственность за то, чтобы все изменения были зафиксированы или

отменены в установленном порядке. Как именно уровень API управляет сеансом базы данных и фиксирует транзакции, мы обсудим в разделе 7.6.

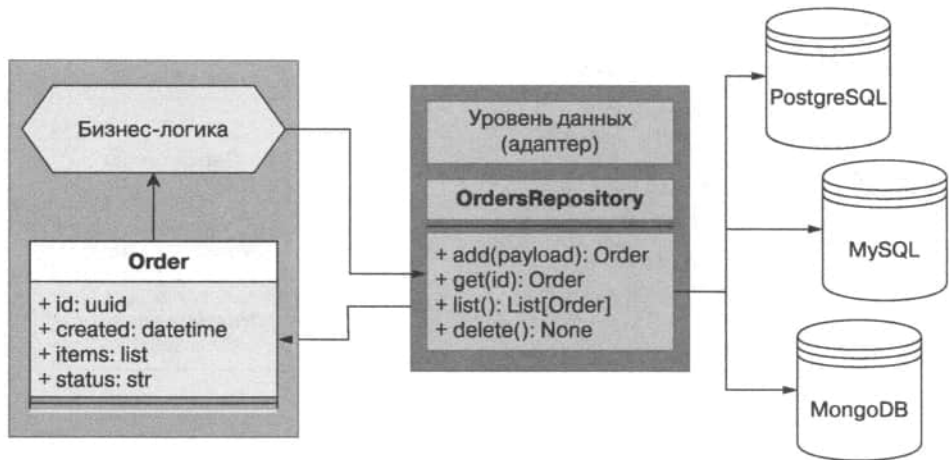


**Рис. 7.9.** В некоторых ситуациях `OrdersService` для выполнения операции должен взаимодействовать с несколькими репозиториями или сервисами. В этом примере он взаимодействует с сервисом платежей для обработки платежа, затем с сервисом кухни для планирования заказа на изготовление и, наконец, обновляет статус заказа через репозиторий заказов. Эти операции должны выполняться успешно или нет все вместе, и уровень API несет ответственность за соответствующую фиксацию или откат

Реализация паттерна Репозиторий как минимум состоит из класса, который предоставляет методы `get()` и `add()` для получения объектов из репозитория и добавления в него соответственно. Для наших целей мы также реализуем следующие методы: `update()`, `delete()` и `list()`. Это упростит CRUD-интерфейс репозитория.

В этом контексте нужно рассмотреть следующий вопрос: когда мы получаем данные через репозиторий, какой тип объекта он должен возвращать? Во многих реализациях вы увидите репозитории, возвращающие экземпляры моделей баз данных (то есть классы, определенные в `orders/repository/models.py`). Но мы не станем так делать. Вместо этого будем возвращать объекты, представляющие заказы из домена уровня бизнес-логики. Почему возвращать экземпляры моделей базы данных через репозиторий — плохая идея? Потому что это противоречит цели репозитория, которая состоит в отделении уровня бизнес-логики от уровня данных. Ведь может появиться необходимость изменить технологию постоянного хранения или ORM-фреймворк. Если это произойдет, классы базы данных, которые мы реализовали в разделе 7.2, перестанут существовать, и нет никакой гарантии,

что новый фреймворк позволит нам вернуть объекты с теми же интерфейсами. По этой причине не стоит связывать с ними уровень бизнес-логики. На рис. 7.10 показана связь между уровнем бизнес-логики и репозиторием заказов.



**Рис. 7.10.** Паттерн Репозиторий инкапсулирует детали реализации используемой технологии постоянного хранения. Наш уровень бизнес-логики всегда имеет дело только с репозиторием, поэтому мы можем спокойно поменять наше решение для постоянного хранения, не затрагивая реализацию основного приложения

Реализация нашего репозитория заказов представлена в листинге 7.3. Этот код хранится в файле `orders/repository/orders_repository.py`. Он принимает один обязательный аргумент, который представляет собой сеанс базы данных. Объекты добавляются в сеанс и удаляются из него. Методы `add()` и `update()` принимают полезную нагрузку, соответствующую заказам в виде словаря Python. Наша полезная нагрузка довольно проста, поэтому словаря здесь достаточно, но, если планируются более сложные данные, стоит воспользоваться объектами.

### Листинг 7.3. Репозиторий заказов

# file: orders/repository/orders\_repository.py

```
from orders.orders_service.orders import Order
from orders.repository.models import OrderModel, OrderItemModel
```

```
class OrdersRepository:
    def __init__(self, session):
        self.session = session

    def add(self, items):
        record = OrderModel(
            items=[OrderItemModel(**item) for item in items]
        )
```

Для метода инициализатора репозитория требуется объект session

При создании записи для заказа мы также создаем запись для каждого товара в заказе

```

self.session.add(record)  ← Добавляем запись в объект session
return Order(**record.dict(), order_=record)  ← Возвращаем экземпляр
                                              класса Order

def _get(self, id_):  ← Универсальный метод для извлечения
                    ← записи по идентификатору
    return (
        self.session.query(OrderModel)
        .filter(OrderModel.id == str(id_))
        .filter_by(**filters)
        .first()
    )  ← Извлекаем запись, используя
        ← метод first() SQLAlchemy

def get(self, id_):
    order = self._get(id_)  ← Извлекаем запись с помощью _get()
    if order is not None:  ← Если заказ существует,
        return Order(**order.dict())  ← возвращаем объект Order

def list(self, limit=None, **filters):  ← Функция list() принимает параметр limit
    query = self.session.query(OrderModel)  ← Динамически создаем наш запрос
    if 'cancelled' in filters:  ← Фильтруем по тому, отменен ли заказ,
        cancelled = filters.pop('cancelled')  ← используя метод filter() SQLAlchemy
    if cancelled:
        query = query.filter(OrderModel.status == 'cancelled')
    else:
        query = query.filter(OrderModel.status != 'cancelled')
    records = query.filter_by(**filters).limit(limit).all()
    return [Order(**record.dict()) for record in records]  ← Возвращаем
                                                            список объектов
                                                            Order

def update(self, id_, **payload):
    record = self._get(id_)
    if 'items' in payload:
        for item in record.items:
            self.session.delete(item)
        record.items = [
            OrderItemModel(**item) for item in payload.pop('items')
        ]
    for key, value in payload.items():
        setattr(record, key, value)  ← Динамически обновляем объект базы
    return Order(**record.dict())  ← данных с помощью функции setattr()

def delete(self, id_):
    self.session.delete(self._get(id_))  ← Чтобы удалить запись, вызываем
                                         ← метод delete() SQLAlchemy

```

Все методы репозитория, за исключением `delete()`, возвращают объекты `Order` с уровня бизнес-логики (подробности реализации `Order` см. в разделе 7.5). Для создания экземпляров `Order` мы передаем словарные представления моделей SQLAlchemy с помощью кастомного метода `dict()` из листинга 7.2. В метод `add()` мы также включаем указатель на фактическую модель SQLAlchemy через параметр `order_`. Как вы увидите в разделе 7.5, этот указатель поможет нам получить доступ к идентификатору заказа после завершения транзакции базы данных.

Методы `get()`, `update()` и `delete()` сервиса `OrdersRepository` используют одну и ту же логику для извлечения записи перед ее возвратом, обновлением или

удалением, поэтому мы определяем общий метод `_get()`, который знает, как получить запись, с заданным идентификатором и необязательными параметрами-фильтрами. Получаем запись с помощью метода `first()` объекта запроса `SQLAlchemy`. Функция `first()` возвращает экземпляр записи, если она существует, или `None` в противном случае. В качестве альтернативы можно также использовать метод `one()`, который выдает ошибку, если запись не найдена. Метод `_get()` возвращает запись из базы данных, поэтому не предназначен для использования на уровне сервиса, и мы сигнализируем об этом, добавляя к имени метода знак подчеркивания.

Метод `list()` принимает параметр `limit` и необязательные фильтры. Мы строим наш запрос динамически, используя объект запроса `SQLAlchemy`. Мы также используем метод `filter_by()` `SQLAlchemy` для включения дополнительных фильтров в запрос и ограничиваем результаты запроса с помощью параметра `limit`. Наконец, преобразуем записи базы данных в объекты `Order` для использования уровнем бизнес-логики с помощью метода `dict()`, который мы реализовали в листинге 7.2.

Реализация репозитория тесно связана с методами объекта `Session` в `SQLAlchemy`, но она также инкапсулирует эти детали, и для уровня бизнес-логики репозиторий является интерфейсом, которому мы передаем идентификаторы и полезную нагрузку, а в ответ получаем объекты `Order`. В этом и заключается смысл репозитория: инкапсулировать и скрывать детали реализации уровня данных от уровня бизнес-логики. Это означает, что, если мы перейдем на другой фреймворк ORM или другую систему баз данных, изменения нужно будет внести только в репозиторий.

На этом реализация нашего уровня данных завершена. Мы внедрили решение для постоянного хранения с помощью `SQLAlchemy` и инкапсулировали его детали с помощью паттерна Репозиторий. Теперь пришло время поработать над уровнем бизнес-логики и посмотреть, как он будет взаимодействовать с репозиторием.

## 7.5. РЕАЛИЗАЦИЯ УРОВНЯ БИЗНЕС-ЛОГИКИ

Мы проделали большую работу по проектированию моделей базы данных для сервиса заказов. В этом разделе мы реализуем уровень бизнес-логики сервиса заказов. Это ядро шестиугольника, который был представлен в разделе 7.1 и проиллюстрирован на рис. 7.1.

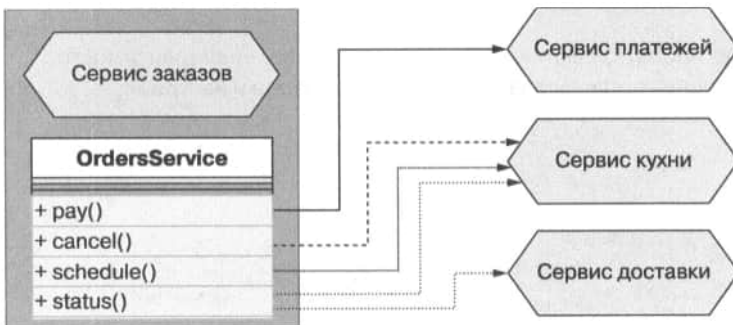
Уровень бизнес-логики реализует возможности сервиса. В чем заключаются бизнес-функции сервиса заказов? Из анализа, проведенного в главе 3 (см. подраздел 3.4.2), мы знаем, что этот сервис позволяет пользователям платформы размещать свои заказы и управлять ими.

Как показано на рис. 7.11, сервис заказов управляет жизненным циклом заказа путем интеграции с другими сервисами. Перечислю функции сервиса заказов



и возможности интеграции с другими сервисами (см. рис. 7.9 для более подробного пояснения).

- *Разместить заказ* — создает запись о заказе в системе. Заказ не будет передан на кухню, пока пользователь не оплатит его.
- *Обработка платежей* — обрабатывает оплату с помощью сервиса платежей. Если сервис платежей подтверждает успешность оплаты, сервис заказов согласовывает заказ на изготовление с сервисом кухни.
- *Изменение заказов* — пользователи могут в любое время обновить свой заказ, чтобы добавить или удалить из него товары. Для подтверждения изменений необходимо провести новый платеж и обработать его с помощью сервиса платежей.
- *Отмена заказов* — пользователи могут в любое время отменить свои заказы. В зависимости от статуса заказа сервис заказов связывается с кухней или сервисом доставки для отмены заказа.
- *Планирование заказа для изготовления на кухне* — после оплаты сервис заказов планирует заказ на изготовление с помощью сервиса кухни.
- *Отслеживание выполнения заказов* — пользователи могут следить за состоянием своих заказов с помощью сервиса заказов. В зависимости от статуса заказа сервис связывается с кухней или сервисом доставки, чтобы получить обновленную информацию о состоянии заказа.



**Рис. 7.11.** Для выполнения некоторых своих функций сервису заказов необходимо взаимодействовать с другими сервисами. Например, чтобы обрабатывать платежи, он должен взаимодействовать с сервисом платежей, а чтобы запланировать заказ на изготовление, он должен взаимодействовать с сервисом кухни

Как лучше всего смоделировать эти действия на нашем уровне бизнес-логики? Можно использовать различные подходы, но, чтобы облегчить другим компонентам взаимодействие с уровнем бизнес-логики, предоставим единый интерфейс через класс `OrdersService`. Определим его в файле `orders/orders_service/orders_service.py`. Для выполнения своих обязанностей `OrdersService` использует репозиторий заказов, позволяющий взаимодействовать с базой данных. Мы могли бы

позволить `OrdersService` импортировать и инициализировать репозиторий заказов, как показано в следующем коде:

```
from repository.orders_repository import OrdersRepository

class OrdersService:
    def __init__(self):
        self.repository = OrdersRepository()
```

Однако это возложило бы слишком большую ответственность на сервис заказов, поскольку ему нужно было бы знать, как настроить репозиторий заказов. Кроме того, реализация репозитория и сервиса заказов будет жестко связана, и мы не сможем использовать различные репозитории, если это понадобится. Как видно из рис. 7.12 и 7.13, лучше всего воспользоваться внедрением зависимостей в сочетании с принципом инверсии управления.



**Рис. 7.12.** При обычном проектировании программного обеспечения зависимости линейны и каждый компонент отвечает за создание экземпляров и настройку собственных зависимостей



**Рис. 7.13.** С помощью инверсии управления мы отделяем компоненты от зависимостей, предоставляя их во время выполнения с помощью таких методов, как внедрение зависимостей. Сплошные линии показывают отношения зависимостей, в то время как пунктирные обозначают, как происходят внедрения зависимостей

## ОПРЕДЕЛЕНИЕ

*Инверсия управления* — это принцип разработки программного обеспечения, который подразумевает отделение компонентов от зависимостей и предоставление их во время выполнения. Так мы можем контролировать, как предоставляются зависимости. Один из популярных паттернов для достижения этой цели — Внедрение зависимостей. Контекст, в котором создаются и поставляются зависимости, называется *контейнером внедрения зависимостей (инверсии управления)*. В сервисе заказов подходящим контейнером инверсии управления является объект запроса, поскольку большинство операций специфичны для контекста запроса.

Принцип инверсии управления гласит, что мы должны отделить зависимости в коде, позволив контексту выполнения предоставить их во время работы приложения. Это означает, что вместо того, чтобы позволять сервису заказов импортировать и создавать репозиторий заказов, мы должны предоставить репозиторий во время выполнения. Как мы это сделаем? Можно использовать различные паттерны, но одним из самых популярных благодаря простоте и эффективности является Внедрение зависимостей.

## ОПРЕДЕЛЕНИЕ

*Внедрение зависимостей* — паттерн разработки программного обеспечения, при котором мы предоставляем зависимости кода во время выполнения. Это помогает нам отделить компоненты от конкретных деталей реализации кода, от которого они зависят, поскольку им не нужно знать, как настраивать и создавать экземпляры своих зависимостей.

Чтобы сделать репозиторий заказов внедренным в сервис заказов, параметризируем его:

```
class OrdersService:
    def __init__(self, orders_repository):
        self.orders_repository = orders_repository
```

Теперь ответственность за создание экземпляра и правильную настройку репозитория заказов лежит на вызывающей стороне. Как показано на рис. 7.11, это очень удобно: в зависимости от контекста мы можем предоставить различные реализации репозитория или добавить различные конфигурации. Это облегчает использование сервиса заказов в различных контекстах<sup>1</sup>.

В листинге 7.4 приведен интерфейс, предоставляемый `OrdersService`. Инициализатор класса в качестве параметра принимает экземпляр репозитория заказов, чтобы сделать его доступным для внедрения. В соответствии с принципом инверсии

---

<sup>1</sup> Более подробно о принципе инверсии управления и паттерне Внедрение зависимостей вы можете прочитать в статье: *Fowler M. Inversion of Control Containers and the Dependency Injection pattern* («Контейнеры инверсии управления и паттерн внедрения зависимостей»), <https://martinfowler.com/articles/injection.html>.

управления, когда мы интегрируем `OrdersService` с уровнем API, ответственность за получение действительного экземпляра репозитория заказов и передачу его в `OrdersService` будет лежать на API. Такой подход удобен, поскольку позволяет нам по желанию менять репозитории, что очень упростит написание тестов, которым мы займемся в следующей главе.

#### Листинг 7.4. Интерфейс класса `OrdersService`

```
# file: orders/orders_service/orders_service.py

class OrdersService:
    def __init__(self, orders_repository):
        self.orders_repository = orders_repository

    def place_order(self, items):
        pass

    def get_order(self, order_id):
        pass

    def update_order(self, order_id, items):
        pass

    def list_orders(self, **filters):
        pass

    def pay_order(self, order_id):
        pass

    def cancel_order(self, order_id):
        pass
```

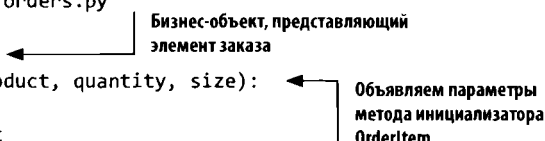
Некоторые действия, прописанные в классе `OrdersService`, такие как оплата или составление расписания, происходят на уровне отдельных заказов. Поскольку заказы содержат данные, будет полезно иметь класс, который соответствует заказу и содержит необходимые методы. В контексте нашего сервиса заказ является основным объектом домена заказов. В предметно-ориентированном проектировании (domain-driven design, DDD) мы называем эти объекты *объектами предметной области (домена)*. Это объекты, возвращаемые репозиторием заказов.

Реализуем наш класс `Order` в файле `orders/orders_service/orders.py` (листинг 7.5).

#### Листинг 7.5. Реализация бизнес-объектов класса `Order`

```
# file: orders/orders_service/orders.py

class OrderItem:
    def __init__(self, id, product, quantity, size):
        self.id = id
        self.product = product
        self.quantity = quantity
        self.size = size
```



Бизнес-объект, представляющий элемент заказа

Объявляем параметры метода инициализатора `OrderItem`

```

class Order:
    def __init__(self, id, created, items, status, schedule_id=None,
                  delivery_id=None, order_=None):
        self._id = id
        self._created = created
        self.items = [OrderItem(**item) for item in items]
        self._status = status
        self.schedule_id = schedule_id
        self.delivery_id = delivery_id

    @property
    def id(self):
        return self._id or self._order.id

    @property
    def created(self):
        return self._created or self._order.created

    @property
    def status(self):
        return self._status or self._order.status

```

Поскольку мы определяем идентификатор динамически, то храним его как закрытое свойство

Создаем объект `OrderItem` для каждого элемента заказа

Параметр `order_` представляет экземпляр модели базы данных

Определяем идентификатор динамически, используя декоратор `property()`

В дополнение к классу `Order` в листинге 7.5 также приведен класс `OrderItem`, который представляет каждый из элементов заказа. Мы будем использовать `Order` для представления заказов до и после их сохранения в базе данных. Некоторые свойства заказа, такие как время создания или его ID, задаются уровнем данных и могут быть известны только после фиксации изменений в базе данных. Как я объяснял в разделе 7.4, фиксация изменений выходит за рамки репозитория, а значит, когда мы добавляем заказ в репозиторий, возвращаемый объект не может иметь этих свойств. Идентификатор заказа и время его создания становятся доступными после завершения транзакции в базе данных заказа. По этой причине инициализатор `Order` связывает идентификатор, время создания и статус заказа как закрытые (`private`) свойства с символом подчеркивания в начале (как в `self._id`), и мы используем параметр `order_` в классе `Order` для хранения указателя на запись в базе данных `order`. Если мы получим детали заказа, уже сохраненного в базе данных, `_id`, `_created` и `_status` будут иметь соответствующие значения в инициализаторе; в противном случае они будут `None` и мы возьмем их значения из `order_`. Вот почему мы определяем свойства `id`, `created` и `status` заказа с помощью декоратора `property()` — это позволяет нам определять их значения в зависимости от состояния объекта. Это единственная связь, которая допустима между уровнем бизнес-логики и уровнем данных. И чтобы убедиться, что такую зависимость легко устранить, если это когда-нибудь понадобится, мы по умолчанию установим `order_` равным `None`.

Помимо хранения данных о заказе, класс `Order` должен обрабатывать такие задачи, как отмена, оплата и планирование заказа. Для их выполнения мы должны взаимодействовать с внешними зависимостями, например с сервисами кухни и платежей. Как объяснялось в разделе 7.1, целью гексагональной архитектуры является инкапсуляция доступа к внешним зависимостям через адаптеры. Однако,

чтобы упростить себе работу, мы реализуем вызовы внешних API в классе `Order`. Хорошим паттерном адаптера для инкапсуляции внешних вызовов API является Фасад<sup>1</sup>. Прежде чем приступить к реализации, мы должны знать, как выглядят эти вызовы API.

Чтобы интегрировать сервис заказов с сервисами кухни и платежей, нам нужно посмотреть, как они работают. Однако нам не нужно запускать сами сервисы. Папка для этой главы в репозитории GitHub содержит три файла OpenAPI: один для API сервиса заказов (`ch07/oas.yaml`), другой для API сервиса кухни (`ch07/kitchen.yaml`) и третий для API сервиса платежей (`ch07/payments.yaml`). Файлы `kitchen.yaml` и `payments.yaml` позволяют понять, как работают API сервисов кухни и платежей, и этой информации для создания интеграции нам будет достаточно. Обязательно возьмите файлы `kitchen.yaml` и `payments.yaml` с GitHub, чтобы иметь возможность работать с приведенными далее примерами.

Как оказалось, мы также можем использовать спецификации API сервисов кухни и платежей для моделирования их поведения с помощью имитационных или mock-серверов. Mock-сервер дублирует сервер, стоящий за API, проверяя наши запросы и возвращая корректные ответы. Чтобы имитировать сервер API для сервисов кухни и платежей, воспользуемся Prism CLI (<https://github.com/stoplightio/prism>), библиотекой, созданной и поддерживаемой компанией Stoplight. Prism — библиотека Node.js, но не волнуйтесь, это просто инструмент CLI и вам не нужно знать JavaScript, чтобы использовать ее. Для установки библиотеки выполните следующую команду:

```
$ yarn add @stoplight/prism-cli
```

Эта команда создаст папку `node_modules/` в папке вашего приложения, куда будет установлен Prism и все его зависимости. Чтобы не коммитить эту папку, добавьте ее в свой файл `.gitignore`. В каталоге приложения вы также увидите новый файл `package.json` и еще один под названием `yarn.lock`. Это те файлы, которые нужно закоммитить, поскольку они позволят вам повторно создать ту же папку `node_modules/` в любой другой среде.

## РАБОТА С ОШИБКАМИ ПРИ ЗАПУСКЕ PRISM

При запуске Prism вы можете столкнуться с ошибками. Распространенная ошибка — отсутствие совместимой версии Node.js. Я рекомендую вам установить npm для управления версиями Node и использовать последнюю стабильную версию Node для запуска Prism. Убедитесь также, что порт, который вы выбрали для запуска Prism, доступен.

Чтобы увидеть работу Prism с API сервиса кухни, выполните следующую команду:

```
$ ./node_modules/.bin/prism mock kitchen.yaml --port 3000
```

Она запустит сервер на порте 3000, на котором будет выполняться mock-сервис для API сервиса кухни. Чтобы получить представление о том, что с этим можно

<sup>1</sup> Гамма Э., Хелм Р., Джонсон Р., Влиссидес Дж. Паттерны объектно-ориентированного проектирования. — СПб.: Питер, 2020. — С. 221–231.

сделать, выполните следующую команду, которая позволит перейти к эндпоинту GET /kitchen/schedules, возвращающей список с расписаниями:

```
$ curl http://localhost:3000/kitchen/schedules
```

## ПРОФЕССИОНАЛЬНО ОТОБРАЖАЕМ В ТЕРМИНАЛЕ JSON

При выводе JSON в терминал либо с помощью сURL для взаимодействия с API, либо при просмотре файла JSON я рекомендую вам использовать JQ — утилиту командной строки, которая парсит JSON и выдает красивые результаты. Вы можете использовать JQ следующим образом:

```
curl http://localhost:3000/kitchen/schedules | jq .
```

Как вы увидите, mock-сервер, запущенный Prism, выдает совершенно корректную нагрузку в виде списка с расписаниями. Впечатляет, если не сказать больше! Теперь, зная, как запустить mock-серверы для API сервисов кухни и платежей, проанализируем требования к интеграции с ними.

- *Сервис кухни (kitchen.yaml)*. Чтобы запланировать заказ с помощью сервиса кухни, мы должны вызвать эндпоинт POST /kitchen/schedules с полезной нагрузкой в виде списка из пунктов заказа. В ответе на этот вызов мы найдем schedule\_id, который можно использовать для отслеживания статуса заказа.
- *Сервис платежей (payments.yaml)*. Чтобы обработать платеж, мы должны вызвать эндпоинт POST /payments с полезной нагрузкой в виде идентификатора заказа. Это имитационный эндпоинт, который нужен нам для интеграционного тестирования.

Прежде чем отменить заказ, необходимо проверить его статус. Если заказ запланирован на изготовление, мы должны выбрать эндпоинт POST /kitchen/schedules/{schedule\_id}/cancel и отменить его в расписании. Если заказ находится в процессе доставки, пользователю уже нельзя отменить заказ, поэтому мы генерируем исключение.

Для реализации API-интеграции мы будем использовать популярную библиотеку requests на Python. Выполните следующую команду для установки библиотеки с помощью pipenv:

```
$ pipenv install requests
```

Листинг 7.6 расширяет реализацию класса Order, добавляя методы, отвечающие за вызовы API для сервисов кухни и платежей. При тестировании мы ожидаем, что API сервиса кухни будет работать на порте 3001, а сервис платежей — на порте 3000. Этого можно добиться, выполнив следующие команды:

```
$ ./node_modules/.bin/prism mock kitchen.yaml --port 3000  
$ ./node_modules/.bin/prism mock payments.yaml --port 3001
```

При каждом вызове API мы убеждаемся, что в ответе есть ожидаемый статус-код, и в том случае, если это не так, генерируем кастомное исключение APIIntegrationError. Кроме того, если пользователь пытается выполнить

недопустимое действие, например отменить заказ, когда он уже передан курьеру, мы выдаем исключение `InvalidActionError`.

**Листинг 7.6.** Инкапсуляция возможностей для каждого заказа в классе `Order`

```
# file: orders/orders_service/orders.py
```

```
import requests
```

```
from orders.orders_service.exceptions import (
    APIIntegrationError, InvalidActionError
)
```

```
...
class Order:
```

```
    ...
```

```
    def cancel(self):
        if self.status == 'progress':
            kitchen_base_url = "http://localhost:3000/kitchen"
            response = requests.post(
                f"{kitchen_base_url}/schedules/{self.schedule_id}/cancel",
                json={"order": [item.dict() for item in self.items]},
            )
            if response.status_code == 200:
                return
            raise APIIntegrationError(
                f'Could not cancel order with id {self.id}'
            )
        if self.status == 'delivery':
            raise InvalidActionError(
                f'Cannot cancel order with id {self.id}'
            )

    def pay(self):
        response = requests.post(
            'http://localhost:3001/payments', json={'order_id': self.id}
        )
        if response.status_code == 201:
            return
        raise APIIntegrationError(
            f'Could not process payment for order with id {self.id}'
        )

    def schedule(self):
        response = requests.post(
            'http://localhost:3000/kitchen/schedules',
            json={'order': [item.dict() for item in self.items]}
        )
        if response.status_code == 201:
            return response.json()['id']
        raise APIIntegrationError(
            f'Could not schedule order with id {self.id}'
        )
```

Если заказ выполняется, мы отменяем его в расписании, вызывая API сервиса кухни

Если ответ от сервиса кухни будет успешным, возвращаем статус-код 200 (OK)

В противном случае выдаем ошибку `APIIntegrationError`

Не разрешаем отменять заказы, которые уже переданы курьеру

Обработываем платеж, вызывая API сервиса платежей

Планируем заказ на изготовление, вызывая API сервиса кухни

Если от сервиса кухни получен успешный ответ, возвращаем идентификатор расписания



Листинг 7.7 содержит реализацию кастомных исключений, которые мы применяем в сервисе заказов, чтобы сигнализировать об ошибках. Будем выдавать `OrderNotFoundError` в классе `OrdersService`, когда пользователь попытается получить детали несуществующего заказа.

### Листинг 7.7. Кастомные исключения сервиса заказов

# file: orders/orders\_service/exceptions.py

```
class OrderNotFoundError(Exception):
    pass
```

← Исключение, сигнализирующее о том, что заказ не существует

```
class APIIntegrationError(Exception):
    pass
```

← Исключение, сигнализирующее о том, что произошла ошибка интеграции

```
class InvalidActionError(Exception):
    pass
```

← Исключение, сигнализирующее о том, что выполняемое действие недопустимо

Как уже говорилось, модуль API не будет использовать класс `Order` напрямую. Вместо этого он задействует единый интерфейс ко всем нашим адаптерам через класс `OrdersService` (см. листинг 7.4). `OrdersService` инкапсулирует возможности домена заказов и заботится об использовании репозитория для получения объектов заказа и выполнения действий над ними. В листинге 7.8 показана реализация класса `OrdersService`.

### Листинг 7.8. Реализация `OrdersService`

# file: orders/orders\_service/orders\_service.py

```
from orders.orders_service.exceptions import OrderNotFoundError

class OrdersService:
    def __init__(self, orders_repository):
        self.orders_repository = orders_repository
```

← Чтобы создать экземпляр класса `OrdersService`, нам требуется экземпляр хранилища заказов

```
    def place_order(self, items):
        return self.orders_repository.add(items)
```

← Размещаем заказ, создавая запись в базе данных

```
    def get_order(self, order_id):
        order = self.orders_repository.get(order_id)
        if order is not None:
            return order
        raise OrderNotFoundError(f'Order with id {order_id} not found')
```

← Получаем подробную информацию о заказе, используя репозиторий заказов и передавая запрошенный идентификатор

← Если порядок не определен, вызываем исключение `OrderNotFoundError`

```
    def update_order(self, order_id, items):
        order = self.orders_repository.get(order_id)
        if order is None:
            raise OrderNotFoundError(f'Order with id {order_id} not found')
        return self.orders_repository.update(order_id, {'items': items})

    def list_orders(self, **filters):
        limit = filters.pop('limit', None)
        return self.orders_repository.list(limit, **filters)
```

← Записываем фильтры в виде словаря, используя именованные аргументы

```

def pay_order(self, order_id):
    order = self.orders_repository.get(order_id)
    if order is None:
        raise OrderNotFoundError(f'Order with id {order_id} not found')
    order.pay()
    schedule_id = order.schedule() ← После планирования заказа
    return self.orders_repository.update(      обновляем его атрибут schedule_id
        order_id, {'status': 'scheduled', 'schedule_id': schedule_id}
    )

def cancel_order(self, order_id):
    order = self.orders_repository.get(order_id)
    if order is None:
        raise OrderNotFoundError(f'Order with id {order_id} not found')
    order.cancel()
    return self.orders_repository.update(order_id, status="cancelled")

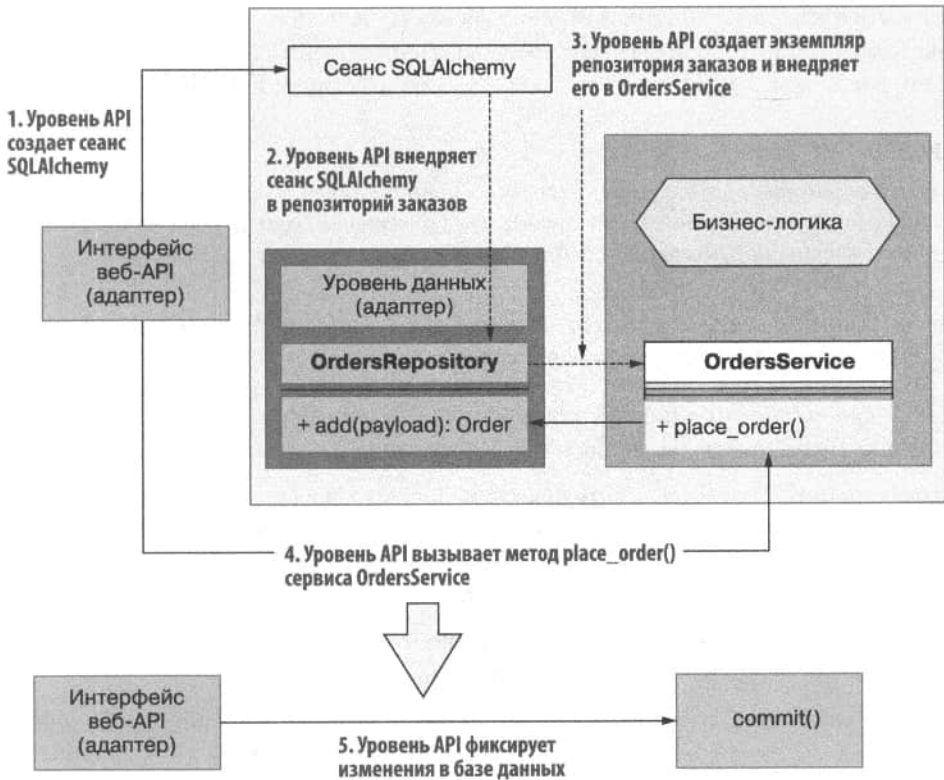
```

Чтобы создать класс `OrdersService`, нам необходим объект репозитория заказов, который можно использовать для добавления заказов в наши записи или удаления оттуда. Для размещения заказа создаем запись в базе данных, используя репозиторий заказов, а для получения информации о заказе запрашиваем соответствующую запись. Если искомый заказ не найден, вызываем исключение `OrderNotFoundError`. Метод `list_orders()` принимает фильтры в виде словаря. Чтобы получить список заказов, репозиторий заказов заставляет нас передать конкретное значение для аргумента `limit`, поэтому мы извлекаем его значение из словаря фильтров с помощью метода `pop()`, который позволяет нам установить значение по умолчанию, а также удаляет ключ из словаря. В методе `pay_order()` мы обрабатываем платеж, используя API сервиса платежей, и, если платеж прошел успешно, планируем заказ, вызывая API сервиса кухни. После планирования заказа обновляем запись заказа, пазначая атрибуту `schedule_id` идентификатор расписания, полученный от API сервиса кухни.

Сервис заказов готов к использованию в нашем модуле API. Однако в этой головоломке есть еще один фрагмент, который нам нужно разгадать. Как говорилось в разделе 7.4, репозиторий заказов не фиксирует в базе данных никаких действий. API как потребитель сервиса `OrdersService` несет ответственность за то, чтобы все действия были зафиксированы по завершении операции. Как именно это происходит? Переходите к разделу 7.6, чтобы узнать!

## 7.6. ВНЕДРЕНИЕ ПАТТЕРНА ЕДИНИЦА РАБОТЫ

В этом разделе мы научимся обрабатывать коммиты и откаты базы данных при взаимодействии с `OrdersService`. Когда мы используем класс `OrdersService` для доступа к любой из его возможностей, мы должны внедрить экземпляр класса `OrdersRepository` (рис. 7.14). Мы также должны открыть сеанс SQLAlchemy перед выполнением каких-либо действий и зафиксировать все изменения в данных, чтобы сохранить их в базе данных.



**Рис. 7.14.** Чтобы сохранить изменения в базе данных, мы могли бы просто заставить уровень API использовать объект сеанса SQLAlchemy для фиксации транзакции. Здесь сплошными линиями показаны вызовы, а пунктирные линии обозначают внедрение зависимостей

Как лучше всего оркестрировать эти операции? Можно использовать различные подходы. Например, просто задействовать объекты сеанса SQLAlchemy для оберты вызовов к `OrdersService`, и, как только операции завершатся успешно, использовать сеанс для выполнения или в ином случае откатиться назад. Это сработает, если `OrdersService` будет иметь дело только с одной базой данных SQL. Однако вдруг нам придется одновременно работать с базами данных другого типа? Придется открыть новый сеанс и для нее. А что, если также придется обрабатывать интеграцию с другими микросервисами в рамках одной и той же операции и убедиться, что мы делаем правильные вызовы API в конце транзакции на случай, если понадобится выполнить откат? Опять же мы могли бы просто добавить специальные условия и защитную блокировку (guards)<sup>1</sup> в наш код. Один и тот же код пришлось бы

<sup>1</sup> Ограничение действий и выбор вариантов с помощью логического выражения. — Примеч. ред.

повторять в каждой функции API, которая взаимодействует с `OrdersService`, так разве было бы плохо, если бы существовал паттерн, который помог бы нам собрать все это в одном месте? И такой паттерн есть, он называется Единица работы.

## ОПРЕДЕЛЕНИЕ

*Единица работы* — это паттерн проектирования, который обеспечивает атомарность наших бизнес-транзакций, гарантируя, что все транзакции одновременно фиксируются или одновременно откатываются, если одна из них не удалась<sup>1</sup>.

Это понятие пришло из мира баз данных, где транзакции реализуются как единицы работы и гарантируется, что каждая транзакция является:

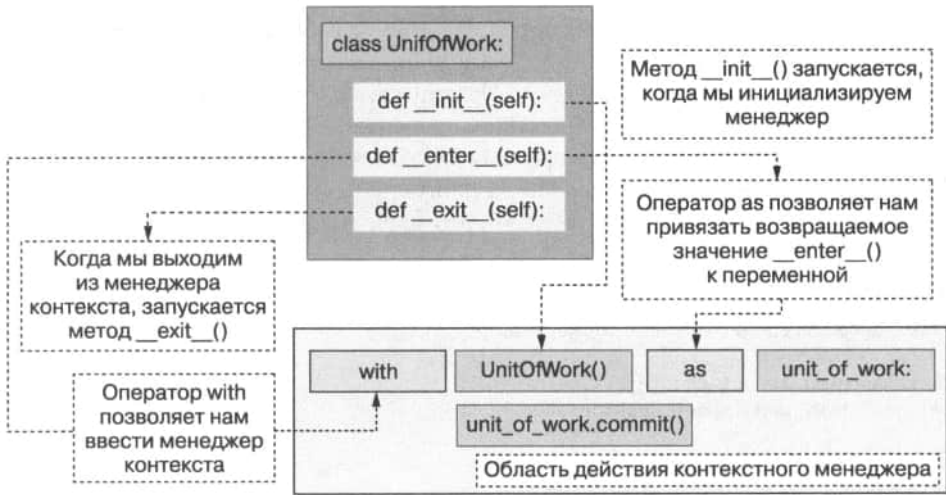
- *атомарной (atomic)* — никакая транзакция не будет зафиксирована в системе частично. Будут либо выполнены все ее подоперации, либо не выполнено ни одной;
- *согласованной (consistent)* — транзакция, достигающая своего нормального завершения и тем самым фиксирующая свои результаты, сохраняет согласованность базы данных. Иначе говоря, каждая успешная транзакция по определению фиксирует только допустимые результаты;
- *изолированной (isolated)* — во время выполнения транзакции параллельные транзакции не должны влиять на ее результат;
- *устойчивой (durable)* — независимо от проблем на нижних уровнях изменения, сделанные успешно завершенной транзакцией, должны остаться сохраненными после возвращения системы в работу. Другими словами, если пользователь получил подтверждение от системы, что транзакция выполнена, он может быть уверен, что сделанные им изменения не будут отменены из-за какого-либо сбоя.

Эти свойства в мире баз данных известны как принципы ACID ([https://en.wikipedia.org/wiki/Database\\_transaction](https://en.wikipedia.org/wiki/Database_transaction), <https://ru.wikipedia.org/wiki/ACID>). Что касается сервисов, то с помощью паттерна Единица работы можно применять эти принципы в нашей деятельности. Объект `Session` в `SQLAlchemy` уже реализует паттерн Единица работы для транзакций базы данных (<https://mng.bz/jA5z>). Это означает, что мы можем добавить столько изменений, сколько требуется, в один и тот же сеанс и зафиксировать их все вместе. Если что-то пойдет не так, мы можем вызвать метод `rollback`, чтобы отменить любые изменения. В Python можно организовать эти шаги с помощью менеджеров контекста.

Менеджер контекста — это паттерн, который позволяет заблокировать ресурс во время операции, обеспечить выполнение всех необходимых работ по очистке на случай, если что-то пойдет не так, и, наконец, освободить блокировку, когда операция будет завершена. Ключевой синтаксической особенностью менеджера контекста является использование оператора `with`. Как видно из рис. 7.15,

<sup>1</sup> Фаулер М., Фоммел М., Райс Д. Шаблоны корпоративных приложений.

контекстные менеджеры могут возвращать объекты, которые мы можем получить, используя оператор `as` в Python. Это полезно, если менеджер контекста предоставляет доступ к ресурсу, например к файлу, над которым мы хотим работать.



**Рис. 7.15.** Менеджер контекста на основе классов имеет методы `__init__()`, `__enter__()` и `__exit__()`

В Python мы можем реализовать менеджеры контекста несколькими способами, в том числе в виде класса или с помощью декоратора `contextmanager()` из модуля `contextlib`<sup>1</sup>. В этом разделе мы реализуем менеджер контекста единицы работы в виде класса. Класс менеджера должен реализовывать как минимум два следующих специальных метода.

- `__enter__()` — определяет операции, которые должны быть выполнены при входе в контекст, например создание сеанса или открытие файла. Если нам нужно выполнить действия над любым из объектов, созданных в методе `__enter__()`, мы можем вернуть объект и получить его значение с помощью оператора `as`, как показано на рис. 7.16.
- `__exit__()` — определяет операции, которые должны быть выполнены при выходе из контекста, например закрытие файла или сеанса. Метод `__exit__()` фиксирует все исключения, возникшие во время выполнения контекста, с помощью трех параметров в сигнатуре метода:
  - `exc_type` — фиксирует тип возникшего исключения;
  - `exc_value` — фиксирует значение, связанное с исключением, обычно сообщение об ошибке;

<sup>1</sup> Рамальо Л. Python. К вершинам мастерства.

- `traceback` — объект обратной трассировки, который может быть использован для точного определения места, где произошло исключение.

Если исключения не возникают, значение этих трех параметров будет равно `None`.

В листинге 7.9 показана реализация паттерна Единица работы в качестве менеджера контекста для сервиса заказов. В методе инициализатора мы получаем объект фабрики сеансов с помощью функции SQLAlchemy `sessionmaker()`. Для нее требуется объект подключения, который мы создаем с помощью функции `create_engine()` SQLAlchemy. Для простоты примера мы жестко кодируем строку подключения к базе данных, чтобы она указывала на нашу локальную базу данных SQLite. В главе 13 вы научитесь параметризовать это значение и извлекать его из среды.

### Листинг 7.9. Паттерн Единица работы в качестве менеджера контекста

```
# file: orders/repository/unit_of_work.py
```

```
from sqlalchemy import create_engine
from sqlalchemy.orm import sessionmaker
```

```
class UnitOfWork:
```

```
    def __init__(self):
        self.session_maker = sessionmaker(
            bind=create_engine('sqlite:// /orders.db')
        )

    def __enter__(self):
        self.session = self.session_maker()
        return self

    def __exit__(self, exc_type, exc_val, traceback):
        if exc_type is not None:
            self.rollback()
            self.session.close()
            self.session.close()

    def commit(self):
        self.session.commit()

    def rollback(self):
        self.session.rollback()
```

Получаем объект фабрики сеансов

Открываем новый сеанс работы с базой данных

Возвращаем экземпляр объекта Единица работы

На выходе из контекста у нас есть доступ к любым исключениям, возникающим во время выполнения

Проверяем, имело ли место исключение

Если произошло исключение, откатить транзакцию

Закрываем сеанс работы с базой данных

Обертка вокруг метода `commit()` SQLAlchemy

Обертка вокруг метода `rollback()` SQLAlchemy

Когда мы входим в контекст, мы создаем новый сеанс базы данных и привязываем его к экземпляру `UnitOfWork`, чтобы получить к нему доступ другими методами. Мы также возвращаем сам объект контекстного менеджера, чтобы вызывающая сторона могла получить доступ к любым его атрибутам, таким как объект `session` или метод `commit()`. При выходе из контекста проверяем, не возникли ли какие-либо исключения при добавлении или удалении объектов в сеанс, и если это так, откатываем изменения, чтобы избежать перехода базы данных в несогласованное состояние.

Мы имеем доступ к типу исключения (`exc_type`) и значению (`exc_val`), а также к контексту обратной трассировки (`traceback`), который можем использовать для записи подробной информации об ошибке. Если исключение не произошло, всем трем параметрам будет присвоено значение `None`. Наконец, мы закрываем сеанс работы с базой данных, чтобы освободить ресурсы базы данных и закрыть область действия транзакции. Мы также добавляем обертки вокруг методов `SQLAlchemy commit()` и `rollback()`, чтобы избежать доступа к внутренним компонентам базы данных на уровне бизнес-логики.

Все это хорошо, но как именно мы должны использовать `UnitOfWork` в сочетании с репозиторием заказов и `OrdersService`? В следующем разделе мы подробнее обсудим этот вопрос, но сначала рассмотрим пример использования этих компонентов вместе в листинге 7.10. Мы вводим контекст единицы работы с помощью синтаксиса Python для менеджеров контекста, используя оператор `with`. Мы также используем оператор `as` для привязки возвращаемого значения метода `__enter__()` класса `UnitOfWork` к переменной `unit_of_work`. Затем получаем экземпляр хранилища `orders`, передавая в него объект сеанса базы данных объекта `UnitOfWork`, и экземпляр класса `OrdersService`, передавая в него объект репозитория заказов. После этого мы используем объект сервиса заказов для размещения заказа и фиксируем транзакцию с помощью метода `commit()` класса `UnitOfWork`.

**Листинг 7.10.** Пример использования паттернов Единица работы и Репозиторий

```

with UnitOfWork() as unit_of_work:
    repo = OrdersRepository(unit_of_work.session)
    orders_service = OrdersService(repo)
    orders_service.place_order(order_details)
    unit_of_work.commit()
  
```

Вводим контекст единицы работы

Получаем экземпляр репозитория заказов, передаваемый в сеанс UnitOfWork

Получаем экземпляр класса OrdersService, передаваемый в объекте хранилища orders

Размещаем заказ

Фиксируем транзакцию

Теперь, когда у нас есть единица работы, которую мы можем использовать для фиксации транзакций, посмотрим, как объединить все это вместе, интегрируя уровень API с уровнем сервиса.

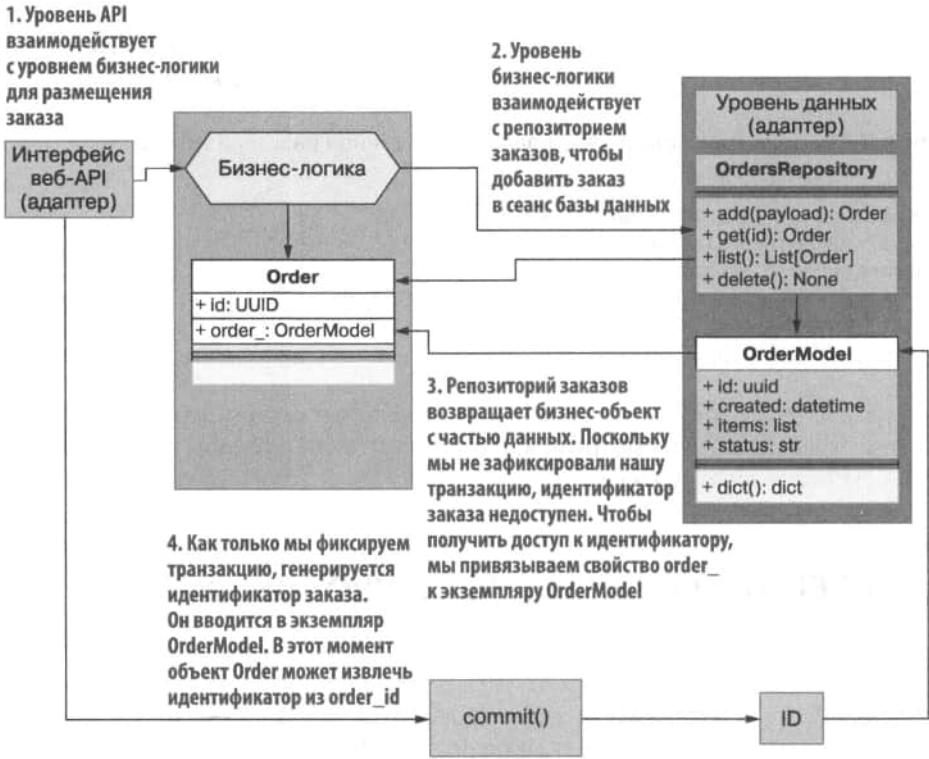
## 7.7. ИНТЕГРАЦИЯ УРОВНЯ API И УРОВНЯ СЕРВИСА

В этом разделе мы объединим все, чему научились в данной главе, чтобы интегрировать уровень сервиса с уровнем API. Для этого воспользуемся шаблоном из листинга 7.10. Когда пользователь пытается выполнить действие с заказом, сначала обязательно проверяем, существует ли такой заказ. Если его нет, возвращаем ответ об ошибке 404 (Not Found).

В листинге 7.11 показана новая версия модуля `orders/web/api/api.py`. В первую очередь в каждой функции мы входим в контекст `UnitOfWork`, убедившись, что

привязываем объект контекста к переменной `unit_of_work`. Затем создаем экземпляры `OrdersRepository`, используя объект `session` из контекстного объекта `UnitOfWork`. Как только у нас есть экземпляр репозитория, мы вводим его в `OrdersService` при создании экземпляра сервиса. Затем используем сервис для выполнения операций, необходимых в каждом эндпоинте. В эндпоинтах, выполняющих действия над определенным заказом, мы защищаемся от ошибки `OrderNotFoundError` в `OrdersService`, возникающей, если запрашиваемый заказ не существует.

В функции `create_order()` извлекаем словарное представление заказа с помощью `order.dict()` до выхода из контекста `UnitOfWork`, чтобы иметь доступ к свойствам, созданным базой данных в процессе фиксации, таким как ID заказа. Помните, что идентификатора заказа не существует до тех пор, пока изменения не зафиксированы в базе данных, то есть он доступен только в рамках сеанса базы данных. В нашей реализации это означает, что мы должны получить доступ к идентификатору перед выходом из контекста `UnitOfWork`, поскольку сессия базы данных закрывается непосредственно перед выходом из контекста. Рисунок 7.16 иллюстрирует этот процесс.



**Рис. 7.16.** Когда мы размещаем заказ, объект, возвращаемый репозиторием заказов, не содержит идентификатора. Он будет назначен, как только мы зафиксироваем транзакцию базы данных через экземпляр `OrderModel`. Поэтому мы привязываем экземпляр модели к объекту `Order`, чтобы он мог извлекать идентификатор из модели после фиксации



**Листинг 7.11.** Интеграция между уровнем API и уровнем сервиса

```
# file: orders/web/api/api.py

from http import HTTPStatus
from typing import List, Optional
from uuid import UUID

from fastapi import HTTPException
from starlette import status
from starlette.responses import Response

from orders.orders_service.exceptions import OrderNotFoundError
from orders.orders_service.orders_service import OrdersService
from orders.repository.orders_repository import OrdersRepository
from orders.repository.unit_of_work import UnitOfWork
from orders.web.app import app
from orders.web.api.schemas import (
    GetOrderSchema, Integrating the API layer and the service layer
    CreateOrderSchema,
    GetOrdersSchema,
)

@app.get('/orders', response_model=GetOrdersSchema)
def get_orders(
    cancelled: Optional[bool] = None,
    limit: Optional[int] = None,
):
    with UnitOfWork() as unit_of_work:
        repo = OrdersRepository(unit_of_work.session)
        orders_service = OrdersService(repo)
        results = orders_service.list_orders(
            limit=limit, cancelled=cancelled
        )
        return {'orders': [result.dict() for result in results]}

@app.post(
    '/orders',
    status_code=status.HTTP_201_CREATED,
    response_model=GetOrderSchema,
)
def create_order(payload: CreateOrderSchema):
    with UnitOfWork() as unit_of_work:
        repo = OrdersRepository(unit_of_work.session)
        orders_service = OrdersService(repo)
        order = orders_service.place_order(payload.dict()['order'])
        order = payload.dict()['order']
        for item in order:
            item['size'] = item['size'].value
        order = orders_service.place_order(order)

        unit_of_work.commit()
        return_payload = order.dict()
    return return_payload
```

Вводим контекст единицы работы

Размещаем заказ

Получаем доступ к словарному представлению заказа перед выходом из контекста единицы работы

```

@app.get('/orders/{order_id}', response_model=GetOrderSchema)
def get_order(order_id: UUID):
    try:
        with UnitOfWork() as unit_of_work:
            repo = OrdersRepository(unit_of_work.session)
            orders_service = OrdersService(repo)
            order = orders_service.get_order(order_id=order_id)
            return order.dict()
    except OrderNotFoundError:
        raise HTTPException(
            status_code=404, detail=f'Order with ID {order_id} not found'
        )

```

Используем блок try/except для перехвата исключения OrderNotFoundError

```

@app.put('/orders/{order_id}', response_model=GetOrderSchema)
def update_order(order_id: UUID, order_details: CreateOrderSchema):
    try:
        with UnitOfWork() as unit_of_work:
            repo = OrdersRepository(unit_of_work.session)
            orders_service = OrdersService(repo)
            order = order_details.dict()['order']
            for item in order:
                item['size'] = item['size'].value
            order = orders_service.update_order(
                order_id=order_id, items=order
            )
            unit_of_work.commit()
            return order.dict()
    except OrderNotFoundError:
        raise HTTPException(
            status_code=404, detail=f'Order with ID {order_id} not found'
        )

```

```

@app.delete(
    "/orders/{order_id}",
    status_code=status.HTTP_204_NO_CONTENT,
    response_class=Response,
)

```

```

def delete_order(order_id: UUID):
    try:
        with UnitOfWork() as unit_of_work:
            repo = OrdersRepository(unit_of_work.session)
            orders_service = OrdersService(repo)
            orders_service.delete_order(order_id=order_id)
            unit_of_work.commit()
        return
    except OrderNotFoundError:
        raise HTTPException(
            status_code=404, detail=f'Order with ID {order_id} not found'
        )

```

```

@app.post('/orders/{order_id}/cancel', response_model=GetOrderSchema)
def cancel_order(order_id: UUID):
    try:
        with UnitOfWork() as unit_of_work:

```

```
repo = OrdersRepository(unit_of_work.session)
orders_service = OrdersService(repo)
order = orders_service.cancel_order(order_id=order_id)
unit_of_work.commit()
return order.dict()
except OrderNotFoundError:
    raise HTTPException(
        status_code=404, detail=f'Order with ID {order_id} not found'
    )

@app.post('/orders/{order_id}/pay', response_model=GetOrderSchema)
def pay_order(order_id: UUID):
    try:
        with UnitOfWork() as unit_of_work:
            repo = OrdersRepository(unit_of_work.session)
            orders_service = OrdersService(repo)
            order = orders_service.pay_order(order_id=order_id)
            unit_of_work.commit()
        return order.dict()
    except OrderNotFoundError:
        raise HTTPException(
            status_code=404, detail=f'Order with ID {order_id} not found'
        )
```

На этом мы завершаем наше путешествие по реализации уровня сервиса для сервиса заказов. Паттерны, рассмотренные в этой главе, применимы не только к миру API и микросервисов, но и ко всем моделям приложений в целом. В частности, паттерн Репозиторий всегда поможет вам убедиться, что уровень доступа к данным полностью отделен от уровня бизнес-логики, а паттерн Единица работы гарантирует, что все бизнес-транзакции обрабатываются атомарно и последовательно.

## РЕЗЮМЕ

- Гексагональная архитектура, или архитектура портов и адаптеров, — это архитектурный паттерн программного обеспечения, который подразумевает отделение уровня бизнес-логики от деталей реализации базы данных и интерфейса приложения.
- Принцип инверсии зависимостей учит нас, что детали реализации компонентов приложения должны зависеть от интерфейсов. Это помогает нам отделить компоненты от деталей реализации их зависимостей.
- Для взаимодействия с базой данных можно использовать библиотеку ORM, такую как SQLAlchemy, позволяющую преобразовывать таблицы и строки базы данных в классы и объекты. Это дает возможность расширить наши модели баз данных полезной функциональностью.

- Репозиторий — это паттерн разработки программного обеспечения, который помогает отделить уровень данных от уровня бизнес-логики путем добавления уровня абстракции, предоставляющего интерфейс списков данных в памяти. Независимо от того, какой движок базы данных используется, уровень бизнес-логики всегда будет получать из репозитория одни и те же объекты.
- Паттерн Единица работы помогает гарантировать, что все бизнес-транзакции, являющиеся частью операций приложения, одновременно завершаются успешно или терпят неудачу. Если одна из транзакций завершается неудачей, паттерн Единица работы обеспечивает откат всех изменений. Благодаря этому механизму данные никогда не остаются в несогласованном состоянии.

## *Часть III*

# *Проектирование и реализация GraphQL API*

В части II вы узнали, что REST — это технология API, которая позволяет изменять состояние ресурса или получать его с сервера. Когда ресурс представлен большой полезной нагрузкой, его извлечение с сервера приводит к передаче большого объема данных. С появлением клиентов API, работающих на мобильных устройствах с ограниченным доступом к сети и ограниченным объемом хранилища и памяти, обмен большой полезной нагрузкой часто приводит к непадежной связи. В 2012 году в Facebook остро осознали эти проблемы и разработали новую технологию, позволяющую клиентам API выполнять детализированные запросы к данным на сервере. Так в 2015 году появился GraphQL.

GraphQL — это язык запросов для API. Вместо полных представлений ресурсов GraphQL позволяет извлекать одно или несколько свойств ресурса, таких как цена товара или статус заказа. С помощью GraphQL мы также можем моделировать взаимосвязи между различными объектами, что позволяет в рамках одного запроса извлекать с сервера свойства различных ресурсов, например ингредиенты товара и его доступность на складе.

Несмотря на преимущества GraphQL, многие разработчики не знакомы с ним или не знают, как он работает, поэтому обычно его не рассматривают в первую очередь при выборе технологии для создания API. В части III вы узнаете все, что нужно знать о проектировании и создании высококачественных API GraphQL, а также о том, как их использовать. Вы также узнаете, что такое GraphQL, как он работает и когда его задействовать, чтобы принимать более обоснованные решения в своей стратегии работы с API.

# Проектирование GraphQL API

---

## В этой главе

- ✓ Принципы работы GraphQL.
- ✓ Подготовка спецификации API с использованием языка определения схем (Schema Definition Language, SDL).
- ✓ Изучение встроенных скалярных типов и структур данных GraphQL и создание кастомных типов объектов.
- ✓ Создание значимых связей между типами GraphQL.
- ✓ Разработка запросов GraphQL и мутаций.

GraphQL — один из самых популярных протоколов для построения веб-API. Это подходящий выбор для обеспечения интеграции между микросервисами и создания интеграций с фронтенд-приложениями. GraphQL предоставляет потребителям API полный контроль над данными, которые они хотят получить с сервера, и над тем, как они хотят их получать.

В этой главе вы научитесь разрабатывать GraphQL API. Вы сделаете это на практическом примере: разработаете GraphQL API для сервиса продукции CoffeeMesh. Этот сервис владеет данными о товарах CoffeeMesh, а также их ингредиентах. У каждого товара и ингредиента богатый список свойств, описывающих их особенности. Однако, когда клиент запрашивает список продуктов, он, скорее всего, заинтересован в получении только нескольких деталей о каждом из продуктов. Кроме того, клиентам может быть интересно отслеживать взаимосвязи между товарами,

ингредиентами и другими объектами, принадлежащими сервису продукции. По этим причинам GraphQL станет отличным выбором для создания API сервиса продукции.

По мере создания спецификации для API сервиса продукции вы узнаете о скалярных типах GraphQL, разработке кастомных типов объектов, а также о запросах и мутациях. К концу этой главы вы поймете, чем GraphQL отличается от других типов API и когда имеет смысл его использовать. Нам еще многое предстоит узнать, поэтому без лишних слов давайте начнем наше путешествие!

Спецификацию, которую мы разрабатываем в этой главе, вы можете взять в репозитории GitHub, прилагаемом к книге. Код для главы доступен в папке `ch08`.

## 8.1. ЗНАКОМСТВО С GRAPHQL

В этом разделе мы рассмотрим, что такое GraphQL, в чем его преимущества и когда его имеет смысл использовать. На официальном сайте GraphQL определяется как «язык запросов для API и среда для выполнения этих запросов с вашими имеющимися данными»<sup>1</sup>. Что это означает на самом деле? Что GraphQL — это спецификация, которая позволяет выполнять запросы на сервере API. Точно так же, как SQL предоставляет язык запросов для баз данных, GraphQL предоставляет язык запросов для API<sup>2</sup>. GraphQL также предоставляет описание того, как эти запросы решаются на сервере, так что каждый может реализовать среду выполнения GraphQL на любом языке программирования<sup>3</sup>.

Точно так же, как SQL можно использовать для определения схем таблиц наших баз данных, GraphQL можно применять для создания спецификаций, описывающих тип данных, которые могут быть запрошены с наших серверов. Спецификация GraphQL API называется схемой и написана согласно стандарту Schema Definition Language (SDL). В этой главе вы узнаете, как использовать SDL для создания спецификации API сервиса продукции.

GraphQL был выпущен в 2015 году и с тех пор завоевал популярность как один из самых популярных вариантов построения веб-API. Следует отметить, что в спецификации ничего не говорится о том, что GraphQL нужно использовать через HTTP, но на практике в GraphQL API этот протокол применяется чаще всего.

Что замечательного в GraphQL? Он дает пользователям полный контроль над тем, какие данные они хотят получить от сервера. Например, в API сервиса продукции мы храним множество сведений о каждом товаре: его название, цену, наличие, ингредиенты и т. д. Если пользователь хочет получить список только названий

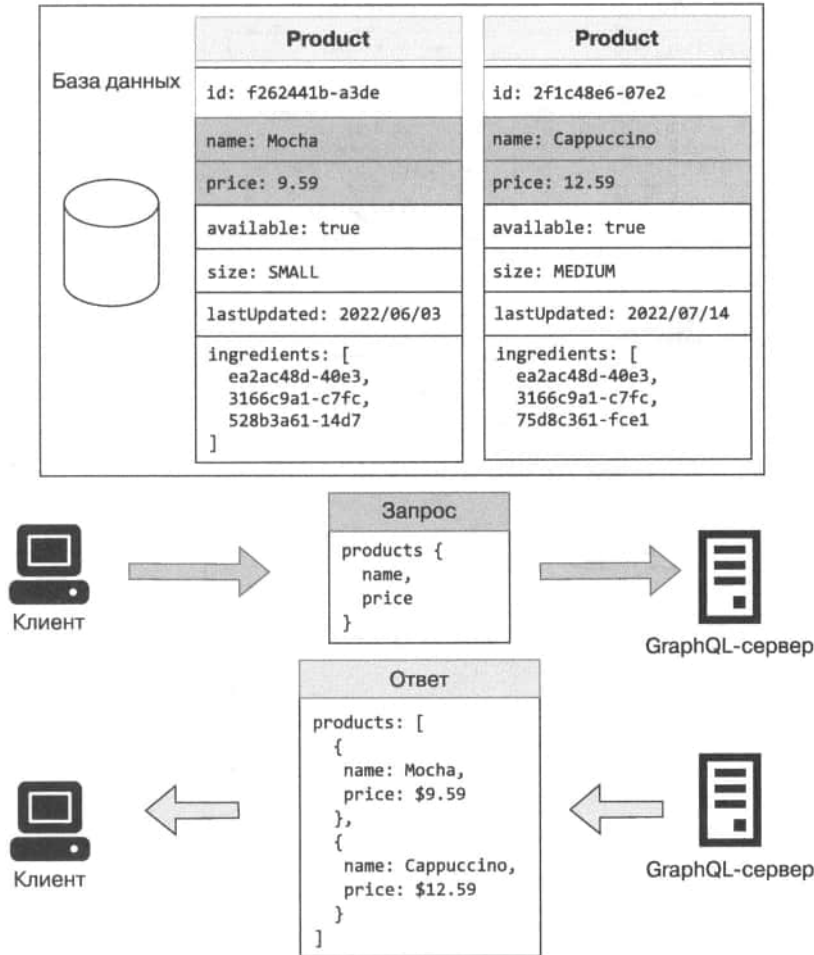
<sup>1</sup> Это определение появляется на главной странице GraphQL: <https://graphql.org/>.

<sup>2</sup> Так GraphQL и SQL сравнивают Ева Порселло и Алекс Бэнкс в своей книге «GraphQL. Язык запросов для современных веб-приложений» (Питер, 2019. — С. 47–50).

<sup>3</sup> На сайте <https://graphql.org/code/> приводится актуальный список программ, доступных для создания серверов GraphQL на разных языках.



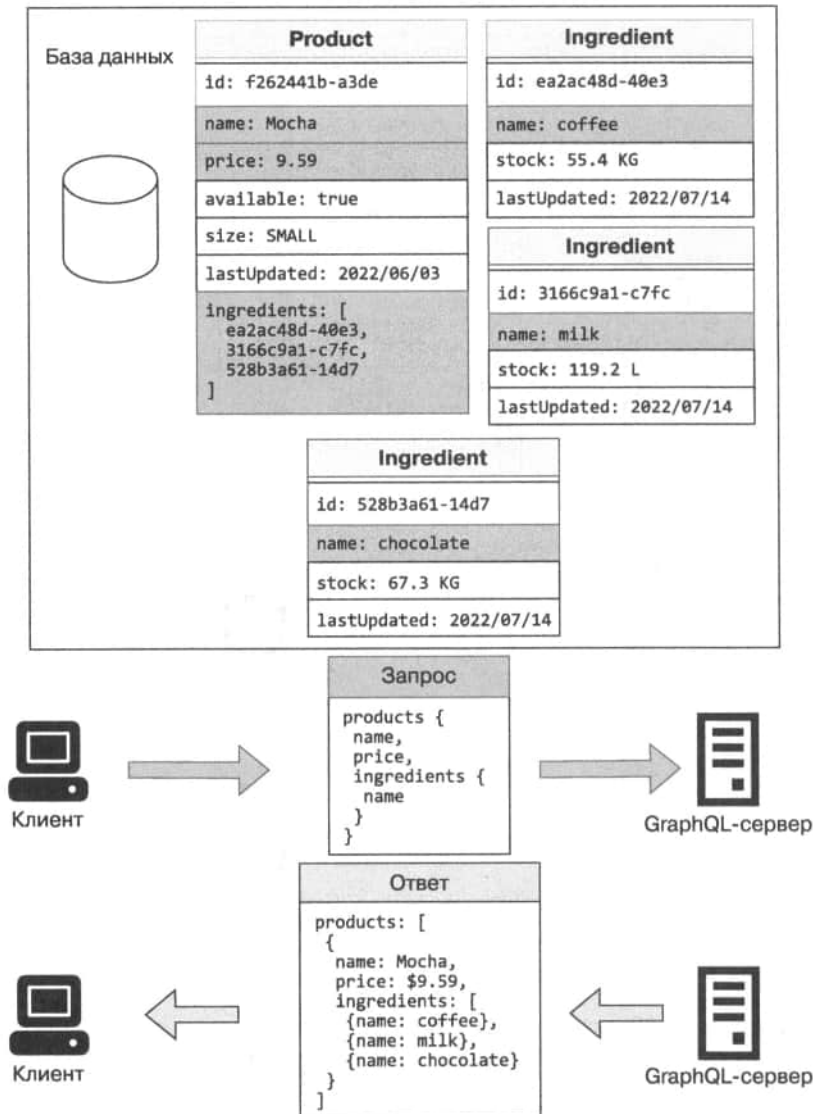
товаров и цен, с помощью GraphQL он может это сделать (рис. 8.1). При использовании же других типов API, таких как REST, вы получите всю информацию о каждом товаре. Поэтому, когда важно предоставить клиенту полный контроль над тем, как получать данные с сервера, лучше всего выбрать GraphQL.



**Рис. 8.1.** Используя GraphQL API, клиент может запросить список элементов с конкретными деталями. В этом примере клиент запрашивает название и цену каждого товара в API сервиса продукции

Еще одним большим преимуществом GraphQL является возможность создавать связи между различными типами ресурсов и предоставлять эти связи клиентам для использования в их запросах. Например, в API сервиса продукции товары и ингредиенты — это разные, но связанные типы ресурсов. Если пользователь хочет получить список товаров, в том числе их названия, цены и ингредиенты, то благодаря

GraphQL он может сделать это, используя связи между ресурсами (рис. 8.2). Поэтому для сервисов, где есть сильно взаимосвязанные ресурсы и где клиентам полезно исследовать и запрашивать эти связи, GraphQL станет отличным выбором.

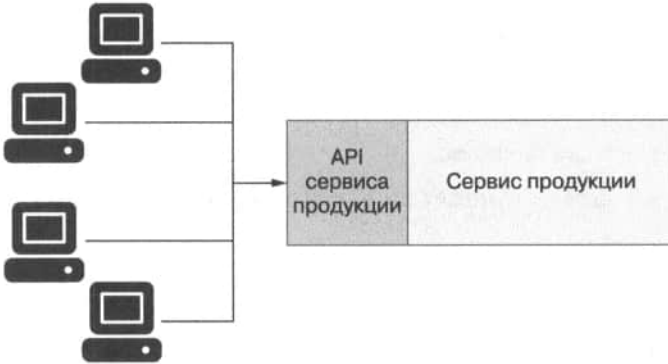


**Рис. 8.2.** Используя GraphQL, клиент может запросить подробную информацию о конкретном ресурсе и других ресурсах, связанных с ним. В этом примере API сервиса продукции имеет два типа ресурсов: товары и ингредиенты, оба связаны через поле ингредиентов товара. Используя эту связь, клиент может запросить название и цену каждого товара, а также перечень ингредиентов

В последующих разделах вы узнаете, как создать спецификацию GraphQL для сервиса продукции. Вы увидите, как определить типы данных, создать значимые связи между ресурсами и определить операции для запроса данных и изменения состояния сервера. Но прежде необходимо понять требования к API сервиса продукции.

## 8.2. ПРЕДСТАВЛЯЕМ API СЕРВИСА ПРОДУКЦИИ

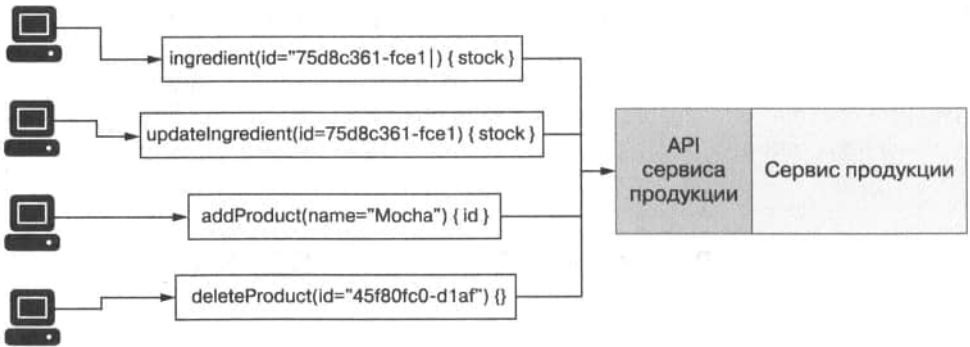
Прежде чем приступить к работе над спецификацией API, важно собрать информацию о требованиях к API. Как показано на рис. 8.3, API сервиса продукции — это интерфейс к одноименному сервису. Чтобы определить требования к API, нам нужно знать, что пользователи сервиса могут делать с его помощью.



**Рис. 8.3.** Для взаимодействия с сервисом продукции клиенты используют соответствующий API

Сервис продукции владеет данными о товарах, предлагаемых платформой CoffeeMesh. Сотрудники CoffeeMesh должны иметь возможность использовать сервис продукции для управления имеющимися запасами каждого товара, а также для поддержания информации об ингредиентах в актуальном состоянии (рис. 8.4). В частности, у них должна быть возможность запрашивать запасы того или иного товара или ингредиента и обновлять их при поступлении новых запасов на склад. Они также должны иметь возможность добавлять в систему новые товары или ингредиенты и удалять старые. Таким образом, у нас уже сложный список условий, поэтому разобьем его на конкретные технические требования.

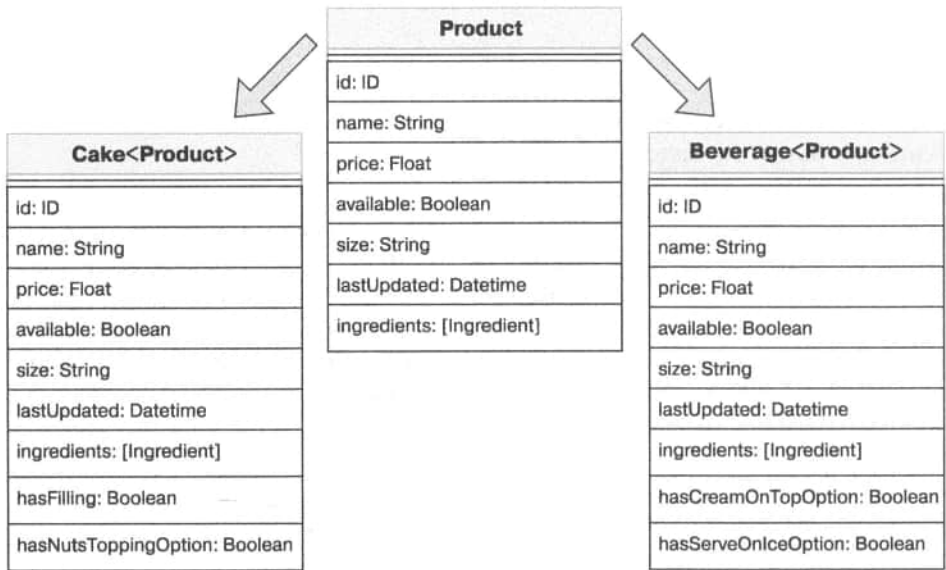
Начнем с моделирования ресурсов, управляемых API сервиса продукции. Нам нужно знать, какой тип ресурсов следует предоставлять через API и свойства товаров. Из описания выше мы знаем, что сервис продукции управляет двумя типами ресурсов: товарами и ингредиентами. Сначала проанализируем товары.



**Рис. 8.4.** Сотрудники CoffeeMesh используют сервис продукции для управления товарами и ингредиентами

Платформа CoffeeMesh предлагает два вида товаров: выпечку и напитки. Как видно на рис. 8.5, и выпечка, и напитки имеют общие свойства: название товара, цену, объем, список ингредиентов и его наличие. У выпечки есть два дополнительных свойства:

- `hasFilling` — указывает, есть ли начинка;
- `hasNutsToppingOption` — указывает, может ли клиент добавить ореховую посыпку.



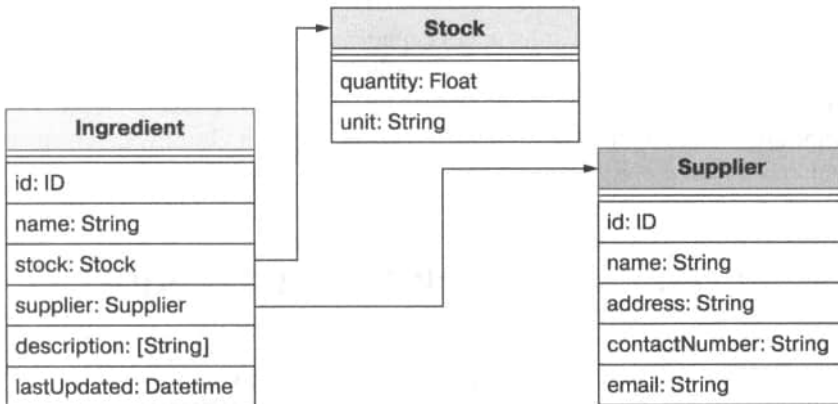
**Рис. 8.5.** CoffeeMesh демонстрирует два типа товаров: выпечку и напитки, которые имеют общие свойства

У напитков есть следующие два дополнительных свойства:

- `hasCreamOnTopOption` — указывает, может ли клиент добавить в напиток сливки;
- `hasServeOnIceOption` — указывает, может ли клиент выбрать подачу напитка со льдом.

А как насчет ингредиентов? Как показано на рис. 8.6, мы можем представить все ингредиенты через одну сущность с такими атрибутами:

- `name` — название ингредиента;
- `stock` — доступный запас ингредиента. Поскольку различные ингредиенты измеряются в разных единицах, таких как килограммы или литры, мы выражаем имеющиеся запасы в единицах измерения;
- `description` — набор заметок, которые сотрудники CoffeeMesh могут использовать для описания и классификации товара;
- `supplier` — информация о компании, которая поставляет ингредиент в CoffeeMesh: ее название, адрес, контактный телефон и адрес электронной почты.



**Рис. 8.6.** Список свойств, характеризующих ингредиент. Поставщик ингредиента определяется ресурсом Supplier, в то время как запасы описываются с помощью объекта Stock

Теперь, когда мы смоделировали основные ресурсы для сервиса продукции, обратим внимание на операции, которые нужно предоставлять через API. Мы будем отличать операции чтения от операций записи/удаления. Это различие будет иметь смысл, когда мы подробнее рассмотрим данные операции в разделах 8.8 и 8.9.

Основываясь на предыдущем обсуждении, рассмотрим следующие операции чтения:

- `allProducts()` — возвращает полный список товаров, доступных в каталоге CoffeeMesh;

- `allIngredients()` — возвращает полный список ингредиентов, используемых CoffeeMesh для приготовления продукции;
- `products()` — позволяет пользователям фильтровать полный список товаров по определенным критериям, таким как наличие, максимальная цена и др.;
- `product()` — позволяет пользователям получить информацию об одном товаре;
- `ingredient()` — позволяет пользователям получить информацию об одном ингредиенте.

Что касается операций записи/удаления, то из предыдущего обсуждения ясно, что мы должны предоставить следующие возможности:

- `addIngredient()` — для добавления новых ингредиентов;
- `updateStock()` — для обновления запасов ингредиента;
- `addProduct()` — для добавления новых товаров;
- `updateProduct()` — для обновления существующих товаров;
- `deleteProduct()` — для удаления товаров из каталога.

Теперь, когда мы понимаем требования к API сервиса продукции, пора переходить к созданию спецификации API. В следующих разделах вы научитесь создавать спецификацию GraphQL для API сервиса продукции и параллельно узнаете, как работает GraphQL. Наша первая остановка — система типов GraphQL, которую мы будем использовать для моделирования ресурсов, управляемых API.

## 8.3. ЗНАКОМСТВО С СИСТЕМОЙ ТИПОВ GRAPHQL

В этом разделе мы рассмотрим систему типов GraphQL. В GraphQL типы — это определения, которые позволяют нам описывать свойства наших данных. Это строительные блоки GraphQL API, и мы используем их для моделирования ресурсов, принадлежащих API. В этом разделе вы научитесь применять систему типов GraphQL для описания ресурсов, определенных в разделе 8.2.

### 8.3.1. Создание определений свойств с помощью скалярных величин

В этом разделе объясняется, как определять тип свойства с помощью системы типов GraphQL. Мы различаем скалярные и объектные типы. Как вы узнаете в подразделе 8.3.2, объектные типы — это коллекции свойств, которые представляют сущности. *Скалярные типы* — это такие типы, как булевы значения или целые числа. Синтаксис для определения типа свойства очень похож на использование

подсказок типов в Python: мы указываем имя свойства, за которым следует двоеточие, а справа от двоеточия — тип свойства. Например, в разделе 8.2 мы обсуждали, что у выпечки есть два разных свойства: `hasFilling` и `hasNutsToppingOption`, оба из которых являются булевыми. В системе типов GraphQL мы описываем эти свойства следующим образом:

```
hasFilling: Boolean
hasNutsToppingOption: Boolean
```

GraphQL поддерживает такие типы скаляров:

- *строки* (`String`) — для текстовых свойств объектов;
- *целые числа* (`Int`) — для числовых свойств объекта;
- *значения с плавающей точкой* (`Float`) — для числовых свойств объекта с десятичной точностью;
- *логические* (`Boolean`) — для бинарных свойств объекта;
- *уникальные идентификаторы* (`ID`) — для описания идентификатора объекта. Технически идентификаторы представляют собой строки, но GraphQL гарантирует, что идентификатор каждого объекта уникален.

Помимо определения типа свойства, мы также можем указать, является ли оно *non-nullable* — не допускающим значения `null`. Nullable-свойства — это свойства, которым может быть присвоено `null`, когда мы не знаем их значения. Мы помечаем свойство как не допускающее значения `null`, ставя восклицательный знак в конце определения свойства:

```
name: String!
```

Эта строка определяет свойство `name` типа `String` и с помощью восклицательного знака помечает его как не допускающее значения `null`. Это означает, что, когда бы мы ни запрашивали это свойство из API, оно всегда будет строкой.

### 8.3.2. Моделирование ресурсов с использованием объектных типов

В этом разделе объясняется, как использовать систему типов GraphQL для моделирования ресурсов. Ресурсы — это объекты, управляемые API, например ингредиенты, выпечка и напитки, о которых мы говорили в разделе 8.2. В GraphQL каждый из этих ресурсов моделируется как объектный тип. *Объектные типы* — это коллекции свойств, и, как видно из названия, мы используем их для определения объектов. Для этого указываем ключевое слово `type`, за которым следуют имя объекта и список его свойств в фигурных скобках. Свойство определяется путем указания имени свойства, за которым следует двоеточие, и его типа справа от

двоеточия. В GraphQL ID — это тип с уникальным значением. Восклицательный знак в конце свойства указывает на то, что свойство не может иметь значение null. В листинге 8.1 показано, как мы описываем ресурс Cake как объектный тип. Листинг содержит основные свойства выпечки, в частности ID, название и стоимость.

### Листинг 8.1. Определение объектного типа Cake

```
type Cake {
  id: ID!
  name: String!
  price: Float
  available: Boolean!
  hasFilling: Boolean!
  hasNutsToppingOption: Boolean!
}
```

← Определяем тип объекта

← Определяем свойство ID, не допускающее обнуления

### ТИПЫ И ОБЪЕКТНЫЕ ТИПЫ

Для удобства везде в книге мы используем понятия «тип» (type) и «объектный тип» (object type) как взаимозаменяемые, если не указано иное.

Некоторые определения свойств в листинге 8.1 заканчиваются восклицательным знаком. Как мы уже говорили, в GraphQL восклицательный знак означает, что свойство необнуляемое, то есть каждый объект Cake, возвращаемый нашим API, будет содержать идентификатор, название, наличие, а также свойства hasFilling и hasNutsToppingOption. Это также гарантирует, что ни одному из этих свойств не будет присвоено значение null. Для разработчиков клиентских API эта информация очень ценна, поскольку они могут рассчитывать на то, что эти свойства всегда будут присутствовать, и создают свои программы с учетом этого предположения. В листинге 8.2 показаны определения для типов Beverage и Ingredient. Здесь также показано определение типа Supplier, который содержит информацию о компании, поставляющей определенный ингредиент, а в подразделе 8.5.1 мы рассмотрим, как связать его с типом Ingredient.

### Листинг 8.2. Определения объектных типов Beverage и Ingredient

```
type Beverage {
  id: ID!
  name: String!
  price: Float
  available: Boolean!
  hasCreamOnTopOption: Boolean!
  hasServeOnIceOption: Boolean!
}
```

```
type Ingredient {
  id: ID!
  name: String!
}
```

```
type Supplier {
```



```
id: ID!  
name: String!  
address: String!  
contactNumber: String!  
email: String!  
}
```

Теперь, когда мы знаем, как определять объектные типы, завершим наше исследование системы типов GraphQL, научившись создавать собственные кастомные типы.

### 8.3.3. Создание кастомных скалярных величин

В подразделе 8.3.1 мы рассмотрели встроенные скаляры GraphQL: `String`, `Int`, `Float`, `Boolean` и `ID`. Чаще всего этого списка скалярных типов будет достаточно для моделирования наших ресурсов API. Однако в некоторых случаях встроенные скалярные типы GraphQL могут оказаться ограниченными. Тогда мы можем определить собственные кастомные скалярные типы. Например, может понадобиться иметь возможность представить тип даты, тип URL-адреса или тип адреса электронной почты.

Поскольку API сервиса продукции используется для управления товарами и ингредиентами и внесения в них изменений, полезно добавить свойство `lastUpdated`, которое сообщает нам о последнем изменении записи. `lastUpdated` должно быть скаляром типа `Datetime`. В GraphQL нет встроенного скаляра такого типа, поэтому мы должны создать его (листинг 8.3). Чтобы объявить скаляр даты-времени, введем следующий оператор:

```
scalar Datetime
```

Нам также необходимо определить, как этот скалярный тип будет проверяться и сериализоваться. Зададим правила проверки и сериализации скаляра в серверной реализации (рассмотрим ее в главе 10).

#### Листинг 8.3. Использование кастомного скалярного типа `Datetime`

```
scalar Datetime  
  
type Cake {  
  id: ID!  
  name: String!  
  price: Float  
  available: Boolean!  
  hasFilling: Boolean!  
  hasNutsToppingOption: Boolean!  
  lastUpdated: Datetime!  
}
```

Объявляем кастомный скаляр  
даты и времени

Объявляем свойство, не допускающее  
значения null, с типом `Datetime`

На этом мы завершаем изучение скаляров и объектных типов GraphQL. Теперь вы можете определять основные типы объектов в GraphQL и создавать собственные скаляры. В следующих разделах вы научитесь создавать связи между различными типами объектов, а также узнаете, как использовать списки, интерфейсы, перечисления и многое другое.

## 8.4. ПРЕДСТАВЛЕНИЕ КОЛЛЕКЦИЙ ЭЛЕМЕНТОВ С ПОМОЩЬЮ СПИСКОВ

В этом разделе мы рассмотрим списки GraphQL. *Списки* — это массивы типов, и они задаются в квадратных скобках. Списки полезны, когда нужно определить свойства, представляющие коллекции элементов. Как обсуждалось в разделе 8.2, объект `Ingredient` содержит свойство `description`, которое хранит коллекции сведений об ингредиенте (листинг 8.4).

**Листинг 8.4.** Представляя список строк, мы определяем список элементов, не допускающих значения `null`

```
type Ingredient {
  id: ID!
  name: String!
  description: [String!]
}
```

↑  
Определяем список элементов,  
не допускающих значения null

Посмотрите внимательно на использование восклицательных знаков в свойстве `description`: мы определяем его как свойство, допускающее значение `null` (nullable) с элементами, не допускающими значения `null` (non-nullable). Что это означает? Когда мы возвращаем ингредиент из API, поле описания может отсутствовать, и если оно есть, то будет содержать список строк.

Когда речь идет о списках, очень важно обратить внимание на использование восклицательных знаков. В свойствах списка мы можем добавить два восклицательных знака: один для самого списка, а другой — для элемента внутри списка. Чтобы сделать и список, и его содержимое необнуляемыми, мы используем восклицательные знаки для обоих. Наличие восклицательных знаков в типах списков — один из самых распространенных источников путаницы у пользователей GraphQL. В табл. 8.1 приведены возможные возвращаемые значения для каждой комбинации восклицательных знаков в определении свойства списка.

### ИСПОЛЬЗУЙТЕ ВОСКЛИЦАТЕЛЬНЫЕ ЗНАКИ С ОСТОРОЖНОСТЬЮ

В GraphQL восклицательный знак указывает на то, что свойство не допускает значения `null`, то есть объект должен иметь это свойство и его значение не может быть равно `null`. Когда речь идет о списках, мы можем использовать два восклицательных знака: один для самого списка, а другой — для элемента внутри списка. Различные комбинации восклицательных знаков дают различные представления свойства.

Таблица 8.1. Допустимые возвращаемые значения для свойств списка

	[Word]	[Word!]	[Word]!	[Word!]!
null	Валидно	Валидно	Невалидно	Невалидно
[]	Валидно	Валидно	Валидно	Валидно
["word"]	Валидно	Валидно	Валидно	Валидно
[null]	Валидно	Невалидно	Валидно	Невалидно
["word", null]	Валидно	Невалидно	Валидно	Невалидно

Теперь, разобравшись с системой типов GraphQL и свойствами списков, мы готовы перейти к изучению одной из самых мощных и захватывающих возможностей GraphQL — связей между типами.

## 8.5. МЫСЛИТЕ ГРАФИЧЕСКИ: ВЫСТРАИВАНИЕ ЗНАЧИМЫХ СВЯЗЕЙ МЕЖДУ ОБЪЕКТНЫМИ ТИПАМИ

В этом разделе объясняется, как создавать связи между объектами в GraphQL. Одним из важных преимуществ GraphQL является возможность соединять объекты. Так мы поясняем, как наши сущности связаны между собой. Как вы увидите в следующей главе, это упрощает использование GraphQL API.

### 8.5.1. Соединение типов с помощью свойств-ребер

В этом разделе объясняется, как соединять типы с помощью *свойств-ребер* (*edge properties*): свойств, указывающих на другой тип. Типы можно соединить, создав свойство, указывающее на другой тип. Как показано на рис. 8.7, свойство, которое соединяется с другим объектом, называется (по теории графов) *ребром* (*edge*). Код в листинге 8.5 показывает, как мы соединяем тип `Ingredient` с типом `Supplier`, добавляя к `Ingredient` свойство `supplier`, которое указывает на `Supplier`.

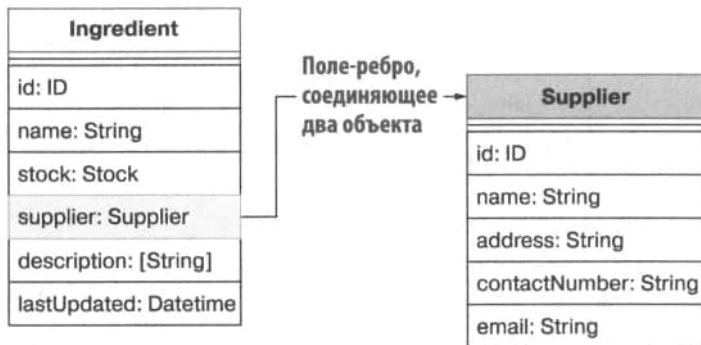
Листинг 8.5. Добавление ребра для соединения «один к одному»

```
type Ingredient {
  id: ID!
  name: String!
  supplier: Supplier!
  description: [String!]
}
```

Используем свойство-ребро для соединения типов Ingredient и Supplier

Это пример *связи «один к одному»*: свойство объекта, которое указывает ровно на один объект. Свойство в данном случае называется *ребром*, потому что соединяет тип `Ingredient` с типом `Supplier`. Это также пример *направленной связи*: как видно

на рис. 8.7, мы можем добраться до типа `Supplier` из типа `Ingredient`, но не наоборот, поэтому связь работает только в одном направлении.



**Рис. 8.7.** Чтобы связать тип `Ingredient` с типом `Supplier`, мы добавляем в `Ingredient` свойство `supplier`, которое указывает на тип `Supplier`. Поскольку свойство `supplier` создает связь между двумя типами, мы называем его ребром

Чтобы сделать связь между `Supplier` и `Ingredient` двунаправленной<sup>1</sup>, нужно добавить свойство к типу `Supplier`, которое указывает на тип `Ingredient` (листинг 8.6). Поскольку поставщик может предоставлять более одного ингредиента, свойство `ingredients` указывает на список типов `Ingredient`. Это пример *связи «один ко многим»*. На рис. 8.8 показано, как выглядит новая связь между типами `Ingredient` и `Supplier`.

#### Листинг 8.6. Двунаправленная связь между `Supplier` и `Ingredient`

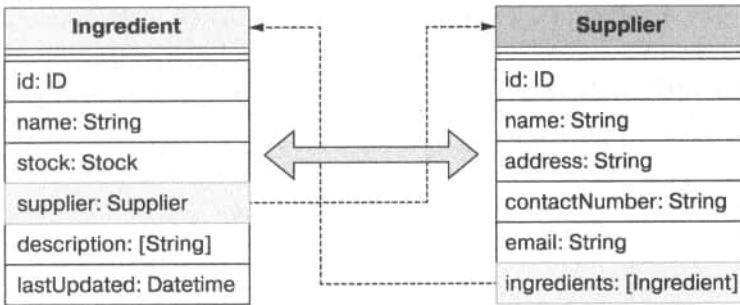
```

type Supplier {
  id: ID!
  name: String!
  address: String!
  contactNumber: String!
  email: String!
  ingredients: [Ingredient!]!
}
  
```

← Создаем двунаправленную связь между типами `Ingredient` и `Supplier`

Теперь, когда мы знаем, как создавать простые связи с помощью свойств-ребер, посмотрим, как формировать более сложные связи с помощью специализированных типов.

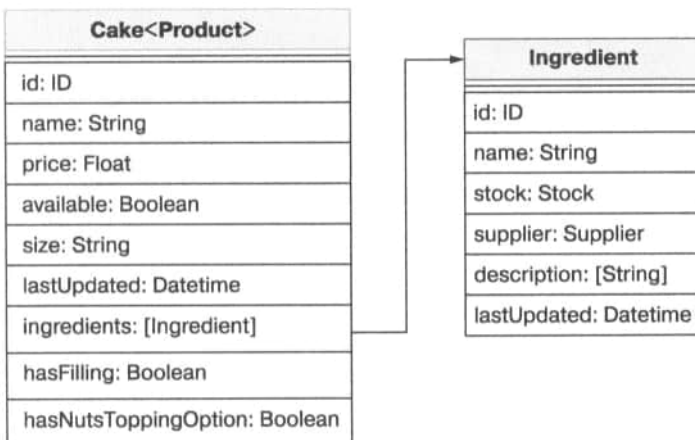
<sup>1</sup> В литературе о GraphQL часто можно встретить ремарку о том, что GraphQL вдохновлен теорией графов и можно использовать некоторые понятия из теории графов для иллюстрации отношений между типами. Следуя этой традиции, двунаправленные отношения, о которых мы здесь говорим, являются примером неориентированного графа, поскольку в тип `Supplier` можно перейти из `Ingredient` и наоборот. Хорошее обсуждение теории графов в контексте GraphQL можно найти в книге: Порселло Е., Бэнкс А. GraphQL. Язык запросов для современных веб-приложений. - СПб.: Питер, 2019. - С. 29–46.



**Рис. 8.8.** Чтобы создать двунаправленную связь между двумя типами, мы добавляем к каждому из них свойства, указывающие друг на друга. В этом примере свойство `supplier` типа `Ingredient` указывает на тип `Supplier`, в то время как свойство `ingredients` типа `Supplier` — на список ингредиентов

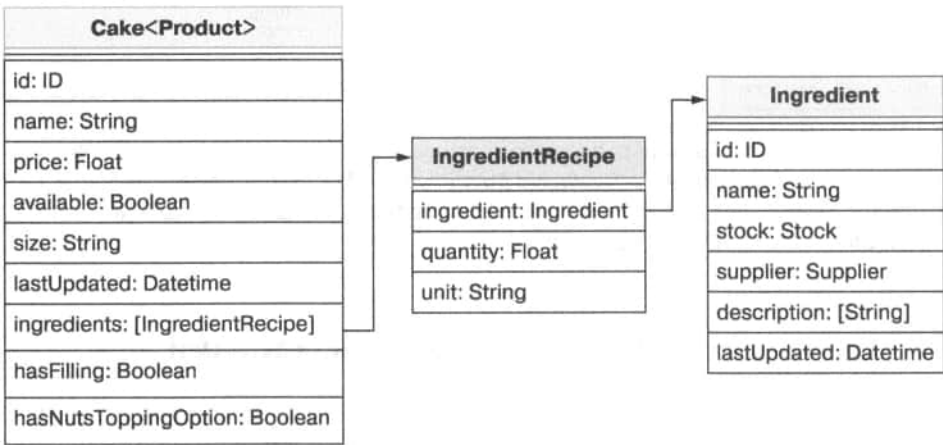
### 8.5.2. Создание соединений со сквозными типами

В этом разделе рассматриваются *сквозные типы* (*through types*): типы, которые сообщают нам, как связаны другие объектные типы. Они добавляют дополнительную информацию о самой связи. Мы будем использовать сквозные типы для связи наших товаров — выпечки и напитков — с их ингредиентами. Мы могли бы связать их, добавив простой список ингредиентов в `Cake` и `Beverage`, как показано на рис. 8.9, но это не даст нам информации о том, какое количество каждого ингредиента нужно для приготовления товара.



**Рис. 8.9.** Мы можем представить поле `ingredients` типа `Cake` в виде списка типов `Ingredient`, но это не скажет нам, какое количество каждого ингредиента понадобится для приготовления торта

Чтобы связать выпечку и напитки с их ингредиентами, мы будем использовать сквозной тип под названием `IngredientRecipe` (листинг 8.7). Как показано на рис. 8.10, `IngredientRecipe` имеет три свойства: сам ингредиент, его объем и единицу измерения объема. Это дает нам более осмысленную информацию о том, как наши товары соотносятся с ингредиентами.



**Рис. 8.10.** Чтобы показать, как `Ingredient` связан с `Cake`, мы используем тип `IngredientRecipe`, который позволяет нам подробно указать, сколько каждого ингредиента требуется для приготовления условного торта

**Листинг 8.7.** Сквозные типы, которые описывают связь между двумя типами

```
type IngredientRecipe {  
  ingredient: Ingredient!  
  quantity: Float!  
  unit: String!  
}  
  
type Cake {  
  id: ID!  
  name: String!  
  price: Float  
  available: Boolean!  
  hasFilling: Boolean!  
  hasNutsToppingOption: Boolean!  
  lastUpdated: Datetime!  
  ingredients: [IngredientRecipe!]!  
}  
  
type Beverage {  
  id: ID!  
  name: String!  
  price: Float
```

← Объявляем `IngredientRecipe`  
сквозным типом

← Объявляем ингредиенты в виде  
списка `IngredientRecipe`

```
available: Boolean!  
hasCreamOnTopOption: Boolean!  
hasServeOnIceOption: Boolean!  
lastUpdated: Datetime!  
ingredients: [IngredientRecipe!]!  
}
```

Создавая связи между различными типами объектов, мы предоставляем пользователям API возможность разобраться в наших данных, просто следуя соединительным ребрам в типах. А создавая двунаправленные связи, мы даем пользователям возможность перемещаться по нашему графу данных туда и обратно. Это одна из самых мощных функций GraphQL, и стоит потратить время на создание осмысленных связей между данными.

Чаще всего есть необходимость создавать свойства, соответствующие нескольким типам. Например, у нас может быть свойство, которое представляет либо выпечку, либо напитки. Это тема следующего раздела.

## 8.6. СОЧЕТАНИЕ РАЗЛИЧНЫХ ТИПОВ С ПОМОЩЬЮ ОБЪЕДИНЕНИЙ И ИНТЕРФЕЙСОВ

В этом разделе обсудим, как быть, когда у нас есть несколько типов одной и той же сущности. Вам часто придется иметь дело со свойствами, которые указывают на коллекцию нескольких типов. Что это означает на практике и как работает? Рассмотрим пример на основе API сервиса продукции.

В API `Cake` и `Beverage` — это два типа товаров. В подразделе 8.4.2 мы рассмотрели, как связать `Cake` и `Beverage` с типом `Ingredient`. Но как связать `Ingredient` с `Cake` и `Beverage`? Мы могли бы просто добавить для типа `Ingredient` свойство `products`, которое будет указывать на список `Cakes` и `Beverage`, например, так:

```
products: [Cake, Beverage]
```

Это рабочий вариант, но мы не сможем представлять `Cakes` и `Beverage` как единое целое. Почему это нужно делать? По следующим причинам.

- `Cake` и `Beverage` — это одно и то же: товар, поэтому имеет смысл относиться к ним как к одной и той же сущности.
- Как вы увидите в разделах 8.8 и 8.9, нам придется ссылаться на товары и в других частях кода, и будет очень полезно иметь возможность использовать для этого единственный тип.
- Если в будущем мы добавим в систему новые типы товаров, то вряд ли захотим изменять все части спецификации, где есть описание товаров. Вместо этого лучше иметь один тип, который представляет их все, и обновлять только его.

GraphQL предлагает два инструмента для объединения различных типов в один: объединения (union) и интерфейсы. Рассмотрим каждый из них подробнее.

Интерфейсы полезны, когда у нас есть типы с общими свойствами. Так обстоит дело с типами `Cake` и `Beverage`, у которых большинство свойств — общие. Интерфейсы GraphQL похожи на интерфейсы классов в таких языках программирования, как Python: они определяют набор свойств, которые должны быть реализованы другими типами. В листинге 8.8 показано, как использовать интерфейс для представления коллекции свойств, общих для `Cake` и `Beverage`. Как видите, мы объявляем типы интерфейсов с помощью ключевого слова `interface`. Типы `Cake` и `Beverage` реализуют `ProductInterface`, поэтому должны определять все свойства, прописанные в типе `ProductInterface`. Посмотрев на тип `ProductInterface`, любой пользователь нашего API может быстро понять, какие свойства доступны для `Beverage` и `Cake`.

**Листинг 8.8.** Представление общих свойств через интерфейсы

```
interface ProductInterface {
  id: ID!
  name: String!
  price: Float
  ingredients: [IngredientRecipe!]
  available: Boolean!
  lastUpdated: Datetime!
}

type Cake implements ProductInterface {
  id: ID!
  name: String!
  price: Float
  available: Boolean!
  hasFilling: Boolean!
  hasNutsToppingOption: Boolean!
  lastUpdated: Datetime!
  ingredients: [IngredientRecipe]!
}

type Beverage implements ProductInterface {
  id: ID!
  name: String!
  price: Float
  available: Boolean!
  hasCreamOnTopOption: Boolean!
  hasServeOnIceOption: Boolean!
  lastUpdated: Datetime!
  ingredients: [IngredientRecipe]!
}
```

Объявляем тип интерфейса `ProductInterface`

Тип `Cake` реализует интерфейс `ProductInterface`

Определяем свойства, специфичные для `Cake`

Тип `Beverage` реализует интерфейс `ProductInterface`



Создавая интерфейсы, мы облегчаем потребителям API понимание общих свойств, специфичных для наших видов товаров. Как вы увидите в следующей главе, интерфейсы также упрощают использование API.

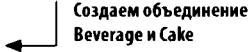
В то время как интерфейсы помогают нам определить общие свойства различных типов, объединения позволяют собрать различные типы в один тип. Это очень полезно, когда необходимо рассматривать различные типы как единое целое. В API сервиса продукции мы хотим иметь возможность рассматривать типы `Cake` и `Beverage` как один тип товаров, и объединения позволяют нам это сделать. Объединение типов задается как комбинация различных типов с использованием оператора `pipe` (`|`) (листинг 8.9).

### Листинг 8.9. Объединение различных типов

```
type Cake implements ProductInterface {
  id: ID!
  name: String!
  price: Float
  available: Boolean!
  hasFilling: Boolean!
  hasNutsToppingOption: Boolean!
  lastUpdated: Datetime!
  ingredients: [IngredientRecipe!]!
}

type Beverage implements ProductInterface {
  id: ID!
  name: String!
  price: Float
  available: Boolean!
  hasCreamOnTopOption: Boolean!
  hasServeOnIceOption: Boolean!
  lastUpdated: Datetime!
  ingredients: [IngredientRecipe!]!
}

union Product = Beverage | Cake
```



Создаем объединение  
Beverage и Cake

Использование объединений и интерфейсов делает наш API проще в обслуживании и потреблении. Если мы когда-нибудь добавим в API новый вид товара, то сможем создать для него интерфейс, аналогичный `Cake` и `Beverage`, реализовав тип `ProductInterface`. А добавив новый товар в объединение `Product`, мы обеспечим его доступность во всех операциях, использующих это объединение.

Теперь, когда мы знаем, как объединять несколько типов объектов, пришло время узнать, как ограничивать значения свойств объектных типов с помощью перечислений.

## 8.7. ОГРАНИЧЕНИЕ ЗНАЧЕНИЙ СВОЙСТВ С ПОМОЩЬЮ ПЕРЕЧИСЛЕНИЙ

В этом разделе рассматривается такой тип GraphQL, как перечисление. Технически *перечисление* — это особый вид скаляра, который может принимать только predetermined количество значений. Перечисления полезны в свойствах, которые могут принимать значение только из ограниченного списка вариантов. В GraphQL мы объявляем перечисления с помощью ключевого слова `enum`, за которым следует имя перечисления, и далее приводим его допустимые значения в фигурных скобках.

В API сервиса продукции нам нужны перечисления для выражения количества ингредиентов. Например, в подразделе 8.5.2 мы определили сквозной тип `IngredientRecipe`, который указывает объем каждого ингредиента, входящего в состав продукта. `IngredientRecipe` выражает объем в единицах измерения. Мы можем измерять ингредиенты разными способами. Например, молоко можно измерять в литрах, пинтах, унциях, галлонах и т. д. Для единообразия нужно, чтобы все использовали одни и те же единицы измерения при описании количества ингредиентов, поэтому создадим тип перечисления `MeasureUnit`, который предназначен для ограничения значений свойства `unit` (листинг 8.10).

**Листинг 8.10.** Использование типа перечисления `MeasureUnit`

```
enum MeasureUnit {
  LITERS
  KILOGRAMS
  UNITS
}
```

← Объявляем перечисление

← Перечисляем допустимые значения в пределах этого перечисления

```
type IngredientRecipe {
  ingredient: Ingredient!
  quantity: Float!
  unit: MeasureUnit!
}
```

← unit — это свойство типа MeasureUnit, значение которого не может быть равно null

Перечисление `MeasureUnit` мы также будем использовать для описания доступного запаса ингредиента. Для этого задаем тип `Stock` и используем его для определения свойства `stock` типа `Ingredient` (листинг 8.11).

Перечисления позволяют обеспечить согласованность определенных значений в интерфейсе. Это помогает избежать ошибок, возникающих в том случае, когда у пользователей есть возможность самостоятельно выбирать и записывать эти значения.

На этом мы завершаем наше путешествие по системе типов GraphQL. Типы — это строительные блоки API, но без механизма запроса или взаимодействия с ними наш API очень ограничен. Чтобы выполнять действия на сервере, нам нужно узнать о запросах и мутациях GraphQL. Этому будет посвящена остальная часть главы.

**Листинг 8.11.** Использование типа перечисления Stock

```

type Stock {
  quantity: Float!
  unit: MeasureUnit!
}

type Ingredient {
  id: ID!
  name: String!
  stock: Stock
  products: [Product!]!
  supplier: Supplier!
  description: [String!]
}

```

Указываем тип товара, чтобы помочь себе получить информацию о доступных запасах ингредиента

Свойство unit типа Stock — это перечисление

Связываем тип Ingredient с типом Stock через свойство запаса ингредиента

## 8.8. ОПРЕДЕЛЕНИЕ ЗАПРОСОВ ДЛЯ ПОЛУЧЕНИЯ ДАННЫХ ИЗ API

В этом разделе представлены *запросы* GraphQL: операции, которые позволяют нам получать или читать данные с сервера. Предоставление данных — одна из важнейших функций любого веб-API, и GraphQL предлагает большую гибкость в создании высокопроизводительного интерфейса запросов. Запросы относятся к группе операций чтения, которые мы обсуждали в разделе 8.2. Напомню, какие операции запросов должен поддерживать API сервиса продукции:

- `allProducts()`;
- `allIngredients()`;
- `products()`;
- `product()`;
- `ingredient()`.

Сначала мы поработаем с запросом `allProducts()`, поскольку он самый простой, а затем перейдем к запросу `products()`. В процессе работы над `products()` вы увидите, как добавлять аргументы к определению запроса, узнаете о пагинации и, наконец, научитесь преобразовывать параметры запроса в собственный тип для улучшения удобочитаемости и обслуживания.

Спецификация запроса GraphQL похожа на определение сигнатуры функции Python: мы задаем имя запроса, по желанию определяем в круглых скобках список параметров и после двоеточия указываем тип возвращаемого значения. Код в листинге 8.12 показывает самый простой запрос в API сервиса продукции: `allProducts()`, который не принимает никаких параметров и просто возвращает список всех товаров.

**Листинг 8.12.** Простой запрос GraphQL для возврата списка товаров

```

type Query {
  allProducts: [Products!]!
}

```

`allProducts()` возвращает список всех товаров, имеющих в базе данных CoffeeMesh. Такой запрос полезен, если мы хотим провести исчерпывающий анализ всех товаров, но в реальной жизни пользователям API понадобится возможность фильтровать результаты. Они могут сделать это с помощью запроса `products()`, который в соответствии с требованиями, собранными в разделе 8.2, возвращает отфильтрованный список товаров.

Аргументы запроса определяются в круглых скобках, как и параметры функции Python. В листинге 8.13 показано, как определяется запрос `products()`. Он включает аргументы, которые позволяют потребителям API фильтровать товары по наличию или максимальной/минимальной цене. Все аргументы необязательные. Потребители API могут использовать как любой из аргументов запроса, так и все или ни одного. Если при использовании запроса `products()` они не укажут ни одного аргумента, то получат список всех товаров.

**Листинг 8.13.** Простой запрос GraphQL для возврата списка товаров

```

type Query {
  products(available: Boolean, maxPrice: Float, minPrice: Float):
    [Product!]
}

```

В дополнение к фильтрации списка товаров потребители API, вероятно, захотят иметь возможность сортировать список и получать результаты постранично. Пагинация — это возможность предоставлять результаты запроса в виде различных наборов заданного размера, и она широко используется в API, чтобы клиенты получали удобный объем данных при каждом запросе. Как показано на рис. 8.11, если в результате запроса выдаются десять или более записей, мы можем разделить его на группы по пять элементов и показывать по одной группе за раз. Каждая такая группа элементов называется *страницей* (*page*).

Итак, включаем разбиение на страницы, добавляя в запрос аргумент `resultsPerPage`, а также аргумент `page`. Чтобы отсортировать набор результатов, добавляем аргумент `sort`. В следующем фрагменте жирным шрифтом выделены изменения в запросе `products()` после добавления этих аргументов:

```

type Query {
  products(available: Boolean, maxPrice: Float, minPrice: Float,
    sort: String, resultsPerPage: Int, page: Int): [Product!]!
}

```



**Рис. 8.11.** Более распространенный подход к разбиению на страницы заключается в том, чтобы позволить пользователям решать, сколько результатов на странице они хотят видеть, и дать им возможность самим выбрать размер страницы

Возможность использовать множество аргументов запроса дает потребителям API большую гибкость, но задавать значения для них всех может быть обременительно. Сделаем наш API проще в использовании, задав для некоторых аргументов значения по умолчанию. Мы установим порядок сортировки по умолчанию, а также значение по умолчанию для аргумента `resultsPerPage` и значение по умолчанию для аргумента `page`. В листинге 8.14 показано, как присвоить значения по умолчанию некоторым аргументам в запросе `products()` и включить перечисление `SortingOrder`, которое ограничивает значения аргумента `sort` вариантами сортировки либо по возрастанию — `ASCENDING`, либо по убыванию — `DESCENDING`.

**Листинг 8.14.** Установка значений по умолчанию для аргументов запроса

```
enum SortingOrder {
  ASCENDING
  DESCENDING
}

type Query {
  products(
    maxPrice: Float
    minPrice: Float
    available: Boolean = true
    sort: SortingOrder = DESCENDING
    resultsPerPage: Int = 10
    page: Int = 1
  ): [Product!]!
}
```

Задаем перечисление `SortingOrder`

Определяем значения по умолчанию для некоторых параметров

Ограничиваем значения параметра `sort`, устанавливая для него тип перечисления `SortingOrder`

Сигнатура запроса `products()` становится громоздкой. Если мы продолжим добавлять в нее аргументы, ее станет трудно читать и поддерживать. Чтобы улучшить читаемость, можем преобразовать аргументы из спецификации запроса в собственный тип (листинг 8.15). В GraphQL допускается определять списки параметров, используя типы входных данных (`input types`), которые имеют такой же стиль, как и любой другой объектный тип GraphQL, но предназначены для использования в качестве входных данных запросов и мутаций.

**Листинг 8.15.** Преобразование аргументов запроса в типы входных данных

```

input ProductsFilter {
  maxPrice: Float
  minPrice: Float
  available: Boolean = true,
  sort: SortingOrder = DESCENDING
  resultsPerPage: Int = 10
  page: Int = 1
}

type Query {
  products(input: ProductsFilter): [Product!]!
}

```

Объявляем тип входных данных ProductsFilter

Определяем параметры ProductsFilter

Некоторым параметрам присваиваем значения по умолчанию

Устанавливаем тип входного параметра как ProductsFilter

Остальные API-запросы, а именно `allIngredients()`, `product()` и `ingredient()`, выделены в листинге 8.16 жирным шрифтом. `allIngredients()` возвращает полный список ингредиентов и поэтому не принимает никаких аргументов, как и запрос `allProducts()`. Наконец, `product()` и `ingredient()` возвращают один товар или ингредиент по ID и поэтому имеют обязательный аргумент `id`. Если по указанному ID найден товар или ингредиент, в результате запроса будет выведена подробная информация о запрашиваемом товаре; в противном случае мы получим `null` (листинг 8.16).

**Листинг 8.16.** Спецификация для всех запросов в API сервиса продукции

```

type Query {
  allProducts: [Product!]!
  allIngredients: [Ingredient!]!
  products(input: ProductsFilter!): [Product!]!
  product(id: ID!): Product
  ingredient(id: ID!): Ingredient
}

```

**product()** возвращает обнуляемый результат типа Product

Теперь, когда вы знаете, как определять запросы, пришло время поговорить о мутациях.

## 8.9. ИЗМЕНЕНИЕ СОСТОЯНИЯ СЕРВЕРА С ПОМОЩЬЮ МУТАЦИЙ

В этом разделе представлены *мутации* GraphQL — операции, которые позволяют нам вызывать действия, изменяющие состояние сервера. Хотя цель запроса — позволить получить данные с сервера, мутации дают возможность создавать новые ресурсы, удалять их или изменять их состояние. Мутации имеют возвращаемое значение, которое может быть скаляром, например булевым числом, или объектом. Это позволяет потребителям API проверить, что операция успешно завершилась, и получить любые значения, созданные сервером, например идентификаторы.

В разделе 8.2 мы говорили о том, что API сервиса продукции должен поддерживать следующие операции для добавления, удаления и обновления ресурсов на сервере:

- `addIngredient();`
- `updateStock();`
- `addProduct();`
- `updateProduct();`
- `deleteProduct();`

В этом разделе мы пропишем мутации `addProduct()`, `updateProduct()` и `deleteProduct()`. Спецификации для других мутаций будут схожими, и вы можете ознакомиться с ними в репозитории GitHub, прилагаемом к книге.

Мутация GraphQL похожа на сигнатуру функции в Python: мы определяем имя мутации, описываем в круглых скобках ее параметры и после двоеточия указываем возвращаемый тип. В листинге 8.17 показана спецификация для мутации `addProduct()`. Функция `addProduct()` принимает длинный список аргументов и возвращает тип `Product`. Все аргументы необязательны, кроме имени и типа. Мы используем тип, чтобы указать, какой вид товара создаем: выпечку или напиток. Мы также добавляем перечисление `ProductType`, чтобы ограничить значения аргумента `type` либо выпечкой, либо напитком. Поскольку эта мутация предназначена для создания тортов и напитков, мы разрешаем пользователям указывать свойства каждого типа, а именно `hasFilling` и `hasNutsToppingOption` для выпечки, а также `hasCreamOnTopOption` и `hasServeOnIceOption` для напитков, но по умолчанию устанавливаем для них значение `false`, чтобы упростить использование мутации.

#### Листинг 8.17. Определение мутации GraphQL

```
enum ProductType {  
  cake  
  beverage  
}  
  
input IngredientRecipeInput {  
  ingredient: ID!  
  quantity: Float!  
  unit: MeasureUnit!  
}  
  
enum Sizes {  
  SMALL  
  MEDIUM  
  BIG  
}  
  
type Mutation {  
  addProduct(  
    ← Объявляем мутации в соответствии  
      со объектным типом Mutation  
    ← Объявляем перечисление  
      ProductType
```

```

    name: String!
    type: ProductType!
    price: String
    size: Sizes
    ingredients: [IngredientRecipeInput!]!
    hasFilling: Boolean = false
    hasNutsToppingOption: Boolean = false
    hasCreamOnTopOption: Boolean = false
    hasServeOnIceOption: Boolean = false
  ): Product!
}

```

← Указываем тип возвращаемого значения функции `addProduct()`

Согласитесь, определение сигнатуры мутации `addProduct()` выглядит несколько громоздким. Мы можем улучшить читаемость и удобство сопровождения, преобразовав список параметров в отдельный тип. В листинге 8.18 показан рефакторинг мутации `addProduct()`: мы перемещаем список параметров в тип `input`. Новый тип `AddProductInput` содержит все необязательные параметры, которые могут быть заданы при создании нового товара. Мы пока не приводим здесь параметр `name`, который является единственным обязательным параметром при создании нового товара. Как вы вскоре увидите, это позволит нам повторно использовать тип входных данных `AddProductInput` в других мутациях, которые не требуют параметра `name`.

#### Листинг 8.18. Рефакторинг параметров с использованием типов входных данных

```

input AddProductInput {
  price: String
  size: Sizes
  ingredients: [IngredientRecipeInput!]!
  hasFilling: Boolean = false
  hasNutsToppingOption: Boolean = false
  hasCreamOnTopOption: Boolean = false
  hasServeOnIceOption: Boolean = false
}

type Mutation {
  addProduct(
    name: String!
    type: ProductType!
    input: AddProductInput!
  ): Product!
}

```

← Объявляем тип ввода `AddProductInput`

← Перечисляем параметры `AddProductInput`

← Некоторым параметрам мы присваиваем значения по умолчанию

← Входной параметр `addProduct()` имеет тип входных данных `addProduct`

Типы входных данных не только помогают нам сделать спецификацию более читабельной и удобной для сопровождения, но и позволяют создавать повторно используемые типы. Так, мы можем повторно использовать тип `AddProductInput` в сигнатуре мутации `updateProduct()`. При обновлении конфигурации товара может понадобиться изменить только некоторые его параметры, например название,



цену или ингредиенты. В следующем фрагменте показано, как повторно использовать параметры `AddProductInput` в мутации `updateProduct()`. В дополнение к `AddProductInput` мы также включаем обязательный параметр `id`, требуемый для идентификации товара, который мы хотим обновить. Кроме того, добавляем параметр `name`, который в данном случае является необязательным:

```
type Mutation {  
  updateProduct(id: ID!, input: AddProductInput!): Product!  
}
```

Теперь рассмотрим мутацию `deleteProduct()`, которая удаляет товар из каталога. Для этого пользователь должен указать ID товара, который он хочет удалить. Если операция прошла успешно, мутация возвращает `true`, в противном случае — `false`. В следующем фрагменте показана спецификация мутации `deleteProduct()`:

```
deleteProduct(id: ID!): Boolean!
```

На этом мы завершаем наше путешествие по SDL GraphQL. Теперь у вас есть все необходимое для определения собственных схем API. В главе 9 вы узнаете, как запустить mock-сервер, используя спецификацию API сервиса продукции, как пользоваться GraphQL API и взаимодействовать с ним.

## РЕЗЮМЕ

- GraphQL — это популярный протокол для создания веб-API. Он отлично подходит, когда важно предоставить клиентам API полный контроль над данными, которые они хотят получить, а также в тех ситуациях, когда есть сильно взаимосвязанные данные.
- Спецификация GraphQL API называется схемой и состоит из использования языка определения схем (SDL).
- Мы используем скалярные типы GraphQL для определения свойств объектного типа: логические значения, строки, числа с плавающей точкой, целые числа и идентификаторы. Кроме того, мы можем создавать собственные скалярные типы.
- Объектные типы GraphQL — это коллекции свойств, и они обычно представляют собой ресурсы или сущности, управляемые сервером API.
- Мы можем соединять объекты с помощью свойств-ребер, то есть свойств, указывающих на другой объект, а также с помощью сквозных типов. Сквозные типы — это типы объектов, которые добавляют дополнительную информацию о том, как связаны два объекта.
- Чтобы ограничить значения свойства, мы используем типы перечислений.

- Запросы GraphQL — это операции, которые позволяют клиентам API получать данные с сервера.
- Мутации GraphQL — это операции, которые позволяют клиентам API инициализировать действия, изменяющие состояние сервера.
- Если запросы и мутации имеют длинные списки параметров, можно реорганизовать их в типы входных данных, чтобы улучшить читаемость и удобство сопровождения. Типы входных данных также можно повторно использовать в нескольких запросах или мутациях.

# Использование GraphQL API

---

## В этой главе

- ✓ Запуск mock-сервера GraphQL для тестирования дизайна API.
- ✓ Применение клиента GraphQL для исследования и использования GraphQL API.
- ✓ Выполнение запросов и мутаций к GraphQL API.
- ✓ Программное использование GraphQL API с помощью cURL и Python.

Как вы узнали в главе 8, GraphQL предлагает язык запросов для веб-API, и в этой главе вы научитесь пользоваться им для выполнения запросов на сервере. В частности, вы узнаете, как делать запросы к GraphQL API, а также научитесь исследовать его, чтобы выявить его доступные типы, запросы и мутации. Освоение GraphQL невозможно без понимания работы GraphQL API на стороне клиента.

Изучая взаимодействие с GraphQL API, вы научитесь использовать API, предоставляемые другими поставщиками, что позволит вам проводить тесты собственных API, а также поможет разрабатывать более совершенные прикладные интерфейсы. Вы разберетесь, как использовать клиент GraphQL для изучения и визуализации API. Как вы увидите, GraphQL предлагает интерактивную панель запросов, которая упрощает выполнение запросов на сервере.

Чтобы проиллюстрировать концепции и идеи, лежащие в основе языка запросов GraphQL, рассмотрим примеры с использованием API сервиса продукции, с которым мы работали в главе 8. Поскольку мы еще не реализовали спецификацию

для API сервиса продукции, попробуем запустить mock-сервер — это важная часть процесса разработки API, упрощающая тестирование и валидацию дизайна. Наконец, вы научитесь выполнять запросы к GraphQL API программно, используя такие инструменты, как cURL и Python.

## 9.1. ЗАПУСК МОСК-СЕРВЕРА GRAPHQL

В этом разделе я объясню, как запустить mock-сервер GraphQL для изучения и тестирования API. Mock-сервер — это имитационный сервер, который эмулирует поведение реального сервера, предлагая те же эндпоинты и возможности, но используя искусственные данные. Например, mock-сервер для API сервиса продукции имитирует реализацию API и предлагает тот же интерфейс, который мы разработали в главе 8.

### ОПРЕДЕЛЕНИЕ

Mock-серверы — это поддельные серверы, которые имитируют поведение реального сервера. Обычно используются для разработки клиентов API во время реализации серверной части. Вы можете запустить mock-сервер, используя спецификацию для API. Mock-серверы возвращают ненастоящие данные и, как правило, не сохраняют их.

Mock-серверы играют важную роль в разработке веб-API, поскольку позволяют потребителям API начать работу с клиентским кодом, пока мы работаем над реализацией бэкенда. В этом разделе мы запустим mock-сервер для API сервиса продукции. Единственное, что нам понадобится для этого, — спецификация API, которую мы разработали в главе 8. Вы найдете ее в разделе `ch08/schema.graphql` в репозитории GitHub для этой книги.

Можете выбрать одну из множества различных библиотек для запуска mock-сервера GraphQL. В этой главе мы будем использовать GraphQL Faker (<https://github.com/APIs-guru/graphql-faker>) — один из самых популярных имитационных инструментов GraphQL. Чтобы установить его, выполните следующую команду:

```
$ npm install graphql-faker
```

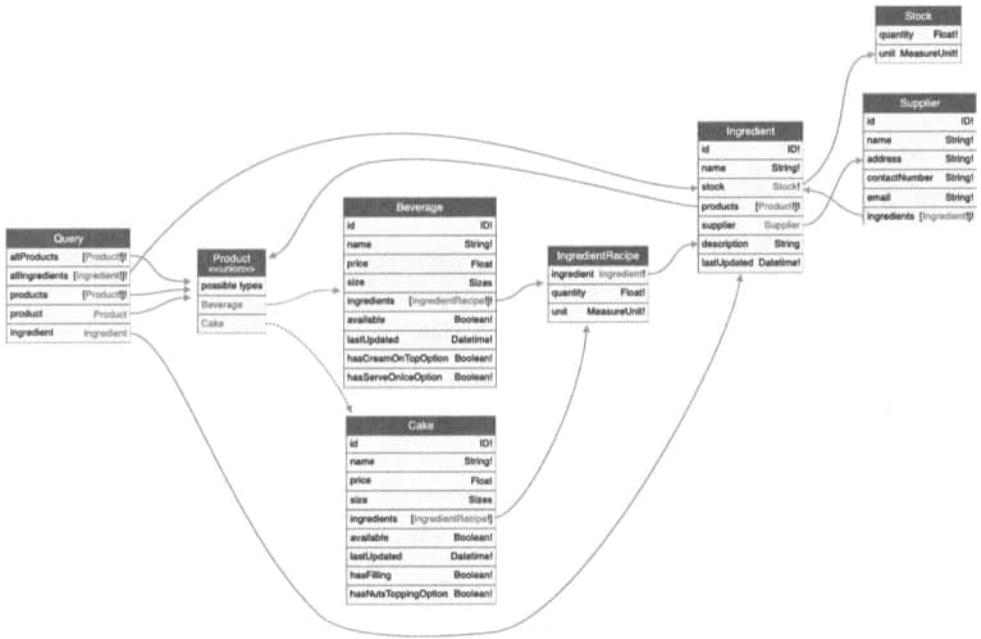
Будут созданы файл `package-lock.json` в вашем текущем каталоге, а также папка `node_modules`. Файл `package-lock.json` содержит информацию о зависимостях, установленных вместе с `graphql-faker`, а `node_modules` — это каталог, в котором эти зависимости установлены. Для запуска mock-сервера выполните следующую команду:

```
$ ./node_modules/.bin/graphql-faker schema.graphql
```

GraphQL Faker обычно работает на порте 9002 и предоставляет доступ к трем эндпоинтам:

- `/editor` — интерактивный редактор, в котором вы можете разрабатывать свой GraphQL API;

- `/graphql` — интерфейс GraphQL для вашего GraphQL API. Это интерфейс, который мы будем использовать для изучения API и выполнения запросов;
- `/voyager` — диалоговое отображение API, которое помогает понять связи и зависимости между типами (рис. 9.1).



**Рис. 9.1.** Пользовательский интерфейс Voyager для API сервиса продукции, который показывает отношения между объектными типами. Следуя по соединительным стрелкам, вы можете увидеть, к каким объектам можно получить доступ в результате каждого запроса

Чтобы начать изучение и тестирование API сервиса продукции, перейдите по следующему адресу в браузере: <http://localhost:9002/graphql> (если вы запускаете GraphQL Faker на другом порте, ваш URL будет другим). Этот эндпоинт загружает интерфейс GraphQL для API сервиса продукции. На рис. 9.2 показано, как выглядит этот интерфейс, и выделены наиболее важные элементы.

Чтобы ознакомиться с запросами и мутациями, предоставляемыми API, нажмите кнопку Docs в правом верхнем углу окна. Появится боковая навигационная панель, предлагающая два варианта: запросы или мутации (рис. 9.3). Если вы выберете запросы, то увидите список запросов, открытых сервером, с их типами возвращаемых данных. Вы можете щелкать на возвращаемых типах, чтобы изучить их свойства, как показано на рис. 9.3. А в следующем разделе мы начнем тестировать GraphQL API.

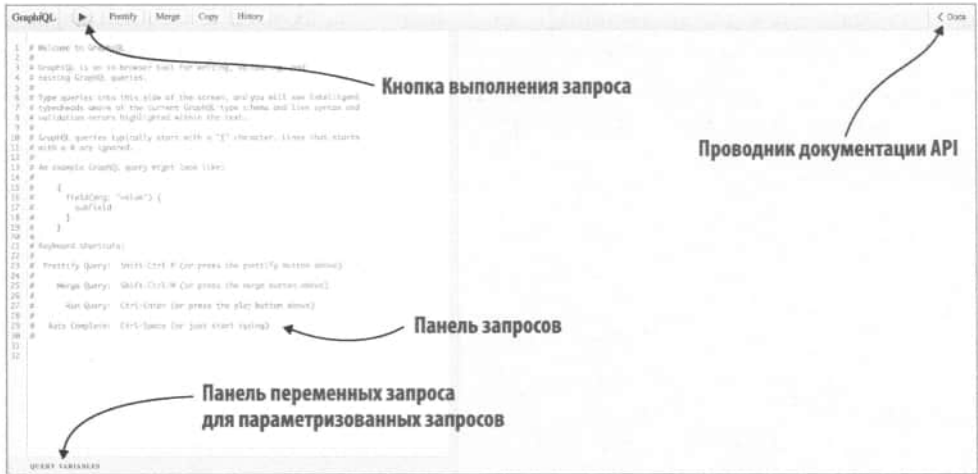


Рис. 9.2. Окно проводника документации API и панели запросов в GraphQL

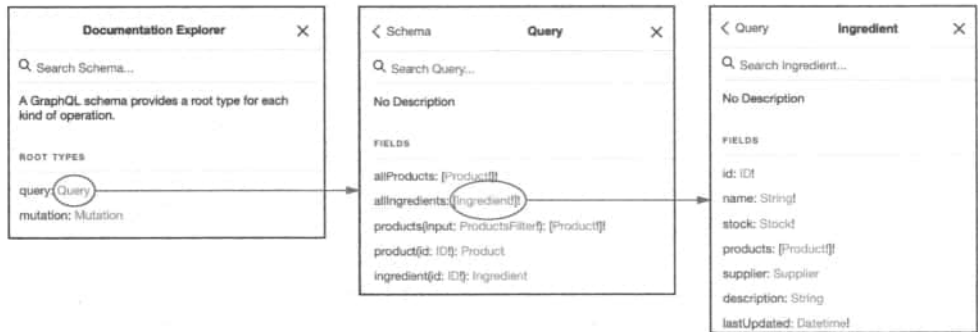


Рис. 9.3. Щелкнув на Documentation Explorer в GraphQL, вы можете просмотреть все запросы и мутации, доступные в API, а также типы, которые они возвращают, и их свойства

## 9.2. ЗАПРОСЫ GRAPHQL

В этом разделе вы научитесь использовать GraphQL API, выполняя запросы с помощью GraphQL.

### 9.2.1. Выполнение простых запросов

Сначала разберем простые запросы, которые не принимают никаких параметров. API сервиса продукции предлагает два таких запроса: `allProducts()`, который возвращает список всех товаров, предлагаемых CoffeeMesh, и `allIngredients()`, возвращающий список всех ингредиентов.

Мы будем использовать GraphQL для выполнения запросов к API. Для выполнения запроса перейдите на панель редактора запросов в окне GraphQL (см. рис. 9.2). В листинге 9.1 показано, как выполняется запрос `allIngredients()`. Как видите, для запуска запроса мы должны использовать его имя, за которым следуют фигурные скобки. В них мы объявляем набор свойств, которые хотим получить от сервера. Блок в фигурных скобках называется *выборкой* (selection set). Запросы GraphQL всегда должны содержать выборку. Если вы не добавите ее, то получите от сервера сообщение об ошибке. Здесь мы выбираем только название каждого ингредиента. Текст, представляющий запрос, называется *документом запроса*.

**Листинг 9.1.** Документ запроса, выполняющий запрос `allIngredients()`

```
{  
  allIngredients {  
    name  
  }  
}
```

← Закладываем запросы в фигурные скобки

← Выполняем запрос `allIngredients()`

← Запрашиваем свойство `name`

Ответ на успешный запрос GraphQL API представляет собой JSON-документ с полем `"data"`, в котором содержится результат запроса (листинг 9.2). Неудачный запрос приводит к получению документа JSON с ключом `"error"`. Поскольку мы запускаем mock-сервер, API возвращает случайные значения.

**Листинг 9.2.** Пример успешного ответа на запрос `allIngredients()`

```
{  
  "data": {  
    "allIngredients": [  
      {  
        "name": "string"  
      },  
      {  
        "name": "string"  
      }  
    ]  
  }  
}
```

← Успешный ответ включает в себя ключ `data`

← Результат запроса проиндексирован по ключу, названному по самому запросу

Теперь, понимая основы запросов GraphQL, добавим в них параметры.

### 9.2.2. Выполнение запросов с параметрами

`allIngredients()` — это простой запрос, который не принимает никаких параметров. Теперь посмотрим, как можно выполнить запрос с параметром. Одним из примеров такого запроса является `ingredient()`, который требует параметр `id`. В листинге 9.3 показано, как мы можем вызвать запрос `ingredient()` со случайным идентификатором. Как вы можете видеть, мы включаем параметры запроса в виде пар (ключ: значение), разделенных двоеточием и взятых в скобки.

**Листинг 9.3.** Выполнение запроса с требуемым параметром

```

{
  ingredient(id: "asdf") {
    name
  }
}

```

← Вызываем ingredient() с параметром ID, равным asdf

Далее рассмотрим, с какими проблемами можно столкнуться при выполнении запросов и как с ними справиться.

**9.2.3. Ошибки запросов**

В этом разделе объясняются некоторые из наиболее распространенных ошибок, встречающихся при выполнении запросов GraphQL, и рассказывается, как их читать и интерпретировать.

Если при выполнении запроса `ingredient()` вы опустите требуемый параметр, то получите сообщение об ошибке от API. Ответы об ошибках содержат ключ ошибки, указывающий на список всех ошибок, обнаруженных сервером. Каждая ошибка представляет собой объект со следующими ключами:

- **message** — включает понятное человеку описание ошибки;
- **locations** — указывает, где именно в запросе была обнаружена ошибка, включая строку и столбец.

В листинге 9.4 показано, что происходит, когда вы запускаете запрос с пустыми круглыми скобками. Как видите, мы получаем синтаксическую ошибку с немного загадочным сообщением: `Expected Name, found )` (Ожидается имя, найдено `)`). Это обычный ответ в случае синтаксической ошибки в GraphQL. В данном случае он означает, что GraphQL ожидал получить после открывающей скобки параметр, но вместо этого нашел закрывающую скобку.

**Листинг 9.4.** Ошибка, касающаяся отсутствующих параметров запроса

```

# Запрос
{
  ingredient() {
    name
  }
}

# Ошибка
{
  "errors": [
    {
      "message": "Syntax Error: Expected Name, found )",

```

← Запускаем запрос ingredient() без обязательного идентификатора параметра

← Неудачный ответ содержит ключ errors

← Получаем общую синтаксическую ошибку



```

"locations": [
  {
    "line": 2,
    "column": 14
  }
]
}

```

← Точное местоположение ошибки в нашем запросе

← Ошибка была обнаружена во второй строке документа запроса

← Ошибка была обнаружена на 14-м символе второй строки

С другой стороны, если вы запустите запрос `ingredient()` вообще без круглых скобок, как показано в листинге 9.5, то получите ошибку, указывающую на отсутствие необходимого параметра `id`.

### ИСПОЛЬЗОВАНИЕ СКОБОК В ЗАПРОСАХ И МУТАЦИЯХ GRAPHQL

В GraphQL параметры запроса приводятся в круглых скобках. Если вы выполняете запрос с обязательными параметрами, такими как `ingredient`, то должны перечислить их в круглых скобках (см. листинг 9.3). В противном случае будет выдана ошибка (см. листинги 9.4 и 9.5). Если вы выполняете запрос без параметров, то круглые скобки можно опустить. Например, при выполнении запроса `allIngredients()` мы опускаем круглые скобки (см. листинг 9.1), поскольку этот запрос не требует никаких параметров.

#### Листинг 9.5. Ошибка, касающаяся отсутствующих параметров запроса

```

# Запрос
{
  ingredient {
    name
  }
}

# Ошибка
{
  "errors": [
    {
      "message": "Field \"ingredient\" argument \"id\" of type \"ID!\" is
➔ required, but it was not provided.",
      "locations": [
        {
          "line": 2,
          "column": 3
        }
      ]
    }
  ]
}

```

← Запускаем запрос `ingredient()` без круглых скобок

← В сообщении об ошибке говорится, что в запросе отсутствует параметр `id`

← Ошибка была обнаружена во второй строке нашего документа запроса

← Ошибка была обнаружена в третьем символе второй строки

Теперь, зная, как читать и интерпретировать сообщения об ошибках, возникающих в наших запросах, изучим запросы, которые возвращают несколько типов.

## 9.3. ИСПОЛЬЗОВАНИЕ ФРАГМЕНТОВ В ЗАПРОСАХ

В этом разделе объясняется, как выполнять запросы, возвращающие несколько типов. Запросы, которые мы рассматривали до сих пор, просты, поскольку возвращают только один тип — `Ingredient`. Однако наши запросы в отношении товаров, такие как `allProducts()` и `product()`, возвращают тип `Product`, который скомбинирован из типов `Cake` и `Beverage`. Как нам выполнить запросы в этом случае?

Когда запрос GraphQL возвращает несколько типов, необходимо создать выборки для каждого типа. Например, если вы выполните запрос `allProducts()` с одним вариантом, то получите сообщение об ошибке, что сервер не знает, как разделить свойства в выборке.

**Листинг 9.6.** Вызов `allProducts()` с помощью одной выборки вариантов

```
# Запрос
{
  allProducts {
    name
  }
}
```

Запускаем запрос `allProducts()` без параметров

Включаем свойство `name` в выборку

```
# Сообщение об ошибке
{
  "errors": [
    {
      "message": "Cannot query field \"name\" on type \"Product\". Did you
        mean to use an inline fragment on \"ProductInterface\", \"Beverage\",
        or \"Cake\"?",
      "locations": [
        {
          "line": 3,
          "column": 5
        }
      ]
    }
  ]
}
```

Получаем ответ об ошибке

Сервер не знает, как разрешить свойства в выборке

Ошибка была обнаружена в третьей строке документа запроса

Ошибка была обнаружена на пятом символе третьей строки

В сообщении об ошибке в листинге 9.6 спрашивается, собирались ли вы использовать встроенный фрагмент в `ProductInterface`, `Beverage` или `Cake`. Что такое *встроенный фрагмент*? Это анонимная выборка определенного типа. В коде встроенные фрагменты обозначаются тремя точками (оператор `spread` в JavaScript), за которыми следует ключевое слово `on` и тип, к которому применяется выборка, а также перечень свойств в фигурных скобках:

```
...on ProductInterface {
  name
}
```

В листинге 9.7 приведен измененный запрос `allProducts()`: добавлены встроенные фрагменты, которые выбирают свойства типов `ProductInterface`, `Cake` и `Beverage`. Тип, возвращаемый `allProducts()`, — `Product`, он является объединением `Cake` и `Beverage`, поэтому мы можем выбирать свойства обоих типов. Из спецификации мы также знаем, что `Cake` и `Beverage` реализуют тип интерфейса `ProductInterface`, поэтому можем выбирать свойства, общие для `Cake` и `Beverage`, прямо в интерфейсе.

**Листинг 9.7.** Добавление встроенных фрагментов для каждого возвращаемого типа

```
{
  allProducts {
    ...on ProductInterface {
      name
    }
    ...on Cake {
      hasFilling
    }
    ...on Beverage {
      hasCreamOnTopOption
    }
  }
}
```

Встроенный фрагмент с выборкой для типа `ProductInterface`

Встроенный фрагмент с выборкой для типа `Cake`

Встроенный фрагмент с выборкой для типа `Beverage`

В листинге 9.7 применяются встроенные фрагменты, но главное достоинство фрагментов состоит в том, что мы можем определять их как независимые переменные. Это позволит многократно их использовать, а также сделать наши запросы более удобочитаемыми. В листинге 9.8 показано, как можно рефакторить листинг 9.7, чтобы использовать фрагменты независимо. Запросы стали намного чище! В реальной жизни вам, скорее всего, придется работать с большими выборками, поэтому организация фрагментов в отдельные, подходящие для повторного использования блоки кода сделает ваши запросы более читабельными.

**Листинг 9.8.** Использование автономных фрагментов

```
{
  allProducts {
    ...commonProperties
    ...cakeProperties
    ...beverageProperties
  }
}

fragment commonProperties on ProductInterface {
  name
}

fragment cakeProperties on Cake {
  hasFilling
}

fragment beverageProperties on Beverage {
  hasCreamOnTopOption
}
```

Итак, надеюсь, вы поняли, как работать с запросами, возвращающими несколько объектных типов. В следующем разделе рассмотрим, как выполнять запросы с входными параметрами.

## 9.4. ВЫПОЛНЕНИЕ ЗАПРОСОВ С ВХОДНЫМИ ПАРАМЕТРАМИ

В разделе 8.8 вы узнали, что типы входных данных похожи на объектные типы, но предназначены для использования в качестве параметров запроса или мутации GraphQL. Одним из примеров типа входных данных в API сервиса продукции является `ProductsFilter`, который позволяет нам фильтровать товары по таким параметрам, как наличие, минимальная или максимальная цена и др. `ProductsFilter` представляет собой параметр запроса `products()`. Как мы вызываем запрос `products()`?

Когда запрос принимает параметры в виде типа входных данных, они должны быть переданы в виде объектов ввода. Это может показаться сложным, но на самом деле все очень просто. Вызываем запрос `products()`, используя параметр `maxPrice` из `ProductsFilter`. Чтобы использовать любой из параметров входного типа, мы просто заключаем их в фигурные скобки (листинг 9.9).

**Листинг 9.9.** Вызов запроса с требуемым параметром

```
{
  products(input: {maxPrice: 10}) {
    ...on ProductInterface {
      name
    }
  }
}
```

← Задаем параметр `maxPrice` для `ProductFilter`

← Встроенный фрагмент для типа `ProductInterface`

Далее мы более подробно разберем отношения между объектами, определенными в спецификации API, и посмотрим, как строить запросы, позволяющие перемещаться по графу данных.

## 9.5. НАВИГАЦИЯ ПО ГРАФУ API

Здесь мы разберемся, как выбирать свойства из нескольких типов, используя связи. В разделе 8.5 вы научились создавать связи между объектными типами с помощью свойств-ребер и сквозных типов. Эти связи позволяют клиентам API перемещаться по графу взаимосвязей между ресурсами, управляемыми API. Например, в API сервиса продукции типы `Cake` и `Beverage` связаны с типом `Ingredient` с помощью сквозного типа `IngredientRecipe`. Используя эту связь, мы можем выполнять запросы, которые получают информацию об ингредиентах каждого товара. В этом разделе мы рассмотрим, как создавать такие запросы.

Всякий раз, когда мы добавляем вариант выбора (селектор) для свойства, указывающего на другой объектный тип, в запросах мы должны включать вложенную выборку для этого типа. Например, если мы добавляем селектор для свойства `ingredient` в отношении типа `ProductInterface`, то должны включить выборку с любым из свойств `IngredientRecipe`, вложенных в `ingredients`. Мы включаем вложенную выборку для свойства `ingredients` типа `ProductInterface` в запрос `allProducts()`. Запрос выбирает название товара, а также название каждого ингредиента, входящего в его состав (листинг 9.10).

**Листинг 9.10.** Запрос к вложенным объектным типам

```
{
  allProducts {
    ...on ProductInterface {
      name,
      ingredients {
        ingredient {
          name
        }
      }
    }
  }
}
```

Встроенный фрагмент для типа `ProductInterface`

Селектор для свойства `ingredients` типа `ProductInterface`

Селектор для свойства `ingredient` типа `IngredientRecipe`

В листинге 9.10 по связи между типами `ProductInterface` и `Ingredient` мы получаем информацию из обоих типов в одном запросе, но можем пойти еще дальше. Тип `Ingredient` содержит свойство `supplier`, которое указывает на тип `Supplier`. Допустим, мы хотим получить список товаров, включая их названия и ингредиенты, а также имя поставщика каждого ингредиента. (Рекомендую вам перейти к пользовательскому интерфейсу Voyager, сгенерированному `graphql-faker`, чтобы визуализировать взаимосвязи, отраженные в этом запросе (см. рис. 9.1).)

**Листинг 9.11.** Обход графа API сервиса продукции с помощью связей между типами

```
{
  allProducts {
    ...on ProductInterface {
      name
      ingredients {
        ingredient {
          name
          supplier {
            name
          }
        }
      }
    }
  }
}
```

Встроенный фрагмент для типа `ProductInterface`

Селектор для свойства `ingredients` типа `ProductInterface`

Селектор для свойства `ingredient` типа `IngredientRecipe`

Селектор для свойства `supplier` типа `Ingredient`

Листинг 9.11 представляет собой обход нашего графа типов. Начиная с типа `ProductInterface`, мы можем получить подробную информацию о других объектах, таких как `Ingredient` и `Supplier`, используя их связи.

Это одна из самых мощных возможностей GraphQL и одно из его главных преимуществ по сравнению с другими типами API, такими как REST. При работе с REST нам пришлось бы сделать несколько запросов, чтобы получить всю информацию, которую мы смогли получить одним запросом в листинге 9.11. GraphQL дает возможность получить всю необходимую информацию и только ту, которая нужна, за один запрос.

## 9.6. ВЫПОЛНЕНИЕ НЕСКОЛЬКИХ ЗАПРОСОВ И АЛИАСИНГ

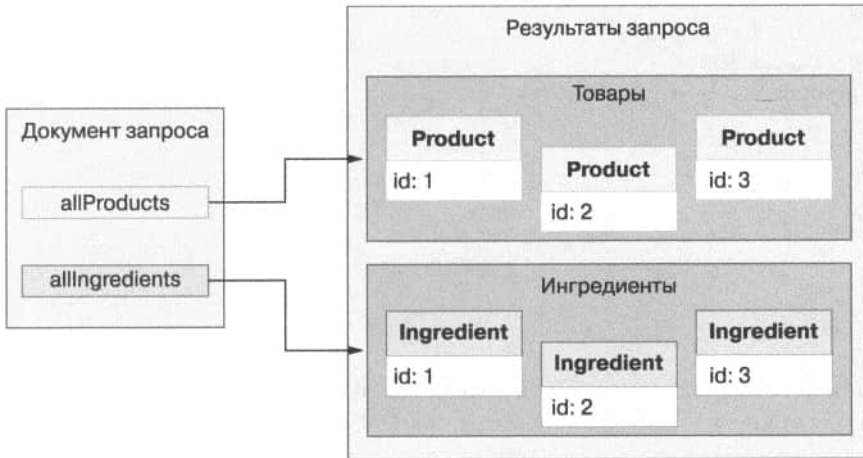
В этом разделе объясняется, как выполнять несколько запросов на один раз и как создавать псевдонимы (*aliases*) для ответов, возвращаемых сервером. Алиасинг (псевдонимирование) запросов означает изменение ключа, по которому индексируется набор данных, возвращаемый сервером. Как вы увидите, псевдонимы могут улучшить читаемость результатов, возвращаемых сервером, особенно когда выполняется несколько запросов за один раз.

### 9.6.1. Выполнение нескольких запросов за один раз

В предыдущих разделах мы выполняли только один запрос за раз. Однако GraphQL также позволяет отправлять сразу несколько запросов. Это еще одна мощная возможность GraphQL, которая позволяет избежать ненужных сетевых обращений к серверу, повышая общую производительность приложений и, следовательно, удобство работы пользователей.

Допустим, мы хотим получить список всех товаров и ингредиентов, доступных на платформе `CoffeeMesh` (рис. 9.4). Для этого мы можем выполнить запрос `allIngredients()` вместе с `allProducts()`.

В листинге 9.12 показано, как включить обе операции в один документ запроса. Включая несколько запросов в один документ запроса, мы гарантируем, что все они будут отправлены на сервер в одном запросе, и, следовательно, экономим время на обращении к серверу. Код также включает именованный фрагмент, который выбирает свойства для типа `ProductInterface`. Именованные фрагменты полезны для сохранения чистоты и сфокусированности запросов.



**Рис. 9.4.** В GraphQL мы можем выполнять несколько запросов за один заход, и ответ будет содержать по одному набору данных для каждого запроса

#### Листинг 9.12. Множественные запросы

```
{
  allProducts {
    ...commonProperties
  }
  allIngredients {
    name
  }
}
```

Выполняем запрос `allProducts()` без параметров

Выбираем свойства, используя именованный фрагмент

Выполняем запрос `allIngredients()`

Именованный фрагмент с выборкой, установленной для типа `ProductInterface`

```
fragment commonProperties on ProductInterface {
  name
}
```

### 9.6.2. Алиасинг запросов

Все запросы, которые мы выполняли в предыдущих разделах, являются анонимными. Когда мы совершаем анонимный запрос, данные, возвращаемые сервером, отображаются под ключом, названным по имени вызываемого запроса (листинг 9.13).

#### Листинг 9.13. Результат анонимного запроса

```
# Запрос
{
  allIngredients {
    name
  }
}
```

Выполняем запрос `allIngredients()`

```
# Результат
{
  "data": {
    "allIngredients": [
      {
        "name": "string"
      },
      {
        "name": "string"
      }
    ]
  }
}
```

Успешный ответ на запрос

Результат запроса

Выполнение анонимных запросов иногда может приводить в замешательство. `allIngredients()` возвращает список ингредиентов, поэтому полезно индексировать по ключу `ingredients` именно этот список, а не `allIngredients()`. Изменение имени ключа называется *алиасингом (псевдонимированием) запроса*. Используя псевдонимы, мы можем сделать запросы более удобочитаемыми. Преимущества псевдонимов становятся более очевидными, когда мы включаем несколько запросов в один. Например, запрос на все товары и ингредиенты, приведенный в листинге 9.12, становится более читабельным, если использовать псевдонимы. Код в листинге 9.14 показывает, как использовать псевдонимы для переименования результатов каждого запроса: результат запроса `allProducts()` появляется под псевдонимом `product`, а результат запроса `allIngredients()` — под псевдонимом `ingredients`.

#### Листинг 9.14. Использование алиасинга для получения более удобочитаемых запросов

```
{
  products: allProducts {
    ...commonProperties
  }
  ingredients: allIngredients {
    name
  }
}
```

Псевдоним для запроса `allProducts()`

Выбираем свойства, используя именованный фрагмент

Псевдоним для запроса `allIngredients()`

```
fragment commonProperties on ProductInterface {
  name
}
```

Именованный фрагмент с выборкой, установленной на `ProductInterface`

В некоторых случаях псевдонимы запросов необходимы для того, чтобы запросы работали. Например, в листинге 9.15 мы дважды запускаем запрос `products()`, чтобы выбрать два набора данных: один для доступных товаров, а другой — для недоступных. Оба набора данных создаются одним и тем же запросом: `products`. Как видите, без алиасинга этот запрос приводит к ошибке конфликта, поскольку оба набора данных возвращаются по одному и тому же ключу: `products`.



**Листинг 9.15.** Ошибка из-за многократного вызова одного и того же запроса без псевдонимов

```
{
  products(input: { available: true }) {
    ...commonProperties
  }
  products(input: { available: false }) {
    ...commonProperties
  }
}
```

Выполняем запрос `products()` для фильтрации доступных товаров

Выбираем свойства, используя фрагмент `commonProperties`

Выполняем запрос `products()` для поиска недоступных товаров

```
fragment commonProperties on ProductInterface {
  name
}
```

Именованный фрагмент с выборкой, установленной для типа `ProductInterface`

# Ошибка

```
{
  "errors": [
    {
      "message": "Fields \"products\" conflict because they have differing
        ➤ arguments. Use different aliases on the fields to fetch both if
        ➤ this was intentional.",
      "locations": [
        {
          "line": 2,
          "column": 3
        },
        {
          "line": 5,
          "column": 3
        }
      ]
    }
  ]
}
```

Запрос возвращает неудачный ответ, поэтому полезная нагрузка содержит ключ ошибки

В сообщении об ошибке говорится о конфликтном обращении к документу запроса

Сервер обнаружил ошибки в строках 2 и 5 документа запроса

Чтобы разрешить конфликт, созданный запросами в листинге 9.15, мы должны использовать псевдонимы. Код из листинга 9.16 исправляет запрос, добавляя псевдоним к каждой операции: `availableProducts` для запроса, который фильтрует доступные товары, и `unavailableProducts` для запроса, который фильтрует недоступные товары.

**Листинг 9.16.** Многократный вызов одного и того же запроса с псевдонимами

```
{
  availableProducts: products(input: { available: true }) {
    ...commonProperties
  }
  unavailableProducts: products(input: { available: false }) {
    ...commonProperties
  }
}
```

Псевдоним для запроса доступных товаров

Псевдоним для запроса недоступных товаров

```
fragment commonProperties on ProductInterface {
  name
}
```

# Результат (наборы данных для краткости опущены)

```
{
  "data": {
    "availableProducts": [...],
    "unavailableProducts": [...]
  }
}
```

На этом мы завершаем обзор запросов GraphQL. Вы разобрались, как выполнять запросы с параметрами, с типами входных данных, со встроенными и именованными фрагментами, с псевдонимами, а также научились включать несколько запросов в один. Мы прошли длинный путь! Но ни один обзор языка запросов GraphQL не будет полным без изучения мутаций.

## 9.7. ВЫПОЛНЕНИЕ МУТАЦИЙ GRAPHQL

*Мутации* — это функции GraphQL, которые позволяют создавать ресурсы или изменять состояние сервера. Выполнение мутации похоже на выполнение запроса. Единственное различие между ними заключается в их предназначении: запросы используются для чтения данных с сервера, а мутации — для создания или изменения данных на сервере.

Проиллюстрируем запуск мутации на примере. В листинге 9.17 показано, как выполняется мутация `deleteProduct()`. В первую очередь мы должны начать документ запроса с определения нашей операции как мутации. Мутация `deleteProduct()` имеет один обязательный аргумент — идентификатор товара, а ее возвращаемое значение — простое булево значение, поэтому в данном случае нам не нужно включать выборку.

### Листинг 9.17. Вызов мутации

```
mutation {
  deleteProduct(id: "asdf")
}
```

Теперь рассмотрим более сложную мутацию `addProduct()`, которая используется для добавления новых товаров в каталог CoffeeMesh. Функция `addProduct()` имеет три необходимых параметра:

- `name` — название товара;
- `type` — тип товара. Значения этого параметра ограничены перечислением `ProductType`, которое предлагает два варианта: выпечку и напиток;

- `input` — дополнительные свойства товара: его цена, объем, список ингредиентов и др. Полный список свойств задается типом `AddProductInput`.

`addProduct()` возвращает значение типа `Product`, что означает, что в данном случае мы должны добавить выборку. Помните, что `Product` — это объединение типов `Cake` и `Beverage`, поэтому в нашей выборке должны использоваться фрагменты, чтобы указать, свойства какого типа мы хотим включить в возвращаемую полезную нагрузку. В листинге 9.18 мы выбираем свойство `name` для типа `ProductInterface`.

**Листинг 9.18.** Вызов мутации с входными параметрами и сложным типом возвращаемого значения

```
mutation {
  addProduct(
    name: "Mocha", type: beverage, input: {price: 10, size: BIG,
      ingredients: [{ ingredient: 1, quantity: 1, unit: LITERS }]}
  ) {
    ...commonProperties
  }
}

fragment commonProperties on ProductInterface {
  name
}
```

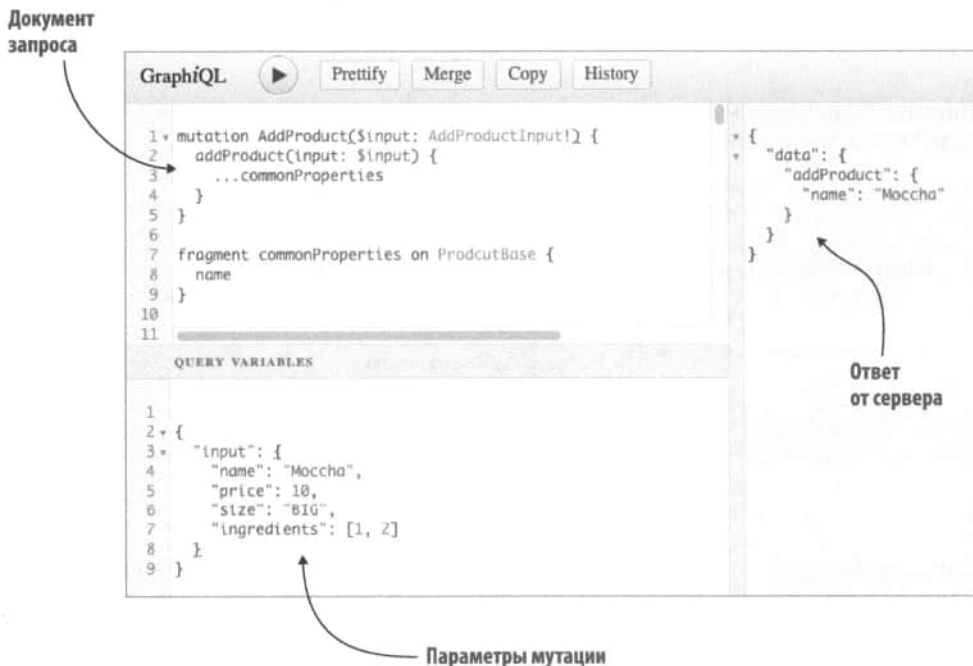
Далее разберемся, как писать более структурированные и удобочитаемые документы запросов, параметризуя аргументы.

## 9.8. ВЫПОЛНЕНИЕ ПАРАМЕТРИЗОВАННЫХ ЗАПРОСОВ И МУТАЦИЙ

В этом разделе описываются параметризованные запросы и объясняется, как использовать их для создания более структурированных и удобочитаемых документов запросов. В предыдущих разделах при использовании запросов и мутаций, требующих параметров, мы определяли значения для каждого параметра в той же строке, в которой вызывали функцию. В запросах с большим количеством аргументов такой подход может привести к тому, что документы запросов будут загромождены, их будет трудно читать и поддерживать. GraphQL предлагает решение этой проблемы — использовать параметризованные запросы.

Параметризованные запросы позволяют отделить вызовы запросов/мутаций от данных. На рис. 9.5 показано, как параметризовать вызов мутации `addProduct()`

с помощью GraphQL (код запроса также показан в листинге 9.19, чтобы вам было легче его просмотреть и скопировать). При параметризации запроса или мутации необходимо сделать две вещи: создать функцию-обертку для запроса/мутации и присвоить значения параметрам запроса/мутации в объекте переменных запроса. На рис. 9.6 показано, как эти части соединяются вместе для привязки параметризованных значений к вызову мутации `addProduct()`.



**Рис. 9.5.** GraphQL предлагает панель переменных запроса, куда мы можем включить входные значения для наших параметризованных запросов

#### Листинг 9.19. Использование параметризованного синтаксиса

```
# Документ запроса
mutation CreateProduct(
  $name: String!
  $type: ProductType!
  $input: AddProductInput!
) {
  addProduct(name: $name, type: $type, input: $input) {
    ...commonProperties
  }
}
```

Создаем обертку с именем `createProduct()`

Вызываем мутацию `addProduct()`

Выбираем свойства, используя именованный фрагмент

```

fragment commonProperties on ProductInterface {
  name
}

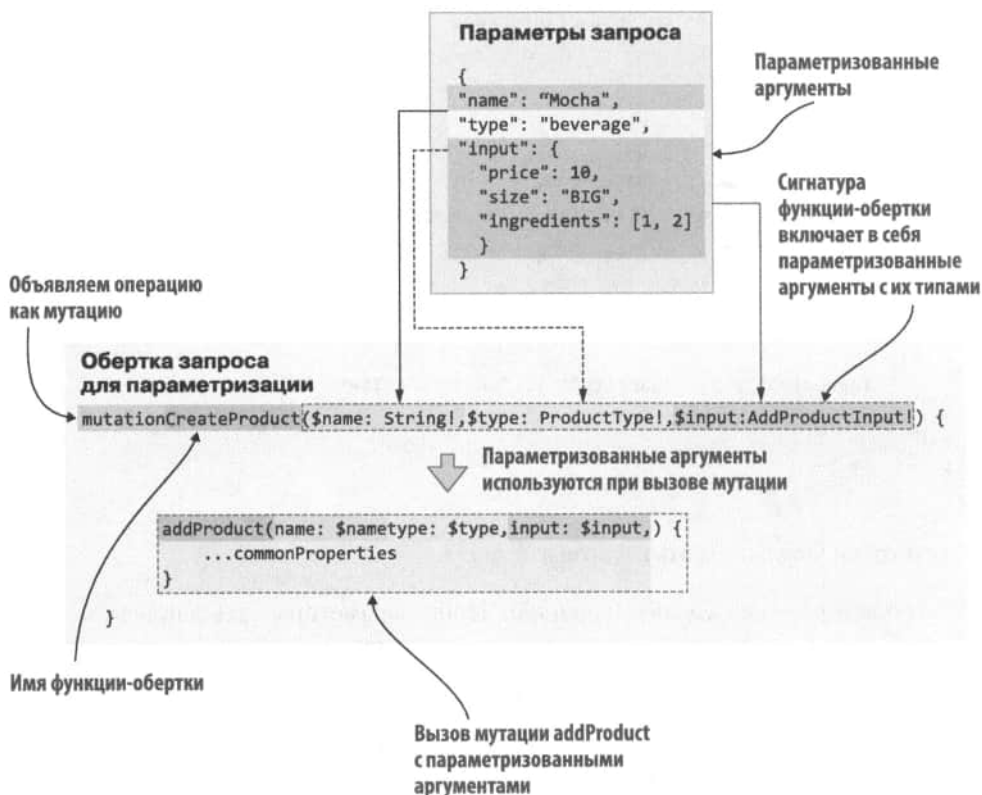
# Переменные запроса
{
  "name": "Mocha",
  "type": "beverage",
  "input": {
    "price": 10,
    "size": "BIG",
    "ingredients": [
      {
        "ingredient": 1, "quantity": 1, "unit": "LITERS"
      }
    ]
  }
}

```

Рассмотрим каждый из этих шагов в деталях.

1. *Создание обертки запроса/мутации.* Чтобы параметризовать запросы, мы создаем функцию-обертку вокруг запроса или мутации. На рис. 9.5 мы вызываем функцию-обертку `CreateProduct()`. Синтаксис обертки очень похож на синтаксис, который используется для определения запроса. Параметризованные аргументы должны быть включены в сигнатуру функции-обертки. На рис. 9.5 мы параметризуем параметры `name`, `type` и `input` мутации `addProduct()`. Параметризованный аргумент обозначен знаком доллара (`$`). В сигнатуре обертки (то есть в `CreateProduct()`) мы указываем ожидаемый тип параметризованных аргументов.
2. *Параметризация через объект переменных запроса.* Отдельно мы определяем переменные запроса в виде документа JSON. Как показано на рис. 9.5, в GraphQL переменные запроса определяются на панели `Query Variables`. Чтобы разобраться, как работают параметризованные запросы, посмотрите на рис. 9.6.

На рис. 9.5 мы использовали параметризованный синтаксис для обертывания только одной мутации, но ничто не запрещает нам обернуть несколько мутаций в пределах одного документа запроса. Когда мы оборачиваем несколько запросов или мутаций, все параметризованные аргументы нужно определить в сигнатуре функции-обертки. Код в листинге 9.20 показывает, как расширить запрос из листинга 9.19, чтобы включить вызов мутации `deleteProduct()`. Здесь мы вызываем обертку `CreateAndDeleteProduct()`, чтобы лучше представить действия в этом запросе.



**Рис. 9.6.** Для параметризации запросов и мутаций мы создаем функцию-обертку для запроса или мутации. В ее сигнатуру включаем параметризованные аргументы. Параметризованные переменные обозначаются символом \$

#### Листинг 9.20. Использование параметризованного синтаксиса

```
# Документ запроса
mutation CreateAndDeleteProduct(
  $name: String!
  $type: ProductType!
  $input: AddProductInput!
  $id: ID!
) {
  addProduct(name: $name, type: $type, input: $input) {
    ...commonProperties
  }
  deleteProduct(id: $id)
}

fragment commonProperties on ProductInterface {
  name
}
```

Создаем обертку с именем CreateAndDeleteProduct()

Вызываем мутацию addProduct()

Выбираем свойства, используя именованный фрагмент

Вызываем мутацию deleteProduct()

```
# Переменные запроса
{
  "name": "Mocha",
  "type": "beverage",
  "input": {
    "price": 10,
    "size": "BIG",
    "ingredients": [
      {
        "ingredient": 1, "quantity": 1, "unit": "LITERS"
      }
    ]
  },
  "id": "asdf"
}
```

← Присваиваем значения параметрам `addProduct()`

← Устанавливаем значение для параметра `id` мутации `deleteProduct()`

Итак, теперь вы можете исследовать любой GraphQL API, изучить его типы и поэкспериментировать с его запросами и мутациями. Прежде чем мы завершим эту главу, я хотел бы показать вам, как работает запрос GraphQL API «за кулисами».

## 9.9. РАСКРЫВАЕМ ТАЙНЫ ЗАПРОСОВ GRAPHQL

В этом разделе объясняется, как запросы GraphQL работают в контексте HTTP-запросов. В предыдущих разделах мы использовали клиент GraphQL для изучения нашего GraphQL API и взаимодействия с ним. GraphQL позволяет перевести документы запросов в HTTP-запросы, которые понимает сервер GraphQL. Клиенты GraphQL, такие как GraphQL, представляют собой интерфейсы, которые упрощают взаимодействие с GraphQL API. Но ничто не мешает вам отправить HTTP-запрос непосредственно к API, скажем, с вашего терминала, используя что-то вроде `cURL`. Вопреки распространенному заблуждению, для работы с GraphQL API не нужны никакие специальные инструменты<sup>1</sup>.

Чтобы отправить запрос к GraphQL API, вы можете использовать один из методов: GET или POST. Если вы выбираете GET, то отправляйте документ запроса с помощью параметров запроса URL, а если предпочитаете POST, включайте запрос в полезную нагрузку. Mock-сервер GraphQL Faker принимает только GET-запросы, поэтому я покажу, как отправить запрос с помощью GET.

Выполним запрос `allIngredients()`, выбирая только свойство `name` каждого ингредиента. Поскольку это GET-запрос, наш документ запроса должен быть включен

<sup>1</sup> Если только вы не планируете использовать подписки (соединения с сервером GraphQL, которые позволяют получать уведомления, когда что-то происходит на сервере, например, при изменении состояния ресурса). Подписки требуют двустороннего соединения с сервером, поэтому вам понадобится что-то более сложное, чем `cURL`. Чтобы больше узнать о подписках на GraphQL, см. книгу: Порселло Е., Бэнкс А. GraphQL. Язык запросов для современных веб-приложений. — СПб: Питер, 2019. — С. 71–73, 197–209.

в URL в качестве параметра запроса. Однако документ запроса содержит специальные символы, например фигурные скобки, которые считаются небезопасными и поэтому не могут быть включены в URL. Чтобы работать со специальными символами в URL, мы кодируем их. *Кодирование URL* — это процесс перевода специальных символов (скобок, знаков препинания и др.) в формат, подходящий для URL. Символы, закодированные в URL, начинаются со знака %, поэтому этот тип кодирования также известен как *процентное кодирование*<sup>1</sup>. cURL позаботится о кодировании URL наших данных, если мы воспользуемся опцией `--data-urlencode`. В результате cURL преобразует нашу команду в GET-запрос с таким URL: `http://localhost:9002/graphql?query=%7BballIngredients%7Bname%7D%7D`. В следующем фрагменте показана команда cURL, которую нужно запустить, чтобы выполнить этот вызов:

```
$ curl http://localhost:9002/graphql --data-urlencode \
'query={allIngredients{name}}'
```

Теперь, когда вы понимаете, как работают запросы GraphQL API, посмотрим, как можно использовать эти знания для написания кода на Python, который требуется GraphQL API.

## 9.10. ВЫЗОВ GRAPHQL API С ПОМОЩЬЮ КОДА НА PYTHON

В этом разделе рассказывается, как можно взаимодействовать с GraphQL API с помощью Python. Такие клиенты GraphQL, как GraphiQL, полезны для изучения GraphQL API, но на практике большую часть времени вы будете проводить за написанием приложений, потребляющих эти API программно. В этом разделе мы научимся использовать API сервиса продукции с помощью клиента GraphQL, написанного на Python.

Для работы с GraphQL API экосистема Python предлагает такие библиотеки, как `gql` (<https://github.com/graphql-python/gql>) и `sgqlc` (<https://github.com/profusion/sgqlc>). Они полезны, когда требуются расширенные возможности GraphQL, например подписки. В контексте микросервисов эти возможности нужны редко, поэтому для целей данного раздела мы обратимся к более простому подходу и воспользуемся популярной библиотекой `requests` (<https://github.com/psf/requests>). Помните, что запросы GraphQL — это просто GET- или POST-запросы с документом запроса.

В листинге 9.21 показано, как мы вызываем запрос `allIngredients()`, добавляя селектор для свойства `name` типа `Ingredient`. Приведенный код доступен в разделе

<sup>1</sup> Berners-Lee T., Fielding R., Masinter L. Uniform Resource Identifier (URI): Generic Syntax («Унифицированный идентификатор ресурса (URI): общий синтаксис») // RFC 3986, section 2.1. <https://datatracker.ietf.org/doc/html/rfc3986#section-2.1>.



`ch09/client.py` в репозитории GitHub этой книги. Поскольку наш mock-сервер GraphQL принимает только GET-запросы, мы отправляем документ запроса в виде данных, закодированных в URL. Для этого используем аргумент `params` метода `get`. Как вы можете видеть, документ запроса выглядит так же, как на панели запросов GraphiQL, и результат от API остается таким же. Это отличная новость, поскольку она означает, что при создании запросов вы можете начать работать с GraphiQL, используя его великолепный функционал в виде подсветки синтаксиса и проверки запросов, а когда будете готовы, перенесете запросы непосредственно в свой код на Python.

### Листинг 9.21. Вызов запроса GraphQL с использованием Python

```
# file: ch09/client.py

import requests

URL = 'http://localhost:9002/graphql'

query_document = '''
{
  allIngredients {
    name
  }
}
...

result = requests.get(URL, params={'query': query_document})

print(result.json())

# Результат
{'data': {'allIngredients': [{'name': 'string'}, {'name': 'string'}],
➡ {'name': 'string'}}}
```

Импортируем библиотеку запросов

Основной URL-адрес нашего GraphQL-сервера

Документ запроса

Отправляем запрос GET на сервер с документом запроса в качестве параметра URL-запроса

Парсируем и выводим полезную нагрузку JSON, которую вернул сервер

На этом мы завершаем наше путешествие по GraphQL. Вы прошли путь от изучения в главе 8 основных скалярных типов, поддерживаемых GraphQL, до создания в этой главе сложных запросов с использованием таких разнообразных инструментов, как GraphiQL, сURL и Python. Попутно мы создали спецификацию для API сервиса продукции и работали с ним с помощью mock-сервера GraphQL. Это отличный результат. Если вы дочитали до этой страницы, то наверняка узнали много нового об API и должны гордиться этим!

GraphQL — один из самых популярных протоколов в мире веб-API, и с каждым годом он становится все востребованнее. Это отличный выбор для построения API микросервисов и интеграции с приложениями фронтенда. В следующей главе мы займемся фактической реализацией API сервиса продукции и самого сервиса.

## РЕЗЮМЕ

- Когда мы вызываем запрос или мутацию, возвращающую объектный тип, наш запрос должен содержать выборку. Выборка — это список свойств, которые мы хотим получить от объекта, возвращаемого запросом.
- Когда запрос или мутация возвращает список из нескольких типов, выборка должна включать фрагменты. Фрагменты — это выборки свойств определенного типа, и для их обозначения используется префикс `spread (...)`.
- При вызове запроса или мутации с аргументами можно параметризовать эти аргументы, создавая обертку вокруг запроса или запросов. Это позволяет писать более читабельные и поддерживаемые документы запросов.
- При разработке GraphQL API удобно запускать его в работу с `mock-сервером`, который позволяет создавать клиенты API.
- Для запуска `mock-сервера GraphQL` можно использовать `graphql-faker`, который также создает интерфейс `GraphQL` для API. Это позволит проверить, соответствует ли дизайн вашим ожиданиям.
- На самом деле GraphQL-запрос — это простой HTTP-запрос, использующий один из методов: GET или POST. При выборе GET следует убедиться, что документ запроса закодирован в URL, а при использовании POST необходимо включить его в полезную нагрузку запроса.

# 10

## Создание GraphQL API с помощью Python

---

### В этой главе

- ✓ Создание GraphQL API с использованием фреймворка Ariadne.
- ✓ Проверка полезной нагрузки запросов и ответов.
- ✓ Создание резольверов для запросов и мутаций.
- ✓ Создание резольверов для сложных типов объектов, таких как типы-объединения.
- ✓ Создание резольверов для кастомных скалярных типов и свойств объектов.

В главе 8 мы разработали GraphQL API для сервиса продукции и составили спецификацию, в которой подробно описали требования к его API. В этой главе мы реализуем API в соответствии со спецификацией. Для создания API воспользуемся фреймворком Ariadne, который представляет собой одну из самых популярных библиотек GraphQL в экосистеме Python. Ariadne позволяет использовать преимущества разработки на основе документации, автоматически загружая модели проверки данных из спецификации. Вы научитесь создавать резольверы, которые представляют собой функции Python, реализующие логику запроса или мутации. Вы также научитесь работать с запросами, которые возвращают несколько типов. После прочтения этой главы у вас будут все необходимые инструменты, чтобы начать разработку собственных GraphQL API!

Код для этой главы доступен в папке `ch10` репозитория GitHub. Если не указано иное, все ссылки на файлы в главе относятся к папке `ch10`. Например, файл `server.py` расположен по адресу `ch10/server.py`, а `web/schema.py` — это `ch10/web/schema.py`. Кроме того, чтобы убедиться, что все команды, упомянутые в этой главе, работают должным образом, используйте команду `cd` для перемещения внутри папки `ch10` в вашем терминале.

## 10.1. АНАЛИЗ ТРЕБОВАНИЙ К API

Прежде чем приступить к реализации API, стоит потратить некоторое время на анализ спецификации API. Давайте сделаем это для API сервиса продукции.

Спецификация API сервиса продукции доступна в разделе `ch10/web/products.graphql` в репозитории GitHub для книги. Она определяет набор объектных типов, представляющих данные, которые мы можем получить из API, а также набор запросов и мутаций, реализующих возможности сервиса продукции. Мы должны создать модели проверки, которые точно соответствуют схемам, определенным в спецификации, а также функции, которые правильно реализуют действия запросов и мутаций. Мы будем работать с фреймворком, который способен автоматически проверять схему на основе спецификации, поэтому нам не нужно беспокоиться о реализации моделей проверки.

Наша реализация будет сосредоточена в основном на запросах и мутациях. Большинство запросов и мутаций, определенных в схеме, возвращают либо массив, либо один экземпляр типов `Ingredient` и `Product`. `Ingredient` проще, поскольку это объектный тип, и мы сначала рассмотрим запросы и мутации, использующие именно его. `Product` — это объединение типов `Beverage` и `Cake`, оба из которых реализуют тип `ProductInterface`. Как вы увидите, реализация запросов и мутаций, возвращающих объединенные типы, немного сложнее. Запрос, возвращающий список объектов `Product`, содержит экземпляры как типов `Beverage`, так и `Cake`, поэтому необходимо реализовать дополнительную функциональность, позволяющую серверу определить, к какому типу принадлежит каждый элемент в списке.

С учетом сказанного проанализируем технологический стек, который будем использовать в этой главе, а затем перейдем непосредственно к реализации.

## 10.2. ТЕХНОЛОГИЧЕСКИЙ СТЕК

В этом разделе мы рассмотрим технологический стек, который будем использовать для реализации API сервиса продукции. Обсудим, какие библиотеки доступны для реализации GraphQL API на Python, и выберем одну из них. Мы также познакомимся с серверным фреймворком, который будем использовать для запуска приложения.

Поскольку мы собираемся внедрять GraphQL API, в первую очередь нам нужна хорошая серверная библиотека GraphQL. Лучшим ресурсом для поиска инструментов и фреймворков экосистемы GraphQL является сайт <https://graphql.org/code/>. Поскольку экосистема постоянно развивается, я рекомендую вам время от времени заглядывать на него. На сайте перечислены четыре библиотеки Python, поддерживающие GraphQL.

- *Graphene* (<https://github.com/graphql-python/graphene>) — одна из первых библиотек GraphQL, созданных для Python. Она проверена на практике многими специалистами и является одной из наиболее широко используемых библиотек.
- *Ariadne* (<https://github.com/mirumee/ariadne>) — библиотека, созданная для разработки schema-first (на основе документации). Это очень популярный инструмент, который автоматически выполняет проверку схемы и сериализацию.
- *Strawberry* (<https://github.com/strawberry-graphql/strawberry>) — это более новая библиотека, которая упрощает реализацию моделей схем GraphQL.
- *Tartiflette* (<https://github.com/tartiflette/tartiflette>) — еще одно свежее дополнение к экосистеме Python, которое позволяет реализовать GraphQL-сервер с использованием подхода schema-first и построено поверх *asyncio*, являющейся основной библиотекой Python для асинхронного программирования.

В этой главе мы будем использовать фреймворк *Ariadne*, поскольку он поддерживает разработку через документирование и это зрелый проект. Спецификация API у нас уже есть, поэтому нам нежелательно тратить время на реализацию каждой модели схемы на Python. То есть нам нужна библиотека, которая может выполнять проверку схемы и сериализацию непосредственно из спецификации API, и *Ariadne* может это сделать.

Запустим сервер *Ariadne* с помощью *Uvicorn*, с которым вы сталкивались в главах 2 и 6 при работе с *FastAPI*. Чтобы установить зависимости для этой главы, можете использовать файлы *Pipfile* и *Pipfile.lock*, доступные в папке *ch10* в репозитории, поставляемом с этой книгой. Скопируйте их в свою папку *ch10*, перейдите в нее и выполните следующую команду:

```
pipenv install
```

Если вы захотите установить последние версии *Ariadne* и *Uvicorn*, просто выполните команду:

```
pipenv install ariadne uvicorn
```

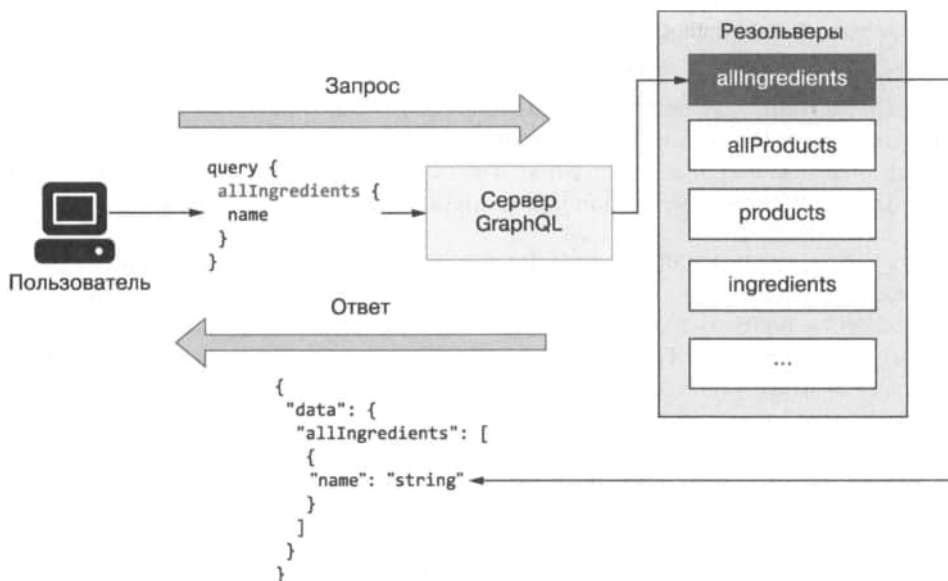
Теперь, когда зависимости установлены, активируем среду:

```
pipenv shell
```

Итак, все зависимости установлены и мы можем приступить к программированию.

## 10.3. ФРЕЙМВОРК ARIADNE

В этом разделе вы познакомитесь с фреймворком Ariadne, мы рассмотрим его работу на простом примере, разберем, как запустить сервер GraphQL с помощью Ariadne, как загрузить спецификацию GraphQL и как реализовать простой резольвер GraphQL. Как вы видели в главе 9, пользователи взаимодействуют с GraphQL API путем выполнения запросов и мутаций. Резольвер GraphQL — это функция, которая знает, как выполнить один из этих запросов или мутаций. В нашей реализации у нас будет столько резольверов, сколько запросов и мутаций указано в спецификации API. Как видно на рис. 10.1, резольверы являются основой сервера GraphQL, поскольку именно с их помощью мы можем возвращать фактические данные пользователям API.



**Рис. 10.1.** Для предоставления данных пользователю сервер GraphQL использует резольверы — функции, которые знают, как создать полезную нагрузку для конкретного запроса

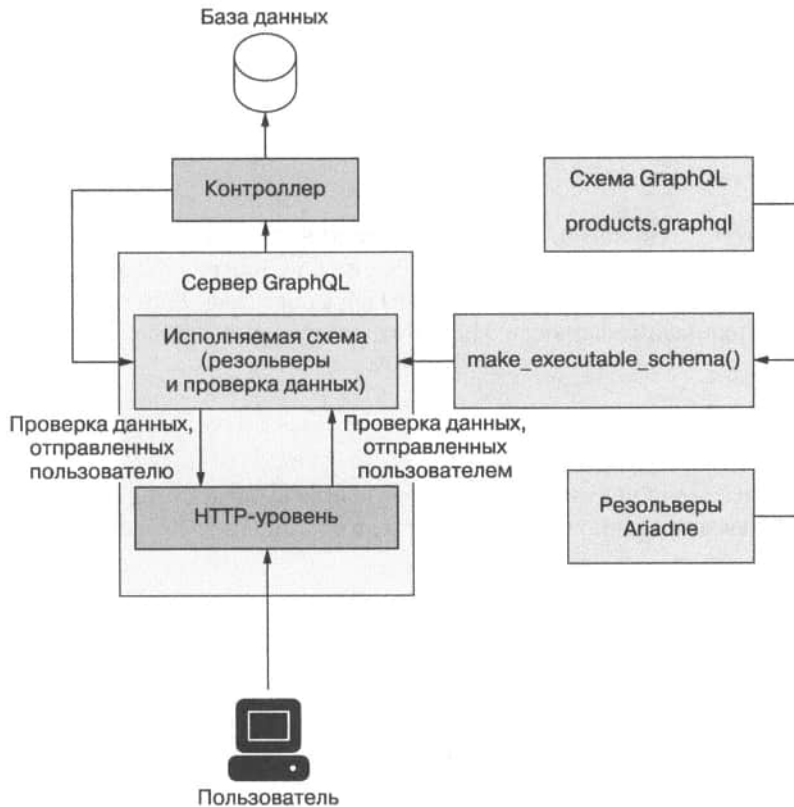
Начнем с написания очень простой схемы GraphQL. Откройте файл `server.py` и скопируйте в него следующее содержимое:

```
# file: server.py

schema = '''
type Query {
  hello: String
}
...
'''
```

Мы задаем переменную под названием `schema` и указываем ей на простую схему GraphQL. Эта схема определяет только один запрос с именем `hello()`, который возвращает строку. Возвращаемое значение запроса `hello()` является необязательным, это означает, что `null` также является допустимым возвращаемым значением. Чтобы предоставить доступ к этому запросу через сервер GraphQL, нам нужно реализовать резольвер, используя `Ariadne`.

`Ariadne` может запустить GraphQL-сервер на основе столь простого определения схемы. Как нам это сделать? Во-первых, нужно загрузить схему с помощью функции `Ariadne make_executable_schema()`. Она парсит документ, проверяет определения и создает внутреннее представление схемы. Как показано на рис. 10.2, `Ariadne` использует результаты этой функции для проверки данных. Например, когда мы возвращаем полезную нагрузку для запроса, `Ariadne` проверяет ее на соответствие схеме.



**Рис. 10.2.** Чтобы запустить сервер GraphQL с помощью `Ariadne`, мы создаем исполняемую схему, загружая схему GraphQL для API и набор резольверов для запросов и мутаций. `Ariadne` задействует ее для проверки данных, отправленных пользователем на сервер, а также данных, отправленных с сервера пользователю

После загрузки схемы мы можем инициализировать сервер с помощью класса GraphQL от Ariadne (листинг 10.1). Ariadne предоставляет две реализации сервера: синхронную реализацию, которая доступна в модуле `ariadne.wsgi`<sup>1</sup>, и асинхронную реализацию, расположенную в модуле `ariadne.asgi`<sup>2</sup>. В этой главе мы будем использовать асинхронную реализацию.

### Листинг 10.1. Инициализация сервера GraphQL с помощью Ariadne

# file: server.py

```
from ariadne import make_executable_schema
from ariadne.asgi import GraphQL
```

```
schema = '''
    type Query {
        hello: String
    }
    ...
'''
```

Объявляем простую схему

Создаем экземпляр сервера GraphQL

```
server = GraphQL(make_executable_schema(schema), debug=True)
```

Чтобы запустить сервер, выполните в терминале следующую команду:

```
$ uvicorn server:server --reload
```

Ваше приложение будет доступно по адресу <http://localhost:8000>. Если вы перейдете по нему, то увидите интерфейс приложения Apollo Playground. Как видно на рис. 10.3, Apollo Playground похож на GraphQL, который мы изучали в главе 8. Левая панель предназначена для написания запросов. Напишите следующий запрос:

```
{
  hello
}
```

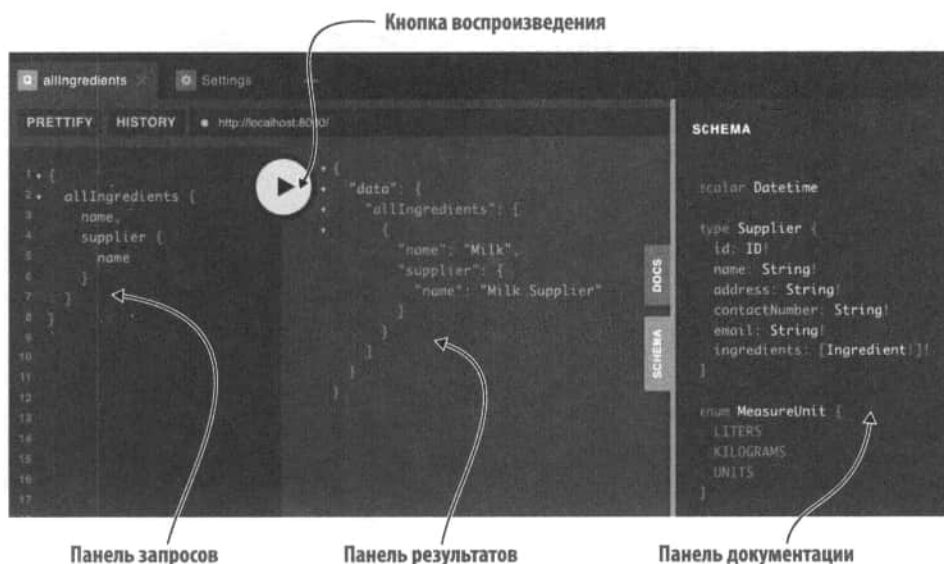
Этот запрос выполняет функцию, которую мы определили в листинге 10.1. Если вы нажмете кнопку **Выполнить**, то получите результаты этого запроса на правой панели:

```
{
  "data": {
    "hello": null
  }
}
```

<sup>1</sup> Wsgi (Web Server Gateway Interface) — стандарт взаимодействия между сервером и веб-приложением на языке Python. Впервые описан в PEP-333 (<http://www.python.org/dev/peps/pep-0333/>). — *Примеч. ред.*

<sup>2</sup> Asgi (Asynchronous Server Gateway Interface) — клиент-серверный протокол взаимодействия веб-сервера и приложения, предоставляет стандарт как для асинхронных, так и для синхронных приложений, поддерживает несколько серверов и фреймворков приложений. — *Примеч. ред.*





**Рис. 10.3.** Интерфейс Apollo Playground содержит панель запросов, где мы выполняем запросы и мутации, панель результатов, где оцениваются запросы и мутации, и панель документации, где мы можем просматривать схемы API

Запрос возвращает `null`. Это не должно удивлять вас, поскольку возвращаемое значение запроса `hello()` — строка, допускающая значение `null`. Как сделать так, чтобы запрос `hello()` возвращал строку? С помощью резольверов. Резольверы позволяют серверу узнать, как получить значение типа или атрибута. Создадим резольвер, который возвращает строку из десяти случайных символов.

В Ariadne резольвер — это вызываемый объект Python (например, функция), который принимает два позиционных параметра: `obj` и `info`.

Резольвер должен быть привязан к соответствующему объектному типу. Ariadne предоставляет классы привязки для каждого типа GraphQL:

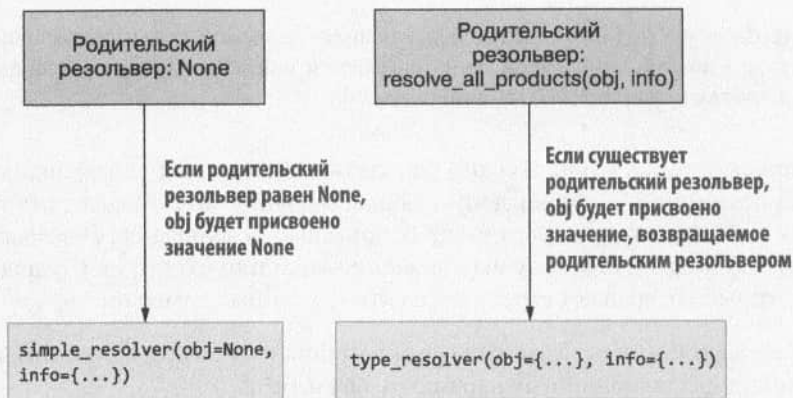
- `ObjectType` — для объектных типов;
- `QueryType` — для типов запросов. В GraphQL тип запроса представляет собой совокупность всех запросов, доступных в схеме. Как вы узнали в разделе 8.8, запрос — это функция, которая считывает данные с сервера GraphQL;
- `MutationType` — для типов мутаций. Как вы видели в разделе 8.9, мутация — это функция, которая изменяет состояние сервера GraphQL;
- `UnionType` — для типов-объединений;
- `InterfaceType` — для типов-интерфейсов;
- `EnumType` — для типов-перечислений.

## ПАРАМЕТРЫ РЕЗОЛЬВЕРА В ARIADNE

Резольверы Ariadne всегда имеют два позиционных параметра, которые обычно называются `obj` и `info`. Сигнатура базового резольвера Ariadne выглядит следующим образом:

```
def simple_resolver(obj: Any, info: GraphQLResolveInfo):
    pass
```

Как видно из рисунка ниже, `obj` обычно устанавливается равным `None`. Если же у резольвера есть родительский резольвер, то для `obj` будет установлено значение, возвращаемое им. С последним случаем мы сталкиваемся, когда резольвер не возвращает явный тип. Например, резольвер для запроса `allProducts()`, который мы реализуем в подразделе 10.4.4, возвращает не явный тип, а объект типа `Product`, который является объединением типов `Cake` и `Beverage`. Чтобы определить тип каждого объекта, Ariadne необходимо вызвать резольвер для типа `Product`.



Если у резольвера нет родительского резольвера, параметру `obj` присваивается значение `None`. При наличии родительского резольвера `obj` будет установлено значение, возвращаемое родительским резольвером

Параметр `info` является экземпляром `GraphQLResolveInfo`, который содержит информацию, необходимую для выполнения запроса. Ariadne использует ее для обработки и обслуживания каждого запроса. Для разработчика приложений наиболее интересным атрибутом объекта `info` является `info.context`, который хранит информацию о контексте, в котором вызывается резольвер, например HTTP-контексте. Чтобы узнать больше об объектах `obj` и `info`, ознакомьтесь с документацией Ariadne: <https://ariadnegraphql.org/docs/resolvers.html>.

Поскольку `hello()` — это запрос, нам нужно связать его резольвер с экземпляром `QueryType` Ariadne. В листинге 10.2 показано, как это сделать. Сначала создаем экземпляр класса `QueryType` и присваиваем его переменной `query`. Затем используем метод декоратора `field()` класса `QueryType`, чтобы привязать наш резольвер. Метод доступен для большинства привязываемых классов Ariadne и позволяет привязать резольвер к определенному полю. По соглашению мы добавляем к именам резольверов префикс `resolve_`. По умолчанию резольверы Ariadne всегда получают два позиционных параметра: `obj` и `info`. Но в данном случае они нам не нужны, поэтому мы добавляем подстановочный знак, за которым следует символ подчеркивания (`*``_`), что является указанием для Python игнорировать список позиционных параметров. Чтобы Ariadne знала о наших резольверах, нам нужно передать связываемые объекты в виде массива в функцию `make_executable_schema()`. Изменения вносятся в файл `server.py`.

### Листинг 10.2. Реализация резольвера GraphQL с помощью Ariadne

# file: server.py

```
import random
import string
```

```
from ariadne import QueryType, make_executable_schema
from ariadne.asgi import GraphQL
```

```
query = QueryType()
@query.field('hello')
def resolve_hello(*_):
    return ''.join(
        random.choice(string.ascii_letters) for _ in range(10)
    )
```

```
schema = '''
type Query {
    hello: String
}
...
'''
```

```
server = GraphQL(make_executable_schema(schema, [query]), debug=True)
```

Поскольку мы запускаем сервер с флагом горячей перезагрузки (`--reload`), сервер автоматически перезагружается, как только изменения в файле сохраняются. Вернитесь к интерфейсу Apollo Playground на странице <http://127.0.0.1:8000> и снова выполните запрос `hello()`. На этот раз вы должны получить случайную строку из десяти символов.

Итак, вы узнали, как загрузить схему GraphQL в Ariadne, как запустить сервер GraphQL и как реализовать резольвер для функции запроса. В оставшейся части главы мы применим эти знания при создании GraphQL API для сервиса продукции.

## 10.4. РЕАЛИЗАЦИЯ API СЕРВИСА ПРОДУКЦИИ

В этом разделе вы проверите на практике все, что узнали в предыдущем разделе, при создании GraphQL API для сервиса продукции. В частности, вы научитесь создавать резольверы для запросов и мутаций API сервиса продукции, обрабатывать параметры запросов и структурировать свой проект. Попутно мы рассмотрим дополнительные возможности фреймворка Ariadne и различные стратегии тестирования и реализации резольверов GraphQL. К концу этого раздела вы сможете создавать GraphQL API для собственных микросервисов.

### 10.4.1. Разработка структуры проекта

В этом подразделе мы структурируем наш проект для реализации API сервиса продукции. До сих пор мы включали весь код в файл `server.py`. Чтобы реализовать весь API, нам нужно разделить код на разные файлы и добавить структуру в проект; иначе кодовую базу будет трудно читать и поддерживать. Для упрощения реализации будем использовать представление данных в памяти.

Удалите код, измененный ранее в файле `server.py`, который представляет собой точку входа в наше приложение и поэтому должен содержать экземпляр сервера GraphQL. Мы инкапсулируем реализацию веб-сервера в папку `web/`. Создайте ее, а в ней — следующие файлы:

- `data.py` — представление данных в памяти;
- `mutations.py` — резольверы для мутаций в API сервиса продукции;
- `queries.py` — резольверы для запросов;
- `schema.py` — весь код, необходимый для загрузки исполняемого файла схемы;
- `types.py` — резольверы для объектных типов, кастомных скалярных типов и свойств объектов.

Файл спецификации `products.graphql` также находится в папке `web`, поскольку обрабатывается кодом из файла `web/schema.py`. Вы можете скопировать спецификацию API из файла `ch10/web/products.graphql` в репозитории GitHub для книги. Структура каталогов для API сервиса продукции выглядит следующим образом:

```

├── Pipfile
├── Pipfile.lock
├── server.py
└── web
    ├── data.py
    ├── mutations.py
    ├── products.graphql
    ├── queries.py
    ├── schema.py
    └── types.py

```

В репозитории GitHub для этой книги содержится дополнительный модуль `exceptions.py`, в котором вы можете найти примеры того, как обрабатывать исключения в GraphQL API. Теперь, когда мы структурировали наш проект, пора приступить к программированию!

### 10.4.2. Создание точки входа для сервера GraphQL

В этом подразделе мы создадим точку входа для сервера GraphQL. Сначала нужно создать экземпляр класса `GraphQL Ariadne` и загрузить исполняемую схему из спецификации API сервиса продукции.

Как уже говорилось в подразделе 10.4.1, точка входа для сервера API сервиса продукции находится в файле `server.py`. Добавьте в него следующее:

```
# file: server.py

from ariadne.asgi import GraphQL

from web.schema import schema

server = GraphQL(schema, debug=True)
```

Далее создадим исполняемую схему в `web/schema.py`:

```
# file: web/schema.py

from pathlib import Path

from ariadne import make_executable_schema

schema = make_executable_schema(
    (Path(__file__).parent / 'products.graphql').read_text()
)
```

Спецификация для API сервиса продукции доступна в файле `web/products.graphql`. Мы читаем содержимое файла схемы и передаем его в функцию `Ariadne make_executable_schema()`. Затем передаем полученный объект схемы в класс `GraphQL Ariadne` для создания экземпляра сервера. Если вы еще не запустили сервер, можете сделать это сейчас, выполнив следующую команду:

```
$ uvicorn server:server --reload
```

Как и раньше, API доступен на сайте <http://localhost:8000>. Если вы снова посетите его, то увидите знакомый пользовательский интерфейс Apollo Playground. На данном этапе можно попробовать выполнить любой из запросов, определенных в спецификации API сервиса продукции; однако большинство из них завершится

неудачей, поскольку мы не добавили резольверы. Например, если вы выполните следующий запрос:

```
{
  allIngredients {
    name
  }
}
```

то получите такое сообщение об ошибке: `Cannot return null for non-nullable field Query.allProducts` (Не удастся вернуть значение `null` для поля `Query.allProducts`). Сервер не знает, как получить значение для типа `Ingredient`, поскольку у нас нет для него резольвера, так что давайте создадим его!

### 10.4.3. Реализация резольверов запросов

В этом подразделе вы научитесь создавать резольверы запросов. Как видно из рис. 10.4, резольвер запроса — это функция Python, которая знает, как вернуть корректную полезную нагрузку для заданного запроса. Создадим резольвер для запроса `allIngredients()`, который является одним из самых простых в спецификации API сервиса продукции (листинг 10.3).

#### Листинг 10.3. Спецификация для типа `Ingredient`

```
# file: web/products.graphql
```

```
type Stock {
  quantity: Float!
  unit: MeasureUnit!
}

type Ingredient {
  id: ID!
  name: String!
  stock: Stock!
  products: [Product!]!
  supplier: Supplier
  description: [String!]
  lastUpdated: Datetime!
}
```

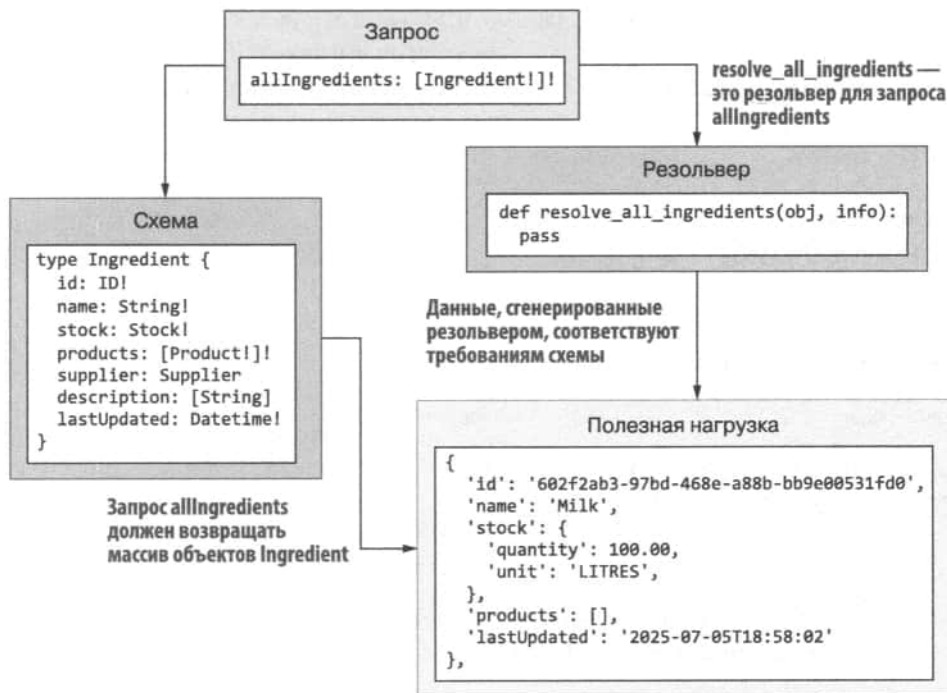
← Объявляем тип `Stock`

← `quantity` — это значение с плавающей запятой, не допускающее `null`

← `products` — список товаров, не допускающий `null`

← `supplier` — сквозной тип, допускающий `null`, который указывает на тип поставщика

Чтобы реализовать резольвер для запроса `allIngredients()`, нам просто нужно создать функцию, которая возвращает структуру данных с формой типа `Ingredient`, имеющую четыре свойства, не допускающих значения `null`: `id`, `name`, `stock` и `products`. Свойство `stock` является экземпляром объектного типа `Stock`, который согласно спецификации должен содержать свойства `quantity` и `unit`. Наконец, свойство `products` должно представлять собой массив объектов `Product`. Массив не допускает значения `null`, но возможно возвращение пустого массива.



**Рис. 10.4.** GraphQL использует резольверы для обработки запросов, отправляемых пользователем на сервер. Резольвер — это функция Python, которая знает, как вернуть допустимую полезную нагрузку для данного запроса

Добавим список ингредиентов к списочному представлению наших данных в памяти в файле `web/data.py`:

```
# file: web/data.py
```

```
from datetime import datetime
```

```
ingredients = [
    {
        'id': '602f2ab3-97bd-468e-a88b-bb9e00531fd0',
        'name': 'Milk',
        'stock': {
            'quantity': 100.00,
            'unit': 'LITRES',
        },
        'supplier': '92f2daae-a4f8-4aae-8d74-51dd74e5de6d',
        'products': [],
        'lastUpdated': datetime.utcnow(),
    },
]
```

Теперь, когда у нас есть некоторые данные, мы можем использовать их в резольвере `allIngredients()`. В листинге 10.4 показано, как он выглядит. Как и в разделе 10.3, сначала мы создаем экземпляр класса `QueryType` и связываем с ним резольвер. Поскольку это резольвер для типа запроса, его реализация находится в файле `web/queries.py`.

#### Листинг 10.4

```
# file: web/queries.py
```

```
from ariadne import QueryType
```

```
from web.data import ingredients
```

```
query = QueryType()
```

```
@query.field('allIngredients')
```

```
def resolve_all_ingredients(*_):
```

```
    return ingredients
```

Привязываем резольвер  
`allIngredients()`  
с помощью декоратора

Возвращаем жестко  
запрограммированный ответ

Чтобы включить резольвер запросов, мы должны передать объект запроса в функцию `make_executable_schema()` в файле `web/schema.py`:

```
# file: web/schema.py
```

```
from pathlib import Path
```

```
from ariadne import make_executable_schema
```

```
from web.queries import query
```

```
schema = make_executable_schema(
    (Path(__file__).parent / 'products.graphql').read_text(), [query]
)
```

Если мы вернемся к пользовательскому интерфейсу Apollo Playground и выполним запрос:

```
{
  allIngredients {
    name
  }
}
```

то получим корректную полезную нагрузку. Запрос выбирает только название ингредиента, что само по себе не очень интересно, и это не говорит нам о том, работает ли наш текущий резольвер для других полей. Напишем более сложный



запрос, чтобы более тщательно протестировать резольвер. Следующий запрос выбирает свойства `id`, `name` и `description` ингредиента, а также название каждого товара, с которым он связан:

```
{
  allIngredients {
    id,
    name,
    products {
      ...on ProductInterface {
        name
      }
    },
    description
  }
}
```

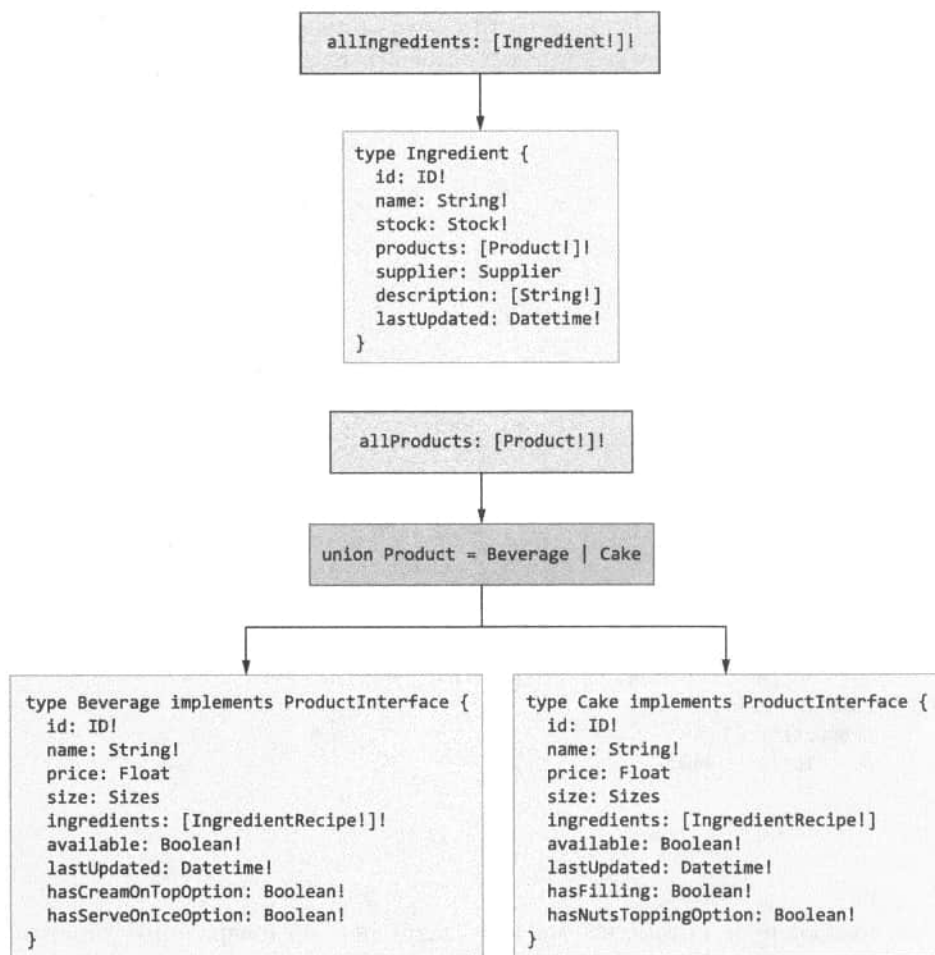
Полезная нагрузка ответа на этот запрос также валидна:

```
{
  "data": {
    "allIngredients": [
      {
        "id": " "602f2ab3-97bd-468e-a88b-bb9e00531fd0",
        "name": "Milk",
        "products": [],
        "description": null
      }
    ]
  }
}
```

Список товаров пуст, потому что мы не связали ни один товар с ингредиентом, а `description` равно `null`, потому что это поле допускает такое значение. Теперь, когда мы знаем, как реализовать резольверы для простых запросов, посмотрим, как реализовывать резольверы для обработки более сложных ситуаций.

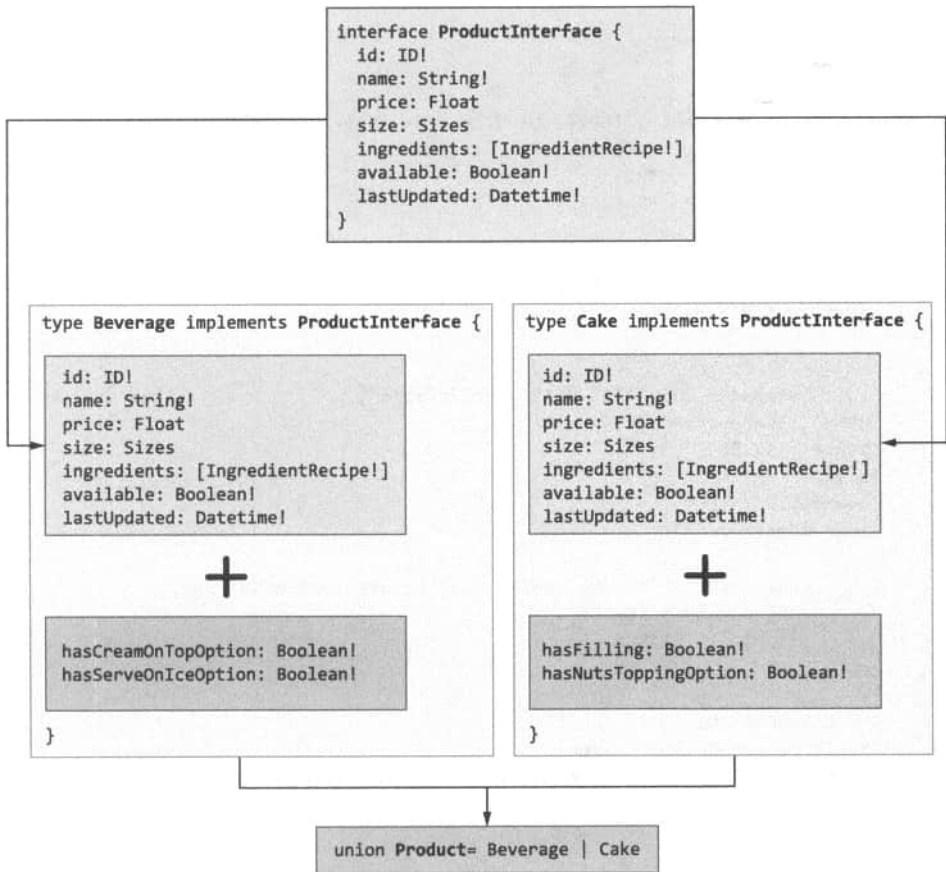
#### 10.4.4. Реализация резольверов типов

В этом подразделе вы научитесь реализовывать резольверы для запросов, возвращающих несколько типов. Запрос `allIngredients()` достаточно прост, поскольку возвращает только один объектный тип: `Ingredient`. Теперь рассмотрим запрос `allProducts()`. Как вы можете видеть из рис. 10.5, он посложнее, поскольку возвращает тип `Product`, который является объединением типов `Beverage` и `Cake`, а они оба реализуют тип `ProductInterface`.



**Рис. 10.5.** Запрос `allIngredients()` возвращает массив объектов `Ingredient`, в то время как запрос `allProducts()` возвращает массив объектов `Product`, где `Product` — это объединение двух типов: `Beverage` и `Cake`

Начнем с добавления списка товаров в наш список данных в памяти в файле `web/data.py`. Мы добавим два товара: один `Beverage` и один `Cake`. Какие поля мы должны включить? Как показано на рис. 10.6, поскольку `Beverage` и `Cake` реализуют тип `ProductInterface`, мы знаем, что они оба требуют идентификатор, название, список ингредиентов и поле `available`, которое показывает наличие товара. Помимо этих общих полей, унаследованных от `ProductInterface`, `Beverage` требует два дополнительных поля: `hasCreamOnTopOption` и `hasServeOnIceOption`, оба — булевого типа. В свою очередь, `Cake` требует наличия свойств `hasFilling` и `hasNutsToppingOption`, которые также являются булевыми (листинг 10.5).



**Рис. 10.6.** Товар представляет собой сочетание видов Beverage и Cake, а они оба реализуют тип ProductInterface. Поскольку Beverage и Cake реализуют один и тот же интерфейс, оба типа совместно используют свойства, унаследованные от интерфейса. В дополнение к этим свойствам каждый тип имеет свои специфические свойства, такие как hasFilling в случае Beverage

#### Листинг 10.5. Резольвер для запроса allProducts()

```
# file: web/data.py
```

```
...
```

```
products = [
    {
        'id': '6961ca64-78f3-41d4-bc3b-a63550754bd8',
        'name': 'Walnut Bomb',
        'price': 37.00,
```

```

'size': 'MEDIUM', 10.4
'available': False,
'ingredients': [
    {
        'ingredient': '602f2ab3-97bd-468e-a88b-bb9e00531fd0',
        'quantity': 100.00,
        'unit': 'LITRES',
    }
],
'hasFilling': False,
'hasNutsToppingOption': True,
'lastUpdated': datetime.utcnow(),
},
{
    'id': 'e4e33d0b-1355-4735-9505-749e3fdf8a16',
    'name': 'Cappuccino Star',
    'price': 12.50,
    'size': 'SMALL',
    'available': True,
    'ingredients': [
        {
            'ingredient': '602f2ab3-97bd-468e-a88b-bb9e00531fd0',
            'quantity': 100.00,
            'unit': 'LITRES',
        }
    ],
    'hasCreamOnTopOption': True,
    'hasServeOnIceOption': True,
    'lastUpdated': datetime.utcnow(),
},
]

```

Этот ID ссылается на ID такого ингредиента, как молоко, который мы ранее добавили в web/data.py

Теперь, когда у нас есть список товаров, используем его в резольвере `allProducts()` (листинг 10.6).

#### Листинг 10.6. Добавление резольвера `allProducts()`

```

# file: web/queries.py

from ariadne import QueryType

from web.data import ingredients, products

query = QueryType()

...

@query.field('allProducts')
def resolve_all_products(*_):
    return products

```

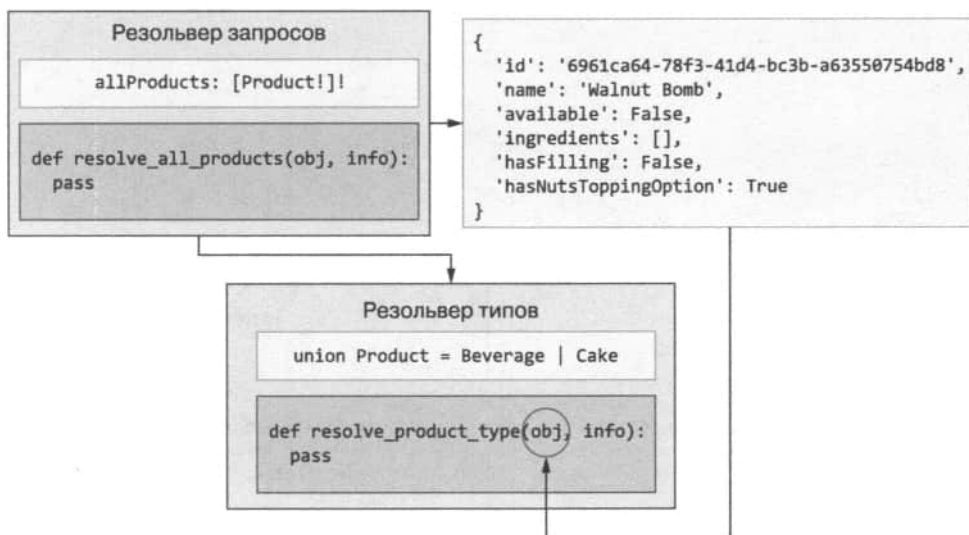
Привязываем резольвер `allProducts()`, используя декоратор `field()`

Возвращаем жестко запрограммированный ответ

Выполним простой запрос, чтобы проверить работу резольвера:

```
{
  allProducts {
    ...on ProductInterface {
      name
    }
  }
}
```

Если вы выполните этот запрос, то получите ошибку, говорящую о том, что сервер не может определить, к какому типу относится каждый из элементов списка. В таких ситуациях нам нужен резольвер типов. Как показано на рис. 10.7, *резольвер типов* — это функция Python, которая определяет тип объекта и возвращает имя типа.



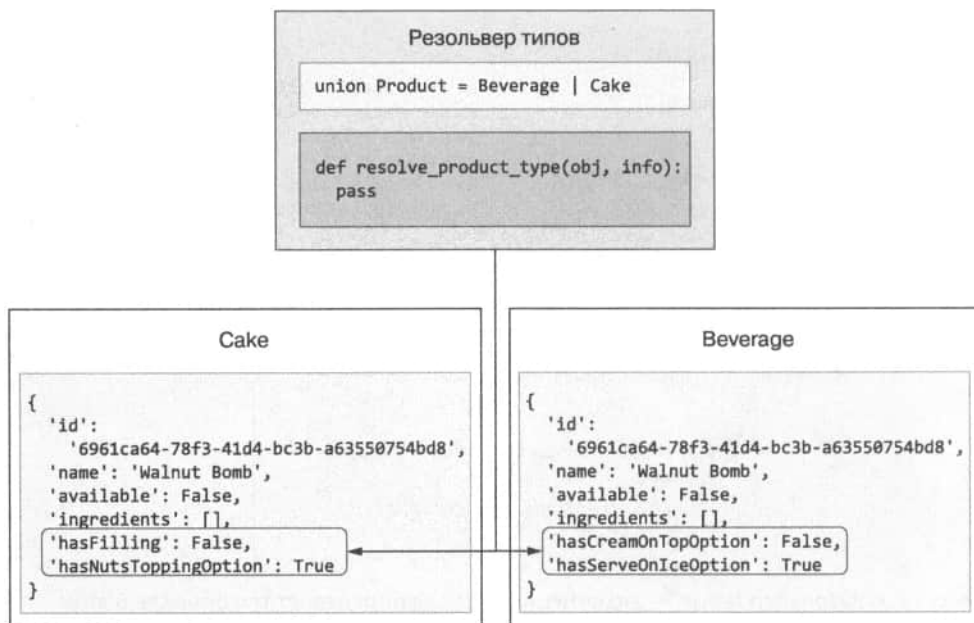
**Рис. 10.7.** Резольвер типов — это функция, которая определяет тип объекта. В этом примере показано, как резольвер `resolve_product_type()` определяет тип объекта, возвращаемого резольвером `resolve_all_products()`

Нам нужны резольверы типов в запросах и мутациях, которые возвращают более одного объектного типа. В API сервиса продукции это касается всех запросов и мутаций, возвращающих тип `Product`: `allProducts()`, `addProduct()` и `product()`.

## ВОЗВРАЩАЕМ МНОЖЕСТВО ТИПОВ

Если запрос или мутация возвращает несколько типов, необходимо реализовать резольвер типов. Это относится к запросам и мутациям, которые возвращают типы-объединения и объектные типы, реализующие интерфейсы.

В листинге 10.7 показано, как мы реализуем резольвер типов для типа `Product` в `Ariadne`. Функция резольвера типов принимает два позиционных параметра, первым из которых является `object`. Нам нужно определить тип этого объекта. Поскольку мы знаем, что `Cake` и `Beverage` имеют разные обязательные поля, можем использовать эту информацию для определения их типов: если объект имеет свойство `hasFilling`, мы понимаем, что это `Cake`; в противном случае это `Beverage` (рис. 10.8).



**Рис. 10.8.** Резольвер типов проверяет свойства полезной нагрузки, чтобы определить ее тип. В этом примере функция `resolve_product_type()` выполняет поиск свойств, которые отличают `Cake` от типа `Beverage`

Резольвер типов должен быть привязан к типу `Product`. Поскольку `Product` является типом-объединением, мы создаем для него привязываемый объект с помощью класса `UnionType`. `Ariadne` гарантирует, что первым аргументом резольвера является объект, и мы проверяем его, чтобы определить тип. Другие параметры

нам не нужны, поэтому мы игнорируем их с помощью синтаксиса Python `*_`. Чтобы определить тип объекта, проверяем, есть ли у него атрибут `hasFilling`. Если да, то мы понимаем, что это объект `Cake`; в противном случае это `Beverage`. Наконец, мы передаем привязываемый товар функции `make_executable_schema()`. Поскольку это резольвер типов, код помещается в файл `web/types.py` (см. листинг 10.7).

#### Листинг 10.7. Реализация резольвера типов для типа-объединения `Product`

# file: web/types.py

```
from ariadne import UnionType

product_type = UnionType('Product')

@product_type.type_resolver
def resolve_product_type(obj, *_):
    if 'hasFilling' in obj:
        return 'Cake'
    return 'Beverage'
```

Создаем привязываемый объект для типа `Product`, используя класс `UnionType`

Привязываем резольвер `Product` с помощью декоратора `resolver()`

Записываем первый позиционный аргумент распознавателя как `obj`

Чтобы включить резольвер типов, нам нужно добавить объект товара в функцию `make_executable_schema()` в файле `web/schema.py`:

# file: web/schema.py

```
from pathlib import Path

from ariadne import make_executable_schema

from web.queries import query
from web.types import product_type

schema = make_executable_schema(
    (Path(__file__).parent / 'products.graphql').read_text(),
    [query, product_type]
)
```

Теперь снова запустим запрос `allProducts()`:

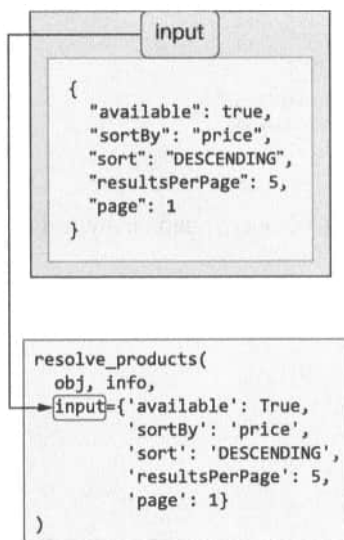
```
{
  allProducts {
    ...on ProductInterface {
      name
    }
  }
}
```

Теперь вы получите успешный ответ. В следующем подразделе продолжим изучение запросов и научимся работать с параметрами запросов.

### 10.4.5. Обработка параметров запроса

В этом подразделе мы разберемся, как обрабатывать параметры запроса в резольверах. Большинство запросов в API сервиса продукции принимают параметры фильтрации, и все мутации требуют по крайней мере одного параметра. Посмотрим, как получить доступ к параметрам, изучив один пример из API сервиса продукции: запрос `products()`, принимающий объект `input`, тип которого — `ProductsFilter`. Как получить доступ к этому объекту фильтра в резольвере?

Когда запрос или мутация принимает параметры, Ariadne передает их резольверам в качестве именованных аргументов (рис. 10.9).



**Рис. 10.9.** Параметры запроса передаются нашим резольверам в качестве именованных аргументов

В листинге 10.8 показано, как мы получаем доступ к параметру `input` для резольвера запросов `products()`. Поскольку параметр `input` является необязательным и, следовательно, допускает значение `null`, по умолчанию мы устанавливаем для него значение `None`. Этот параметр является экземпляром входного типа `ProductsFilter`, поэтому, когда он присутствует в запросе, он представлен в виде словаря. Из спецификации API мы знаем, что `ProductsFilter` гарантирует наличие следующих полей:

- `available` — логическое поле, которое фильтрует товары по наличию;
- `sortBy` — тип перечисления, который позволяет нам сортировать товары по цене или названию;



- `sort` — тип перечисления, который позволяет сортировать результаты в порядке возрастания или убывания;
- `resultsPerPage` — указывает, сколько результатов должно отображаться на странице;
- `page` — указывает, какую страницу результатов мы должны вернуть.

Помимо этих параметров, `ProductsFilter` может включать два необязательных параметра: `maxPrice`, который фильтрует результаты по максимальной цене, и `minPrice`, фильтрующий результаты по минимальной цене. Поскольку `maxPrice` и `minPrice` не являются обязательными, мы проверяем их наличие с помощью метода `get()` словаря Python, который возвращает `None`, если они не найдены. Сначала реализуем функции фильтрации и сортировки, а потом разберемся с разбивкой по страницам. Код, приведенный в листинге 10.8, находится в папке `web/queries.py`.

#### Листинг 10.8. Доступ к входным параметрам в резольвере

```
# file: web/queries.py
```

```
...
```

```
Query = QueryType()
```

```
...
```

```
@query.field('products')
def resolve_products(*, input=None):
    filtered = [product for product in products]
    if input is None:
        return filtered
    filtered = [
        product for product in filtered
        if product['available'] is input['available']
    ]
    if input.get('minPrice') is not None:
        filtered = [
            product for product in filtered
            if product['price'] >= input['minPrice']
        ]
    if input.get('maxPrice') is not None:
        filtered = [
            product for product in filtered
            if product['price'] <= input['maxPrice']
        ]
    filtered.sort(
        key=lambda product: product.get(input['sortBy'], 0),
        reverse=input['sort'] == 'DESCENDING'
    )
    return filtered
```

Привязываем резольвер `products()`, используя декоратор `field()`

Игнорируем позиционные аргументы по умолчанию и вместо этого фиксируем параметр `input`

Если `input` равен `None`, возвращаем весь набор данных целиком

Копируем список товаров

Фильтруем товары по наличию

Фильтруем товары по минимальной цене

Сортируем отфильтрованный набор данных

Возвращаем отфильтрованный набор данных

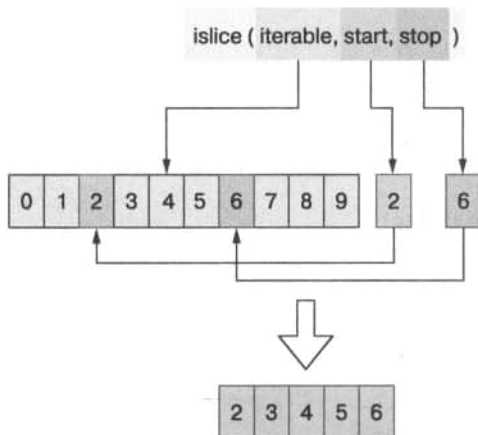
Выполним запрос для проверки этого резольвера:

```
{
  products(input: {available: true}) {
    ...on ProductInterface {
      name
    }
  }
}
```

Вы должны получить правильный ответ от сервера.

Теперь, когда мы отфильтровали результаты, нам нужно разбить их на страницы. В листинге 10.9 добавлена универсальная функция разбивки на страницы `get_page()` в `web/queries.py`. Небольшое предупреждение: в обычных условиях вы будете хранить данные в базе данных и делегировать фильтрацию и разбивку на страницы именно ей. Приведенные здесь примеры иллюстрируют использование параметров запроса в резольвере. Мы размещаем результаты постранично с помощью функции `islice()` из модуля `itertools`.

Как показано на рис. 10.10, `islice()` позволяет извлечь фрагмент итерируемого объекта. Функция требует, чтобы мы указали начальный и конечный индексы фрагмента, для которого хотим сделать срез. Например, список из десяти элементов, состоящий из чисел от 0 до 9, с начальным индексом 2 и конечным 6, даст нам срез со следующими элементами: [2, 3, 4, 5]. API разбивает результаты на страницы, начиная с 1, в то время как в `islice()` индексация начинается с нуля, поэтому `get_page()` вычитает одну единицу из параметра `page`, чтобы учесть эту разницу.



**Рис. 10.10.** Функция `islice()` из модуля `itertools` позволяет получить фрагмент итерируемого объекта, выбрав начальный и конечный индексы требуемого подмножества

**Листинг 10.9.** Разбивка результатов на страницы

```
# file: web/queries.py

from itertools import islice  ← Импортируем islice()

from ariadne import QueryType

from web.data import ingredients, products

...

def get_page(items, items_per_page, page):
    page = page - 1
    start = items_per_page * page if page > 0 else page  ← Определяем
                                                         начальный индекс
    stop = start + items_per_page  ← Вычисляем конечный индекс
    return list(islice(items, start, stop))  ← Возвращаем
                                           фрагмент списка

@query.field('products')
def resolve_products(*_, input=None):
    ...
    return get_page(filtered, input['resultsPerPage'], input['page'])  ← Разбиваем результаты
                                                                           на страницы
```

Наш жестко запрограммированный набор данных содержит только два товара, поэтому протестируем разбиение на страницы, установив `resultsPerPage` равным 1, что разделит список на две страницы:

```
{
  products(input: {resultsPerPage: 1, page: 1}) {
    ...on ProductInterface {
      name
    }
  }
}
```

Вы должны получить только один результат. Когда мы реализуем мутацию `addProduct()` в следующем подразделе, сможем добавлять больше товаров через API и более активно использовать параметры пагинации.

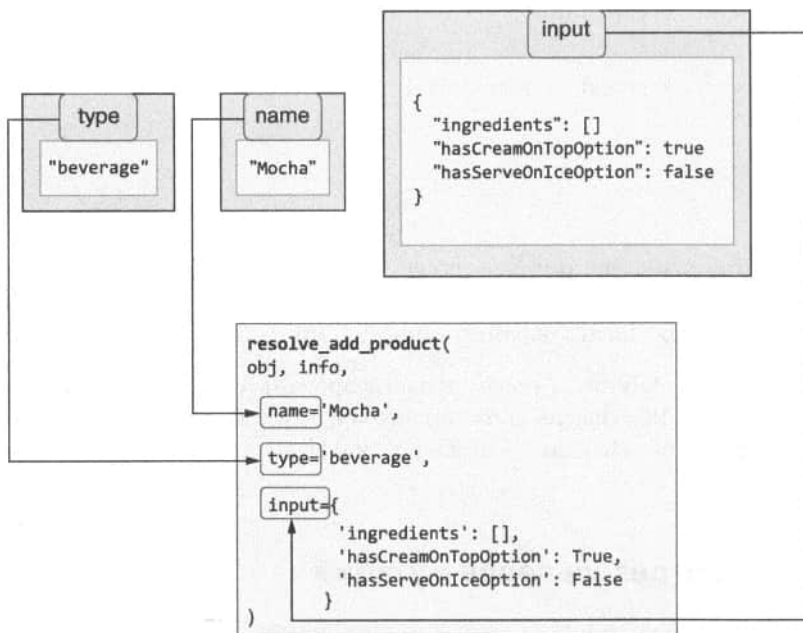
Вы только что узнали, как работать с параметрами запроса, и теперь готовы узнать, как реализовать мутации. Резольверы мутаций похожи на резольверы запросов, но у них всегда есть параметры. Но хватит спойлеров; читайте дальше, чтобы больше узнать о мутациях.

### 10.4.6. Реализация резольверов мутаций

В этом подразделе вы научитесь реализовывать резольверы мутаций. При этом нужно придерживаться тех же рекомендаций, которые приводились для запросов. Единственное различие состоит в классе, который мы используем для привязки резольверов мутаций. Если запросы привязываются к экземпляру класса `QueryType`, то мутации привязываются к экземпляру класса `MutationType`.

Рассмотрим реализацию резольвера для мутации `addProduct()`. Из спецификации мы знаем, что эта мутация имеет три обязательных параметра: `name`, `type` и `input`. Форма параметра `input` задается объектным типом `AddProductInput`. Он определяет дополнительные свойства, которые могут быть заданы при создании нового товара, все они являются необязательными и, следовательно, допускают значение `null`. Наконец, мутация `addProduct()` должна возвращать тип `Product`.

В листинге 10.10 показано, как реализуется резольвер для мутации `addProduct()` (рис. 10.11). Сначала импортируем класс с возможностью привязки `MutationType` и инстанцируем его. Затем объявляем наш резольвер и привязываем его к `MutationType` с помощью декоратора `field()`. Нам не нужно использовать позиционные параметры `obj` и `info` Ariadne, поэтому пропускаем их, используя синтаксис Python `*_`. Мы не задаем значения по умолчанию для параметров `addProduct()`, поскольку в спецификации говорится, что все они обязательны. Мутация `addProduct()` должна возвращать корректный объект `Product`, поэтому создаем объект с его ожидаемыми атрибутами в теле резольвера. Поскольку `Product` является объединением типов `Cake` и `Beverage` и для каждого типа требуются разные наборы свойств, проверяем параметр `type`, чтобы определить, какие поля необходимо добавить в наш объект. Код, приведенный в листинге 10.10, находится в файле `web/mutations.py`.



**Рис. 10.11.** Параметры мутации передаются нашим резольверам в качестве именованных аргументов. В этом примере показано, как вызывается резольвер `resolve_add_product()` с параметрами `name`, `type` и `input`

**Листинг 10.10.** Резольвер мутации addProduct()

```
# file: web/mutations.py
```

```
import uuid
from datetime import datetime

from ariadne import MutationType

from web.data import products

mutation = MutationType()

@mutation.field('addProduct')
def resolve_add_product(*_, name, type, input):
    product = {
        'id': uuid.uuid4(),
        'name': name,
        'available': input.get('available', False),
        'ingredients': input.get('ingredients', []),
        'lastUpdated': datetime.utcnow(),
    }
    if type == 'cake':
        product.update({
            'hasFilling': input['hasFilling'],
            'hasNutsToppingOption': input['hasNutsToppingOption'],
        })
    else:
        product.update({
            'hasCreamOnTopOption': input['hasCreamOnTopOption'],
            'hasServeOnIceOption': input['hasServeOnIceOption'],
        })
    products.append(product)
    return product
```

Привязываемый объект для мутаций

Привязываем распознаватель addProduct() с помощью декоратора field()

Фиксируем параметры addProduct()

Определяем новый товар как словарь

Устанавливаем свойства на стороне сервера, такие как ID

Парсируем необязательные параметры и устанавливаем их значения по умолчанию

Проверяем, чем является товар: выпечкой или напитком

Возвращаем только что созданный товар

Чтобы включить резольвер, реализованный в листинге 10.10, нужно добавить объект `mutation` для функции `make_executable_schema()` в `web/schema.py`:

```
# file: web/schema.py
```

```
from pathlib import Path
```

```
from ariadne import make_executable_schema
```

```
from web.mutations import mutation
```

```
from web.queries import query
```

```
from web.types import product_type
```

```
schema = make_executable_schema(
    (Path(__file__).parent / 'products.graphql').read_text(),
    [query, mutation, product_type]
)
```

Проверим работу новой мутации, выполнив простой тест. Перейдите на Apollo Playground по адресу <http://127.0.0.1:8000> и запустите следующую мутацию:

```
mutation {
  addProduct(name: "Mocha", type: beverage, input:{ingredients: []}) {
    ...on ProductInterface {
      name,
      id
    }
  }
}
```

Вы получите правильный ответ, и новый товар будет добавлен в список. Чтобы убедиться, что все работает правильно, выполните следующий запрос и проверьте, что в ответе есть только что созданный новый элемент:

```
{
  allProducts {
    ...on ProductInterface {
      name
    }
  }
}
```

Помните, что мы запускаем сервис со списком данных в памяти, поэтому, если вы остановите или перезагрузите сервер, список будет сброшен и вы потеряете все вновь созданные данные.

Итак, вы узнали, как создавать мутации. Это мощная функция: с помощью мутаций можно создавать и обновлять данные на сервере GraphQL. На текущий момент мы рассмотрели почти все основные аспекты реализации сервера GraphQL. В следующем подразделе продолжим эту тему и попробуем реализовать резольверы для кастомных скалярных типов.

### 10.4.7. Создание резольверов для кастомных скалярных типов

В этом подразделе вы узнаете, как реализовать резольверы для кастомных скалярных типов. Как вы видели в главе 8, GraphQL предоставляет приличное количество скалярных типов, таких как `boolean`, `integer` и `string`. И во многих случаях скалярных типов GraphQL по умолчанию достаточно для разработки API. Однако иногда необходимо определить собственные скалярные типы. API сервиса продукции содержит кастомный скаляр `Datetime`. Поле `lastUpdated` в типах `Ingredient` и `Product` имеет тип `Datetime`. Поскольку это кастомный скаляр, Ariadne не знает, как с ним работать, то есть нужно реализовать для него резольвер. Как это сделать?



**Рис. 10.12.** Когда сервер GraphQL получает данные от пользователя, он проверяет и десериализует данные в нативные объекты Python. В этом примере сервер десериализует имя "Mocha" в строку, а дату "2021-01-01" — в объект `datetime`

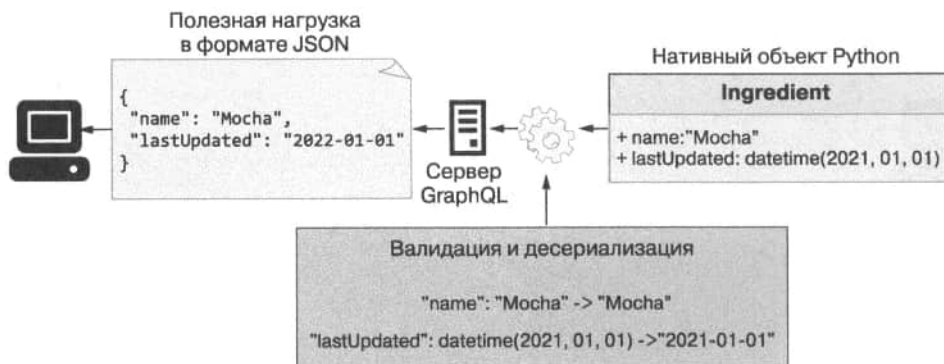
Как видно на рис. 10.12 и 10.13, когда мы встречаем кастомный скалярный тип в GraphQL API, нужно убедиться, что мы можем выполнить с ним следующие три действия:

- *сериализацию* — когда пользователь запрашивает данные с сервера, Ariadne должна уметь сериализовать их. Ariadne знает, как сериализовать кастомные скаляры GraphQL, но для пользовательских скаляров придется реализовать кастомный сериализатор. В случае со скаляром `Datetime` в API сервиса продукции нужно реализовать метод для сериализации объекта `datetime`;
- *десериализацию* — когда пользователь отправляет данные на сервер, Ariadne десериализует их и предоставляет нам в виде нативной структуры данных Python, например словаря. Если данные включают кастомный скаляр, нам нужно реализовать метод, который позволит Ariadne узнать, как парсить и загружать скаляр в нативную структуру данных Python. Что касается скаляра `Datetime`, мы хотим загружать его как объект `datetime`;
- *валидацию* — GraphQL обеспечивает валидацию каждого скаляра и типа, и Ariadne знает, как проверять встроенные скаляры GraphQL. Для кастомных скаляров придется реализовать собственные методы валидации. В случае со скаляром `Datetime` нам требуется убедиться, что он имеет валидный формат ISO.

Ariadne предоставляет простой API для обработки этих действий через свой класс `ScalarType`. В первую очередь нам нужно создать экземпляр этого класса:

```
from ariadne import ScalarType

datetime_scalar = ScalarType('Datetime')
```



**Рис. 10.13.** Когда сервер GraphQL отправляет данные пользователю, он преобразует нативные объекты Python в сериализуемые данные. В этом примере сервер сериализует как имя, так и дату в виде строк

`ScalarType` предоставляет методы декораторов, которые позволяют нам реализовать сериализацию, десериализацию и валидацию. Для сериализации воспользуемся декоратором `serializer()` от `ScalarType`. Нам нужно сериализовать объекты `datetime` в стандартный формат даты ISO, а библиотека `datetime` в Python предоставляет удобный метод для форматирования ISO — `isoformat()`:

```
@datetime_scalar.serializer
def serialize_datetime(value):
    return value.isoformat()
```

Для проверки и десериализации `ScalarType` предоставляет декоратор `value_parser()`. Когда пользователь отправляет на сервер данные, содержащие скаляр `Datetime`, мы ожидаем, что дата будет в формате ISO и, следовательно, будет парсирована Python-методом `datetime.fromisoformat()`:

```
from datetime import datetime

@datetime_scalar.value_parser
def parse_datetime_value(value):
    return datetime.fromisoformat(value)
```

Если дата указана в неправильном формате, `fromisoformat()` вызовет ошибку `ValueError`, которая будет перехвачена `Ariadne`, и пользователь получит такое сообщение: `Invalid isoformat string (Недопустимая строка ISO-формата)`. Код, приведенный в листинге 10.11, находится в `web/types.py`, поскольку он реализует резольвер типов.

**Листинг 10.11.** Сериализация и парсинг кастомного скаляра

```
# file: web/types.py

import uuid
from datetime import datetime
```



```
from ariadne import UnionType, ScalarType
```

```
...
```

```
datetime_scalar = ScalarType('Datetime')
```

Создаем привязываемый объект для скаляра  
Datetime, используя класс ScalarType

```
@datetime_scalar.serializer
```

Привязываем сериализатор Datetime  
с помощью декоратора serializer()

Сериализуем  
объект date

```
def serialize_datetime_scalar(date):
```

```
    return date.isoformat()
```

Фиксируем аргумент  
сериализатора как дату

Привязываем парсер Datetime,  
используя декоратор  
value\_parser()

```
@datetime_scalar.value_parser
```

```
def parse_datetime_scalar(date):
```

Фиксируем аргумент парсера

```
    return datetime.fromisoformat(date)
```

Парсируем дату

Чтобы включить резольверы Datetime, мы добавляем datetime\_scalar в массив привязываемых объектов для функции make\_executable\_schema() в web/schema.py:

```
from pathlib import Path
```

```
from ariadne import make_executable_schema
```

```
from web.mutations import mutation
```

```
from web.queries import query
```

```
from web.types import product_type, datetime_scalar
```

```
schema = make_executable_schema(
    (Path(__file__).parent / 'products.graphql').read_text(),
    [query, mutation, product_type, datetime_scalar]
)
```

Проверим новые резольверы! Вернитесь в Apollo Playground, запущенный на <http://127.0.0.1:8000>, и выполните следующий запрос:

```
# Документ запроса
```

```
{
  allProducts {
    ...on ProductInterface {
      name,
      lastUpdated
    }
  }
}
```

```
# Результат
```

```
{
  "data": {
    "allProducts": [
      {
        "name": "Walnut Bomb",
        "lastUpdated": "2022-06-19T18:27:53.171870"
      },
      {
        "name": "Cappuccino Star",
```

```

        "lastUpdated": "2022-06-19T18:27:53.171871"
      }
    ]
  }
}

```

Вы должны получить список всех товаров с их названиями и датой в формате ISO в поле `lastUpdated`. Теперь у вас есть возможность реализовать собственные кастомные скалярные типы в GraphQL. Используйте ее с умом! Но прежде, чем завершить главу, рассмотрим еще одну тему: реализацию резольверов для полей объектного типа.

### 10.4.8. Реализация резольверов полей

В этом подразделе вы научитесь реализовывать резольверы для полей объектного типа. Мы реализовали почти все резольверы, необходимые нам для выполнения всевозможных запросов к API сервиса продукции, но есть еще один тип запросов, который наш сервер не может преобразовать: запросы с полями, которые сопоставляются с другими типами GraphQL. Например, тип `Products` имеет поле `ingredients`, которое сопоставляется с массивом объектов `IngredientRecipe`. Согласно спецификации тип `IngredientRecipe` описывается следующим образом:

```
# file: web/products.graphql
```

```

type IngredientRecipe {
  ingredient: Ingredient!
  quantity: Float!
  unit: String!
}

```

Каждый объект `IngredientRecipe` имеет поле `ingredient`, которое соответствует объектному типу `Ingredient`. Это означает, что при запросе поля `ingredient` товара мы должны иметь возможность получить информацию о каждом ингредиенте, например его название, описание или сведения о поставщике. Другими словами, мы должны иметь возможность выполнить следующий запрос к серверу:

```

{
  allProducts {
    ...on ProductInterface {
      name,
      ingredients {
        quantity,
        unit,
        ingredient{
          name
        }
      }
    }
  }
}

```

Если вы выполните этот запрос в Apollo Playground сейчас, то получите такое сообщение об ошибке: `Cannot return null for non-nullable field Ingredient.name` (Не удастся вернуть значение `null` для поля `Ingredient.name`).

Почему это происходит? Если вы посмотрите на список товаров в листинге 10.5, то заметите, что поле `ingredients` соответствует массиву объектов с тремя полями: `ingredient`, `quantity` и `unit`. Например, ореховая бомбочка содержит следующие ингредиенты:

```
# file: web/data.py

ingredients = [
    {
        'ingredient': '602f2ab3-97bd-468e-a88b-bb9e00531fd0',
        'quantity': 100.00,
        'unit': 'LITRES',
    }
]
```

Поле `ingredient` сопоставляется с ID ингредиента, а не с объектом целиком. Это наше внутреннее представление ингредиентов товара. Так мы храним данные о товаре в нашей базе данных (список в памяти в текущей реализации). И это удобно, поскольку мы можем идентифицировать каждый ингредиент по ID. Однако спецификация API говорит нам, что поле `ingredients` должно сопоставляться с массивом объектов `IngredientRecipe` и что каждый ингредиент должен представлять собой объект `Ingredient`, а не просто ID.

Как решить эту проблему? Мы можем использовать различные подходы. В частности, убедиться, что полезная нагрузка каждого ингредиента правильно сформирована в резольверах для каждого запроса, который возвращает тип `Product`. Например, в листинге 10.12 показано, как можно изменить резольвер `allProducts()`. В этом фрагменте кода мы изменим свойство `ingredients` каждого товара, чтобы увидеть, что оно содержит полную полезную нагрузку ингредиентов. Поскольку каждый товар представлен словарем, мы делаем глубокую копию каждого товара, чтобы убедиться, что изменения, которые мы внесем в эту функцию, не повлияют на наш список товаров в памяти.

**Листинг 10.12.** Обновление товаров, чтобы они содержали полную полезную нагрузку, а не только ID

```
# file: web/queries.py

...

@query.field('allProducts')
def resolve_all_products(*_):
    products_with_ingredients = [deepcopy(product) for product in products]
    for product in products_with_ingredients:
        for ingredient_recipe in product['ingredients']:
```

Делаем глубокую  
копию каждого объекта  
в списке товаров

```

for ingredient in ingredients:
    if ingredient['id'] == ingredient_recipe['ingredient']:
        ingredient_recipe['ingredient'] = ingredient
return products_with_ingredients

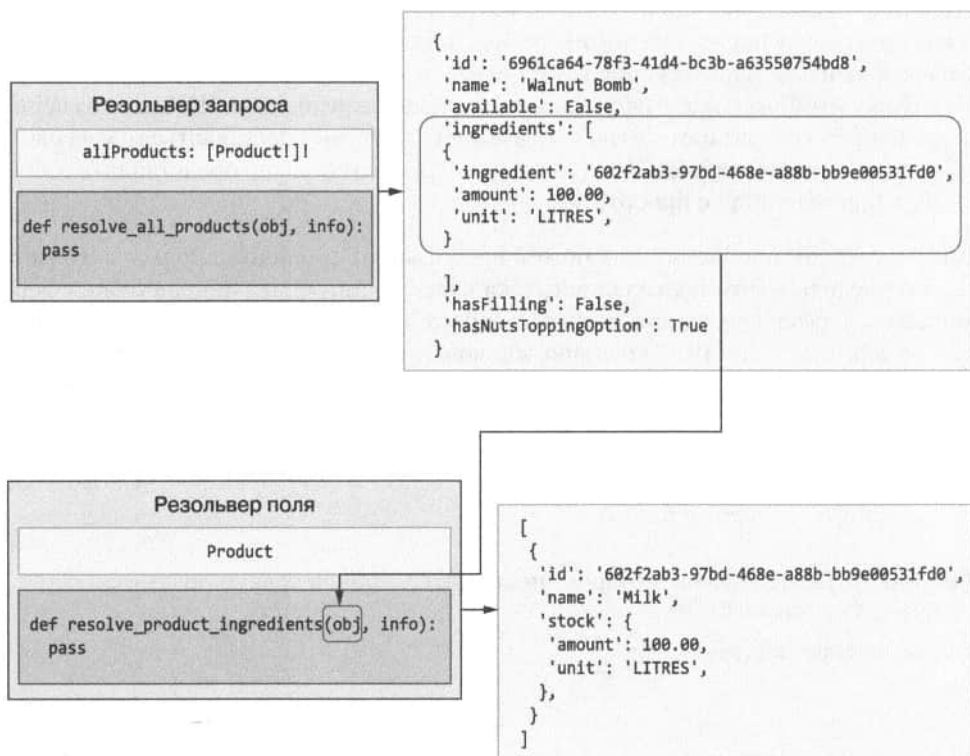
```

Возвращаем список товаров с ингредиентами

Обновляем свойство ingredient, предоставляя полную информацию об ингредиенте

Описанный подход хорош, но, как видите, при его использовании код усложняется. Если бы нам пришлось сделать то же самое для нескольких свойств, функция быстро стала бы сложной для понимания и сопровождения.

Как показано на рис. 10.14, GraphQL предлагает альтернативный способ преобразования свойств объекта. Вместо того чтобы изменять полезную нагрузку товара в резольвере `allProducts()`, мы можем создать специальный резольвер для свойства `ingredients` товара и внести в него все необходимые изменения. В листинге 10.13 приводится код резольвера для свойства `ingredients`. Он находится в файле `web/types.py`, поскольку реализует резольвер для свойств объекта.



**Рис. 10.14.** GraphQL позволяет создавать резольверы для определенных полей объекта. В этом примере резольвер `resolve_product_ingredients()` возвращает валидную полезную нагрузку для свойства `ingredients` товара

**Листинг 10.13.** Реализация резольвера поля

```
# file: web/types.py
```

```
...
```

```
@product_interface.field('ingredients')
def resolve_product_ingredients(product, _):
    recipe = [
        copy.copy(ingredient)
        for ingredient in product.get("ingredients", [])
    ]

    for ingredient_recipe in recipe:
        for ingredient in ingredients:
            if ingredient['id'] == ingredient_recipe['ingredient']:
                ingredient_recipe['ingredient'] = ingredient
    return recipe
```

Создаем точную  
копию каждого  
ингредиента

Резольверы свойств объектов помогают нам сделать код более модульным, поскольку каждый резольвер выполняет только одну задачу. Они также позволяют избежать повторений. Имея один резольвер, который обновляет свойство `ingredients` в полезной нагрузке товара, мы избегаем необходимости выполнять эту операцию в каждом резольвере, возвращающем тип `Product`. С другой стороны, резольверы свойств могут быть более сложными для отслеживания и отладки. Если что-то не так с полезной нагрузкой `ingredients`, вы не сможете найти ошибку в резольвере `allProducts()`. Вам нужно знать, где находится резольвер для ингредиентов товаров, и изучить его. Журналы приложений помогут указать вам верное направление при отладке такого рода проблем, но имейте в виду, что этот вариант не будет очевиден для других разработчиков, не знакомых с GraphQL. Как и во всем остальном при разработке программного обеспечения, убедитесь, что возможность повторного использования кода не ухудшает его читаемость и простоту обслуживания.

## РЕЗЮМЕ

- Экосистема Python предлагает различные фреймворки для реализации GraphQL API. Последние новости о доступных фреймворках можно найти на официальном сайте GraphQL: <https://graphql.org/code/>.
- Вы можете использовать фреймворк Ariadne для реализации GraphQL API, следуя подходу `schema-first`, что означает, что сначала мы разрабатываем API, а затем реализуем сервер в соответствии со спецификацией. Этот подход позволяет командам разработчиков сервера и клиента работать параллельно.
- Ariadne может автоматически проверять полезную нагрузку запросов и ответов, используя спецификацию, то есть вам не нужно тратить время на внедрение кастомных моделей валидации.

- Для каждого запроса и мутации в спецификации API необходимо реализовать резольвер. Резольвер — это функция, которая знает, как обработать данный запрос или мутацию.
- Для привязки резольвера мы используем один из привязываемых классов Ariadne, например `QueryType` или `MutationType`. Они предоставляют соответствующие декораторы, которые позволяют нам привязать функцию резольвера.
- Спецификации GraphQL могут содержать сложные типы-объединения, которые объединяют два объектных типа или более. Если спецификация API содержит тип-объединение, нам нужно реализовать резольвер, который знает, как определить тип объекта; в противном случае сервер GraphQL не сумеет его преобразовать.
- В GraphQL мы можем определять кастомные скаляры. Если спецификация содержит кастомный скаляр, следует реализовать резольверы, которые знают, как сериализовать, парсировать и проверять этот тип; в противном случае сервер GraphQL не будет знать, как обращаться с этими скалярами.

## *Часть IV*

# *Защита, тестирование и развертывание API микросервисов*

Как вы узнали из главы 1, API являются программными интерфейсами для приложений, и, публикуя свои API, мы позволяем другим организациям создавать интеграции с ними. API открывают новые возможности для бизнеса, но в то же время представляют угрозу безопасности. Отсутствие надлежащего тестирования или неправильно внедренные протоколы безопасности делают наши API уязвимыми. В части IV вы познакомитесь с основными принципами тестирования, безопасности и эксплуатации API.

В главе 11 мы рассмотрим стандарты OpenID Connect и Open Authorization (OAuth) 2.1, используемые для аутентификации и авторизации API. По моему опыту, это одна из наиболее неправильно понимаемых областей разработки API, в которой высок риск уязвимостей и нарушений безопасности. В главе 11 вы узнаете все, что нужно знать для реализации надежной стратегии аутентификации и авторизации ваших API.

Когда вы проводите интеграцию с использованием API, вам нужен надежный метод их тестирования и валидации. Вы должны убедиться, что серверный API обслуживает интерфейс, определенный в вашей спецификации. Как это сделать? Как вы узнаете из главы 12, лучше всего использовать инструменты контрактного тестирования, такие как Dredd и Schemathesis, и выполнять тестирование на основе свойств. Так вы сможете тщательно протестировать и проверить свой код, прежде чем запускать его в производство.

И наконец, как насчет развертываний и операций? В заключительных главах этой книги вы узнаете, как контейнеризовать и развернуть API микросервисов с помощью Kubernetes. Вы научитесь развертывать кластер Kubernetes и управлять им с помощью Elastic Kubernetes Service (EKS) AWS, одного из самых популярных решений для запуска Kubernetes в облаке. После прочтения части IV вы будете готовы к масштабному тестированию, защите и эксплуатации своих API микросервисов.



# 11

## Авторизация и аутентификация API

---

### В этой главе

- ✓ Использование открытой авторизации для предоставления доступа к API.
- ✓ Использование OpenID Connect для проверки личности пользователей API.
- ✓ Какие типы схем авторизации существуют и какая больше подходит для каждого сценария авторизации.
- ✓ Понимание веб-токенов JSON (JWT) и использование библиотеки PyJWT на Python для их создания и проверки.
- ✓ Добавление в API промежуточного слоя аутентификации и авторизации.

В 2018 году в системе аутентификации API почтовой системы США (<https://usps.com>) была обнаружена уязвимость, которая позволила хакерам получить данные 60 миллионов пользователей: их адреса электронной почты, номера телефонов и другие личные сведения<sup>1</sup>. Атаки на безопасность API, подобные этой, возникают все чаще,

<sup>1</sup> Первым о проблеме сообщил Брайан Кребс (Brian Krebs) в своей статье [USPS Site Exposed Data on 60 Million Users](https://krebsonsecurity.com/2018/11/usps-site-exposed-data-on-60-million-users/) («Сайт USPS раскрыл данные о 60 миллионах пользователей») // KrebsOnSecurity, 21 ноября 2018 г. <https://krebsonsecurity.com/2018/11/usps-site-exposed-data-on-60-million-users/>.

и, по оценкам, в 2021 году количество атак возрастет более чем на 300 %<sup>1</sup>. Из-за уязвимостей API вы рискуете не только раскрыть конфиденциальные данные пользователей; они также могут привести к закрытию вашего бизнеса!<sup>2</sup> Хорошая новость заключается в том, что не все потеряно и можно снизить риск взлома API. Первая линия защиты — это надежная система аутентификации и авторизации. В этой главе вы узнаете, как предотвратить несанкционированный доступ к API с помощью стандартных протоколов аутентификации и авторизации.

По моему опыту, аутентификация и авторизация API — две самые сложные темы для разработчиков, и в этих областях часто случаются ошибки при реализации. Я настоятельно рекомендую вам, прежде чем внедрять уровень безопасности вашего API, прочитать эту главу, чтобы убедиться, что вы знаете, что делаете, и знаете, как это делать правильно. Я сделал все возможное, чтобы подробно описать суть аутентификации и авторизации API, и к концу этой главы вы сможете добавить поддержку авторизации в собственные API.

Аутентификация — это процесс проверки личности пользователя, в то время как авторизация — процесс определения того, есть ли у пользователя доступ к определенным ресурсам или операциям. Принципы и стандарты аутентификации и авторизации, с которыми вы познакомитесь в этой главе, применимы ко всем типам веб-API.

Вы узнаете о различных протоколах и схемах аутентификации и авторизации, а также о том, как проверять токены авторизации. Вы также научитесь использовать библиотеку PyJWT в Python для создания подписанных токенов и их проверки. Мы рассмотрим пример добавления аутентификации и авторизации в API сервиса заказов. Нам предстоит многое обсудить, так что давайте начнем!

## 11.1. НАСТРОЙКА СРЕДЫ ДЛЯ ПРИМЕРОВ ЭТОЙ ГЛАВЫ

Код для этой главы доступен в каталоге `ch11` в репозитории GitHub для книги. В главе 7 мы внедрили полнофункциональный сервис заказов, дополненный уровнем бизнес-логики, базой данных и API. В этой главе мы начнем работать с ним в том виде, в каком оставили его в главе 7. Для этого предлагаю вам скопировать код из главы 7 в новую папку `ch11`:

```
$ cp -r ch07 ch11
```

<sup>1</sup> *Doerfeld B.* API Attack Traffic Grew 300+% In the Last Six Months («Трафик атак через API вырос более чем на 300 % за последние шесть месяцев») // Security Boulevard, 30 июля 2021 года. <https://securityboulevard.com/2021/07/api-attack-traffic-grew-300-in-the-last-six-months/>.

<sup>2</sup> *Galvin J.* 60 Percent of Small Businesses Fold Within 6 Months of a Cyber Attack («60 процентов малых предприятий сворачиваются в течение 6 месяцев после кибератаки») // Inc., 7 мая 2018 года. <https://www.inc.com/joe-galvin/60-percent-of-small-businesses-fold-within-6-months-of-a-cyber-attack-heres-how-to-protect-yourself.html>.

Перейдите в `ch11` и установите зависимости, запустив `pipenv install`. Для этой главы вам понадобится несколько дополнительных зависимостей, поэтому выполните следующую команду, чтобы установить их:

```
$ pipenv install cryptography pyjwt
```

`PyJWT` — это библиотека Python, которая позволяет работать с веб-токенами JSON, в то время как `cryptography` позволит проверять подписи токенов. (Список альтернативных библиотек JWT в экосистеме Python смотрите здесь: <https://jwt.io/libraries?language=Python>.)

Теперь среда готова, так что начнем наше путешествие по удивительному миру аутентификации и авторизации пользователей. Это путешествие, полное подводных камней, но необходимое.

## 11.2. ПРОТОКОЛЫ АУТЕНТИФИКАЦИИ И АВТОРИЗАЦИИ

Когда дело доходит до аутентификации API, важно знать два главных протокола: OAuth (Open Authorization) и OpenID Connect (OIDC). В этом разделе я объясню, как работает каждый протокол и как они вписываются в схемы аутентификации и авторизации для наших API.

### 11.2.1. Что такое Open Authorization

OAuth — это стандартный протокол для предоставления доступа<sup>1</sup>. Как вы можете видеть на рис. 11.1, OAuth позволяет пользователю предоставлять стороннему приложению доступ к защищенным ресурсам на другом сайте, которыми он владеет, без необходимости указывать свои учетные данные.

#### ОПРЕДЕЛЕНИЕ

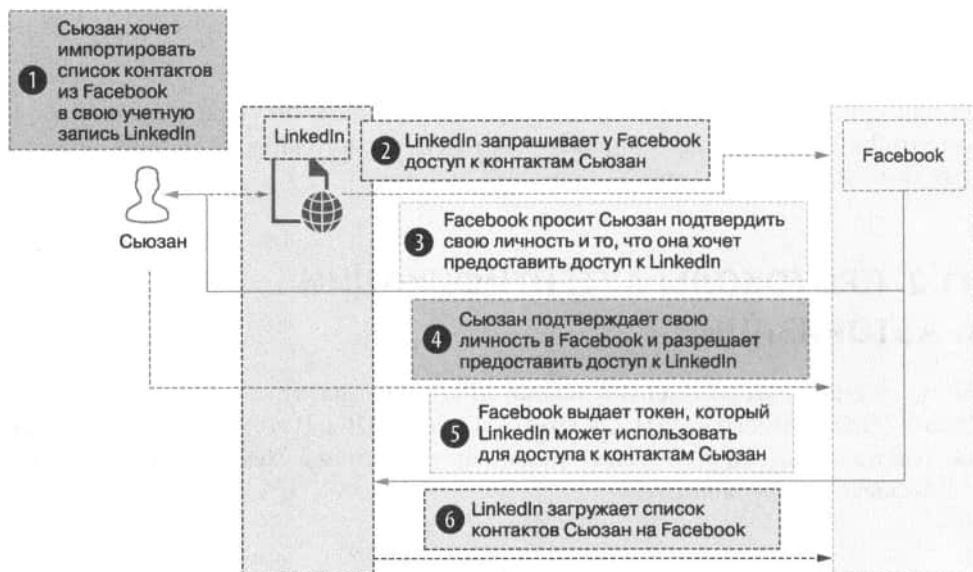
*OAuth* — это открытый стандарт, который позволяет пользователям предоставлять сторонним приложениям доступ к своей информации на других сайтах. Как правило, доступ предоставляется путем выдачи токена, который стороннее приложение использует для доступа к информации пользователя.

Предположим, у Сьюзан есть список контактов в ее аккаунте Facebook. Она заходит в LinkedIn и хочет импортировать свой список контактов из Facebook. Чтобы разрешить LinkedIn импортировать ее контакты с Facebook, Сьюзан должна предоставить LinkedIn доступ к этому ресурсу. Как она может дать LinkedIn доступ

---

<sup>1</sup> <https://oauth.net/> — довольно хороший сайт с множеством ресурсов, позволяющих больше узнать о спецификации OAuth.

к своему списку контактов? Она могла бы предоставить LinkedIn свои учетные данные на Facebook. Но это было бы серьезной угрозой безопасности. Вместо этого OAuth определяет протокол, через который Сьюзан сообщает Facebook, что LinkedIn может получить доступ к ее списку контактов. С помощью OAuth Facebook выдает временный токен, который LinkedIn может использовать для импорта контактов Сьюзан.



**Рис. 11.1.** С помощью OAuth пользователь может предоставить стороннему приложению доступ к своей информации на другом сайте

В процессе предоставления доступа к ресурсу OAuth различает такие роли.

- *Владелец ресурса* — пользователь, предоставляющий доступ к ресурсу. В нашем примере это Сьюзан.
- *Сервер ресурсов* — сервер, на котором размещены защищенные ресурсы пользователя. В нашем примере это Facebook.
- *Клиент* — приложение или сервер, запрашивающий доступ к ресурсам пользователя. В нашем примере это LinkedIn.
- *Сервер авторизации* — сервер, который предоставляет клиенту доступ к ресурсам. В примере это Facebook.

OAuth предлагает четыре различных схемы (flows) предоставления авторизации пользователю в зависимости от условий доступа. Важно знать, как работает каждый вариант и в каких случаях вы можете его использовать. По моему опыту, схемы OAuth являются одной из самых больших проблем, связанных с авторизацией,

и одним из самых больших источников проблем безопасности на современных сайтах. Это схемы OAuth:

- схема с кодом авторизации;
- схема PKCE;
- схема клиентских полномочий;
- схема с обновляемым токеном.

## **OAuth**

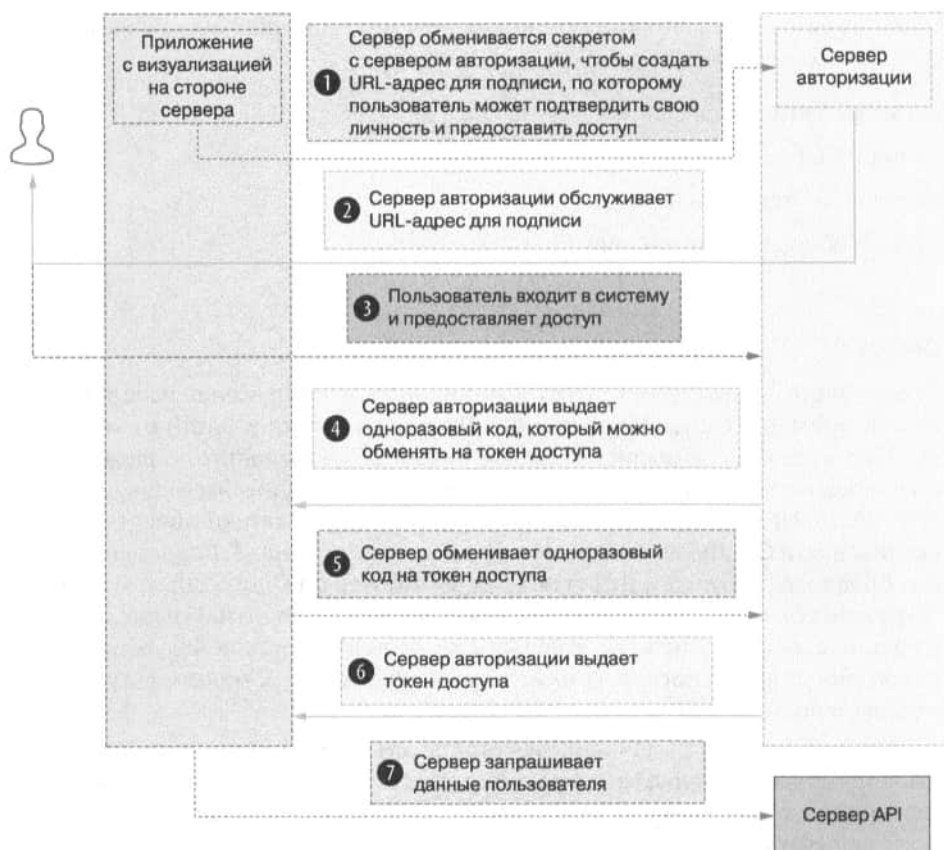
Схемы OAuth — это стратегии, которые клиентское приложение использует для авторизации своего доступа к API. Лучшие практики в OAuth меняются по мере того, как специалисты больше узнают об уязвимостях приложений и совершенствуют протокол. Текущие рекомендации описаны в документе *IETF OAuth 2.0 Security Best Current Practice* («Лучшие практики обеспечения безопасности OAuth 2.0»). (<http://mng.bz/o58v>) за авторством Т. Лоддерштедта, Дж. Брэдли, А. Лабунца и Д. Фетта. Если вы читаете об OAuth 2.0, то можете встретить ссылки на две схемы, которые я не описываю в этой главе: схему с предоставлением клиенту пароля и схему неявного доступа. Оба варианта на сегодня устарели, поскольку их использование чревато серьезными уязвимостями, и поэтому вам не следует их применять.

Другим популярным расширением, которое мы не обсуждаем в этой главе, является предоставление авторизации устройству (<http://mng.bz/5mZD>), при котором устройства с ограниченным вводом данных, такие как смарт-телевизоры, могут получать токены доступа. Последней версией OAuth является 2.1, которая описана в документе *IETF The OAuth 2.1 Authorization Framework* («Система авторизации OAuth 2.1») (<http://mng.bz/69m6>).

Давайте подробнее рассмотрим каждую схему, чтобы понять, как они работают и когда их нужно использовать.

### **Схема с кодом авторизации**

В схеме с кодом авторизации клиентский сервер обменивается секретными данными с сервером авторизации для получения URL-адреса подписи. Как вы можете видеть на рис. 11.2, когда пользователь вошел в систему по этому URL-адресу, клиентский сервер получает одноразовый код, который может обменять на токен доступа. В этом методе используются конфиденциальные данные клиента, и, следовательно, он подходит только для тех приложений, в которых код не является общедоступным, например, для традиционных веб-приложений, где пользовательский интерфейс отображается в серверной части. OAuth 2.1 рекомендует использовать схему кода авторизации в сочетании со схемой PKCE, которая описана далее.



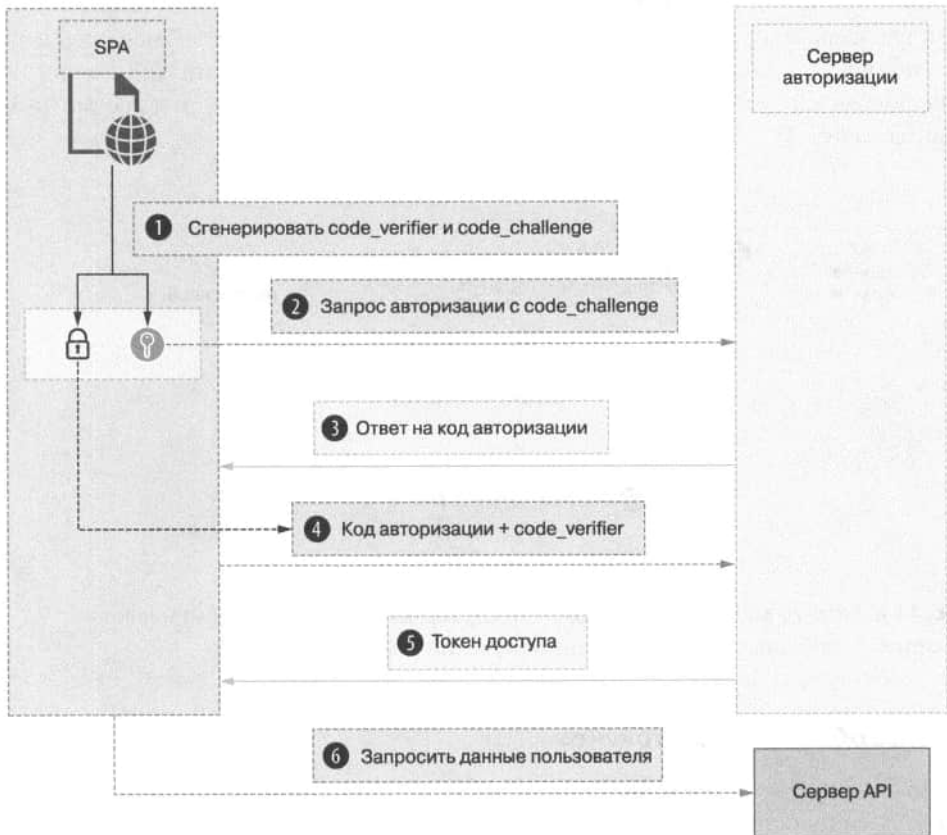
**Рис. 11.2.** В схеме с кодом авторизации сервер авторизации выдает URL-адрес для подписи, который пользователь может использовать для подтверждения своей личности и предоставления доступа к стороннему приложению

## Подтверждение ключа для обмена кодом

*Proof of Key for Code Exchange* (ПКСЕ, произносится как «пикси», подтверждение ключа для обмена кодом) — это расширение схемы с кодом авторизации, предназначенное для защиты приложений, исходный код которых является общедоступным, то есть мобильных и одностраничных приложений (SPA)<sup>1</sup>. Поскольку источник является общедоступным, клиент не может использовать секретные данные, ведь они также были бы общедоступны.

<sup>1</sup> Sakimura N., Bradley J., Agarwal N. Proof Key for Code Exchange by OAuth Public Clients («Пробный ключ для обмена кодом общедоступными клиентами OAuth») // IETF RFC 7636, сентябрь 2015 года. <https://datatracker.ietf.org/doc/html/rfc7636>.

Как вы можете видеть на рис. 11.3, в схеме PKCE клиент генерирует секрет, называемый *средством проверки кода* (*code verifier*), и кодирует его. Закодированный код называется *вызовом кода* (*code challenge*)<sup>1</sup>. При отправке запроса на авторизацию на сервер клиент включает в запрос как средство проверки кода, так и вызов кода. В ответ сервер выдает код авторизации, который клиент может обменять на токен доступа. Для этого он должен отправить как код авторизации, так и вызов кода.



**Рис. 11.3.** В схеме PKCE SPA, обслуживаемый клиентом, запрашивает доступ к данным пользователя непосредственно с сервера авторизации путем обмена `code_verifier` и `code_challenge`

<sup>1</sup> `Code_verifier` — случайное число, которое используется только один раз. Для каждого запроса на получение кода клиент должен генерировать новый `code_verifier`. `Code_challenge_method` — название функции преобразования, чаще всего SHA-256. `Code_challenge` — это `code_verifier`, к которому применили преобразование `code_challenge_method` и закодировали в URL Safe Base64. — *Примеч. ред.*

Благодаря `code_challenge` схема PKCE также позволяет предотвращать атаки с внедрением кода авторизации, при которых злоумышленник перехватывает код авторизации и использует его для получения токена доступа. Из-за преимуществ этой схемы PKCE также рекомендуется для серверных приложений. Мы рассмотрим пример этой схемы с использованием SPA в приложении В.

### Схема клиентских полномочий

Схема клиентских полномочий (client credentials) предназначена для взаимодействия сервера с сервером и, как видно из рис. 11.4, предполагает обмен секретом для получения токена доступа. Эта схема подходит для организации взаимодействия между микросервисами по защищенной сети. Пример мы рассмотрим в приложении В.



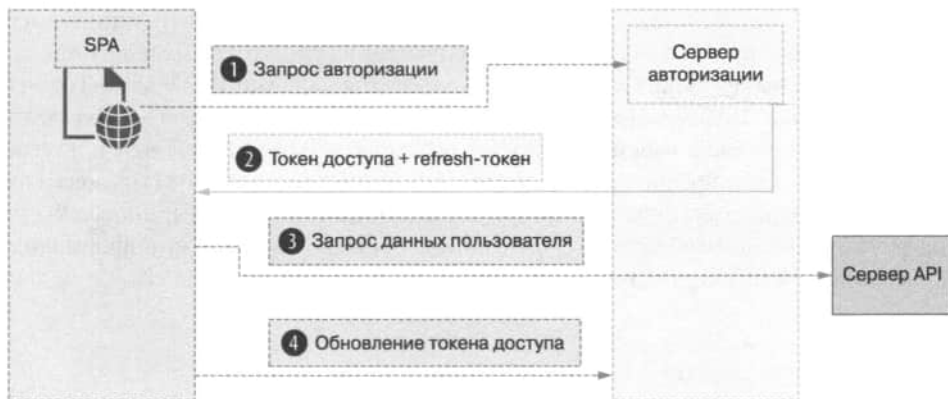
**Рис. 11.4.** В схеме клиентских полномочий серверное приложение обменивается секретом с сервером авторизации для получения токена доступа

### Схема с обновляемым токеном

Схема с обновляемым токеном позволяет клиентам обменивать `refresh`-токен на новый токен доступа. По соображениям безопасности токены доступа действительны в течение ограниченного периода времени. Однако клиентам API часто нужно взаимодействовать с сервером API после истечения срока действия токена доступа, и для получения нового токена они используют эту схему.

Как вы можете видеть на рис. 11.5, клиенты API обычно получают как токен доступа, так и `refresh`-токен, когда успешно получают доступ к API. `Refresh`-токены нередко действительны в течение ограниченного периода времени и только для однократного использования. Каждый раз, когда вы обновляете свой токен доступа, вы получаете новый `refresh`-токен.





**Рис. 11.5.** Чтобы разрешить клиентам API использовать refresh-токены для продолжения связи с сервером API после истечения срока действия токена доступа, сервер авторизации выдает новый refresh-токен каждый раз, когда клиент запрашивает новый токен доступа

Теперь, когда мы понимаем, как работает OAuth, рассмотрим OpenID Connect.

### 11.2.2. Что такое OpenID Connect

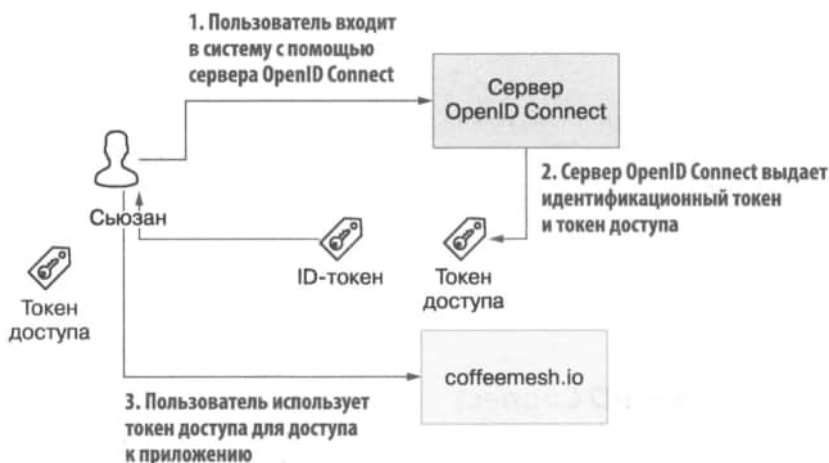
OpenID Connect (OIDC) — это открытый стандарт для проверки личности, построенный поверх OAuth. Как вы можете видеть на рис. 11.6, OIDC позволяет пользователям проходить аутентификацию на сайте с помощью стороннего поставщика удостоверений (identity provider). Если вы использовали свою учетную запись Facebook, Twitter или Google для входа на другие сайты, то уже сталкивались с OIDC. В данном случае Facebook, Twitter и Google являются поставщиками удостоверений. Вы используете их, чтобы перенести свою идентификационную информацию на новый сайт. OIDC — это удобная система аутентификации, поскольку позволяет пользователям применять одни и те же идентификационные данные на разных сайтах, а не создавать новые логины и пароли и управлять ими.

#### ОПРЕДЕЛЕНИЕ

*OpenID Connect (OIDC)* — это протокол проверки подлинности, который позволяет пользователям передавать свои данные с одного сайта (поставщика идентификационных данных) на другой. OIDC построен поверх OAuth, и мы можем использовать схемы, определенные OAuth.

Поскольку OIDC построен поверх OAuth, мы можем использовать любую из схем авторизации, описанных в предыдущем разделе, для аутентификации и авторизации

пользователей. Как вы можете видеть на рис. 11.6, когда мы аутентифицируемся с использованием протокола OIDC, мы различаем два типа токенов: идентификационные и токены доступа. Оба токена выпускаются в форме JSON Web Tokens, но служат разным целям. *ID-токены* идентифицируют пользователя и содержат такую информацию, как имя пользователя, его адрес электронной почты и другие личные данные. Они предназначены только для проверки личности пользователя и никогда для определения того, имеет ли пользователь доступ к API. Доступ к API проверяется с помощью *токенов доступа*, которые обычно содержат не информацию о пользователе, а набор утверждений о правах доступа пользователя.



**Рис. 11.6.** Пользователь входит в систему с помощью сервера OIDC, который выдает идентификационный токен и токен доступа. Их пользователь может использовать для доступа к приложению

## ID-ТОКЕНЫ ПРОТИВ ТОКЕНОВ ДОСТУПА

Распространенная проблема безопасности — неправильное использование ID-токенов и токенов доступа. ID-токены содержат идентификационную информацию пользователя. Они должны использоваться для проверки исключительно его личности, а не доступа к API. Доступ к API проверяется с помощью токенов доступа. Токены доступа редко содержат идентификационные данные пользователя, а вместо этого содержат утверждения о правах пользователя на доступ к API. Фундаментальным различием между токенами ID и токенами доступа является аудитория: у токена ID это сервер авторизации, в то время как у токена доступа это будет наш сервер API.

Поставщики удостоверений, предлагающие интеграцию с OIDC, предоставляют эндпоинт `/well-known/openid-configuration`, также известную как *эндпоинт обнаружения (discovery endpoint)*. Он сообщает потребителю API, как пройти аутентификацию и получить свои токены доступа. Например, хорошо известным эндпоинтом OIDC для аккаунтов Google является <https://accounts.google.com/.well-known/>

`openid-configuration`. Если вы вызовете этот эндпоинт, то получите следующую полезную нагрузку (пример приводится в сокращенном виде):

```
{
  "issuer": "https://accounts.google.com",
  "authorization_endpoint": "https://accounts.google.com/o/oauth2/v2/auth",
  "device_authorization_endpoint":
➔ "https://oauth2.googleapis.com/device/code",
  "token_endpoint": "https://oauth2.googleapis.com/token",
  "userinfo_endpoint": "https://openidconnect.googleapis.com/v1/userinfo",
  "revocation_endpoint": "https://oauth2.googleapis.com/revoke",
  "jwks_uri": "https://www.googleapis.com/oauth2/v3/certs",
  "response_types_supported": [
    "code",
    "token",
    "id_token",
    "code token",
    "code id_token",
    "token id_token",
    "code token id_token",
    "none"
  ],
  ...
}
```

Как видите, эндпоинт сообщает нам, какой URL мы должны использовать при получении токена доступа для авторизации, какой URL возвращает информацию о пользователе или какой URL мы используем для отмены токена доступа. В этой полезной нагрузке есть и другие единицы информации, например доступные заявки (`claims`) или JSON Web Keys URI (JWKS). Как правило, пользователи задействуют библиотеку для обработки этих эндпоинтов от своего имени или поставщика удостоверений личности, чтобы позаботиться об этих интеграциях. Если вы хотите больше узнать об OpenID Connect, рекомендую прочесть книгу Прабата Сиравардены *OpenID Connect in Action* (Manning, 2022).

Теперь, когда вы знаете принципы работы OAuth и OpenID Connect, пришло время подробнее поговорить о том, как работают аутентификация и авторизация. Начнем с изучения JSON Web Tokens.

## 11.3. РАБОТА С JSON WEB TOKENS

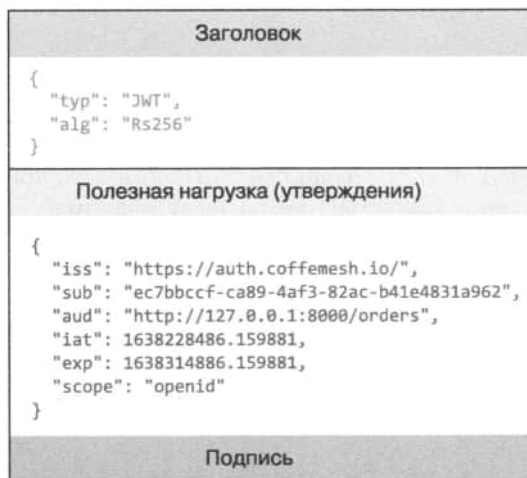
В OAuth и OpenID Connect доступ пользователя проверяется с помощью токена, известного как *JSON Web Token*, или *JWT*. В этом разделе объясняется, что такое JSON-токены, как они структурированы, какие типы полномочий подразумевают и как их создавать и проверять.

JWT — это токен, представляющий собой документ в формате JSON. Он содержит утверждения (`claims`): кто выдал токен, аудитория токена или когда истекает срок его действия. Документ JSON обычно кодируется как строка Base64. JWT

подписываются личным секретом или криптографическим ключом<sup>1</sup>. Типичный веб-токен JSON выглядит следующим образом:

```
eyJ0eXAiOiJKV1QiLCJhbGciOiJSUzI1NiJ9.eyJpc3MiOiJodHRwczovL2F1dGguY29mZmVlbw
➔ VzaC5pby8iLCJzdWIiOiJlYzdiYmNjZi1jYTg5LTRhZjMtODJhYy1lNDFlNDgzMWE5NjIiL
➔ CJhdWQiOiJodHRwOi8vMTIzLjAuMC4xOjgwMDAvb3JkZXJzIiwiaWF0IjoxNjM4MjI4NDg2
➔ LjE10tg4MSwiZXBhIjoxNjM4MzE0ODg2LjE10tg4Mswic2NvcGUiOiJvcGVuawQifQ.oblJ
➔ 5wV9GqrhIDzNSzc1rpEQTMK8hZGzn1S707tDtQE__OCDS9J2Wa70aBua6X81-
➔ zrvWBfzrcX--nSyT-
➔ A9uQxL5j3RHHycToqSVi87I9H6jgP4FEKH6ClwZfabVwzNIy5ZzS7zRdcSI4WRz10pHoCM-
➔ 2hNtZ67dMJQgBVIIrXcwKAeKQWP8SxSDgFbwnyRTZJt6zjJrncJQqV4KrK_M4pv2UQYqf9t
➔ Qpj2uf1TsVcZq6XsrFLAgqvAg-YsIarYw9d63rs4H_I2aB3_T_1dGPY6ic2R8WDT1_Axzi-
➔ crjowq9A51SN-kMaTLhE_v2MSBB3A0zrjbdC4ZvusAqQ
```

Если вы внимательно посмотрите на пример, то увидите, что строка содержит две точки. Они действуют как разделители, которые отделяют каждый компонент JWT. Как вы можете видеть на рис. 11.7, документ JWT состоит из трех разделов.



**Рис. 11.7.** JWT состоит из трех частей: заголовка, который содержит информацию о самом токене, полезной нагрузки с утверждениями о доступе пользователя к сайту и подписи, подтверждающей подлинность токена

- *Заголовок* — идентифицирует тип токена, а также алгоритм и ключ, которые были использованы для подписи токена. Мы используем эту информацию, чтобы применить правильный алгоритм для проверки подписи токена.
- *Полезная нагрузка* — содержит набор утверждений документа. Спецификация JWT включает в себя список зарезервированных утверждений, которые иден-

<sup>1</sup> Полное описание того, как должны создаваться и проверяться JSON-токены, можно найти в статье: Jones J., Bradley J., Sakimura N. JSON Web Token (JWT) // RFC-7519, май 2015 года. <https://datatracker.ietf.org/doc/html/rfc7519>.

тифицируют издателя токена (сервер авторизации), аудиторию токена или предполагаемого получателя (наш сервер API) и среди прочего дату истечения срока его действия. В дополнение к стандартным утверждениям JWT полезная нагрузка может включать пользовательскую информацию. По ней мы определяем, имеет ли пользователь доступ к API.

- *Подпись* — строка, представляющая подпись токена.

Теперь, когда мы понимаем, что такое JWT и какова его структура, рассмотрим его свойства. В следующих разделах разбираются основные типы утверждений и свойств, которые можно найти в полезной нагрузке и заголовках JWT, и примеры их использования.

### 11.3.1. Заголовок JWT

JWT содержат заголовок, который описывает тип токена, а также алгоритм и ключ, используемые для подписи. JWT обычно подписываются с применением алгоритмов HS256 и RS256. HS256 использует секрет для шифрования токена, в то время как RS256 использует пару «закрытый/открытый ключ» для подписи токена. Попробуем на основе этой информации применить правильный алгоритм для проверки подписи токена.

#### АЛГОРИТМЫ ПОДПИСИ ДЛЯ JWT

HS256 и RS256 — два наиболее распространенных алгоритма, используемых для подписи JWT. HS256 расшифровывается как HMAC-SHA256, и это форма шифрования, в которой для создания хеша применяется ключ.

RS256 расшифровывается как RSA-SHA256. RSA (Rivest-Shamir-Adleman) — это форма шифрования, в которой для шифрования полезной нагрузки применяется закрытый ключ. Здесь мы можем проверить правильность подписи токена с помощью открытого ключа.

Больше о HMAC и RSA вы можете узнать в книге Дэвида Вонга «Реальная криптография»<sup>1</sup>.

Типичный заголовок JWT выглядит следующим образом:

```
{
  "alg": "RS256",
  "typ": "JWT",
  "kid": "ZweIFRR411dJlVPn0oZqf"
}
```

<sup>1</sup> Вонг Д. Реальная криптография. — СПб.: Питер, 2024.

Проанализируем его.

- `alg` — сообщает нам, что токен был подписан с использованием алгоритма RS256;
- `typ` — говорит о том, что это токен JWT;
- `kid` — сообщает нам, что ключ, используемый для подписи токена, имеет идентификатор `ZweIFRR411dJ1VPH0oZqf`.

Подпись токена может быть проверена только с использованием того же секрета или ключа, который применялся для его подписи. В целях безопасности мы часто используем набор секретов или ключей для подписи токенов. Поле `kid` сообщает нам, какой секрет или ключ использовать для подписи токена, чтобы мы могли выбрать правильное значение при проверке подписи токена.

Некоторые токены также содержат в заголовке поле `nonce`. Если вы видите один из этих токенов, скорее всего, он не предназначен для вашего сервера API, если только вы не являетесь создателем токена и не знаете, каково значение `nonce`. Это поле обычно хранит зашифрованный секрет, что добавляет дополнительный уровень безопасности к JWT. Например, токены, выданные Azure Active Directory для доступа к своему Graph API, содержат поле `nonce`, что означает, что вы не должны использовать их для авторизации доступа к своим пользовательским API.

### 11.3.2. Утверждения JWT

Полезная нагрузка JWT содержит набор утверждений. Поскольку полезная нагрузка JWT представляет собой документ JSON, утверждения представляются в виде пар «ключ — значение».

Существует два типа утверждений: *зарезервированные*, которые являются частью спецификации JWT, и *пользовательские*, которые мы можем добавить, чтобы добавить к токенам дополнительную информацию<sup>1</sup>. Спецификация JWT определяет семь зарезервированных утверждений.

- `iss` (издатель) — идентифицирует издателя JWT. Если вы используете поставщика удостоверений личности как услуги (*identity-as-a-service*), издатель идентифицирует эту услугу. Обычно представлено идентификатором или URL-адресом.
- `sub` (субъект) — идентифицирует субъекта JWT (то есть пользователя, отправляющего запрос на сервер). Обычно представлено в виде закрытого идентификатора, который не предоставляет личные данные пользователя.

<sup>1</sup> Вы можете ознакомиться с полным списком наиболее часто используемых утверждений JWT по адресу <https://www.iana.org/assignments/jwt/jwt.xhtml>.

- **aud** (аудитория) — указывает получателя, для которого предназначен JWT. Это наш сервер API. Обычно представлено в виде идентификатора или URL-адреса. Очень важно проверить это поле, чтобы подтвердить, что токен предназначен для наших API. Если мы не распознаем значение в этом поле, значит, токен не для нас и мы должны проигнорировать запрос.
- **exp** (время истечения срока действия) — временная метка UTC, которая указывает, когда истекает срок действия JWT. Запросы с истекшим сроком действия токенов должны быть отклонены.
- **nbf** (не раньше времени) — временная метка UTC, указывающая время, до наступления которого JWT не должен приниматься.
- **iat** (выдано по времени) — временная метка UTC, указывающая, когда было выдано JWT. Утверждение может быть использовано для определения возраста JWT.
- **jti** (JWT ID) — уникальный идентификатор для JWT.

Зарезервированные утверждения не обязательно приводить в полезной нагрузке JWT, но рекомендуется включать их для обеспечения взаимодействия со сторонними интеграциями (листинг 11.1).

#### Листинг 11.1. Пример утверждений в полезной нагрузке JWT

```
{
  "iss": "https://auth.coffeemesh.io/",
  "sub": "ec7bbccf-ca89-4af3-82ac-b41e4831a962",
  "aud": "http://127.0.0.1:8000/orders",
  "iat": 1667155816,
  "exp": 1667238616,
  "azp": "7c2773a4-3943-4711-8997-70570d9b099c",
  "scope": "openid"
}
```

Проанализируем утверждения, приведенные в листинге 11.1.

- **iss** сообщает нам, что токен был выпущен службой идентификации сервера компании <https://auth.coffeemesh.io>.
- **sub** сообщает нам, что идентификатор пользователя — **ec7bbccf-ca89-4af3-82ac-b41e4831a962**. Значение этого идентификатора принадлежит службе идентификации. Наши API могут использовать его для управления доступом к ресурсам, принадлежащим данному пользователю, неявным способом. Мы говорим, что этот идентификатор закрытый, поскольку он не раскрывает никакой личной информации о пользователе.
- **aud** говорит нам, что токен был выдан для предоставления доступа к API сервиса заказов. Если значением этого поля является другой URL-адрес, API отклонит запрос.
- **iat** сообщает нам, что токен был выпущен 30 октября 2022 года в 18:50 по Гринвичу.

- `exp` говорит нам, что срок действия токена истекает 31 октября 2022 года в 17:50 по Гринвичу.
- `azp` сообщает нам, что токен был запрошен приложением с идентификатором `7c2773a4-3943-4711-899770570d9b099c`. Обычно это интерфейсное приложение. Это утверждение часто встречается в токенах, которые были выпущены с использованием протокола OpenID Connect.
- `scope` сообщает нам, что токен был выпущен с использованием протокола OpenID Connect.

### 11.3.3. Создание JWT

Чтобы сформировать окончательный JWT, мы кодируем заголовок, полезную нагрузку и подпись, используя кодировку `base64url`. Как прописано в RFC 4648 (<http://mng.bz/aPRj>), кодировка `base64url` аналогична кодировке Base64, но в ней используются не алфавитно-цифровые символы и опускаются символы отступа. Заголовок, полезная нагрузка и подпись затем объединяются (разделителями служат точки). Библиотеки, подобные `PyJWT`, берут на себя тяжелую работу по созданию JWT. Допустим, мы хотим создать токен для полезной нагрузки, которую видели в листинге 11.1:

```
payload = {
    "iss": "https://auth.coffeemesh.io/",
    "sub": "ec7bbccf-ca89-4af3-82ac-b41e4831a962",
    "aud": "http://127.0.0.1:8000/orders",
    "iat": 1667155816,
    "exp": 1667238616,
    "azp": "7c2773a4-3943-4711-8997-70570d9b099c",
    "scope": "openid"
}
```

Чтобы создать подписанный токен с этой полезной нагрузкой, мы задействуем функцию `PyJWT encode()`, передавая токен, ключ для подписи токена и алгоритм, который мы хотим использовать для подписи токена:

```
>>> import jwt
>>> jwt.encode(payload=payload, key='secret', algorithm='HS256')
➔ 'eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJpc3MiOiJodHRwczovL2F1dGguY29mZmVlbnVzaC5pby8iLCJzdWUiOiJlYzdiYmNjZi1jYtTg5LTRhZjMtODJhYy1iNDFlNDgzMWE5NjIiLCJhdWQiOiJodHRwOi8vMTI3LjAuMC4xOjgwMDAvb3JkZXJzIiwiaWF0IjoxNjY3MTU1ODE2LCJleHAiOjE2NjcyMzg2MTYsImF6cCI6IjdlbmNjc3M2E0LTMSNDMTNDcxMS04OTk3LWtCWNTcwZDliMDk5YyIsInNjb3BlIjoib3BlbmklIn0.sZEXZVtCv0iVrbxGN54GJr8QecZfHA_pdvfEMZT1dI'
```

В этом случае мы подписываем токен секретным ключевым словом, используя алгоритм HS256. С целью более безопасного шифрования мы применяем пару секретных/открытых ключей для подписи токена с помощью алгоритма RS256. Для подписи JWT мы обычно используем сертификаты, соответствующие



стандарту X.509, который позволяет привязывать идентификатор к открытому ключу. Чтобы создать пару «закрытый/открытый ключ», запустите следующую команду со своего терминала:

```
$ openssl req -x509 -nodes -newkey rsa:2048 -keyout private_key.pem \
-out public_key.pem -subj "/CN=coffeemesh"
```

Минимальным вводимым значением для сертификата X.509 является общее имя субъекта (common name, CN), которое в нашем случае будет `coffeemesh`. Если вы опустите флаг `-subj`, то увидите ряд вопросов об удостоверении, к которому вы хотите привязать сертификат. Эта команда создает закрытый ключ в файле с именем `private_key.pem` и соответствующий открытый сертификат в файле `public_key.pem`. Если вы не можете выполнить эти команды, найдите пример пары ключей в репозитории GitHub, предоставленном вместе с книгой, в разделах `ch11/private_key.pem` и `ch11/public_key.pem`.

Теперь, получив пару «закрытый/открытый ключ», можем использовать их для подписи наших токенов и их проверки. Создайте файл с именем `jwt_generator.py` и вставьте в него код из листинга 11.2, в котором показано, как генерировать токены JWT, подписанные закрытым ключом. В списке определена функция `generate_jwt()`, которая генерирует JWT для полезной нагрузки, определенной внутри функции. В полезной нагрузке мы задаем свойства `iat` и `exp`: `iat` устанавливается на текущее время UTC; для `exp` устанавливается 24 часа с этого момента. Загружаем закрытый ключ, используя функцию `serialization()` и передавая в качестве параметров содержимое нашего файла закрытого ключа, закодированное в байтах, а также кодовую фразу, закодированную в байтах. Наконец, кодируем полезную нагрузку с помощью функции `PuJWT.encode()`, передавая полезную нагрузку, загруженный закрытый ключ и алгоритм, который мы хотим использовать для подписи токена (RS 256).

### Листинг 11.2. Генерация JWT, подписанных закрытым ключом

```
# file: jwt_generator.py

from datetime import datetime, timedelta
from pathlib import Path

import jwt
from cryptography.hazmat.primitives import serialization

def generate_jwt():
    now = datetime.utcnow()
    payload = {
        "iss": "https://auth.coffeemesh.io/",
        "sub": "ec7bbccf-ca89-4af3-82ac-b41e4831a962",
        "aud": "http://127.0.0.1:8000/orders",
        "iat": now.timestamp(),
        "exp": (now + timedelta(hours=24)).timestamp(),
        "scope": "openid",
    }
}
```

```
private_key_text = Path("private_key.pem").read_text()
private_key = serialization.load_pem_private_key(
    private_key_text.encode(),
    password=None,
)
return jwt.encode(payload=payload, key=private_key, algorithm="RS256")
```

```
print(generate_jwt())
```

Чтобы увидеть этот код в действии, активируйте свою виртуальную среду, запустив `pipenv shell`, и выполните следующую команду:

```
$ python jwt_generator.py
➔ eyJ0eXAiOiJKV1QiLCJhbGciOiJSUzI1NiJ9.eyJpc3MiOiJodHRwczovL2F1dGguY29mZm
➔ V1bWVzaC5pby8iLCJzdWIiOiJlYzdiYmNjZi1jYTg5LTRhZjMtODJhYy1iNDFlNDgzMWE5N
➔ jIiLCJhdwQiOiJodHRwOi8vMTI3LjAuMC4xOjgwMDAvb3JkZXJzIiwiaWF0IjoxNjM4MDMx
➔ LjgzOTY5ODczOTEsImV4cCI6MTYzODExOC4yMzk2OTg5OTMsInNjb3B1Ijoib3B1bm1kIn0
➔ .GipMvEvZG8ErMMA99geYUq5IkeWpRrnHoViLb1CkRuFqC5vgM9555re4IsLLa7yVxNAXIp
➔ FVFBqaoWrloJl6dSQ5r00dvUBSM1EM78KMZ7f0gQqUDFWNoKWcYQu1QCBzuHTouS41_mzz
➔ Ii75Sal3DJLTaj4zr6c_bQduUdU1GyrIOJiPSCHS1nKPgg9tjrx8e0cB_ESGSo9ipnCbpAl
➔ uWp0cDjPRPBNRuiU53sbl1-
➔ dTy7WoCD1mXAbqhztw039kG3DZBkysB4vTnKU4Eul2yNNYK2hHVZQEAvAq8TJjETUS7iekf
➔ 0NSt1qQArJ7cxg6Jh5D7y5pbKmYYsB1FohPg
```

Теперь вы знаете, как генерировать JWT. Генератор JWT из листинга 11.2 удобен для запуска тестов, и мы будем использовать его в следующих разделах для тестирования нашего кода. Далее посмотрим, как проверять полезную нагрузку JWT и как их проверять.

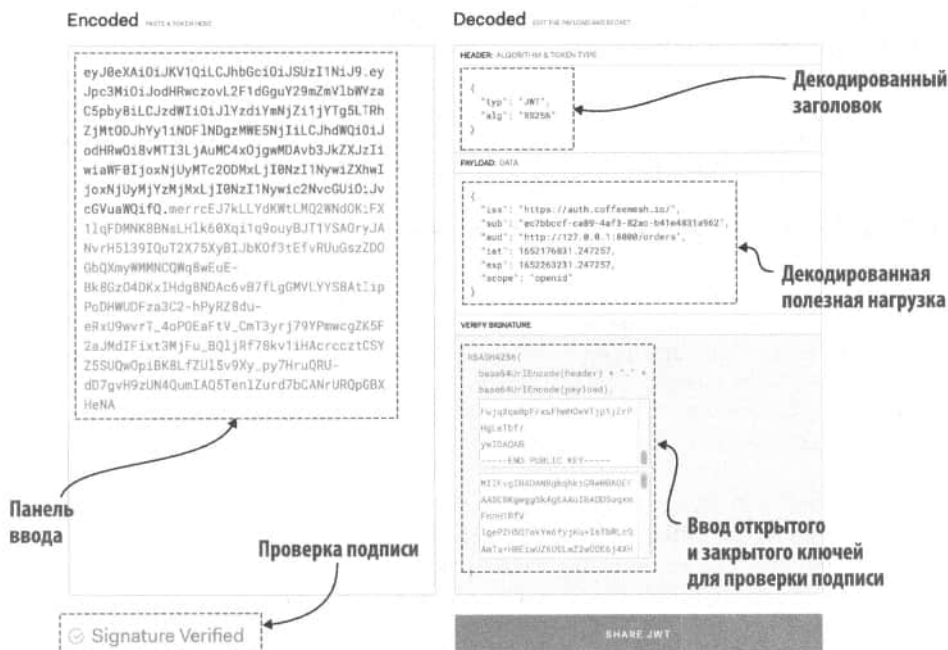
### 11.3.4. Проверка JWT

Часто при работе с JWT возникают проблемы с валидацией. Чтобы понять, почему проверка токена завершается неудачей, полезно проверить полезную нагрузку и убедиться в правильности содержащихся в ней утверждений. В этом разделе вы научитесь проверять JWT с помощью трех различных инструментов: `jwt.io` (<https://jwt.io>), команды терминала `base64` и `Python`. Чтобы опробовать эти инструменты, запустите скрипт `jwt_generator.py`, который мы создали в подразделе 11.3.3 для создания нового токена.

`jwt.io` — это отличный инструмент, который предлагает простой способ проверки JWT. Как вы можете видеть на рис. 11.8, вам нужно лишь вставить JWT на панель ввода слева. На панель отображения справа будут выведены содержимое заголовка токена и полезная нагрузка. Вы также можете проверить подпись токена, предоставив свой открытый ключ. Чтобы извлечь открытый ключ из нашего открытого сертификата, используйте следующую команду:

```
$ openssl x509 -pubkey -noout < public_key.pem > pubkey.pem
```

Эта команда выводит открытый ключ в файл `pubkey.pem`. Вам нужно скопировать содержимое этого файла на панель ввода открытого ключа в `jwt.io`, чтобы проверить подпись токена.



**Рис. 11.8.** `jwt.io` — это инструмент, который поможет вам легко проверять и визуализировать JWT. Просто вставьте токен на левую боковую панель. Вы также можете проверить подпись токена, вставив открытый ключ в поле `VERIFY SIGNATURE` справа

Вы также можете проверить содержимое JWT, расшифровав заголовок и полезную нагрузку в терминале с помощью команды `base64`. Например, чтобы расшифровать заголовок токена в терминале, выполните следующую команду:

```
$ echo eyJ0eXAiOiJKV1QiLCJhbGciOiJSUzI1NiJ9 | base64 --decode
{"alg": "RS256", "typ": "JWT"}
```

Кроме того, можно проверить содержимое JWT с помощью библиотеки Python `base64`. Чтобы декодировать заголовок JWT с помощью Python, откройте оболочку Python и выполните следующий код:

```
>>> import base64
>>> base64.decodebytes('eyJ0eXAiOiJKV1QiLCJhbGciOiJSUzI1NiJ9'.encode())
b'{"alg": "RS256", "typ": "JWT",}'
```

Поскольку полезная нагрузка JWT также закодирована в `base64url`, те же методы используются для ее декодирования.

### 11.3.5. Валидация JWT

Проверка подлинности JWT состоит из двух частей. С одной стороны, вы должны подтвердить его подпись, а с другой стороны, следует подтвердить правильность его утверждений, например убедившись, что срок действия токена не истек и что аудитория указана правильно. Обязательны оба этапа процесса валидации. Токен с истекшим сроком действия и действительной подписью не должен приниматься сервером API, в то время как активный токен с недействительной подписью также никуда не годится. Любой пользовательский запрос к серверу должен содержать токен, который нужно проверять при каждом запросе.

#### ВАЛИДАЦИЯ ТОКЕНОВ JWT ПРИ КАЖДОМ ЗАПРОСЕ

Когда пользователь взаимодействует с API сервера, он должен отправлять JWT в каждом запросе, а мы должны проверять токен. Некоторые реализации, особенно те, что используют схему с кодом авторизации, которую мы обсуждали в подразделе 11.2.1, хранят токены в кэше сеанса и сверяют токен запроса с кэшем. Но для JWT такой вариант не подойдет. JWT предназначены для обмена данными без сохранения состояния между клиентом и сервером и поэтому должны быть проверены с использованием методов, которые мы описываем в этом разделе.

Как мы видели в подразделе 11.3.3, токены могут быть подписаны секретным ключом или парой «закрытый/открытый ключ». В целях безопасности большинство веб-сайтов используют токены, подписанные закрытыми/открытыми ключами, и для проверки подписи таких токенов мы используем открытый ключ.

Посмотрим, как проверить токен в коде. Мы будем использовать ключ подписи, который создали в подразделе 11.3.3, для генерации и проверки токена. Активируйте свою среду Pipenv, запустив `pipenv shell`, и выполните скрипт `jwt_generator.py` для выдачи нового токена.

Чтобы проверить токен, сначала нужно загрузить открытый ключ, используя следующий код:

```
>>> from cryptography.x509 import load_pem_x509_certificate
>>> from pathlib import Path
>>> public_key_text = Path('public_key.pem').read_text()
>>> public_key = load_pem_x509_certificate(public_key_text.encode('utf-
➡ 8')).public_key()
```

Теперь, когда у нас есть открытый ключ, мы можем использовать его для проверки токена с помощью следующего кода:

```
>>> import jwt
>>> access_token = "eyJ0eXAiOiJKV1QiLCJhbGciOiJSUzI1NiJ..."
>>> jwt.decode(access_token, key=public_key, algorithms=['RS256'],
➡ audience=["http://127.0.0.1:8000/orders"])
{'iss': 'https://auth.coffeemesh.io/', 'sub': 'ec7bbccf-ca89-4af3-82ac-
➡ b41e4831a962', 'aud': 'http://127.0.0.1:8000/orders', 'iat':
➡ 1638114196.49375, 'exp': 1638200596.49375, 'scope': 'openid'}
```

Как вы можете видеть, если токен действителен, будет возвращена полезная нагрузка JWT. Если токен недействителен, этот код вызовет исключение. Теперь, когда вы знаете, как работать с JWTs и проверять их подлинность, посмотрим, как выполнять авторизацию запросов на сервере API.

## 11.4. ДОБАВЛЕНИЕ АВТОРИЗАЦИИ НА API СЕРВЕРА

Итак, вы разобрались, как проверять токены доступа, поэтому далее мы объединим весь изученный код на нашем сервере API. В этом разделе мы добавим авторизацию в API сервиса заказов. Одни эндпоинты API защищены, в то время как другие должны быть доступны для всех. Наша цель — убедиться, что сервер проверяет наличие действительных токенов доступа в защищенных эндпоинтах.

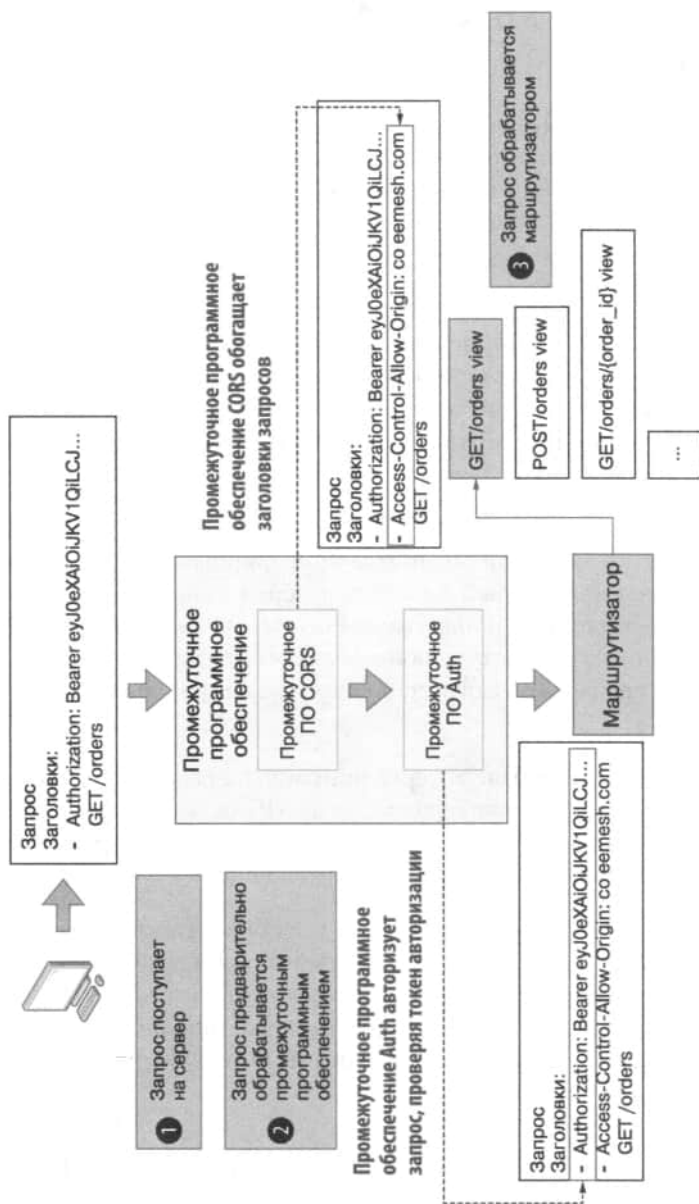
Мы разрешим общий доступ к эндпоинтам `/docs/orders` и `/openapi/orders.json`, поскольку они обслуживают документацию API, которая должна быть доступна для всех потребителей. Для всех остальных эндпоинтов требуются действительные токены. Если токен недействителен или отсутствует в запросе, мы должны отклонить запрос с кодом 401 (Unauthorized), который указывает на отсутствие учетных данных.

Как нам добавить авторизацию в API? Существует два основных метода: обработка проверки в шлюзе API или обработка проверки в каждом сервисе. *Шлюз API* — это сетевой слой, который находится перед нашими API<sup>1</sup>. Основная задача шлюза API — облегчать обнаружение сервисов, но его также можно использовать для авторизации доступа пользователей, проверки токенов доступа и дополнения запроса кастомными заголовками, расширяющими информацию о пользователе.

Второй метод состоит в обработке авторизации внутри каждого API. Нужно обрабатывать авторизацию на уровне сервиса, когда API-шлюз не может это сделать или когда он не вписывается в вашу архитектуру. В этом разделе мы разберем, как обрабатывать авторизацию внутри сервиса, поскольку у нас нет API-шлюза.

Часто возникает вопрос: где именно в коде мы обрабатываем авторизацию? Поскольку авторизация необходима для подтверждения доступа пользователя к сервису через API, мы реализуем ее в промежуточном программном обеспечении API. Как вы можете видеть на рис. 11.9, *промежуточное программное обеспечение* (middleware) — это слой кода, который обеспечивает общую функциональность для обработки всех наших запросов.

<sup>1</sup> См. статью: *Richardson C. Pattern: API Gateway/Backends for Frontends* («Паттерн: API-шлюз/бэкенд для фронтенда»). <https://microservices.io/patterns/apigateway.html>.



**Рис. 11.9.** Запрос сначала обрабатывается серверным промежуточным программным обеспечением, таким как CORS и Auth, а затем отправляется на маршрутизатор, который сопоставляет запрос с соответствующей функцией просмотра

Большинство веб-серверов поддерживают возможность использования промежуточного программного обеспечения или препроцессоров запросов, и именно к ним относится наш код авторизации. Компоненты промежуточного программного обеспечения чаще всего выполняются по порядку, и обычно мы можем выбирать этот порядок. Поскольку авторизация контролирует доступ к нашему серверу, следует заблаговременно запустить промежуточное программное обеспечение для авторизации.

### 11.4.1. Создание модуля авторизации

Сначала создадим модуль для инкапсуляции нашего кода авторизации. Создайте файл с именем `orders/web/api/auth.py` и скопируйте в него код из листинга 11.3. Мы начинаем с загрузки открытого ключа, который создали в подразделе 11.3.3. Чтобы проверить токен, сначала извлекаем заголовки и загружаем открытый ключ. Для проверки токена мы применяем функцию `PyJWT decode()`, передавая в качестве параметров сам токен, открытый ключ, необходимый для его проверки, ожидаемый список адресатов и алгоритмы, используемые для подписи ключа.

#### Листинг 11.3. Добавление модуля авторизации в API

```
# file: orders/web/api/auth.py

from pathlib import Path

import jwt
from cryptography.x509 import load_pem_x509_certificate

public_key_text = (
    Path(__file__).parent / "../../../public_key.pem"
).read_text()
public_key = load_pem_x509_certificate(
    public_key_text.encode()
).public_key()

def decode_and_validate_token(access_token):
    """
    Проверяет токен доступа. Если токен действителен,
    возвращает полезную нагрузку токена.
    """
    return jwt.decode(
        access_token,
        key=public_key,
        algorithms=["RS256"],
        audience=["http://127.0.0.1:8000/orders"],
    )
```

Теперь, когда мы создали модуль, который инкапсулирует функциональность, необходимую для проверки JWT, включим его в API, добавив промежуточное программное обеспечение для проверки доступа.

## 11.4.2. Создание промежуточного программного обеспечения для авторизации

Чтобы добавить авторизацию в наш API, мы создаем соответствующее промежуточное программное обеспечение. Код, приведенный в листинге 11.4, находится в файле `orders/web/app.py`, а добавленные строки выделены жирным шрифтом. Мы реализуем промежуточное программное обеспечение в виде простого класса под названием `AuthorizeRequestMiddleware`, который наследуется от класса `BaseHTTPMiddleware` Starlette. Точка входа для промежуточного программного обеспечения должна быть реализована в функции `dispatch()`.

Как определить, следует ли нам включать авторизацию? Воспользуемся флагом в виде переменной окружения с именем `AUTH_ON`, которая по умолчанию хранит значение `False`. Часто при работе над новой функцией или при отладке проблемы в API удобно запускать сервер локально без авторизации. Наличие флага позволяет нам включать и выключать аутентификацию в соответствии с нашими потребностями. Если авторизация отключена, мы добавляем проверку ID по умолчанию для запрашивающего пользователя.

Далее мы проверяем, запрашивает ли пользователь документацию по API. В этом случае мы не блокируем запрос, так как хотим сделать документацию видимой для всех пользователей; иначе они не знали бы, как правильно формировать свои запросы.

Мы также проверяем метод запроса. Если это запрос параметров, мы не будем пытаться авторизовать его. Запросы параметров — это предварительные запросы, также известные как запросы на совместное использование ресурсов между разными источниками (`cross-origin resource sharing`, CORS). Цель такого запроса — проверить, какие источники, методы и заголовки запросов принимаются сервером API, и в соответствии со спецификацией W3 запросы CORS не должны требовать учетных данных (<https://www.w3.org/TR/2020/SPSD-cors-20200602/>). Запросы CORS обычно обрабатываются платформой веб-сервера.

### ОПРЕДЕЛЕНИЕ

*Запросы CORS* — это запросы, отправляемые браузером, чтобы понять, какие методы, источники и заголовки принимаются сервером API. Если мы неправильно обрабатываем запросы CORS, браузер прервет связь с API. К счастью, большинство веб-фреймворков содержат плагины или расширения, которые корректно обрабатывают запросы CORS. Эти запросы не проходят валидацию, поэтому, добавляя авторизацию на сервер, мы должны убедиться, что для предварительных запросов не требуются учетные данные.

Если это не запрос CORS, мы пытаемся перехватить токен из заголовков запроса. Мы ожидаем токен в заголовке авторизации. Если заголовок авторизации не найден, отклоняем запрос, выдавая ответ со статус-кодом 401 (`Unauthorized`).

Формат значения заголовка авторизации — `Bearer <ACCESS_TOKEN>`, поэтому, если он найден, мы перехватываем токен, разделяя значение заголовка пробелом, и пытаемся его проверить. Если токен недействителен, `PyJWT` вызовет исключение.



В промежуточном программном обеспечении мы фиксируем исключения недействительности PyJWT, чтобы убедиться, что можем вернуть ответ со статус-кодом 401. Если исключение не выброшено, значит, токен действителен и мы можем обработать запрос, поэтому передаем вызов следующему обратному вызову. Мы также сохраняем идентификатор пользователя из полезной нагрузки токена в объекте `state` запроса, чтобы получить к нему доступ позже в представлениях API. Наконец, чтобы зарегистрировать промежуточное программное обеспечение, используем метод `FastAPI add_middleware()`.

## КУДА ПОМЕЩАЮТСЯ JSON WEB TOKENS

JWT размещаются в заголовках запросов, обычно под заголовком авторизации. Заголовок авторизации с JWT имеет такой формат: `Authorization: Bearer <JWT>`.

**Листинг 11.4.** Добавление промежуточного программного обеспечения для авторизации в API сервиса заказов

```
# file: orders/web/app.py
```

```
import os
```

```
from fastapi import FastAPI
```

```
from jwt import (
    ExpiredSignatureError,
    ImmatureSignatureError,
    InvalidAlgorithmError,
    InvalidAudienceError,
    InvalidKeyError,
    InvalidSignatureError,
    InvalidTokenError,
    MissingRequiredClaimError,
```

```
)
```

```
from starlette import status
```

```
from starlette.middleware.base import (
    RequestResponseEndpoint,
    BaseHTTPMiddleware,
```

```
)
```

```
from starlette.requests import Request
```

```
from starlette.responses import Response, JSONResponse
```

```
from orders.api.auth import decode_and_validate_token
```

```
app = FastAPI(debug=True)
```

Создаем класс промежуточного ПО, наследуя от базового класса Starlette `BaseHTTPMiddleware`

```
class AuthorizeRequestMiddleware(BaseHTTPMiddleware):
```

```
    async def dispatch(
```

```
        self, request: Request, call_next: RequestResponseEndpoint
```

```
    ) -> Response:
```

```
        if os.getenv("AUTH_ON", "False") != "True":

```

Авторизуем запрос, если `AUTH_ON` имеет значение `True`

```
            request.state.user_id = "test"
```

```
            return await call_next(request)
```

Если авторизация отключена, привязываем к запросу пользователя с именем `test`  
Возвращаем, вызывая следующий обратный вызов

```

if request.url.path in ["/docs/orders", "/openapi/orders.json"]:
    return await call_next(request)
if request.method == "OPTIONS":
    return await call_next(request)

bearer_token = request.headers.get("Authorization")
if not bearer_token:
    return JSONResponse(
        status_code=status.HTTP_401_UNAUTHORIZED,
        content={
            "detail": "Missing access token",
            "body": "Missing access token",
        },
    )
try:
    auth_token = bearer_token.split(" ")[1].strip()
    token_payload = decode_and_validate_token(auth_token)
except (
    ExpiredSignatureError,
    ImmatureSignatureError,
    InvalidAlgorithmError,
    InvalidAudienceError,
    InvalidKeyError,
    InvalidSignatureError,
    InvalidTokenError,
    MissingRequiredClaimError,
) as error:
    return JSONResponse(
        status_code=status.HTTP_401_UNAUTHORIZED,
        content={"detail": str(error), "body": str(error)},
    )
else:
    request.state.user_id = token_payload["sub"]
    return await call_next(request)

app.add_middleware(AuthorizeRequestMiddleware)
from orders.api import api

```

Конечные точки документации находятся в открытом доступе, поэтому мы их не авторизуем

Пытаемся получить заголовок авторизации

Если заголовок авторизации не задан, возвращаем ответ 401

Извлекаем токен из заголовка авторизации

Проверяем и извлекаем полезную нагрузку токена

Если токен недействителен, возвращаем ответ 401

Извлекаем ID пользователя из субполя токена

Регистрируем промежуточное ПО, используя метод FastAPI add\_middleware()

Наш сервер готов начать проверку запросов с помощью JWT! Запустим тест, чтобы увидеть код авторизации в действии. Активируйте виртуальную среду с помощью `pipenv shell` и запустите сервер такой командой:

```
$ AUTH_ON=True uvicorn orders.web.app:app --reload
```

С другого терминала выполните не прошедший проверку запрос, используя с URL (приводится только часть выходных данных) с флагом `-i`, что позволит отобразить дополнительную информацию, например статус-код ответа:

```
$ curl -i http://localhost:8000/orders
HTTP/1.1 401 Unauthorized
[...]
```

```
{"detail":"Missing access token","body":"Missing access token"}
```

Как вы можете видеть, запрос с отсутствующим токеном отклоняется с ошибкой 401 и сообщением о том, что токен доступа отсутствует. Теперь сгенерируем токен с помощью скрипта `jwt_generator.py`, который мы реализовали в подразделе 11.3.3, и используем токен для создания нового запроса:

```
curl http://localhost:8000/orders -H 'Authorization: Bearer
➔ eyJ0eXAiOiJKV1QiLCJhbGciOiJSUzI1NiIsImtpZCI6ImI3NTQwM2QxLWUzZDktNDgzYy0
➔ 5MjZhLTm4NDhM2Q4OWY1YyJ9.eyJpc3MiOiJodHRwczovL2F1dGguY29mZmVlbWVzaC5pb
➔ y8iLCJzdWIiOiJlYzdiYmNjZi1jYTg5LTRhZjMtODJhYy1iNDFlNDgzMWE5NjIiLCJhdWQi
➔ OiJodHRwOi8vMTI3LjAuMCA4x0JgwMDAvb3JkZXJzIiwiaWF0IjoxNjM4MTE3MjE5Ljc5OTE
➔ 3OSwiczXhwIjoxNjM4MjAzNjE5Ljc5OTE3OSwic2NvcGUiOiJvcGVuawQi fQ. F1bmgYm1acf
➔ i1NMm5JGkbYQYWFNvG1-7BAXEnIqNdF0th_DYcnEm_p3YZ5hQ93v4QWxDx9tmuit6InKs-
➔ MHqhChP2k6DakpSocaqbgJ_IHpnHtAEzByqZjoNfZFyQLZMo3yEaQB8S_x0LcK00qeoPY1
➔ GSWM1eAUy7VFBXmVMUZrUj-yoK721U9vevgM-wdVyyFVtpTRuyjCoWMjJEVadNn-
➔ Zrxr0ghlRQnwEx-YdtbbEMkk_vVLWoWeEgj7mkBE167fr-fyGUKBqa2F71Zwh8DaDQz79Ph-
➔ ST0Y6BT1CnAVL8Xwnl1OhJWpShuc90Kynn_RX49_yJrQHKF-xLoflWg'
{"orders":[]}
```

Если токен действителен, на этот раз вы получите успешный ответ со списком заказов. Код авторизации работает! Далее нужно сделать так, чтобы пользователи могли получать доступ только к своим ресурсам на сервере. Однако перед этим добавим еще одну часть промежуточного программного обеспечения для обработки запросов CORS.

### 11.4.3. Добавление промежуточного программного обеспечения CORS

Поскольку мы собираемся разрешить взаимодействие с клиентским приложением, нам также необходимо разрешить использование промежуточного программного обеспечения CORS. Как вы видели в подразделе 11.4.2, браузер отправляет запросы CORS, чтобы узнать, какие заголовки, методы и источники разрешены сервером. Промежуточное ПО CORS от FastAPI отвечает за то, чтобы наши ответы были заполнены нужной информацией. В листинге 11.5 показано, как изменить файл `orders/web/app.py` для регистрации промежуточного программного обеспечения CORS. Недавно добавленный код выделен жирным шрифтом, а часть кода обрезана (вместо него стоят многоточия).

#### Листинг 11.5. Добавление промежуточного программного обеспечения CORS

```
# file: orders/web/app.py

import os

from fastapi import FastAPI
from jwt import (
    ExpiredSignatureError,
    ImmatureSignatureError,
    InvalidAlgorithmError,
```

```

InvalidAudienceError,
InvalidKeyError,
InvalidSignatureError,
InvalidTokenError,
MissingRequiredClaimError,
)
from starlette import status
from starlette.middleware.base import RequestResponseEndpoint,
    BaseHTTPMiddleware
from starlette.middleware.cors import CORSMiddleware ← Импортируем класс
from starlette.requests import Request                CORSMiddleware or Starlette
from starlette.responses import Response, JSONResponse

from orders.api.auth import decode_and_validate_token

app = FastAPI(debug=True)

...

app.add_middleware(AuthorizeRequestMiddleware)

app.add_middleware(
    CORSMiddleware,
    allow_origins=["*"],
    allow_credentials=True,
    allow_methods=["*"],
    allow_headers=["*"],
)

from orders.api import api

```

Регистрируем CORSMiddleware, используя метод add\_middleware() FastAPI

Допускаем любой источник

Поддерживаем файлы cookie для запросов из разных источников

Разрешаем все HTTP-методы

Разрешаем все заголовки

Как и ранее, мы используем метод `add_middleware()` FastAPI для регистрации промежуточного программного обеспечения CORS и передаем необходимую конфигурацию. В целях тестирования мы используем подстановочные знаки, чтобы разрешить все источники, методы и заголовки, но в своей производственной среде вы должны быть более конкретны. В частности, вы должны ограничить разрешенные источники доменом вашего сайта и другими надежными источниками.

Порядок, в котором мы регистрируем наше промежуточное ПО, имеет значение. Промежуточное ПО выполняется в порядке, обратном регистрации, то есть первым выполняется последнее зарегистрированное. Поскольку промежуточное ПО CORS требуется для всех взаимодействий между клиентом и сервером API, мы регистрируем его последним, чтобы гарантировать его постоянное выполнение.

Почти все готово! Теперь наш сервер может авторизовывать пользователей и обрабатывать запросы CORS.

Следующий шаг — убедиться, что каждый пользователь может получить доступ только к своим данным.

## 11.5. АВТОРИЗАЦИЯ ДОСТУПА К РЕСУРСАМ

Мы защитили наш API, убедившись, что доступ к нему могут получить только те пользователи, что прошли аутентификацию. Теперь мы должны убедиться, что детали каждого заказа доступны только тому пользователю, который его разместил, — не стоит разрешать пользователям получать доступ к данным друг друга. Такая проверка называется *авторизацией*, и в этом разделе вы узнаете, как добавить ее в свои API.

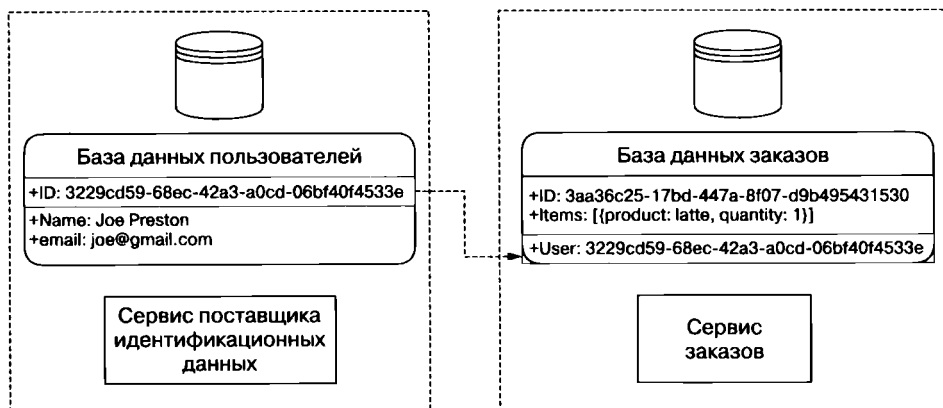
### 11.5.1. Обновление базы данных для привязки пользователей и заказов

Начнем с удаления заказов, которые в данный момент присутствуют в базе данных. Они не связаны с пользователем и, следовательно, не будут действительны, пока мы не установим связь между каждым заказом и пользователем. Перейдите в каталог `ch11`, активируйте виртуальную среду, запустив `pipenv shell`, и откройте оболочку Python, выполнив команду `python`. В оболочке Python запустите следующий код:

```
>>> from orders.repository.orders_repository import OrdersRepository
>>> from orders.repository.unit_of_work import UnitOfWork
>>> with UnitOfWork() as unit_of_work:
...     orders_repository = OrdersRepository(unit_of_work.session)
...     orders = orders_repository.list()
...     for order in orders: order.delete(order.id)
...     unit_of_work.commit()
```

Теперь наша база данных очищена, так что мы готовы приступить к работе. Как мы свяжем каждый заказ с пользователем? Типичная стратегия — создать пользовательскую таблицу и привязать заказы к пользовательским записям с помощью внешних ключей. Но действительно ли имеет смысл создавать пользовательскую таблицу для сервиса заказов? Нужна ли нам пользовательская таблица для каждого сервиса?

Нет, нам это не нужно, поскольку это чревато большим дублированием. Как вы можете видеть на рис. 11.10, лучше иметь только одну пользовательскую таблицу, и она должна принадлежать пользовательскому сервису. Наш пользовательский сервис — это наш поставщик идентификационных данных как услуги, поэтому наша таблица пользователей уже существует. У каждого пользователя уже есть ID, и, как вы видели в подразделе 11.3.1, ID присутствует в полезной нагрузке JWT в поле `sub`. Нам нужно лишь добавить новый столбец в таблицу заказов, чтобы сохранить ID пользователя, сделавшего заказ.



**Рис. 11.10.** Чтобы избежать дублирования, мы храним только одну пользовательскую таблицу у поставщика идентификационных данных. И чтобы избежать тесной связи между сервисами, стараемся не использовать внешние ключи между таблицами, принадлежащими разным сервисам

## ПРИВЯЗКА ПОЛЬЗОВАТЕЛЕЙ К ИХ РЕСУРСАМ

Два распространенных антипаттерна в микросервисной архитектуре — это создание одной пользовательской таблицы для каждого сервиса и наличие общей пользовательской таблицы, к которой напрямую обращаются несколько сервисов, для создания внешних ключей между пользователями и другими ресурсами. Создание пользовательских таблиц для каждого сервиса влечет за собой дублирование, в то время как наличие общей пользовательской таблицы для нескольких сервисов обеспечивает тесную связь между сервисами и рискует нарушить их при следующем изменении схемы этой таблицы. Поскольку JWT уже содержат ID пользователей в поле `sub`, рекомендуется полагаться на этот ID для привязки пользователей к их ресурсам.

В листинге 11.6 показано, как мы добавляем поле `user_id` в класс `OrderModel`. Следующий код должен храниться в файле `orders/repository/models.py`, а недавно добавленный код выделен жирным шрифтом.

### Листинг 11.6. Добавление внешнего ключа ID пользователя в таблицу заказов

# file: orders/repository/models.py

```
class OrderModel(Base):
    __tablename__ = 'order'

    id = Column(String, primary_key=True, default=generate_uuid)
    user_id = Column(String, nullable=False)
    items = relationship('OrderItemModel', backref='order')
    status = Column(String, nullable=False, default='created')
    created = Column(DateTime, default=datetime.utcnow)
    schedule_id = Column(String)
    delivery_id = Column(String)
```

Добавляем  
новый столбец  
с именем `user_id`

Теперь, когда мы обновили модели, нам нужно обновить базу данных, выполнив миграцию. Как вы видели в главе 7, запуск миграции — это процесс обновления схемы базы данных. Если помните, для управления миграциями мы использовали Alembic, которая является лучшей библиотекой управления миграцией баз данных в Python. Alembic проверяет разницу между моделью `OrderModel` и текущей схемой таблицы заказов и выполняет обновления, необходимые для добавления столбца `user_id`.

## ИЗМЕНЕНИЕ ТАБЛИЦ В SQLite

SQLite имеет ограниченную поддержку инструкций ALTER, например, не позволяет добавлять новый столбец в таблицу с помощью инструкции ALTER. Как вы можете видеть на рис. 11.11, чтобы обойти эту проблему, нужно скопировать данные таблицы во временную таблицу и удалить исходную. Затем мы заново создаем таблицу с новыми полями, копируем данные из временной таблицы и удаляем ее.

Прежде чем мы сможем запустить миграцию, нам необходимо обновить конфигурацию Alembic. В листинге 11.6 в таблицу `order` добавлен новый столбец, который преобразуется в SQL-инструкцию `ALTER TABLE`. Для локальной разработки мы работаем с SQLite, который имеет ограниченную поддержку инструкций ALTER. Чтобы гарантировать, что Alembic генерирует правильные миграции для SQLite, нам необходимо обновить его конфигурацию для активации пакетной обработки. *Вам нужно сделать это только в том случае, если вы работаете с SQLite.*



**Рис. 11.11.** При работе с SQLite для внесения изменений в таблицы мы используем пакетную обработку. Она позволяет нам копировать данные из исходной таблицы во временную; затем мы удаляем исходную таблицу и создаем ее заново с новыми полями; и, наконец, копируем обратно данные из временной таблицы

Чтобы обновить конфигурацию Alembic и получить возможность запустить миграцию, откройте файл `migrations/env.py` и найдите функцию с именем `run_migrations_online()`. Она отвечает за миграцию в базу данных. В рамках этой функции выполните поиск следующего блока:

```
# file: migrations/env.py

with connectable.connect() as connection:
    context.configure(
        connection=connection,
        target_metadata=target_metadata
    )
```

Добавьте следующую строку (выделена жирным шрифтом) в вызов метода `configure()`:

```
# file: migrations/env.py

with connectable.connect() as connection:
    context.configure(
        connection=connection,
        target_metadata=target_metadata,
        render_as_batch=True
    )
```

Теперь мы можем сгенерировать миграцию Alembic и обновить базу данных. Запустите следующую команду, чтобы создать новую миграцию:

```
$ PYTHONPATH=`pwd` alembic revision --autogenerate -m "Add user id to order table"
```

Далее мы запускаем миграцию с помощью такой команды:

```
$ PYTHONPATH=`pwd` alembic upgrade heads
```

Итак, наша база данных готова к тому, чтобы мы начали связывать заказы и пользователей. В следующем подразделе объясняется, как извлечь ID пользователя из объекта запроса и передать его в хранилища данных.

## 11.5.2. Ограничение доступа пользователей к их собственным ресурсам

Теперь, когда наша база данных готова, нам нужно обновить представления API, чтобы захватить ID пользователя при создании или обновлении заказа или при извлечении списка заказов. Поскольку все изменения, которые понадобится внести в функции просмотра, довольно схожи, я покажу, как применить изменения к некоторым представлениям. С полным списком изменений вы можете ознакомиться в репозитории этой книги на GitHub.



В листинге 11.7 показано, как обновить функцию просмотра `create_order()`, чтобы получить ID пользователя при размещении заказа. Недавно добавленный код выделен жирным шрифтом. Как вы видели в подразделе 11.4.2, ID пользователя хранится в свойстве `state` запроса, поэтому первое изменение, которое мы вносим, затрагивает подпись функции `create_order()` — мы включаем объект `request`. Второе изменение состоит в передаче ID пользователя методу `place_order()` `OrderService`.

### Листинг 11.7. Захват ID пользователя при размещении заказа

# file: orders/web/api/api.py

```
@app.post(
    "/orders", status_code=status.HTTP_201_CREATED,
    response_model=GetOrderSchema
)
def create_order(request: Request, payload: CreateOrderSchema):
    with UnitOfWork() as unit_of_work:
        repo = OrdersRepository(unit_of_work.session)
        orders_service = OrdersService(repo)
        order = payload.dict()["order"]
        for item in order:
            item["size"] = item["size"].value
            order = orders_service.place_order(order, request.state.user_id)
            unit_of_work.commit()
            return_payload = order.dict()
        return return_payload
```

Фиксируем объект запроса в сигнатуре функции

Извлекаем идентификатор пользователя из объекта состояния запроса

Нам также нужно изменить `OrdersService` и `OrdersRepository`, чтобы убедиться, что они также захватывают ID пользователя. Следующий код показывает, как обновить `OrdersService` для получения ID пользователя:

# file: orders/orders\_service/orders\_service.py

```
class OrdersService:
    def __init__(self, orders_repository: OrdersRepository):
        self.orders_repository = orders_repository

    def place_order(self, items, user_id):
        return self.orders_repository.add(items, user_id)
```

Следующий код показывает, как обновить `OrderRepository`, чтобы захватить ID пользователя:

# file: orders/repository/orders\_repository.py

```
class OrdersRepository:
    def __init__(self, session):
        self.session = session

    def add(self, items, user_id):
        record = OrderModel(
```

```

        items=[OrderItemModel(**item) for item in items],
        user_id=user_id
    )
    self.session.add(record)
    return Order(**record.dict(), order_=record)

```

Теперь, когда мы знаем, как сохранить заказ с ID пользователя, посмотрим, как убедиться, что пользователь получает только список своих заказов при вызове эндпоинта GET /orders. В листинге 11.8 показаны изменения, необходимые для функции `get_orders()`, которая реализует эндпоинт GET /orders. Недавно добавленный код выделен жирным шрифтом. Как вы можете видеть, нам также нужно изменить сигнатуру функции, чтобы захватить объект запроса. Затем мы просто передаем ID пользователя в качестве одного из фильтров запроса. Никаких дополнительных изменений больше нигде в коде не требуется, поскольку и `OrdersService`, и `OrdersRepository` предназначены для приема произвольных словарей фильтров.

**Листинг 11.8.** Обеспечение того, чтобы пользователь получал только список своих собственных заказов

```

# file: orders/web/api/api.py

@app.get("/orders", response_model=GetOrdersSchema)
def get_orders(
    request: Request,
    cancelled: Optional[bool] = None,
    limit: Optional[int] = None
):
    with UnitOfWork() as unit_of_work:
        repo = OrdersRepository(unit_of_work.session)
        orders_service = OrdersService(repo)
        results = orders_service.list_orders(
            limit=limit, cancelled=cancelled, user_id=request.state.user_id
        )
    return {"orders": [result.dict() for result in results]}

```

Теперь обратите внимание на эндпоинт GET /orders/{order\_id}. Что произойдет, если пользователь попытается получить сведения о заказе, который ему не принадлежит? Возможны два сценария: можно вернуть ответ 404 (Not Found), указывающий на то, что запрошенный заказ не существует, или ответ 403 (Forbidden), указывающий на то, что у пользователя нет доступа к запрошенному ресурсу.

Технически ответ 403 будет более правильным, чем 404, ведь пользователь пытается получить доступ к ресурсу, который ему не принадлежит. Но это также предоставляет ненужную информацию. Злоумышленник, у которого есть действительные учетные данные, может использовать наши ответы 403 для построения карты существующих ресурсов на сервере. Чтобы избежать этого,

предпочтительнее раскрывать меньше информации и возвращать ответ 404. Когда мы попытаемся получить заказ из базы данных, ID пользователя станет дополнительным фильтром.

Код в листинге 11.9 показывает изменения в функции `get_order()`, чтобы включить ID пользователя в наши запросы. Новый код выделен жирным шрифтом. Опять же, включаем объект запроса в сигнатуру функции и передаем ID пользователя методу `get_order()` `OrderService`.

#### Листинг 11.9. Фильтрация заказов по ID заказа и ID пользователя

```
# file: orders/web/api/api.py

@app.get("/orders/{order_id}", response_model=GetOrderSchema)
def get_order(request: Request, order_id: UUID):
    try:
        with UnitOfWork() as unit_of_work:
            repo = OrdersRepository(unit_of_work.session)
            orders_service = OrdersService(repo)
            order = orders_service.get_order(
                order_id=order_id, user_id=request.state.user_id
            )
            return order.dict()
    except OrderNotFoundError:
        raise HTTPException(
            status_code=404, detail=f"Order with ID {order_id} not found"
        )
```

Чтобы иметь возможность запрашивать заказы и по ID пользователя, нам также необходимо обновить классы `OrdersService` и `OrdersRepository`. Изменим их методы, чтобы они принимали необязательный словарь произвольных фильтров. Метод `get_order()` `OrdersService` изменяем следующим образом:

```
# file: orders/orders_service/orders_service.py

def get_order(self, order_id, **filters):
    order = self.orders_repository.get(order_id, **filters)
    if order is not None:
        return order
    raise OrderNotFoundError(f"Order with id {order_id} not found")
```

А методы `get()` и `_get()` `OrdersRepository` требуют следующих изменений:

```
# file: orders/repository/orders_repository.py

def _get(self, id_, **filters):
    return (
        self.session.query(OrderModel)
        .filter(OrderModel.id == str(id_)).filter_by(**filters)
        .first()
    )
```

```
def get(self, id_, **filters):
    order = self._get(id_, **filters)
    if order is not None:
        return Order(**order.dict())
```

Остальную часть функций представления в файле `orders/web/api/api.py` нужно изменить аналогичным образом, и то же самое касается остальных методов классов `OrdersService` и `OrdersRepository`. В качестве упражнения рекомендую вам попытаться внести изменения, необходимые для добавления авторизации к оставшимся эндпоинтам API. Репозиторий GitHub для этой книги содержит полный список изменений, поэтому вы в любой момент можете обратиться к нему для согласования своих действий.

На этом наше изучение аутентификации и авторизации API завершается. Вы узнали, что такое OAuth и OpenID Connect и как они работают. Вы узнали о схемах OAuth и о том, когда использовать каждую. Вы разобрались с тем, что такое JWT, как проверять их полезную нагрузку, а также как их создавать и проверять достоверность. Наконец, вы узнали, как авторизовывать запросы API и разрешать доступ пользователей к определенным ресурсам. У вас есть все необходимое, чтобы добавить надежную аутентификацию и авторизацию в свои собственные API.

В приложении В рассказывается, как интегрироваться с поставщиком удостоверений, таким как Auth0. Вы также найдете там примеры того, как использовать схемы учетных данных PKCE и клиента, и научитесь авторизовывать свои запросы с помощью пользовательского интерфейса Swagger UI.

## РЕЗЮМЕ

- Мы авторизуем доступ к нашим API, используя стандартные протоколы OAuth и OpenID Connect.
- OAuth — это протокол предоставления доступа, который позволяет пользователю предоставлять приложению доступ к ресурсам, которыми он владеет на другом сайте. Различают четыре схемы авторизации:
  - *с кодом авторизации* — сервер API обменивается кодом с сервером авторизации, чтобы запросить токен доступа пользователя;
  - *PKCE* — клиентское приложение, обычно SPA, использует `code_verifier` и `code_challenge` для получения токена доступа с сервера авторизации;
  - *клиентских полномочий* — клиент, обычно другой микросервис, обменивается личным секретом в обмен на токен доступа;
  - *с обновляемым токеном* — клиент получает новый токен доступа в обмен на refresh-токен.

- OpenID Connect — протокол проверки подлинности, основанный на OAuth. Помогает пользователям легко проходить аутентификацию на новых сайтах, указывая данные с других сайтов, таких как Google или Facebook.
- JWT — это документы в формате JSON, содержащие утверждения о разрешениях доступа пользователя. JWT кодируются с использованием кодировки base64url и обычно подписываются с помощью закрытого/открытого ключа.
- Чтобы аутентифицировать запрос, пользователи отправляют свои токены доступа в заголовке авторизации запроса. Ожидаемый формат этого заголовка — `Authorization: Bearer <ACCESS_TOKEN>`.
- Мы используем PyJWT для проверки токенов доступа. PyJWT убеждается, что срок действия токена не истек, что аудитория указана правильно и что подпись можно проверить с помощью одного из доступных открытых ключей. Если токен недействителен, мы отклоняем запрос с ответом 401 (Unauthorized).
- Чтобы привязать пользователей к их ресурсам, мы используем ID пользователя, представленный в дополнительном утверждении JWT.
- Если пользователь пытается получить доступ к ресурсу, который ему не принадлежит, выдаем ответ 403 (Forbidden).
- Запросы параметров (OPTIONS) известны как запросы CORS или предварительные запросы. Они не должны быть защищены учетными данными.

# Тестирование и валидация API

---

## В этой главе

- ✓ Генерация автоматических тестов для REST API с использованием Dredd и Schemathesis.
- ✓ Написание перехватчиков Dredd для настройки поведения набора тестов Dredd.
- ✓ Использование для API тестирования на основе свойств.
- ✓ Использование ссылок OpenAPI для улучшения набора тестов Schemathesis.
- ✓ Тестирование GraphQL API с помощью Schemathesis.

В этой главе вы узнаете, как тестировать и проверять реализации API. К настоящему времени вы научились проектировать и создавать API для обеспечения интеграции между микросервисами. Попутно мы провели несколько ручных тестов, чтобы убедиться, что реализации демонстрируют правильное поведение. Однако эти тесты были простейшими и, самое главное, ручными, следовательно, не повторялись в автоматическом режиме.

В этой главе вы узнаете, как запустить полный набор тестов для своих реализаций API, используя такие инструменты, как Dredd и Schemathesis, которые должны быть в арсенале каждого разработчика API. И Dredd, и Schemathesis работают, просматривая спецификацию API и автоматически генерируя тесты для сервера

API. Для разработчика это очень удобно, поскольку он может сосредоточиться на создании API, а не на их тестировании.

Используя такие инструменты, как Dredd и Schemathesis, вы можете сэкономить время и энергию, оставаясь при этом уверенными в правильности созданной реализации. Вы можете запустить Dredd и Schemathesis вместе или выбрать один из них. Как вы увидите, Dredd предоставляет базовый набор тестов, который подойдет на ранних стадиях цикла разработки API, в то время как Schemathesis предлагает более расширенный набор тестов — его лучше использовать перед запуском API в производство.

Чтобы продемонстрировать, как тестируются REST API, воспользуемся API сервиса заказов, который мы реализовали в главах 2 и 6. Чтобы проиллюстрировать, как тестируются GraphQL API, воспользуемся API сервиса продукции, который реализовали в главе 10. Напомню, что оба API являются частью CoffeeMesh, вымышленной платформы доставки кофе, которую мы создаем в книге. API сервиса заказов — это интерфейс к сервису заказов, который управляет заказами клиентов, в то время как API сервиса продукции — это интерфейс к сервису продукции, управляющему каталогом товаров, предлагаемых CoffeeMesh.

Код для этой главы доступен на GitHub, в папке `ch12`. В разделе 12.1 мы настроим структуру папок и среду для работы, поэтому обязательно ознакомьтесь с этим разделом, если хотите следовать примерам из этой главы.

## 12.1. НАСТРОЙКА СРЕДЫ ДЛЯ ТЕСТИРОВАНИЯ API

В этом разделе мы настроим среду для работы с примерами главы. Начнем со структуры папок. Создайте новую папку с именем `ch12` и перейдите в нее. В нее мы скопируем API сервисов заказов и продукции. Чтобы упростить задачу, возьмем реализацию API сервиса заказов в том виде, в каком оставили ее в главе 6. Там мы создали полную реализацию API, но нет реальной базы данных и интеграции с другими сервисами (эти функции были добавлены в главе 7). Поскольку цель этой главы — разобраться с тестированием API, нам будет достаточно реализации, описанной в главе 6, — не придется настраивать базу данных и запускать дополнительные сервисы. В реальной жизни пришлось бы протестировать уровень API изолированно и запустить интеграционные тесты, в том числе в базе данных. Посмотрите файл `README.md` в папке `ch12/orders` в репозитории GitHub — он содержит инструкции по запуску тестов на соответствие состоянию приложения после изучения глав 7 и 11.

В папке `ch12` скопируйте реализацию API сервиса заказов из `ch06/orders`, выполнив следующую команду:

```
$ cp -r ../ch06/orders orders
```

Перейдите в `ch12/orders` и запустите следующую команду для установки зависимостей:

```
$ pipenv install --dev
```

Не забудьте включить флаг `--dev` при запуске `pipenv install` — он указывает Pipenv устанавливать зависимости как для производства, так и для разработки. В этой главе мы будем использовать комплект разработки для тестирования API сервиса заказов. Для запуска тестов нам понадобятся `pytest`, `dredd_hooks` и `schemathesis`, которые вы можете установить с помощью такой команды:

```
$ pipenv install --dev dredd_hooks pytest schemathesis
```

Для запуска тестов воспользуемся слегка измененной версией спецификации API сервиса заказов без схемы безопасности `bearerAuth`, которую вы можете найти в разделе `ch12/orders/oas.yaml` в репозитории GitHub для книги. В этой главе мы сосредоточимся на тестировании соответствия реализации API его спецификации, а именно на обеспечении того, чтобы API использовал правильные схемы, правильные статус-коды и т. д. Тестирование безопасности API — это совершенно другая тема, и чтобы ознакомиться с ней, рекомендую вам прочесть главу 11 книги Марка Уинтерингема *Testing Web APIs* (Manning, 2022) и книгу Кори Дж. Болла *Hacking APIs: Breaking Web Application Programming Interfaces* (No Starch Press, 2022).

Теперь скопируем реализацию API сервиса продукции из главы 10. Вернитесь на верхний уровень каталога `ch12`, введя `cd ..`, а затем выполните следующую команду:

```
$ cp -r ../ch10 products
```

Перейдите в `ch12/products` и запустите `pipenv install --dev` для установки зависимостей. Для тестирования API сервиса продукции мы будем использовать `pytest` и `schemathesis`, которые вы можете установить, выполнив команду:

```
$ pipenv install pytest schemathesis
```

Теперь можно приступать к тестированию API. Начнем наше путешествие с изучения фреймворка тестирования Dredd API.

## 12.2. ТЕСТИРОВАНИЕ REST API С ПОМОЩЬЮ DREDD

В этом разделе объясняется, что такое Dredd и как его можно использовать для тестирования REST API. Dredd — это фреймворк для тестирования API, который автоматически генерирует тесты для проверки поведения сервера API. Он генерирует тесты, парсируя спецификацию API и извлекая из нее информацию о том, как должен работать API. Если использовать Dredd во время разработки, можно сосредоточить усилия на создании API, в то время как фреймворк позволит гарантировать, что наша работа идет в верном направлении.

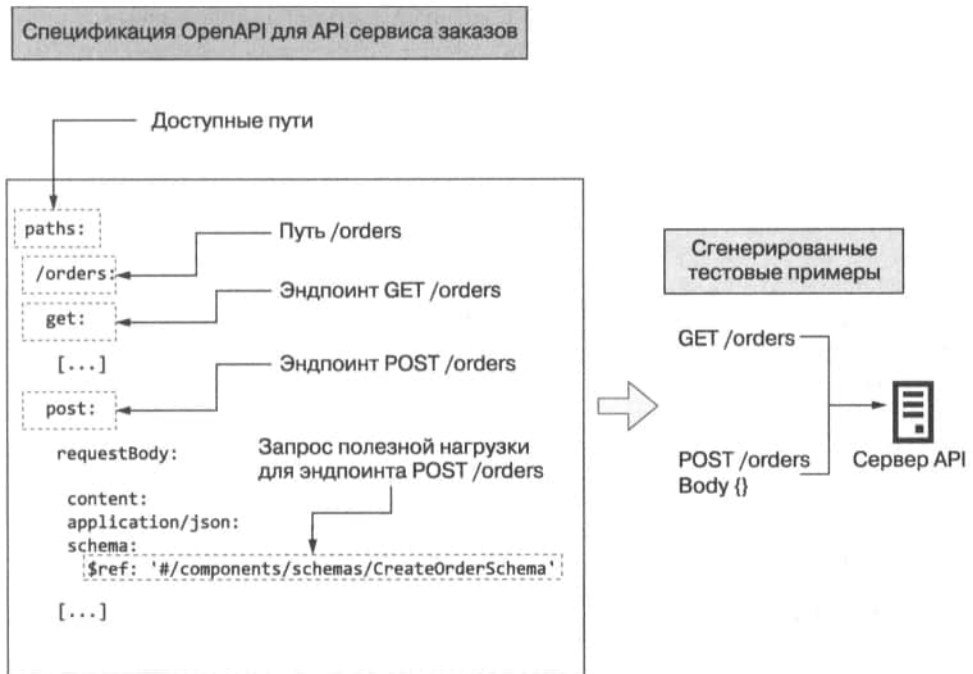


Dredd был выпущен компанией Apiary в 2017 году (<http://mng.bz/5maq>) и с тех пор стал частью необходимого набора инструментов каждого разработчика API.

В этом разделе мы узнаем, как работает Dredd, используя его для проверки реализации API сервиса заказов. Сначала запустим базовый набор тестов для API, а затем изучим более продвинутые функции фреймворка.

### 12.2.1. Что такое Dredd

Итак, Dredd – фреймворк для тестирования API. Как показано на рис. 12.1, он работает, парсируя спецификацию API и обнаруживая доступные URL-пути и методы HTTP, которые они принимают.



**Рис. 12.1.** Dredd работает, парсируя спецификацию API, обнаруживая доступные эндпоинты и запуская тесты для каждого из них

Чтобы протестировать API, Dredd отправляет запросы каждому эндпоинту, определенному в спецификации API, с ожидаемой полезной нагрузкой, если таковая имеется, а также с любыми параметрами запроса, принятыми эндпоинтом. Наконец, он проверяет, соответствуют ли ответы, которые получает API, схемам, объявленным в спецификации, и содержат ли они ожидаемые статус-коды.

## 12.2.2. Установка и запуск набора тестов Dredd по умолчанию

В этом подразделе мы установим Dredd и запустим его набор тестов по умолчанию с использованием API сервиса заказов. Перейдите в папку `ch12/orders` и запустите `pipenv shell`, чтобы активировать среду. Dredd — это пакет `npm`, то есть необходимо иметь на компьютере среду выполнения `Node.js`, а также инструмент управления пакетами для `JavaScript`, такой как `npm` или `Yarn`. Чтобы установить Dredd с помощью `npm`, запустите следующую команду из каталога `ch12/orders`:

```
$ npm install dredd
```

Это приведет к установке Dredd в папку с именем `node_modules/`. Как только установка будет завершена, можно использовать Dredd для тестирования API. Dredd поставляется с интерфейсом CLI, который доступен в следующем каталоге: `node_modules/.bin/Dredd`. Dredd CLI позволяет указывать некоторые аргументы (необязательные), которые дают нам большую гибкость в том, как запускать тесты. Воспользуемся некоторыми из них позже в этом разделе. А пока выполним простейшую команду Dredd для запуска теста:

```
$ ./node_modules/.bin/dredd oas.yaml http://127.0.0.1:8000 --server \
  "uvicorn orders.app:app"
```

Первым аргументом для Dredd CLI является путь к файлу спецификации API, а второй аргумент представляет собой базовый URL-адрес сервера API. С помощью опции `--server` мы сообщаем Dredd, какую команду необходимо использовать для запуска сервера API сервиса заказов. Если вы выполните эту команду сейчас, то получите несколько предупреждений от Dredd такого вида (под многоточием подразумевается путь к файлу спецификации API, который у вас будет другим):

```
warn: [...] (Orders API > /orders/{order_id}/cancel > Cancels an order >
  ➔ 200 > application/json): Ambiguous URI parameter in template:
  ➔ /orders/{order_id}/cancel
No example value for required parameter in API description document:
  ➔ order_id
```

Dredd не нравится, что мы не предоставили пример параметра URL `order_id`, который требуется в некоторых URL-путях. Фреймворк ругается, потому что не может генерировать случайные значения из спецификации. Чтобы ответить Dredd, добавляем пример параметра `order_id` в каждый URL, где он используется. Например, для URL-адреса `/orders/{order_id}` вносим изменения, показанные в листинге 12.1 (многоточия представляют пропущенный код). URL-адреса `/orders/{order_id}/pay` и `/orders/{order_id}/cancel` также содержат описания параметра `order_id`, поэтому добавьте примеры и к ним. Dredd будет использовать точное значение, указанное в примерах, для тестирования API.

**Листинг 12.1.** Добавление примеров для параметра URL-пути `order_id`

```
# file: orders/oas.yaml
```

```
[...]
```

```
  /orders/{order_id}:
```

```
    parameters:
```

```
      - in: path
```

```
        name: order_id
```

```
        required: true
```

```
        schema:
```

```
          type: string
```

```
          example: d222e7a3-6afb-463a-9709-38eb70cc670d
```

```
    get:
```

```
      [...]
```

Добавляем пример  
для параметра URL `order_id`

Как только мы добавим примеры для параметра `order_id`, мы сможем снова запустить Dredd CLI. На этот раз набор тестов запускается без проблем, и ваш результат должен быть подобен этому:

```
complete: 7 passing, 5 failing, 0 errors, 0 skipped, 12 total
```

```
complete: Tests took 90ms
```

```
INFO: Shutting down
```

```
INFO: Finished server process [23593]
```

В этом резюме говорится, что Dredd провел 18 тестов, из которых семь прошли успешно, а 11 провалились. Полный вывод теста слишком длинный, чтобы воспроизводить его здесь, но если вы прокрутите страницу вверх в терминале, то увидите, что неудачные тесты выполняются в эндпоинтах, нацеленных на определенные ресурсы:

- GET, PUT и DELETE `/orders/{order_id}`;
- POST `/orders/{order_id}/pay`;
- POST `/orders/{order_id}/cancel`.

Dredd запускает три теста для каждой из этих эндпоинтов и ожидает получить по одному успешному ответу на каждую. Однако при предыдущем выполнении фреймворк получил только ответы 404, а это означает, что сервер не смог найти запрошенные ресурсы. При тестировании этих эндпоинтов Dredd использует ID, который мы предоставили в качестве примера в листинге 12.1. Чтобы решить проблему, мы могли бы добавить жестко закодированный заказ с этим ID в наш список заказов в памяти (мы бы добавили его в базу данных, если бы использовали ее). Однако, как вы увидите в следующем разделе, лучше всего использовать хуки (hooks) Dredd.

Кроме того, произошел сбой теста для эндпоинта POST `/orders`, в которой Dredd ожидает ответа 422. Неудачные тесты в таких случаях происходят потому, что фреймворк не знает, как создавать тесты, генерирующие ответы 422, и в этом нам также помогут хуки Dredd.

### 12.2.3. Настройка набора тестов Dredd с помощью хуков

Поведение Dredd по умолчанию может быть ограничено. Как вы видели в подразделе 12.2.1, Dredd не знает, как обрабатывать эндпоинты с параметрами URL-пути, такими как `order_id` в URL-адресе `/orders/{order_id}`. Dredd также не знает, как создать случайный ID ресурса, и, если мы предоставим пример, фреймворк будет ожидать, что образец ID будет присутствовать в системе во время выполнения набора тестов. Такое ожидание бесполезно, поскольку это означает, что наш API доступен для тестирования только тогда, когда находится в определенном состоянии — когда конкретные ресурсы или фикстуры загружены в базу данных.

#### ОПРЕДЕЛЕНИЕ

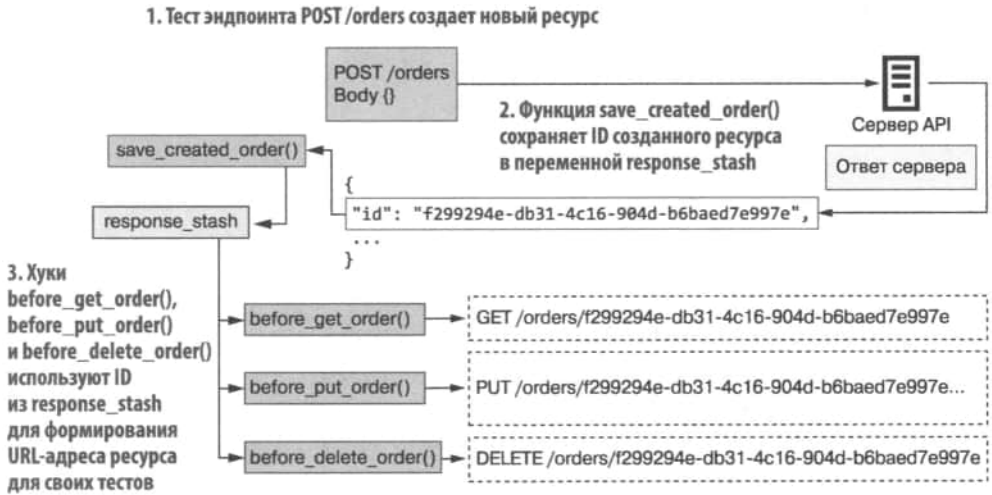
При тестировании программного обеспечения *фикстуры* определяют предварительные условия, необходимые для запуска теста. Как правило, фикстуры — это данные, которые мы загружаем в базу данных для тестирования, но они также могут представлять собой конфигурацию, каталоги и файлы или ресурсы инфраструктуры.

Вместо фикстур мы можем использовать хуки Dredd. Это скрипты, которые позволяют настраивать поведение Dredd во время выполнения набора тестов. Используя хуки Dredd, мы можем создавать ресурсы для применения во время теста, сохранять их ID и очищать по завершении тестирования.

Хуки Dredd позволяют нам запускать действия до и после всего набора тестов, а также до и после каждого теста, специфичного для эндпоинта. Их удобно применять для тестов с отслеживанием состояния, которые включают создание ресурсов и выполнение операций с ними. Например, можно использовать хуки для размещения заказа с помощью эндпоинта `POST /orders`, сохранения ID заказа и повторного использования ID для выполнения таких операций с заказом, как его оплата и отмена, с другими эндпоинтами. Используя этот подход, мы можем убедиться, что эндпоинт `POST /orders` выполняет свою работу по созданию ресурса, и протестировать другие эндпоинты с реальным ресурсом. Создадим несколько хуков, выполнив следующие шаги (рис. 12.2–12.4).

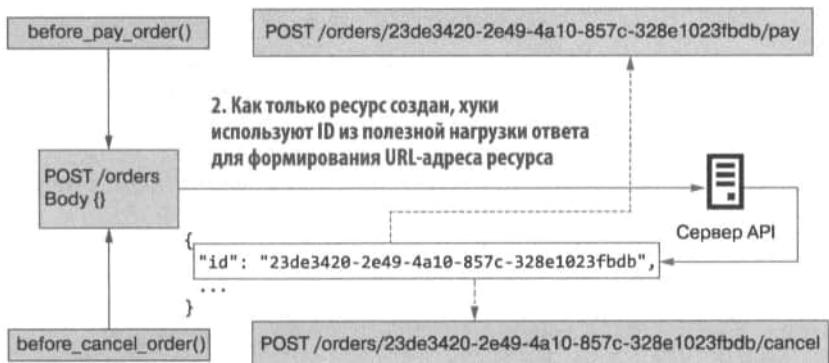
1. После проверки `POST /orders` мы используем хук для сохранения ID, возвращаемого сервером для вновь созданного заказа.
2. Перед тестами `GET`, `PUT` и `DELETE /orders/{order_id}` применим хуки, чтобы указать Dredd использовать ID из заказа, созданного в пункте 1. Эти эндпоинты применяются для получения сведений о заказе (`GET`), для изменения заказа (`PUT`) и его удаления с сервера (`DELETE`). Следовательно, после запуска теста `DELETE /orders/{order_id}` заказ больше не будет существовать на сервере.
3. Перед эндпоинтами `POST /orders/{order_id}/pay` и `POST /orders/{order_id}/cancel` используем хуки для создания новых заказов. Мы не сможем повторно использовать ID из пункта 1, так как тест `DELETE /orders/{order_id}` из пункта 2 удаляет заказ с сервера.

4. Для ответов 422 нам нужно сделать так, чтобы они генерировались с сервера. Будем использовать два подхода: для эндпоинта POST /orders отправим невалидную полезную нагрузку, в то время как для остальных эндпоинтов изменим URI заказа и укажем недопустимый ID.



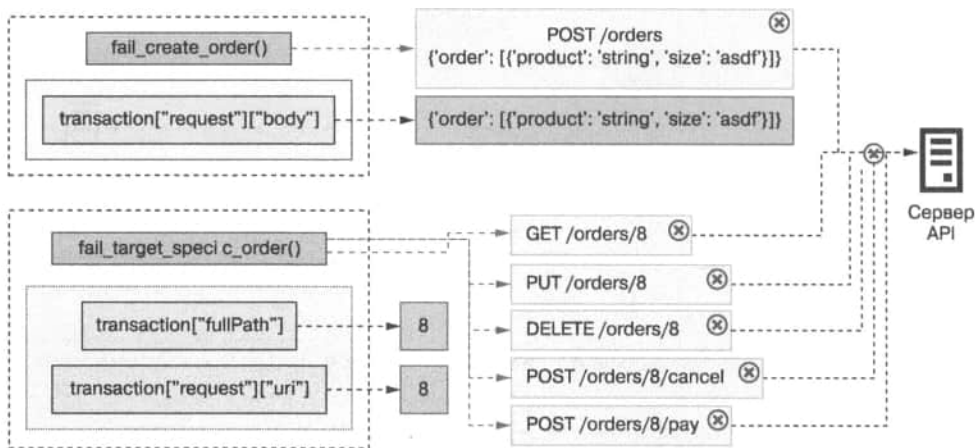
**Рис. 12.2.** После проверки эндпоинта POST /orders функция save\_created\_order() сохраняет ID из тела ответа сервера в переменной response\_stash. Хуки before\_get\_order(), before\_put\_order() и before\_delete\_order() используют ID из response\_stash для формирования URL-адресов своих ресурсов

- 1. Перед выполнением теста хуки before\_pay\_order() и before\_cancel\_order() используют эндпоинт POST /orders для создания нового ресурса, чтобы использовать его во время тестов**



**Рис. 12.3.** Перед выполнением теста хуки before\_pay\_order() и before\_cancel\_order() используют эндпоинт POST /orders для размещения нового заказа и используют ID из полезной нагрузки ответа в качестве URL-адресов своих ресурсов

Перед выполнением теста функция `fail_create_order()` вводит невалидную полезную нагрузку для теста эндпоинта `POST /orders`, в то время как функция `fail_target_specific_order()` вводит невалидный ID заказа для тестов одиночных эндпоинтов



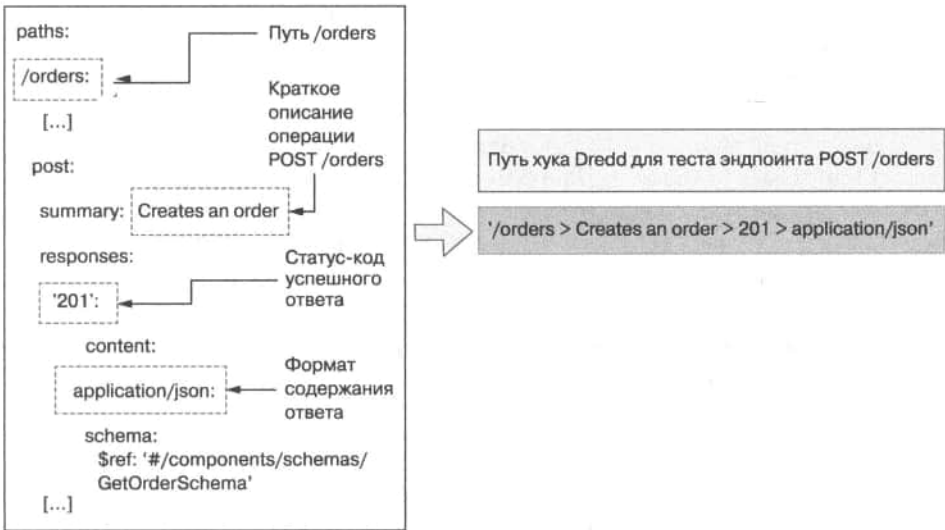
**Рис. 12.4.** Хуки `fail_create_order()` и `fail_target_specific_order()` вводят недопустимую полезную нагрузку и недопустимые ID заказов, чтобы вызвать ответ 422 от сервера

## Использование хуков Dredd для сохранения ID созданного ресурса

Теперь давайте наконец напомним хуки. Во-первых, если вы еще этого не сделали, перейдите в папку `ch12/orders` и активируйте виртуальную среду, запустив `shell1 ripenv`. Создайте файл с именем `orders/hooks.py` — в нем мы и добавим хуки. Хотя Dredd — это пакет `npm`, мы можем написать хуки на Python, используя библиотеку `dredd-hooks`. В разделе 12.1 мы уже настроили среды для этой главы, так что `dredd-hooks` должен быть установлен.

Чтобы понять, как работают хуки Dredd, рассмотрим один из них подробнее. В листинге 12.2 показана реализация хука `after` для эндпоинта `POST /orders`. Этот код находится в файле `orders/hooks.py`.

Сначала мы объявляем переменную с именем `response_stash`, которую будем использовать для хранения данных из запроса `POST /orders`. Библиотека `dredd-hooks` предоставляет функции декоратора: `dredd_hooks.before()` и `dredd_hooks.after()`, которые позволяют нам привязать функцию к определенной операции. Декораторы `dredd-hooks` принимают аргумент, представляющий путь к конкретной операции, к которой мы хотим привязать хук. Как вы можете видеть на рис. 12.5, в Dredd операция определяется как эндпоинт URL со статус-кодом ответа и форматом кодирования содержимого. В листинге 12.2 мы привязываем функцию `save_created_order()` к ответу 201 эндпоинта `POST /orders`.



**Рис. 12.5.** Чтобы сформировать путь для конкретной операции в хуке Dredd, мы используем URL-адрес с краткой информацией об операции, статус-кодом ответа и кодировкой содержимого ответа

### Листинг 12.2. Реализация хука after для эндпоинта POST /orders

# file: orders/hooks.py

```
import json
import dredd_hooks

response_stash = {}

@dredd_hooks.after('/orders > Creates an order > 201 > application/json')
def save_created_order(transaction):
    response_payload = transaction['real']['body']
    order_id = json.loads(response_payload)['id']
    response_stash['created_order_id'] = order_id
```

Импортируем библиотеку `dredd_hooks`

Создаем глобальный объект для хранения состояния набора тестов и управления им

Создаем хук, который будет запущен после теста эндпоинта `POST /orders`

Получаем доступ к полезной нагрузке ответа из эндпоинта `POST /orders`

Загружаем ответ, используя библиотеку `json` в Python, и извлекаем ID заказа

Храним ID заказа в нашем глобальном объекте `response_stash`

### ОПРЕДЕЛЯЕМ ПУТИ ОПЕРАЦИЙ В ХУКАХ DREDD

При определении пути для операции с использованием `dredd-hooks` нельзя применять HTTP-методы, то есть следующий синтаксис не будет работать: `/orders > post > 201 > application/json`. Вместо этого используйте другие свойства эндпоинта POST, такие как `summary` или `OperationId`, как в примере: `/orders > Creates an order > 201 > application/json`.

Хуки Dredd принимают аргумент, который обозначает транзакцию, выполненную Dredd во время теста. Аргумент представлен в виде словаря. В листинге 12.2 мы назвали аргумент хука `transaction`. Поскольку наша цель в хуке `save_created_order()` — получить ID созданного заказа, мы проверяем полезную нагрузку, возвращаемую эндпоинтом `POST /orders`, который можно найти в разделе `transaction['real']['body']`. Наш API возвращает полезные данные в формате JSON, поэтому загружаем его содержимое с помощью библиотеки JSON Python. Как только мы получаем ID заказа, сохраняем его для последующего использования в нашем глобальном словаре состояний, который мы назвали `response_stash`.

## Использование хуков для передачи Dredd пользовательских URL

Теперь, зная, как сохранить ID заказа, созданного в POST-запросе, посмотрим, как использовать этот ID для формирования URL-адреса ресурса заказа. В листинге 12.3 показано, как создать хуки для эндпоинтов ресурса заказа. Код, показанный в листинге 12.3, находится в файле `orders/hooks.py`. Код из листинга 12.2 опущен, а изменения выделены жирным шрифтом.

Чтобы указать, какой URL-адрес Dredd следует использовать при тестировании пути `/orders/{order_id}`, нам нужно изменить полезную нагрузку транзакции. В частности, изменим свойства `fullPath` транзакции и `uri` ее запроса и убедимся, что они указывают на правильный URL. Чтобы сформировать URL-адрес, мы получаем доступ к ID заказа из словаря `response_stash`.

**Листинг 12.3.** Использование хуков для того, чтобы сообщить Dredd, какой URL нам нужен

```
# file: orders/hooks.py
```

```
import json
import dredd_hooks
```

```
response_stash = {}
```

```
[...]
```

```
@dredd_hooks.before(
    '/orders/{order_id}' > Returns the details of a specific order > 200 > '
    'application/json'
```

```
)
def before_get_order(transaction):
    transaction["fullPath"] = (
```

```
        "/orders/" + response_stash["created_order_id"]
```

```
    )
    transaction['request']['uri'] = (
        '/orders/' + response_stash['created_order_id']
    )
```

← Создаем хук, который будет запущен перед тестированием эндпоинта GET /orders/{order\_id}

← Изменяем URL-адрес теста эндпоинта GET /orders/{order\_id}, чтобы включить ID заказа, который мы создали ранее



```

@dredd_hooks.before(
    '/orders/{order_id}' > Replaces an existing order > 200 > '
    'application/json'
)
def before_put_order(transaction):
    transaction['fullPath'] = (
        '/orders/' + response_stash['created_order_id']
    )
    transaction['request']['uri'] = (
        '/orders/' + response_stash['created_order_id']
    )

@dredd_hooks.before('/orders/{order_id}' > Deletes an existing order > 204')
def before_delete_order(transaction):
    transaction['fullPath'] = (
        '/orders/' + response_stash['created_order_id']
    )
    transaction['request']['uri'] = (
        '/orders/' + response_stash['created_order_id']
    )

```

## Использование хуков Dredd для создания ресурсов перед тестированием

Эндпоинт DELETE `/orders/{order_id}` удаляет заказ из базы данных, поэтому мы не можем использовать один и тот же ID заказа для проверки эндпоинтов `/orders/{order_id}/pay` и `/orders/{order_id}/cancel`. Вместо этого добавим хуки для создания новых заказов перед тестированием этих эндпоинтов. В листинге 12.4 показано, как это сделать. Код в нем относится к файлу `orders/hooks.py`. Новый код выделен жирным шрифтом, в то время как код из предыдущих листингов сокращен.

Чтобы создавать новые заказы, вызовем эндпоинт POST `/orders`, используя библиотеку `requests`, которая упрощает выполнение HTTP-запросов. Для запуска POST-запроса воспользуемся функцией `post()`, передавая целевой URL-адрес запроса и полезную нагрузку JSON, необходимую для создания заказа. В этом случае мы жестко кодируем базовый URL сервера в `http://127.0.0.1:8000`, но вы можете указать свое значение, если хотите иметь возможность запускать набор тестов в разных средах.

Как только мы создали заказ, извлекаем его ID из полезной нагрузки ответа и используем этот ID для изменения свойств `fullPath` транзакции и `uri` ее запроса.

### Листинг 12.4. Использование хука для создания ресурсов перед тестированием

```
# file: orders/hooks.py
```

```

import json
import dredd_hooks
import requests

```

Импортируем  
библиотеку requests

```

response_stash = {}

[...]

@dredd_hooks.before(
    '/orders/{order_id}/pay > Processes payment for an order > 200 > '
    'application/json'
)
def before_pay_order(transaction):
    response = requests.post(  ← Размещаем новый заказ
        "http://127.0.0.1:8000/orders",
        json={
            "order": [{"product": "string", "size": "small", "quantity":1}]
        },  ← Извлекаем идентификатор вновь созданного заказа
    )
    id_ = response.json()['id']  ← Изменяем URL-адрес тестируемого эндпоинта
    transaction['fullPath'] = '/orders/' + id_ + '/pay'  ← POST /orders/{order_id}/pay, включив в него
    transaction['request']['uri'] = '/orders/' + id_ + '/pay'  ← идентификатор заказа, который создали ранее

@dredd_hooks.before(
    '/orders/{order_id}/cancel > Cancels an order > 200 > application/json'
)
def before_cancel_order(transaction):
    response = requests.post(
        "http://127.0.0.1:8000/orders",
        json={
            "order": [{"product": "string", "size": "small", "quantity":1}]
        },
    )
    id_ = response.json()['id']
    transaction['fullPath'] = '/orders/' + id_ + '/cancel'
    transaction['request']['uri'] = '/orders/' + id_ + '/cancel'

```

## Использование хуков для генерации ответов 422

Некоторые эндпоинты в API сервиса заказов принимают полезную нагрузку запроса или параметры URL-пути. Если клиент API отправляет невалидную полезную нагрузку или использует недопустимый параметр URL-пути, API выдает ответ 422. Как мы видели ранее, Dredd не знает, как сгенерировать ответ 422 с сервера, поэтому мы создадим для этого хуки.

Как вы можете видеть в листинге 12.5, нам нужны только две функции:

- `fail_create_order()` перехватывает запрос для эндпоинта `POST /orders` до того, как он достигнет сервера, и изменяет его полезную нагрузку с недопустимым значением для свойства `size`;
- `fail_target_specific_order()` изменяет URI заказа на недопустимый ID. Поскольку мы знаем, что Dredd запускает этот тест, используя образец ID, который мы указали в спецификации API, нам просто нужно заменить этот ID недопустимым значением. Тип параметра пути `order_id` — UUID, поэтому, если заменить его целым числом, сервер ответит статус-кодом 422.

**Листинг 12.5.** Генерирование ответов 422 с помощью перехватчиков Dredd

```
# file: orders/hooks.py

@dredd_hooks.before('/orders > Creates an order > 422 > application/json')
def fail_create_order(transaction):
    transaction["request"]["body"] = json.dumps(
        {"order": [{"product": "string", "size": "asdf"]}}
    )

@dredd_hooks.before(
    "/orders/{order_id} > Returns the details of a specific order > 422 > "
    "application/json"
)

@dredd_hooks.before(
    "/orders/{order_id}/cancel > Cancels an order > 422 > application/json"
)

@dredd_hooks.before(
    "/orders/{order_id}/pay > Processes payment for an order > 422 > "
    "application/json"
)

@dredd_hooks.before(
    "/orders/{order_id} > Replaces an existing order > 422 > "
    "application/json"
)

@dredd_hooks.before(
    "/orders/{order_id} > Deletes an existing order > 422 > "
    "application/json"
)

def fail_target_specific_order(transaction):
    transaction["fullPath"] = transaction["fullPath"].replace(
        "d222e7a3-6afb-463a-9709-38eb70cc670d", "8"
    )
    transaction["request"]["uri"] = transaction["request"]["uri"].replace(
        "d222e7a3-6afb-463a-9709-38eb70cc670d", "8"
    )
```

Хуки позволяют протестировать, как сервер ведет себя с различными типами полезной нагрузки и параметров, и, если нужно, вы можете создать специальные тесты для каждого эндпоинта, чтобы полнее охватить тестированием API.

**Запуск Dredd с пользовательскими хуками**

Теперь, когда у нас есть хуки Dredd, позволяющие убедиться, что каждый URL сформирован правильно, мы можем снова запустить набор тестов Dredd. Следующая команда показывает, как запустить Dredd с помощью файла хуков:

```
$ ./node_modules/.bin/dredd oas.yaml http://127.0.0.1:8000 --server \
"uvicorn orders.app:app" --hookfiles=./hooks.py --language=python
```

Нам просто нужно передать путь к нашему файлу, используя флаг `--hookfiles`. Кроме того, следует указать язык, на котором написаны хуки, используя флаг `--language`. Если вы запустите команду сейчас, то увидите, что все тесты пройдены.

## 12.2.4. Использование Dredd для тестирования API

Dredd — фантастический инструмент для тестирования реализаций API, но его набор тестов ограничен. Он проверяет только удачный сценарий работы каждого эндпоинта («счастливый путь»). Например, чтобы протестировать эндпоинт `POST /orders`, Dredd отправляет в него только валидную полезную нагрузку и ожидает, что она будет корректно обработана. Он не отправляет некорректную полезную нагрузку, поэтому, используя только Dredd, мы не сможем узнать, как сервер реагирует в таких ситуациях. Это некритично, если мы находимся на ранней стадии разработки сервиса.

Однако, прежде чем выпустить код, мы должны убедиться, что он работает должным образом во всех ситуациях, и для запуска тестов, выходящих за рамки «счастливого пути», нам стоит использовать другую библиотеку: `schemathesis`. Мы поговорим о ней в разделе 12.4, но сейчас попробуем разобраться с основным подходом к тестированию, который использует Schemathesis: с тестированием на основе свойств.

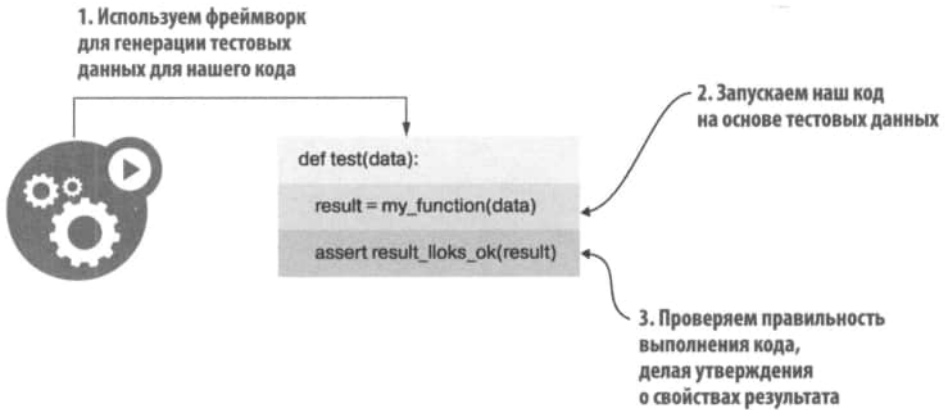
## 12.3. ВВЕДЕНИЕ В ТЕСТИРОВАНИЕ НА ОСНОВЕ СВОЙСТВ

В этом разделе объясняется, что такое тестирование на основе свойств, как оно работает и как помогает писать более исчерпывающие тесты для API. Попутно вы также познакомитесь с отличной Python-библиотекой тестирования на основе свойств — `hypothesis`. Как вы увидите, тестирование на основе свойств помогает создавать надежные наборы тестов для API, позволяя легко генерировать сотни тестовых примеров с множеством комбинаций свойств и типов. Этот раздел подготавливает почву для последующих разделов главы, где вы узнаете о Schemathesis, платформе тестирования API, которая использует тестирование на основе свойств.

### 12.3.1. Что такое тестирование на основе свойств

Как вы можете видеть на рис. 12.6, тестирование на основе свойств — это стратегия тестирования, при которой мы вводим тестовые данные в код и разрабатываем тесты таким образом, чтобы они предъявляли требования к результату выполнения

нашего кода<sup>1</sup>. Как правило, фреймворк, основанный на свойствах, генерирует для нас тестовые примеры с учетом набора условий, которые мы определяем.



**Рис. 12.6.** При тестировании на основе свойств мы используем фреймворк для генерации тестовых примеров для наших функций и делаем утверждения о результате выполнения нашего кода в таких случаях

## ОПРЕДЕЛЕНИЕ

Тестирование на основе свойств — это подход к тестированию, при котором мы предъявляем требования к свойствам возвращаемого значения наших функций или методов. Вместо того чтобы вручную писать множество различных тестов с разными входными данными, мы позволяем фреймворку генерировать для нас входные данные и определяем ожидаемый способ их обработки. В Python отличной библиотекой для тестирования на основе свойств является Hypothesis (<https://github.com/HypothesisWorks/hypothesis>).

### 12.3.2. Традиционный подход к тестированию API

Допустим, мы хотим протестировать наш эндпоинт `POST /orders`, чтобы убедиться, что он принимает только действительную полезную нагрузку. Как вы можете видеть из спецификации API сервиса заказов в файле `ch012/orders/oas.yaml`, допустимая полезная нагрузка для эндпоинта `POST /orders` содержит ключ с именем `order`, который представляет собой массив упорядоченных элементов (листинг 12.6). У каждого товара есть два обязательных ключа: товар и объем.

<sup>1</sup> Смотрите превосходную статью: *MacIver D. R. What is Property Based Testing?* («Что такое тестирование на основе свойств?»). <https://hypothesis.works/articles/what-is-property-based-testing/>.

**Листинг 12.6.** Схема для полезной нагрузки запроса  
эндпоинта POST /orders

# file: orders/oas.yaml

```

components:
  schemas:
    OrderItemSchema:
      type: object
      additionalProperties: false
      required:
        - product
        - size
      properties:
        product:
          type: string
        size:
          type: string
          enum:
            - small
            - medium
            - big
        quantity:
          type: integer
          format: int64
          default: 1
          minimum: 1

    CreateOrderSchema:
      type: object
      additionalProperties: false
      required:
        - order
      properties:
        order:
          type: array
          minItems: 1
          items:
            $ref: '#/components/schemas/OrderItemSchema'

```

При традиционном подходе мы бы записывали различную полезную нагрузку вручную, затем отправляли ее в эндпоинт POST /orders и фиксировали ожидаемый результат для каждой полезной нагрузки. В листинге 12.7 показано, как протестировать эндпоинт POST /orders с двумя видами полезной нагрузки. Если вы хотите опробовать код, приведенный в листинге 12.7, создайте файл с именем orders/test.py и запустите тесты с помощью команды `pytest test.py`.

**Листинг 12.7.** Тестирование эндпоинта POST /orders с различной полезной нагрузкой

# file: orders/test.py

```

from fastapi.testclient import TestClient

```

Импортируем класс  
TestClient из FastAPI

```

from orders.app import app

test_client = TestClient(app=app)

def test_create_order_fails():
    bad_payload = {
        'order': [{'product': 'coffee'}]
    }
    response = test_client.post('/orders', json=bad_payload)
    assert response.status_code == 422

def test_create_order_succeeds():
    good_payload = {
        'order': [{'product': 'coffee', 'size': 'big'}]
    }
    response = test_client.post('/orders', json=good_payload)
    assert response.status_code == 201

```

Создаем экземпляр тестового клиента

Создаем тест

Определяем плохую полезную нагрузку для эндпоинта POST /orders

Тестируем полезную нагрузку

Подтверждаем, что статус-код ответа равен 422

Определяем допустимую полезную нагрузку для эндпоинта POST /orders

Подтверждаем, что статус-код ответа равен 201

В листинге 12.7 мы определяем два тестовых примера: один с недопустимой полезной нагрузкой, в которой отсутствует требуемое свойство `size` элемента заказа, и другой — с допустимой полезной нагрузкой. В обоих случаях мы используем тестовый клиент FastAPI для отправки полезной нагрузки на наш сервер API и тестируем поведение сервера, проверяя статус-код из ответа. Мы ожидаем, что ответ для недопустимой полезной нагрузки будет содержать код 422 (Unprocessable Entity), а для допустимой — код 201 (Created). FastAPI использует `pydantic` для проверки полезной нагрузки и автоматически генерирует ответ 422 для тех, что неправильно сформированы. Таким образом, этот тест служит для проверки правильности реализации наших моделей `pydantic`.

### 12.3.3. Тестирование на основе свойств с использованием Hypothesis

Стратегия тестирования, приведенная в листинге 12.7, где мы пишем все тестовые примеры вручную, является распространенным подходом к тестированию API. Проблема с ним в том, что он довольно ограничен, если только мы не готовы потратить много часов на написание исчерпывающих наборов тестов. Набор тестов в листинге 12.7 далек от совершенства: он не проверяет, что произойдет, если свойство `size` содержит недопустимое значение, или если свойству `quantity` присвоено отрицательное значение, или если список позиций заказа пуст.

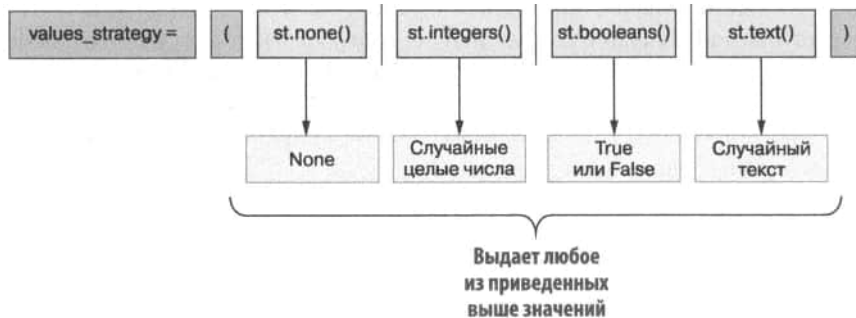
Для более комплексного подхода к тестированию API хотелось бы иметь фреймворк, который может генерировать все возможные типы полезной нагрузки и тестировать их на нашем сервере API. Это именно то, что позволяет нам делать тестирование на основе свойств. В Python мы можем запускать тесты, основанные на свойствах, с помощью отличной библиотеки `hypothesis`.

Hypothesis использует для генерации тестовых данных стратегии. Например, если мы хотим сгенерировать случайные целые числа, мы используем стратегию `integers()`, а если мы хотим сгенерировать текстовые данные, используем стратегию `text()`. Стратегии Hypothesis предоставляют также метод `example()`, который можно использовать, чтобы посмотреть, какие значения они генерируют. Вы можете получить представление о том, как работают стратегии Hypothesis, поэкспериментировав с ними в оболочке Python (поскольку Hypothesis генерирует случайные значения, вы увидите разные результаты):

```
>>> from hypothesis import strategies as st
>>> st.integers().example()
0
>>> st.text().example()
'н'
```

Как вы можете видеть на рис. 12.7, Hypothesis также позволяет комбинировать различные стратегии с помощью оператора `pipe()`. Например, мы можем определить стратегию, которая генерирует либо целые числа, либо текст:

```
>>> strategy = st.integers() | st.text()
>>> strategy.example()
-2781
```



**Рис. 12.7.** Мы можем объединить различные стратегии Hypothesis в одну. Результирующая стратегия произвольно выдаст значение любой из скомбинированных стратегий

Чтобы проверить эндпоинт `POST /orders` с помощью Hypothesis, определим стратегию, которая создает словари со случайными значениями. Для работы со словарями мы можем использовать либо `dictionaries()` из Hypothesis, либо стратегии `fixed_dictionaries()`. Например, если мы хотим сгенерировать словарь с двумя ключами, `product` и `size`, где каждый ключ может быть либо целым числом, либо текстом, можем воспользоваться следующим объявлением:

```
>>> strategy = st.fixed_dictionaries(
    {
        "product": st.integers() | st.text(),
```



```

        "size": st.integers() | st.text(),
    }
)

>>> strategy.example()
{'product': -7958791642907854994, 'size': 16875}

```

### 12.3.4. Использование Hypothesis для тестирования эндпоинта REST API

Соберем все вместе, чтобы создать реальный тест для эндпоинта `POST /orders`. Сначала определим стратегию для всех значений, которые может принимать свойство в нашей полезной нагрузке. Упростим пример и предположим, что свойства могут содержать только `null`, логические значения, текст или целые числа:

```

>>> values_strategy = (
    st.none() |
    st.booleans() |
    st.text() |
    st.integers()
)

```

Теперь определим стратегию для схемы, представляющей элемент `order`. Чтобы упростить задачу, воспользуемся фиксированным словарем с допустимыми ключами, то есть `product`, `size` и `quantity`. Поскольку свойство `size` может принимать только значения `small`, `medium` или `big`, добавим стратегию, которая позволит Hypothesis выбирать значение либо из этого перечня, либо из стратегии `values_strategy`, которую мы определили ранее:

```

>>> order_item_strategy = st.fixed_dictionaries(
    {
        "product": values_strategy,
        "size": st.one_of(st.sampled_from(("small", "medium", "big")))
        | values_strategy,
        "quantity": values_strategy,
    }
)

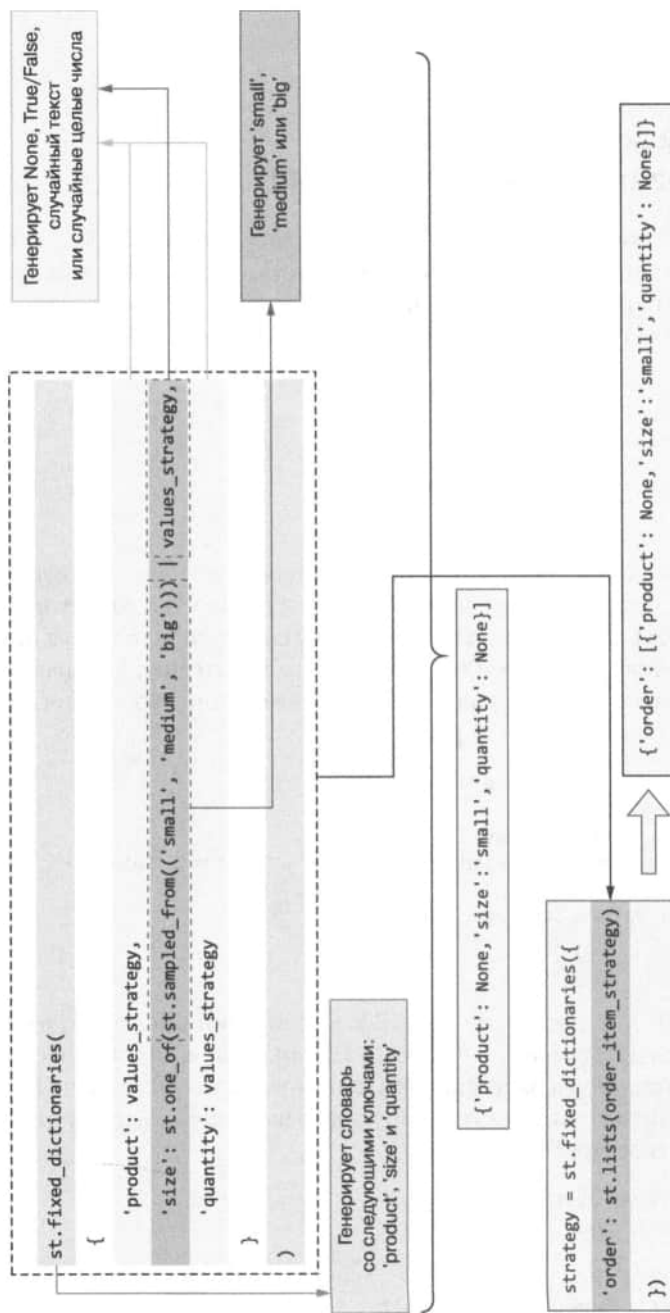
```

Наконец, как вы можете видеть на рис. 12.8, мы объединили все это в стратегию для схемы `CreateOrderSchema`. Из листинга 12.4 мы знаем, что для нее требуется свойство `order`, которое содержит список элементов заказа. Используя Hypothesis, мы можем определить стратегию, генерирующую полезную нагрузку для тестирования схемы `CreateOrderSchema`:

```

>>> strategy = st.fixed_dictionaries({
    'order': st.lists(order_item_strategy)
})
>>> strategy.example()
{'order': [{'product': None, 'size': 'small', 'quantity': None}]}

```



**Рис. 12.8.** Комбинируя стратегию `Hypothesis fixed_dictionaries()` со стратегиями `lists()` и `values_strategy`, мы можем создавать полезную нагрузку, которая напоминает схему `CreateOrderSchema`

Теперь мы готовы переписать наш набор тестов из листинга 12.6 в более общий и всеобъемлющий тест для эндпоинта `POST /orders`. В листинге 12.7 показано, как мы вводим стратегии `Hypothesis` в тестовую функцию. Этот код нужно добавить в файл `orders/test.py`. Я опустил в листинге определения переменных `values_strategy` и `order_item_strategy`, поскольку мы уже сталкивались с ними в предыдущих примерах.

Стратегия тестирования, приведенная в листинге 12.8, использует библиотеку `jsonschema` для проверки полезной нагрузки, сгенерированной `Hypothesis`. Чтобы проверить полезную нагрузку с помощью `jsonschema`, мы сначала загружаем спецификацию OpenAPI для API сервиса заказов, которая находится в `ch012/orders/oas.yaml`. Считываем содержимое файла, используя метод `Path().read_text()` от `pathlib`, и парсируем его с помощью библиотеки `yaml` в Python. Чтобы проверить допустимость полезной нагрузки, создаем служебную функцию `is_valid_payload()`, которая будет возвращать `True`, если полезная нагрузка допустима, и `False` в ином случае.

Мы проверяем полезную нагрузку с помощью функции `validate()` от `jsonschema`, а она требует два аргумента: полезную нагрузку, которую мы хотим проверить, и схему, по которой хотим провести проверку. Поскольку `CreateOrderSchema` содержит ссылку на другую схему в спецификации API, а именно на `OrderItemSchema`, мы также предоставляем резольвер, который `jsonschema` может использовать для разрешения ссылок на другие схемы в документе. Функция `validate()` выдает `ValidationError`, если полезная нагрузка недопустима, поэтому мы вызываем ее в блоке `try/except` и возвращаем `True` или `False` в зависимости от результата.

### Листинг 12.8. Использование `Hypothesis` для запуска тестов на основе свойств с использованием API

```
# file: orders/test.py
```

```
from pathlib import Path

import hypothesis.strategies as st
import jsonschema
import yaml
from fastapi.testclient import TestClient
from hypothesis import given, Verbosity, settings
from jsonschema import ValidationError, RefResolver

from orders.app import app

orders_api_spec = yaml.full_load(
    (Path(__file__).parent/ 'oas.yaml').read_text()
)
create_order_schema = (
    orders_api_spec['components']['schemas']['CreateOrderSchema']
)

def is_valid_payload(payload, schema):
    try:
        jsonschema.validate(
```

Загружаем спецификацию API

Указатель на схему `CreateOrderSchema`

Вспомогательная функция для определения того, является ли полезная нагрузка допустимой

```

        payload, schema=schema,
        resolver=RefResolver('', orders_api_spec)
    )
except ValidationError:
    return False
else:
    return True

test_client = TestClient(app=app)
values_strategy = [...]
order_item_strategy = [...]
strategy = [...]

@given(strategy)
def test(payload):
    response = test_client.post('/orders', json=payload)
    if is_valid_payload(payload, create_order_schema):
        assert response.status_code == 201
    else:
        assert response.status_code == 422

```

Проверяем полезную нагрузку с помощью функции `validate() jsonschema`

Создаем экземпляр тестового клиента

Вводим стратегии Hypothesis в нашу тестовую функцию

Фиксируем каждый тестовый пример с помощью аргумента `payload`

Отправляем полезную нагрузку в эндпоинт `POST/orders`

Задаем ожидаемый статус-код в зависимости от того, является ли полезная нагрузка допустимой

Чтобы мы могли ввести данные в тестовые функции, Hypothesis предоставляет декоратор `given()`, который принимает стратегию Hypothesis в качестве аргумента и использует ее для передачи тестовых примеров в нашу функцию. Если полезная нагрузка валидна, мы ожидаем, что наш API вернет ответ со статус-кодом 201, в то время как для некорректной полезной нагрузки стоит ожидать статус-код 422.

Как оказалось, Hypothesis очень подходит для генерации наборов данных на основе JSON Schema, и уже существует библиотека, которая преобразует схемы в стратегии Hypothesis, так что вам не нужно делать это самостоятельно. Это `hypothesis-jsonschema` (<https://github.com/ZacHD/hypothesis-jsonschema>). Я настоятельно рекомендую вам ознакомиться с этой библиотекой, прежде чем пытаться генерировать собственные стратегии Hypothesis для тестирования веб-API.

Теперь, когда вы понимаете, что такое тестирование на основе свойств и как работает Hypothesis, вы готовы узнать о Schemathesis, который является темой нашего следующего раздела.

## 12.4. ТЕСТИРОВАНИЕ REST API С ПОМОЩЬЮ SCHEMATHESIS

В этом разделе мы рассмотрим Schemathesis: как он работает и как пользоваться им для тестирования REST API. Schemathesis — это фреймворк, который использует тестирование на основе свойств. В него встроена библиотека `hypothesis`, и он способен запускать более полный набор тестов, чем Dredd. Как только вы будете

готовы выпустить свои API в производство, рекомендую вам протестировать их с помощью Schemathesis, чтобы убедиться, что вы охватываете все возможные сценарии.

### 12.4.1. Запуск набора тестов Schemathesis по умолчанию

В этом разделе мы запустим стандартный набор тестов Schemathesis. Поскольку мы уже установили зависимости в разделе 12.1, нам нужно лишь перейти в папку `orders` и активировать нашу среду, выполнив команду `pipenv shell`. В отличие от Dredd, Schemathesis требует, чтобы перед запуском набора тестов был запущен сервер API. Вы можете запустить сервер в новом окне терминала или переведя его в фоновый режим с помощью следующей команды:

```
$ uvicorn orders.app:app &
```

Символ `&` переводит процесс в фоновый режим. Затем вы можете запустить Schemathesis следующей командой:

```
$ schemathesis run oas.yaml --base-url=http://localhost:8000 \
--hypothesis-database=none
```

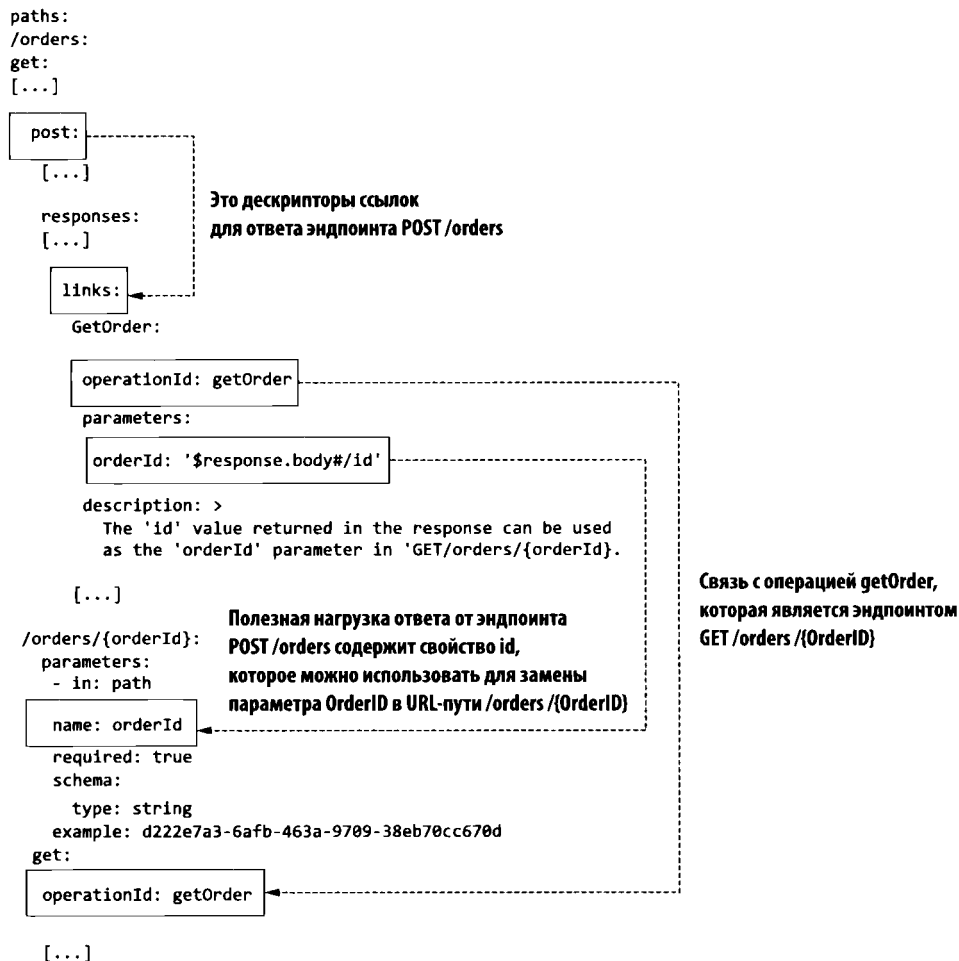
Hypothesis, библиотека, которую Schemathesis использует для создания тестовых примеров, создает папку с именем `.hypothesis/`, где она кэширует некоторые из своих тестов. По моему опыту, кэширование иногда приводит к результатам, вводящим в заблуждение при последующем выполнении тестов, поэтому до тех пор, пока это не будет исправлено, лучше избегать его. Мы устанавливаем флаг `--hypothesis-database=none`, чтобы Schemathesis не кэшировал тестовые случаи.

После выполнения команды вы увидите, что Schemathesis запускает около 700 тестов в отношении API, тестируя все возможные комбинации параметров, типов и форматов. Все тесты должны выполняться. Как только Schemathesis завершит работу, вы можете вывести процесс Uvicorn на передний план, выполнив команду `fg`, и остановить его. (Надеюсь, вы помните, что для остановки процесса нужно нажать `Ctrl+C`.)

### 12.4.2. Использование ссылок для расширения набора тестов Schemathesis

Набор тестов, который мы запускаем с помощью Schemathesis, имеет одно серьезное ограничение: он не проверяет, правильно ли эндпоинт `POST /orders` создает заказы и можем ли мы выполнять ожидаемые операции с заказом, в частности оплачивать и отменять его. Это просто запуск независимых и несвязанных запросов к каждому из эндпоинтов в API сервиса заказов. Чтобы проверить, правильно ли мы создаем ресурсы, нам нужно дополнить спецификацию API ссылками. Как вы можете видеть на рис. 12.9, в стандарте OpenAPI ссылки — это объявления

(declarations), которые позволяют нам описывать взаимосвязи между различными эндпоинтами<sup>1</sup>.



**Рис. 12.9.** В OpenAPI мы можем использовать ссылки для описания взаимосвязей между эндпоинтами. Например, ответ POST /orders содержит свойство id, которое можно использовать для замены параметра order\_id в URL-адресе /orders/{order\_id}

Например, с помощью ссылок мы можем указать, что эндпоинт POST /orders возвращает полезную нагрузку с ID и что мы можем использовать этот ID для

<sup>1</sup> Как работают ссылки и как использовать их в документации по API, подробнее объясняется по адресу <https://swagger.io/docs/specification/links/>.

формирования URL-адреса ресурса заказа, только что созданного в эндпоинте GET /orders/{order\_id}. Мы используем ID операций для описания взаимосвязей между нашими эндпоинтами. Как вы узнали в разделе 5.3, ID операций являются уникальными для каждого эндпоинта в API. В листинге 12.9 показано, как добавить в API сервиса заказов ссылку, описывающую взаимосвязь между эндпоинтами POST /orders и GET /orders/{order\_id}. Полный список ссылок вы найдете в файле ch12/orders/oas\_with\_links.yaml в репозитории GitHub для этой книги. Я скрыл за многоточиями части кода, которые не имеют отношения к примеру, а новый код выделил жирным шрифтом.

**Листинг 12.9.** Добавление ссылок для создания связей между эндпоинтами в OpenAPI

```
# file: orders/oas.yaml
```

```
paths:
```

```
  /orders:
```

```
    get:
```

```
      [...]
```

```
    post:
```

```
      operationId: createOrder
```

```
      summary: Creates an order
```

```
      requestBody:
```

```
        required: true
```

```
        content:
```

```
          application/json:
```

```
            schema:
```

```
              $ref: '#/components/schemas/CreateOrderSchema'
```

```
      responses:
```

```
        '201':
```

```
          description: A JSON representation of the created order
```

```
          content:
```

```
            application/json:
```

```
              schema:
```

```
                $ref: '#/components/schemas/GetOrderSchema'
```

```
      links: ← Добавляем ссылки на эндпоинт POST /orders
```

```
        GetOrder:
```

```
          operationId: getOrder ←
```

```
          parameters:
```

```
            order_id: '$response.body#/id' ←
```

```
            description: >
```

```
              The `id` value returned in the response can be used as  
              the `order_id` parameter in `GET/orders/{order_id}`
```

```
          [...]
```

```
/orders/{order_id}:
```

```
  [...]
```

```
    get:
```

```
      operationId: getOrder
```

```
      [...]
```

Объясняем,  
как работает  
эта ссылка

Определяем связь с эндпоинтом  
GET /orders/{order\_id}

Параметр order\_id URL в эндпоинте  
getOrder может быть заменен  
свойством id полезной нагрузки ответа

В листинге 12.9 мы даем название связи между эндпоинтами POST /orders и GET /order/{order\_id} — getOrder. Свойство operationId идентифицирует эндпоинт, на который ведет ссылка getOrder. Эндпоинт GET /order/{order\_id} имеет параметр URL с именем order\_id, а свойство parameters сообщает нам, что тело ответа от эндпоинта POST /orders содержит свойство id, которое мы можем использовать для замены order\_id в эндпоинте GET /order/{order\_id}.

Теперь мы можем запустить Schemathesis и оценить работу наших ссылок, выполнив следующую команду:

```
$ schemathesis run oas_with_link.yaml --base-url=http://localhost:8000 \
--stateful=links
```

Флаг --stateful=links указывает Schemathesis искать ссылки в документации и использовать их для запуска тестов на ресурсах, созданных через эндпоинт POST /orders. Если вы запустите Schemathesis сейчас, то увидите, что он выполняет более тысячи тестов с использованием API. Поскольку фреймворк генерирует случайные тесты, количество тестовых случаев в разное время может быть разным. В листинге 12.10 показан вывод набора тестов Schemathesis после его запуска с параметром --stateful, установленным равным links. В списке опущены первые несколько строк, поскольку они содержат только системные метаданные. Обратите внимание, что некоторые тесты отображаются вложенными в эндпоинт POST /orders (строки, начинающиеся с символа ->). Вложенные тесты — это тесты, которые используют ссылки из нашей документации по API. Если тесты по ссылкам эндпоинта POST /orders пройдут успешно, мы можем быть уверены, что наши ресурсы созданы правильно.

Листинг 12.10. Вывод набора тестов Schemathesis

[...]  
Base URL: http://localhost:8000  
Specification version: Open API 3.0.3  
Workers: 1  
Collected API operations: 7

Базовый URL-адрес сервера  
Версия OpenAPI, используемая нашим сервером  
Количество процессов, параллельно выполняющих набор тестов  
Количество операций, определенных в спецификации API

GET /orders .  
POST /orders .  
    -> GET/orders/{order\_id} .  
    -> PUT/orders/{order\_id} .  
    -> DELETE/orders/{order\_id} .  
    -> POST/orders/{order\_id}/cancel .  
    -> POST/orders/{order\_id}/pay .  
GET/orders/{order\_id} .  
PUT/orders/{order\_id} .  
DELETE/orders/{order\_id} .  
POST/orders/{order\_id}/pay .  
POST/orders/{order\_id}/cancel .

Тестируем эндпоинт GET /orders  
[ 14%]  
[ 28%]  
[ 37%]  
[ 44%]  
[ 50%]  
[ 54%]  
[ 58%]  
[ 66%]  
[ 75%]  
[ 83%]  
[ 91%]  
[100%]  
Тестируем эндпоинт GET /orders/{order\_id}, связанный с эндпоинтом POST /orders



```

===== SUMMARY =====
Performed checks:
    not_a_server_error          1200/ 1200 passed          PASSED
===== 12 passed in 57.57s =====

```

←

В наборе 1200 тестов, и все они проходят успешно

В результатах предыдущего теста говорится, что наш API прошел все проверки в категории `not_a_server_error`. По умолчанию Schemathesis проверяет только то, что API не вызывает ошибок сервера, но его можно настроить и для проверки того, что API использует правильные статус-коды, типы контента, заголовки и схемы, как описано в спецификации API. Чтобы активировать все эти проверки, мы добавляем флаг `--checks` и устанавливаем для него значение `all`:

```
$ schemathesis run oas_with_link.yaml --base-url=http://localhost:8000 \
--hypothesis-database=none --stateful=links --checks=all
```

Как видите, на этот раз Schemathesis запускает более тысячи тестов за раз:

```

===== SUMMARY =====
Performed checks:
    not_a_server_error          1200 / 1200 passed          PASSED
    status_code_conformance     1200 / 1200 passed          PASSED
    content_type_conformance     1200 / 1200 passed          PASSED
    response_headers_conformance 1200 / 1200 passed          PASSED
    response_schema_conformance  1200 / 1200 passed          PASSED
===== 12 passed in 70.54s =====

```

В некоторых случаях Schemathesis может выдавать сообщение, что создание тестовых примеров занимает слишком много времени. Вы можете заблокировать его, используя флаг `--hypothesis-suppress-healthcheck=too_slow`. Выполнив весь набор тестов Schemathesis в отношении своего API, вы можете быть уверены, что он работает как надо и соответствует спецификации API. Имейте в виду, что есть возможность расширить тесты дополнительной полезной нагрузкой или сценариями. Поскольку `schemathesis` — это библиотека Python, очень легко добавить дополнительные кастомные тесты. Ознакомьтесь с документацией, чтобы на примерах посмотреть, как это сделать (<http://mng.bz/69Q5>).

На этом мы завершаем наше путешествие по миру тестирования REST API. Пришло время проверить работу GraphQL API, что и станет темой следующего раздела.

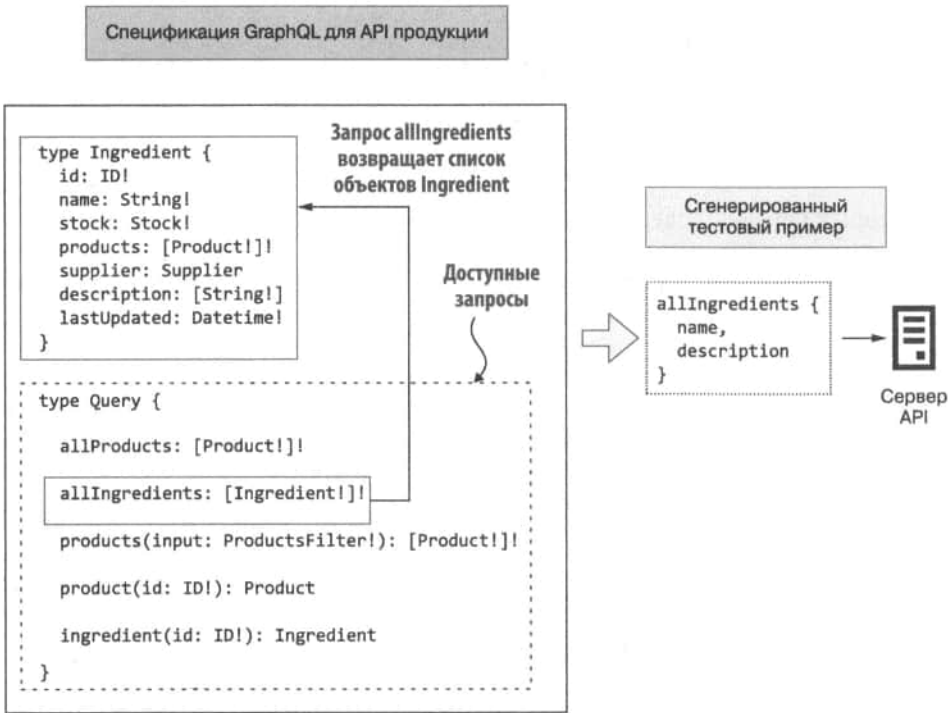
## 12.5. ТЕСТИРОВАНИЕ GRAPHQL API

В этом разделе мы протестируем GraphQL API, чтобы убедиться, что они работают должным образом, и только потом запускать их в производство. Для примера будем использовать API сервиса продукции, который создали в главе 10. Перед началом работы перейдите в раздел `ch12/products` и активируйте среду, выполнив команду `pipenv shell`.

В разделах 12.2 и 12.4 вы узнали о Dredd и Schemathesis, которые автоматически генерируют тесты для REST API на основе спецификации API. В GraphQL меньше поддержки автоматической генерации тестов. В частности, Dredd не работает с GraphQL API, в то время как Schemathesis может проверить их лишь частично. Однако сегодня это активная область разработки, поэтому в будущем можно ожидать увеличения поддержки автоматического тестирования GraphQL.

### 12.5.1. Тестирование GraphQL API с помощью Schemathesis

Как я объяснял в разделе 12.4, Schemathesis — это фреймворк для тестирования API, который использует тестирование на основе свойств. С помощью Schemathesis можно проверять как REST, так и GraphQL API. В обоих случаях, как вы можете видеть на рис. 12.10, Schemathesis просматривает спецификацию API, чтобы узнать о его эндпоинтах и схемах и решить, какие тесты запускать.



**Рис. 12.10.** Schemathesis парсит спецификацию GraphQL API в поисках доступных операций и генерирует документы запроса как с допустимыми, так и с недопустимыми параметрами и выборками для проверки ответа сервера

Чтобы сгенерировать тесты для GraphQL API, Schemathesis использует `hypothesis-graphql` (<http://mng.bz/o5Pj>), библиотеку, которая генерирует стратегии Hypothesis на основе схемы GraphQL. Прежде чем мы запустим тест, нам нужно запустить сервер GraphQL API. Это можно сделать в другом окне терминала или запустив процесс в фоновом режиме с помощью следующей команды:

```
$ uvicorn server:server &
```

Символ `&` означает, что процесс Uvicorn будет выполняться в фоновом режиме. Чтобы протестировать GraphQL API с помощью Schemathesis, нам просто нужно указать ему URL-адрес, по которому размещена спецификация API. В нашем случае GraphQL API размещен по адресу <http://127.0.0.1:8000/graphql>. Вооружившись этой информацией, теперь мы можем запустить тесты:

```
$ schemathesis run --hypothesis-deadline=None http://127.0.0.1:8000/graphql
```

Флаг `--hypothesis-deadline=None` указывает Schemathesis избегать синхронизации запросов. Это полезно, когда запросы выполняются медленно, что иногда случается с GraphQL API. В листинге 12.11 показан вывод набора тестов, за исключением первых нескольких строк, содержащих метаданные, которые относятся к конкретной платформе. Как видите, Schemathesis тестирует все запросы и мутации, предоставляемые API сервиса продукции, генерируя солидный набор тестов: 1100 тестовых примеров!

### Листинг 12.11. Вывод набора тестов Schemathesis для GraphQL API

```
[...]
Schema location: http://127.0.0.1:8000/graphql
Base URL: http://127.0.0.1:8000/graphql
Specification version: GraphQL
Workers: 1
Collected API operations: 11
Query.allProducts . [ 9%]
Query.allIngredients . [ 18%]
Query.products . [ 27%]
Query.product . [ 36%]
Query.ingredient . [ 45%]
Mutation.addSupplier . [ 54%]
Mutation.addIngredient . [ 63%]
Mutation.addProduct . [ 72%]
Mutation.updateProduct . [ 81%]
Mutation.deleteProduct . [ 90%]
Mutation.updateStock . [100%]

===== SUMMARY =====

Performed checks:
not_a_server_error. 1100/ 1100 passed PASSED

===== 11 passed in 36.82s =====
```

После запуска набора тестов Schemathesis с использованием API сервиса продукции мы можем быть уверены, что наши запросы и мутации работают должным образом. Вы можете дополнительно настроить тесты, чтобы убедиться, что приложение корректно работает при определенных условиях. Как добавлять пользовательские тесты, рассказывается в документации Schemathesis (<https://schemathesis.readthedocs.io/en/stable/>).

## 12.6. РАЗРАБОТКА СТРАТЕГИИ ТЕСТИРОВАНИЯ API

Вы многому научились в этой главе, в частности научились использовать такие фреймворки, как Dredd и Schemathesis, которые запускают автоматизированные наборы тестов для API на основе документации. Вы также узнали о тестировании на основе свойств и о том, как использовать Hypothesis для автоматической генерации тестовых примеров для проверки ваших REST и GraphQL API.

Как вы видели в разделе 12.2, Dredd предлагает простой набор тестов для API. Этот фреймворк позволяет проверить только «счастливый путь»: он гарантирует, что ваш API как принимает, так и выдает ожидаемую полезную нагрузку. Он не проверяет, что происходит, когда на сервер поступают некорректные данные.

Стратегия тестирования Dredd подходит на ранней стадии разработки API, когда нужно сосредоточиться на общей функциональности приложения, а не увязать в конкретных примерах интеграции с помощью API. Однако, прежде чем запустить свой API в производство, следует протестировать их с помощью Schemathesis. Этот фреймворк предлагает более полный набор тестов, которые гарантируют, что ваш API работает именно так, как ожидалось.

Я рекомендую вам запускать Dredd и Schemathesis локально во время разработки, а также на сервере непрерывной интеграции (CI) перед выпуском кода. С примером использования Dredd и Schemathesis на сервере CI вы можете ознакомиться в моем выступлении *API Development Workflows for Successful Integrations* («Процессы разработки API для успешной интеграции») на конференции Manning's API Conference 3 августа 2021 года ([https://youtu.be/SUKqmEX\\_uwg](https://youtu.be/SUKqmEX_uwg)).

## РЕЗЮМЕ

- Dredd и Schemathesis — это инструменты тестирования API, которые автоматически генерируют тесты для API. Благодаря им вам не придется писать тесты вручную и вы сможете сосредоточиться на создании API и сервисов.
- Dredd — это фреймворк для тестирования REST API. Он предоставляет базовый набор тестов для вашего API, не охватывая крайние случаи, поэтому удобен на ранних стадиях цикла разработки API.

- Можно настроить поведение Dredd, добавив в тесты хуки. Хотя Dredd — это пакет npm, вы можете написать хуки на Python. Хуки Dredd подойдут, если нужно сохранить информацию из одного теста для повторного использования в другом, а также для создания или удаления ресурсов до и после каждого теста.
- Schemathesis — это более общий фреймворк тестирования API, который предоставляет обширный набор тестов для ваших API. Прежде чем запускать API в производство, вам нужно протестировать их с помощью Schemathesis. Можете использовать Schemathesis для тестирования как REST, так и GraphQL API.
- Чтобы проверить, что ваши эндпоинты POST правильно создают ресурсы, вы можете дополнить спецификацию OpenAPI ссылками и настроить Schemathesis на их использование в наборе тестов. Ссылки — это свойства, которые описывают взаимосвязь между различными операциями в спецификации OpenAPI.
- Тестирование на основе свойств — подход, при котором вы позволяете фреймворку генерировать случайные тестовые наборы и проверяете поведение своего кода, делая утверждения о результатах. Благодаря такому подходу можно сэкономить время, как как нет нужды вручную писать тестовые примеры. В Python есть отличная библиотека `hypothesis`, позволяющая запускать тесты на основе свойств.

# 13

## Контейнеризация API микросервисов

---

### В этой главе

- ✓ Как упаковать приложение.
- ✓ Как запускать контейнеры Docker.
- ✓ Как запускать приложение с помощью Docker Compose.
- ✓ Публикация образа Docker в AWS Elastic Container Registry.

Docker — это контейнеризатор приложений, программное обеспечение для автоматизации развертывания приложений и управления ими в средах с поддержкой контейнеризации. Он позволяет запускать приложения в любом месте, просто используя среду выполнения Docker, и избавляет от необходимости настраивать среду для запуска кода. Так развертывания становятся более предсказуемыми, поскольку Docker создает воспроизводимые артефакты (образы контейнеров), которые можно запускать как локально, так и в облаке.

В этой главе вы научитесь контейнеризовать приложение на Python. Контейнеризация (докеризация) — это процесс упаковки приложения в виде образа Docker. Вы можете думать об образе Docker как о сборке или артефакте, который готов к развертыванию и выполнению. Чтобы запустить образ, Docker создает запущенные экземпляры образа, известные как *контейнеры*. Для развертывания образов Docker обычно используют оркестратор контейнеров, такой как Kubernetes, который управляет жизненным циклом контейнера. В следующей главе вы узнаете, как развертывать Docker-образы с помощью Kubernetes. Мы разберемся, как

контейнеризовать приложение с помощью сервиса заказов платформы CoffeeMesh. Вы также научитесь публиковать свои Docker-образы в реестре контейнеров, загружая образы в Elastic Container Registry AWS (ECR).

Все примеры кода доступны в папке `ch13` в репозитории GitHub для этой книги. Начнем с настройки среды для работы.

## 13.1. НАСТРОЙКА СРЕДЫ ДЛЯ ЭТОЙ ГЛАВЫ

В этом разделе мы настроим среду так, чтобы вы могли следовать примерам главы. Мы продолжим работу над сервисом заказов в том состоянии, в каком оставили его в главе 11, где добавили возможность аутентификации и авторизации. Сначала скопируйте код из главы 11 в новую папку `ch13`:

```
$ cp -r ch11 ch13
```

Перейдите в нее, установите зависимости и активируйте виртуальную среду, выполнив следующие команды:

```
$ cd ch13 && pipenv install --dev && pipenv shell
```

Когда мы развертываем приложение, мы используем движок PostgreSQL, который является одним из самых популярных движков SQL для запуска приложений в рабочей среде. Для связи с базой данных используем `psycopg2` — один из самых популярных драйверов PostgreSQL для Python:

```
$ pipenv install psycopg2
```

### УСТАНОВКА PSYCOPG2

Если вы столкнетесь с проблемами при установке и компиляции `psycopg2`, попробуйте установить скомпилированный пакет, запустив `pipenv install psycopg2-binary`, или возьмите файлы `ch13/Pipfile` и `ch13/Pipfile.lock` в репозитории к этой книге на GitHub и запустите `pipenv install --dev`. Еще два мощных драйвера PostgreSQL — это `asyncpg` (<https://github.com/MagicStack/asyncpg>) и `psycopg3` (<https://github.com/psycopg/psycopg>), оба поддерживают асинхронные операции. Очень рекомендую вам ознакомиться с ними.

Чтобы создавать и запускать контейнеры Docker, вам понадобится среда выполнения Docker на компьютере. Инструкции по установке зависят от конкретной платформы, поэтому ознакомьтесь с официальной документацией, чтобы узнать, как установить Docker в своей системе (<https://docs.docker.com/get-docker/>).

Поскольку мы собираемся опубликовать наши образы Docker в ECR AWS, нам необходимо установить AWS CLI:

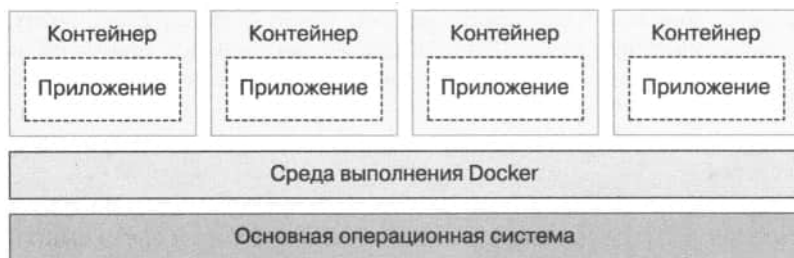
```
$ pipenv install --dev awscli
```

Далее перейдите на сайт <https://aws.amazon.com/>. Создайте учетную запись AWS и получите ключ доступа, чтобы иметь возможность программного доступа к сервисам AWS. Пользователь, через которого вы создаете учетную запись AWS, будет root-пользователем. В целях безопасности рекомендуется не использовать пользователя root для генерации ключа доступа. Вместо этого создайте пользователя IAM и сгенерируйте ключ доступа для него. IAM — это сервис управления идентификацией и контролем доступа AWS, который позволяет создавать пользователей, роли и детализированные политики для предоставления доступа к другим сервисам в вашей учетной записи. Ознакомьтесь с документацией AWS, чтобы узнать, как создать пользователя IAM (<http://mng.bz/neP8>) и как сгенерировать ключи доступа и настроить AWS CLI (<http://mng.bz/vXxq>).

Теперь, когда среда готова, пришло время упаковать наши приложения!

## 13.2. КОНТЕЙНЕРИЗАЦИЯ МИКРОСЕРВИСА

Что такое *контейнеризация* приложения? Это процесс упаковки приложения в виде образа Docker. Представьте себе образ Docker как сборку или артефакт, который может быть развернут и выполнен во время работы Docker. Все системные зависимости уже установлены в образе, и для его запуска нужна только среда выполнения Docker. Запуская образ, Docker создает контейнер, который является запущенным экземпляром образа. Как вы можете видеть на рис. 13.1, работать с Docker очень легко, поскольку он позволяет запускать приложения в изолированных процессах. Существуют различные варианты установки среды выполнения Docker в зависимости от вашей платформы, поэтому ознакомьтесь с официальной документацией, чтобы определить, какой вариант вам лучше всего подходит (<https://docs.docker.com/get-docker/>).



**Рис. 13.1.** Контейнеры Docker запускаются в изолированных процессах поверх основной операционной системы

В этом разделе мы создадим оптимизированный Docker-образ сервиса заказов. Попутно вы узнаете, как написать Dockerfile, который представляет собой документ со всеми инструкциями на тему создания образа Docker. Вы также узнаете, как



запускать контейнеры Docker и сопоставлять порты из контейнера с операционной системой хоста, чтобы вы могли взаимодействовать с приложением, запущенным внутри контейнера. Наконец, вы узнаете, как управлять контейнерами с помощью Docker CLI.

## ОСНОВЫ DOCKER

Если вы хотите узнать больше о том, как работает Docker и как он взаимодействует с операционной системой хоста, ознакомьтесь с главой Docker Fundamentals («Основы Docker») из книги Прабата Сиривардены и Нувана Диаса *Microservices Security in Action* (Manning, 2020, <http://mng.bz/49Ag>).

Прежде чем мы создадим образ, нам нужно внести два небольших изменения в код приложения, чтобы подготовить его к развертыванию. До сих пор сервис заказов использовал жестко запрограммированный URL-адрес базы данных, но для работы сервиса в различных средах нам необходимо настроить этот параметр. В листинге 13.1 показан код с изменениями в файле `orders/repository/unit_of_work.py` для извлечения URL-адреса базы данных из среды. Недавно добавленный код выделен жирным шрифтом. Мы используем оператор контроля ошибок (`assert`) для немедленного выхода из приложения, если URL-адрес базы данных не указан.

### Листинг 13.1. Извлечение URL-адреса базы данных из среды

```
# file: orders/repository/unit_of_work.py
```

```
import os
```

```
from sqlalchemy import create_engine
from sqlalchemy.orm import sessionmaker
```

```
DB_URL = os.getenv('DB_URL')
```

← Извлекаем URL-адрес базы данных  
из переменной среды DB\_URL

```
assert DB_URL is not None, 'DB_URL environment variable needed.'
```

← Выходим из приложения,  
если DB\_URL не задан

```
class UnitOfWork:
```

```
    def __init__(self):
```

```
        self.session_maker = sessionmaker(bind=create_engine(DB_URL))
```

```
    def __enter__(self):
```

```
        self.session = self.session_maker()
```

```
        return self
```

```
    ...
```

← Используем значение из DB\_URL  
для подключения к базе данных

Нам также нужно обновить файлы Alembic, чтобы извлечь из среды URL-адрес базы данных. В листинге 13.2 показано, какие изменения необходимо внести в файл `migrations/env.py`, новый код выделен жирным шрифтом. Я опустил несущественные части кода, чтобы было легче наблюдать за изменениями.

**Листинг 13.2.** Извлечение URL-адреса базы данных из среды для Alembic

```
# file: migrations/env.py
```

```
import os
from logging.config import fileConfig

from sqlalchemy import create_engine
from sqlalchemy import pool

from alembic import context
...

def run_migrations_online():
    """...
    """
    url = os.getenv('DB_URL')

    assert url is not None, 'DB_URL environment variable needed.'

    connectable = create_engine(url)

    context.configure(
        url=url,
        target_metadata=target_metadata,
        literal_binds=True,
        dialect_opts={"paramstyle": "named"},
    )

    ...
```

Извлекаем URL-адрес базы данных из переменной среды DB\_URL

Выходим из приложения, если DB\_URL не задан

Теперь, когда наш код готов, пришло время его упаковать. Чтобы создать образ Docker, нужно написать Dockerfile. В листинге 13.3 приводится его содержимое. Мы используем неполную версию официального образа Python 3.9 Docker в качестве базового. Неполные образы содержат только те зависимости, которые требуются для запуска приложений, то есть мы получим более легкий образ. Чтобы использовать базовый образ, мы используем директиву Docker FROM. Затем создаем папку для кода приложения под названием /orders/orders. Для запуска команд bash, в частности mkdir в нашем случае, мы используем директиву RUN. Устанавливаем /orders/orders в качестве рабочего каталога, используя директиву WORKDIR. Рабочий каталог — это каталог, из которого запускается приложение.

Далее устанавливаем pipenv, копируем наши файлы Pipenv и определяем зависимости. Для копирования файлов из файловой системы в образ Docker мы используем директиву COPY. Поскольку мы работаем в Docker, нам не нужна виртуальная среда, поэтому устанавливаем зависимости, используя флаг pipenv --system. Добавляем также флаг pipenv -deploy — так мы проверим актуальность файлов

Pipenv. Наконец, копируем исходный код и указываем команду, которую следует выполнить, чтобы запустить сервис заказов. Она будет указана с помощью директивы `CMD`. Используем также директиву `EXPOSE`, чтобы убедиться, что запущенный контейнер прослушивает порт 8000 — именно на нем работает наш API. Если мы не откроем порт, то не сможем взаимодействовать с API.

Порядок наших инструкций в файле `Dockerfile` имеет значение, потому что Docker кэширует каждый шаг сборки. Docker выполнит шаг повторно только в том случае, если на предыдущем шаге что-то изменилось, например, если мы установили новую зависимость или изменился один из наших файлов. Поскольку код нашего приложения, скорее всего, будет меняться чаще, чем зависимости, мы копируем код в конце сборки. Таким образом, Docker установит зависимости только один раз и выполнит кэширование до внесения изменений.

### Листинг 13.3. Dockerfile для сервиса заказов

# file: Dockerfile

```
FROM python:3.9-slim  ← Базовый образ

RUN mkdir -p/orders/orders  ← Базовая структура папок
                               для нашего приложения

WORKDIR/orders  ← Рабочий каталог, из которого мы запустим код

RUN pip install -U pip && pip install pipenv

COPY Pipfile Pipfile.lock/orders/  ← Копируем наши файлы pipenv

RUN pipenv install --system --deploy  ← Устанавливаем зависимости

COPY orders/orders_service/orders/orders/orders_service/  ←
COPY orders/repository/orders/orders/repository/             Копируем остальные
COPY orders/web/orders/orders/web/                           файлы приложения
COPY oas.yaml/orders/
COPY public_key.pem/orders/public_key.pem
COPY private.pem/orders/private.pem

EXPOSE 8000  ← Предоставляем порт
              приложения хост-машине

CMD ["uvicorn", "orders.web.app:app", "--host", "0.0.0.0"]  ← Команда запуска
                                                             сервера API
```

Чтобы создать образ Docker из листинга 13.3, вам необходимо запустить из каталога `ch13` следующую команду:

```
$ docker build -t orders:1.0 .
```

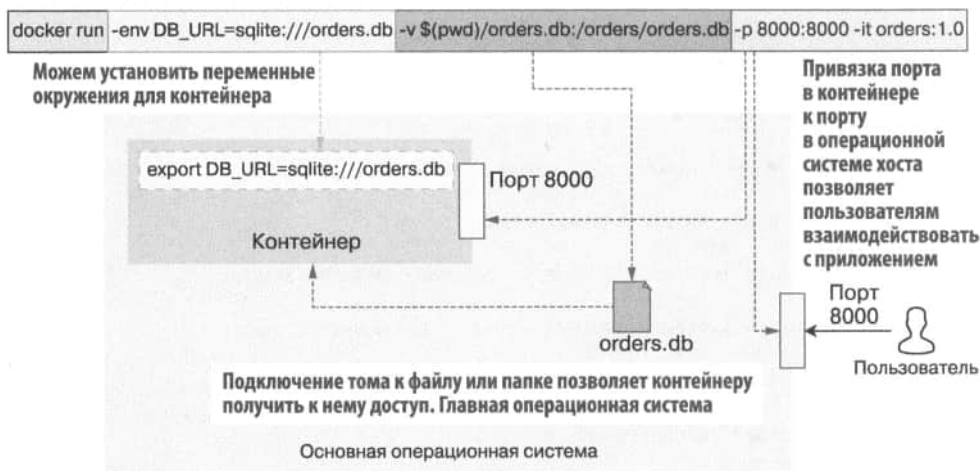
Флаг `-t` обозначает *тег*. Тег Docker состоит из двух частей: названия образа (слева от двоеточия) и названия тега (справа от двоеточия). Имя тега — это обычно версия образа. В данном случае мы присваиваем порядком образов имена и помечаем их

значением 1.0. Убедитесь, что вы не пропустили точку в конце инструкции `build`: она означает текущий каталог в пути к исходному коду образа (*контекст* на языке Docker).

Как только образ будет создан, вы можете выполнить его с помощью такой команды:

```
$ docker run --env DB_URL=sqlite:///orders.db \
-v $(pwd)/orders.db:/orders/orders.db -p 8000:8000 -it orders:1.0
```

Как видно на рис. 13.2, флаг `--env` позволяет устанавливать переменные окружения в контейнере, и мы используем его для установки URL-адреса базы данных. Чтобы сделать приложение доступным для хост-машины, добавляем флаг `-p`, который позволяет нам привязать порт, на котором приложение запущено внутри контейнера, к порту на хост-машине. Мы также указываем флаг `-v` для подключения тома к файлу базы данных SQLite. Тома Docker позволяют контейнерам получать доступ к файлам из файловой системы хост-компьютера.



**Рис. 13.2.** Когда мы запускаем контейнер, можем включать различные конфигурации для установки переменных окружения внутри контейнера или для того, чтобы разрешить ему доступ к файлам основной операционной системы

Теперь вы можете получить доступ к приложению по URL-адресу `http://127.0.0.1:8000/docs/orders`. Предыдущая команда запускает контейнер, подключенный к текущему сеансу терминала, что позволяет вам видеть, как меняются журналы по мере взаимодействия с приложением. Вы можете остановить контейнер так же, как и любой другой процесс, нажав `Ctrl+C`.

Вы также можете запускать контейнеры в автономном режиме, то есть этот процесс не будет связан с вашим сеансом работы в терминале, и, когда вы закроете терминал, процесс продолжит выполняться. Это удобно, если нужно просто

запустить контейнер и не нужно просматривать журналы. Обычно в автономном режиме запускают контейнеризованные базы данных. Чтобы запустить контейнер в автономном режиме, воспользуйтесь флагом `-d`:

```
$ docker run -d --env DB_URL=sqlite:///orders.db \  
-v $(pwd)/orders.db:/orders/orders.db -p 8000:8000 orders:1.0
```

В этом случае вам нужно будет остановить контейнер с помощью команды `docker stop`. Сначала следует выяснить ID запущенного контейнера с помощью такой команды:

```
$ docker ps
```

Она выведет список всех запущенных в данный момент контейнеров на вашем компьютере. Вывод выглядит следующим образом (приводится в сокращенном виде):

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS...
83e6189a02ee	orders:1.0	"uvicorn..."	7 seconds ago	Up 6 seconds

Скопируйте ID контейнера (в данном случае `83e6189a02ee`) и используйте его, чтобы остановить процесс:

```
$ docker stop 83e6189a02ee
```

Это все, что нужно знать для создания и запуска контейнеров Docker. Он предоставляет гораздо больше возможностей, чем описано в этом разделе, и, если вам интересно узнать подробнее об этой технологии, прочтите книгу *Docker in Practice*<sup>1</sup> и *Docker in Action* Джеффа Николоффа и Стивена Куэнзли (Manning, 2019).

## 13.3. ЗАПУСК ПРИЛОЖЕНИЙ С ПОМОЩЬЮ DOCKER COMPOSE

В предыдущем разделе мы запустили контейнер сервиса заказов, связав его с нашей локальной базой данных SQLite. Так можно делать при быстром тестировании, но на самом деле мы не сможем узнать, будет ли приложение работать с базой данных PostgreSQL. Распространенная стратегия подключения контейнерных приложений к базе данных — использование Docker Compose, который позволяет запускать несколько контейнеров в общей сети, чтобы они могли взаимодействовать друг с другом. В этом разделе вы узнаете, как запустить сервис заказов с базой данных PostgreSQL с помощью `docker-compose`.

Чтобы использовать Docker Compose, нужно установить его. Это пакет на Python, поэтому устанавливаем его с помощью команды `pip`:

```
$ pip install docker-compose
```

---

<sup>1</sup> Милл И. Сейерс Э. Х. Docker на практике.

Далее напишем файл Docker Compose — объявим ресурсы, необходимые для запуска нашего приложения. В листинге 13.4 показан файл `docker-compose` для сервиса заказов. Мы используем последний формат спецификации Docker Compose версии 3.9 и объявляем два сервиса: `database` и `api`. Первый запускает официальный образ Docker от PostgreSQL, в то время как `api` запускает сервис заказов. Добавляем ключевое слово `build`, чтобы указать на контекст Docker-образа, и присваиваем ему значение `.`. С помощью этой точки мы даем Docker Compose команду выполнить поиск файла `Dockerfile` и создать образ относительно текущего каталога. Под ключевым словом `environment` мы настраиваем переменные среды, необходимые для запуска приложений. Предоставляем порт 5432 для `database`, чтобы подключиться к базе данных с нашего хост-компьютера, а также порт 8000 для `api`, чтобы получить доступ к API. Наконец, прописываем том под названием `database-data`, который `docker-compose` будет использовать для сохранения наших данных. Это означает, что, перезапустив `docker-compose`, вы не потеряете свои данные.

#### Листинг 13.4. Файл `docker-compose` для сервиса заказов

```
# file: docker-compose.yaml

version: "3.9"

services:
  database:
    image: postgres:14.2
    ports:
      - 5432:5432
    environment:
      POSTGRES_PASSWORD: postgres
      POSTGRES_USER: postgres
      POSTGRES_DB: postgres
    volumes:
      - database-data:/var/lib/postgresql/data

  api:
    build: .
    ports:
      - 8000:8000
    depends_on:
      - database
    environment:
      DB_URL: postgresql://postgres:postgres@database:5432/postgres
volumes:
  database-data:
```

Версия формата `docker-compose` для этого файла

Объявляем наши сервисы

Сервис базы данных

Образ Docker сервиса базы данных

В сервисе базы данных предоставляем порты базы данных хост-компьютеру

Конфигурация среды базы данных

Монтируем папку нашей базы данных в локальном томе

Сервис API

Контекст сборки API

API зависит от базы данных

Конфигурация среды API

Объем базы данных

Предоставляем порт API хост-машине

Выполните следующую команду, чтобы запустить файл Docker Compose:

```
$ docker-compose up --build
```

Флаг `--build` указывает Docker Compose перестроить ваши образы, если файлы изменились. Как только веб-API будет запущен, вы сможете получить к нему доступ

на `http://localhost:8000/docs/orders`. Если вы попытаетесь использовать любую из эндпоинтов, то не обнаружите таблицы. Это потому, что мы не запускали миграцию с нашей новой базой данных PostgreSQL! Чтобы запустить миграцию, откройте новое окно терминала, перейдите в папку `ch13`, активируйте среду `pipenv` и выполните следующую команду:

```
$ PYTHONPATH=`pwd` \
DB_URL=postgresql://postgres:postgres@localhost:5432/postgres alembic \
upgrade heads
```

Как только миграции будут применены, вы можете снова обратиться к эндпоинтам API, и теперь все должно сработать. Чтобы остановить `docker-compose`, запустите следующую команду из другого окна терминала и внутри папки `ch13`:

```
$ docker-compose down
```

Это все, что нужно для запуска Docker Compose. Итак, вы только что научились пользоваться одним из самых мощных инструментов автоматизации. Docker Compose часто используется для запуска интеграционных тестов и позволяет без труда выполнить серверную часть для разработчиков, работающих над клиентскими приложениями, такими как SPA.

Теперь, когда наш стек Docker готов и образы протестированы, разберемся, как помещать образы в реестр контейнеров.

## 13.4. ПУБЛИКАЦИЯ DOCKER-ОБРАЗОВ В РЕЕСТРЕ КОНТЕЙНЕРОВ

Чтобы развернуть Docker-образы, нам нужно сначала опубликовать их в реестре контейнеров Docker. Реестр контейнеров — это хранилище образов Docker. В следующей главе мы развернем наши приложения в сервисе AWS Elastic Kubernetes, поэтому опубликуем образы в ECR AWS. Хранение образов Docker в AWS упростит их развертывание в EKS.

Сначала создадим ECR-репозиторий для образов с помощью следующей команды:

```
$ aws ecr create-repository --repository-name coffeemesh-orders
{
  "repository": {
    "repositoryArn":
➤ "arn:aws:ecr:<aws_region>:<aws_account_id>:repository/coffeemesh-orders",
    "registryId": "876701361933",
    "repositoryName": "coffeemesh-orders",
    "repositoryUri":
➤ "<aws_account_id>.dkr.ecr.<aws_region>.amazonaws.com/coffeemesh-orders",
    "createdAt": "2021-11-16T10:08:42+00:00",
```

```

    "imageTagMutability": "MUTABLE",
    "imageScanningConfiguration": {
        "scanOnPush": false
    },
    "encryptionConfiguration": {
        "encryptionType": "AES256"
    }
}
}
}

```

Здесь мы создаем репозиторий ECR с именем `coffeemesh-orders`. Результатом выполнения команды будет полезная нагрузка, описывающая репозиторий, который мы только что создали. Когда вы запустите команду, `<aws_account_id>` в выходной полезной нагрузке будет содержать ваш ID учетной записи AWS, а `<aws_region>` — ваш регион AWS по умолчанию. Чтобы опубликовать Docker-образ в ECR, нужно пометить ее именем репозитория ECR:

```

$ docker tag orders:1.0 \
<aws_account_id>.dkr.ecr.<aws_region>.amazonaws.com/coffeemesh-orders:1.0

```

Чтобы публиковать образы в ECR, нам нужно получить учетные данные для входа в систему с помощью следующей команды:

```

$ aws ecr get-login-password --region <aws_region> | docker login \
--username AWS --password-stdin \
<aws_account_id>.dkr.ecr.<region>.amazonaws.com

```

Убедитесь, что заменили `<aws_region>` в этой команде на регион AWS, в котором вы создали репозиторий Docker, например `eu-west-1` для Европы (Ирландия) или `us-east-2` для Востока США (штат Огайо). Также замените `<aws_account_id>` своим ID учетной записи AWS. Ознакомьтесь с документацией AWS, чтобы узнать, как найти свой ID учетной записи AWS (<http://mng.bz/Qnye>).

## РЕГИОНЫ AWS

Когда вы развертываете сервисы в AWS, вы делаете это в определенных регионах. У каждого региона есть ID, например `eu-west-1` для Ирландии и `us-east-2` для Огайо. Обновленный список регионов, доступных в AWS, смотрите по адресу <http://mng.bz/XaPM>.

Команда `aws ecr get-login-password` выдает инструкцию, которую Docker использует для входа в ECR. Теперь вы готовы опубликовать образ! Выполните следующую команду, чтобы переместить образ в ECR:

```

$ docker push \
<aws_account_id>.dkr.ecr.<aws_region>.amazonaws.com/coffeemesh-orders:1.0

```

Вуаля! Наш Docker-образ теперь находится в ECR. В следующей главе вы узнаете, как развернуть его в кластере Kubernetes в AWS.



## РЕЗЮМЕ

- Docker — это контейнеризатор приложений, программное обеспечение для автоматизации развертывания приложений и управления ими в средах с поддержкой контейнеризации. Сборка Docker называется образом, который выполняется в процессах, называемых контейнерами Docker.
- Docker Compose — это платформа для оркестрации контейнеров, которая позволяет запускать несколько контейнеров одновременно, таких как базы данных и API. Использование Docker Compose — это простой и эффективный способ запустить весь ваш сервер без необходимости устанавливать и настраивать дополнительные зависимости.
- Чтобы развернуть образы Docker, мы публикуем их в реестре контейнеров. ECR AWS — надежный и безопасный реестр контейнеров, который упрощает развертывание контейнеров в сервисах AWS.

# Развертывание API микросервисов с помощью Kubernetes

---

## В этой главе

- ✓ Создание кластера с помощью сервиса Elastic Kubernetes Service от AWS (EKS).
- ✓ Предоставление доступа к сервисам с помощью контроллера балансировки нагрузки AWS (AWS Load Balancer Controller).
- ✓ Развертывание сервисов в кластере Kubernetes.
- ✓ Безопасное управление секретами в Kubernetes.
- ✓ Развертывание бессерверной базы данных Aurora.

Kubernetes — это платформа для оркестрации контейнеров с открытым исходным кодом, и сегодня это популярный инструмент для развертывания приложений и управления ими на разных платформах. Вы можете самостоятельно развернуть Kubernetes на своих серверах или воспользоваться сервисом, управляемым Kubernetes. В любом случае вы получите согласованный интерфейс для своих сервисов, то есть переход от одного облачного провайдера к другому становится менее проблематичным. Вы также можете развернуть кластер Kubernetes на своем компьютере и запускать тесты локально во многом так же, как делали бы это в облаке.

## ЗАПУСК KUBERNETES ЛОКАЛЬНО С ПОМОЩЬЮ MINIKUBE

Вы можете запустить кластер Kubernetes локально с помощью minikube. Хотя мы не будем рассматривать здесь minikube, это отличный инструмент для работы с Kubernetes. Можете ознакомиться с официальной документацией по minikube на сайте <https://minikube.sigs.k8s.io/docs/start/>.

Самостоятельное развертывание Kubernetes — хорошее упражнение для знакомства с технологиями, но на практике большинство компаний используют управляемый сервис. В этой главе мы будем использовать управляемый сервис Kubernetes для развертывания кластера. Такие управляемые сервисы предлагают многие поставщики, но самые популярные — Google Cloud с движком Google Kubernetes (GKE), Azure Kubernetes Service (AKS) и Elastic Kubernetes Service (EKS) от AWS. Все три сервиса очень надежны и предлагают схожие функции<sup>1</sup>. В этой главе мы будем использовать EKS, который сегодня является самым популярным управляемым сервисом Kubernetes<sup>2</sup>.

Чтобы разобраться, как развертывать приложения в кластере Kubernetes, воспользуемся сервисом заказов. Кроме того, мы создадим бессерверную базу данных Aurora и посмотрим, как безопасно передавать в сервис учетные данные для подключения к базе данных, используя секреты Kubernetes.

Для изучения материала этой главы не требуется предварительных знаний об AWS или Kubernetes. Я постарался подробно объяснить каждую концепцию Kubernetes и AWS, чтобы вы могли выполнить примеры, даже если у вас ранее не было опыта работы ни с одной из технологий. На эти темы написаны целые книги, поэтому здесь мы лишь кратко ознакомимся с ними, но я привожу ссылки на другие ресурсы, с помощью которых вы сможете подробнее во всем разобраться.

Прежде чем продолжить, хочу предупредить, что EKS и другие сервисы AWS, упомянутые в этой главе, платные. Таким образом, это единственная глава в книге, выполнение примеров которой обойдется вам в некоторую сумму, если вы, конечно, решите их выполнить. Кластер Kubernetes в AWS EKS обойдется вам примерно в 0,10 доллара в час, или 2,40 доллара в день и примерно 72 доллара в месяц. В то же время не торопитесь платить: я рекомендую вам сначала прочитать главу, чтобы получить представление о том, что мы делаем, а затем уже опробовать примеры EKS.

Если вы впервые работаете с EKS и Kubernetes, выполнение примеров может занять у вас один или два дня, поэтому постарайтесь запланировать это время. В разделе 14.9 описано, как удалить кластер EKS и все другие ресурсы, созданные в этой главе, чтобы избежать дополнительных затрат.

Ну и наконец давайте начнем! И начнем с настройки среды.

---

<sup>1</sup> Краткое сравнение GKE, AKS и EKS см. в статье: *Postasnick A. AWS vs EKS vs GKE: Managed Kubernetes Services Compared* («AWS, EKS или GKE: сравнение управляемых сервисов Kubernetes»), 9 июня 2021 года. <https://acloudguru.com/blog/engineering/aks-vs-eks-vs-gke-managed-kubernetes-services-compared>.

<sup>2</sup> Flexera, 2022 State of the Cloud Report («Отчет о состоянии облака за 2022 год») (с. 52–53). <https://info.flexera.com/CM-REPORT-State-of-the-Cloud>.

## 14.1. НАСТРОЙКА СРЕДЫ ДЛЯ ЭТОЙ ГЛАВЫ

В этом разделе мы настроим среду так, чтобы вы могли выполнить примеры. Даже если вы не планируете это делать, рекомендую вам хотя бы бегло ознакомиться с разделом, чтобы узнать об инструментах, которые можно использовать. В этой главе описывается много инструментов, и здесь мы устанавливаем наиболее важные зависимости, а в следующих разделах вы найдете дополнительные инструкции по использованию других инструментов.

Сначала скопируйте код из главы 13 в папку `ch14`, выполнив следующую команду:

```
$ cp -r ch13 ch14
```

Перейдите в `ch14`, установите зависимости и активируйте виртуальную среду:

```
$ cd ch14 && pipenv install --dev && pipenv shell
```

Поскольку мы будем развертываться на AWS, необходимо иметь программный доступ к сервисам AWS. В главе 13 мы установили и настроили AWS CLI. Если вы еще этого не сделали, вернитесь к разделу 13.1 и следуйте инструкциям по установке и настройке AWS CLI.

Вы узнаете, как развертывать сервисы в Kubernetes, поэтому вам также необходимо установить Kubernetes CLI, известный как `kubectl`. Существуют разные способы установки `kubectl` в зависимости от используемой платформы, поэтому обратитесь к официальной документации, чтобы узнать, какой вариант подходит вам лучше всего (<https://kubernetes.io/docs/tasks/tools/>).

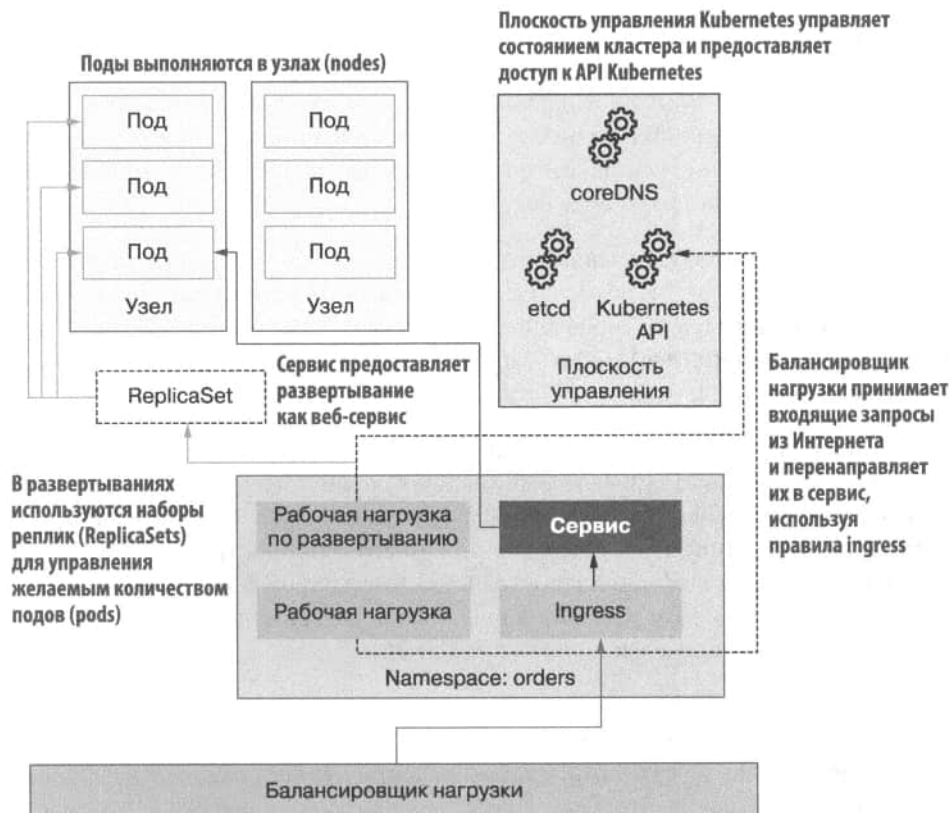
Наконец, в этой главе мы будем активно использовать `jq` — инструмент CLI, который помогает парсить документы в формате JSON и делать запросы к ним. Не обязательно использовать именно `jq`, но он действительно упрощает работу, и, если вы раньше не пользовались этим инструментом, я настоятельно рекомендую вам ознакомиться с ним. Мы будем применять `jq` в основном для фильтрации полезных данных JSON и извлечения из них определенных свойств. Как и в случае с Kubernetes, существуют различные варианты установки этого инструмента в зависимости от вашей платформы, поэтому обратитесь к официальной документации, чтобы узнать, какой вариант вам лучше всего выбрать (<https://stedolan.github.io/jq/download/>).

Теперь, когда наша среда готова, пришло время развертывания! Прежде чем мы создадим кластер, давайте познакомимся с Kubernetes и рассмотрим основные концепции. Если у вас уже есть опыт работы с Kubernetes, можете пропустить раздел 14.2.

## 14.2. КАК РАБОТАЕТ KUBERNETES: ВЕРСИЯ CLIFFSNOTES

Итак, что же такое Kubernetes? Это инструмент для оркестрации контейнеров с открытым исходным кодом. *Оркестрация контейнеров* — это процесс запуска контейнеризованных приложений. В дополнение к оркестрации контейнеров Kubernetes также позволяет автоматизировать развертывания и обеспечивает плавное развертывание и откат, масштабирование приложений и многое другое.

На рис. 14.1 показаны основные компоненты кластера Kubernetes. Ядром является плоскость управления (control plane) — мастер-узел, который среди прочего запускает API Kubernetes для нашего кластера, контролирует его состояние и управляет доступными ресурсами. Здесь можно также установить плагины и определенные DNS-серверы, такие как CoreDNS.



**Рис. 14.1.** Высокоуровневая архитектура кластера Kubernetes, показывающая, как связаны между собой все компоненты кластера

## ОПРЕДЕЛЕНИЕ

Плоскость управления Kubernetes — это мастер-узел, который запускает API Kubernetes и контролирует состояние кластера, а также управляет доступными ресурсами, планированием и многими другими задачами. Дополнительные сведения о плоскости управления см. в главах 11 (<http://mng.bz/yayE>) и 12 (<http://mng.bz/M0dm>) книги *Core Kubernetes*<sup>1</sup> Джея Вьяса и Криса Лава.

Наименьшей вычислительной единицей в Kubernetes является *под* (*pod*) — обертка вокруг контейнеров, которая может включать один или несколько контейнеров. Чаще всего запускают один контейнер на под, и в этой главе мы развернем сервис заказов именно так.

Для развертывания модулей в кластере мы используем *рабочую нагрузку* (*workloads*). В Kubernetes есть четыре типа рабочей нагрузки: Deployment, StatefulSet, DaemonSet и Job/CronJob. Deployment — наиболее распространенный тип рабочей нагрузки Kubernetes, подходит для запуска распределенных приложений без сохранения состояния. StatefulSet используется для запуска распределенных приложений, состояние которых необходимо синхронизировать. DaemonSet подойдет для определения процессов, которые должны выполняться на всех или большинстве узлов кластера, таких как сборщики журналов. Job и CronJob помогают определять разовые процессы или приложения, которые необходимо запускать по расписанию, например раз в день или раз в неделю.

Для развертывания микросервиса мы используем либо Deployment, либо StatefulSet. Поскольку все наши сервисы не имеют состояния, в этой главе мы развернем сервис заказов как Deployment. Для управления количеством модулей в развертываниях используется концепция ReplicaSet — процесса, который поддерживает желаемое количество подов в кластере.

Рабочая нагрузка обычно ограничена *пространствами имен* (*namespaces*). В Kubernetes пространства имен представляют собой логические группировки ресурсов, которые позволяют изолировать наши развертывания и расширить область их применения. Например, мы можем создать пространство имен для каждого сервиса на платформе. Пространства имен упрощают управление развертываниями и позволяют избежать конфликтов имен: имена ресурсов должны быть уникальными в пределах пространства имен.

Для запуска приложений в качестве веб-сервисов Kubernetes предлагает концепцию *сервисов* — процессов, которые управляют интерфейсами наших модулей и обеспечивают связь между ними. Чтобы предоставить доступ к своим сервисам через Интернет, мы используем *балансировщик нагрузки*, который находится перед кластером Kubernetes и перенаправляет трафик к сервисам на основе правил входа.

<sup>1</sup> Вьяс Дж., Лав К. Kubernetes изнутри. — 2022.

Последней частью системы Kubernetes является *узел (node)*, представляющий собой фактические вычислительные ресурсы, на которых выполняются сервисы. Мы определяем узлы как вычислительные ресурсы, поскольку они могут быть чем угодно — начиная с физических серверов и заканчивая виртуальными машинами. Например, при запуске кластера Kubernetes в AWS узлы будут представлены машинами EC2.

Теперь, когда мы знаем основные компоненты Kubernetes, давайте создадим кластер.

## 14.3. СОЗДАНИЕ КЛАСТЕРА KUBERNETES С ПОМОЩЬЮ EKS

В этом разделе вы узнаете, как создать кластер Kubernetes с помощью AWS EKS. Мы запустим кластер Kubernetes с помощью `eksctl`, который является рекомендуемым инструментом для управления Kubernetes в AWS.

`eksctl` — это инструмент с открытым исходным кодом, созданный и поддерживаемый Weaveworks. Он работает на базе сервиса CloudFormation, позволяющего вносить изменения в кластеры Kubernetes и управлять ими. Это означает, что мы можем повторно использовать шаблоны CloudFormation для репликации одной и той же инфраструктуры в разных средах. Кроме того, все наши изменения в кластере будут видимыми через CloudFormation.

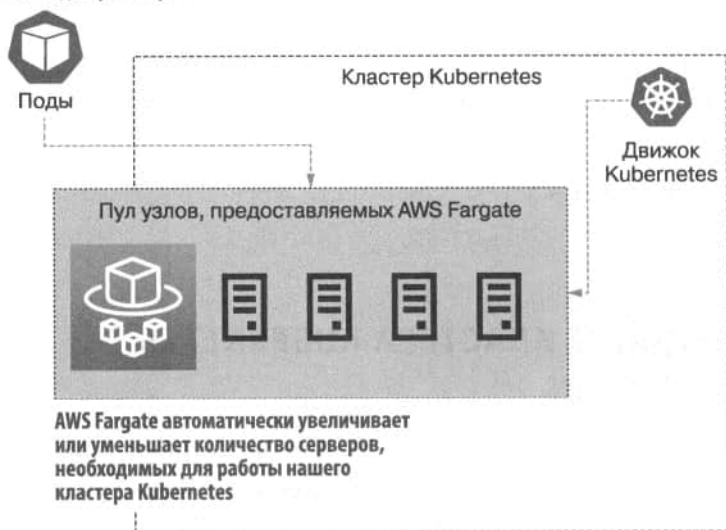
### ОПРЕДЕЛЕНИЕ

*CloudFormation* — это сервис «инфраструктура как код» от AWS. С помощью CloudFormation мы можем объявлять ресурсы в файлах YAML или JSON, называемых *шаблонами*. Когда мы отправляем шаблоны в CloudFormation, AWS создает *стек* — коллекцию ресурсов, определенных в шаблонах. Шаблоны CloudFormation не должны содержать конфиденциальной информации и могут быть сохранены в репозиториях кода, что делает изменения в инфраструктуре заметными и воспроизводимыми в различных средах.

Существуют различные способы установки `eksctl` в зависимости от используемой платформы, поэтому обратитесь к официальной документации, чтобы узнать, какой вариант подходит вам лучше всего (<https://github.com/weaveworks/eksctl>).

Для запуска контейнеров в кластере Kubernetes мы используем AWS Fargate (рис. 14.2). Это сервис AWS, который позволяет запускать контейнеры в облаке без необходимости предоставления серверов. С AWS Fargate вам не нужно беспокоиться о масштабировании серверов, поскольку об этом позаботится Fargate.

Наши модули автоматически  
развертываются на доступных узлах



**Рис. 14.2.** AWS Fargate автоматически подготавливает серверы, необходимые для работы кластера Kubernetes

Чтобы создать кластер Kubernetes с помощью `eksctl`, выполните следующую команду:

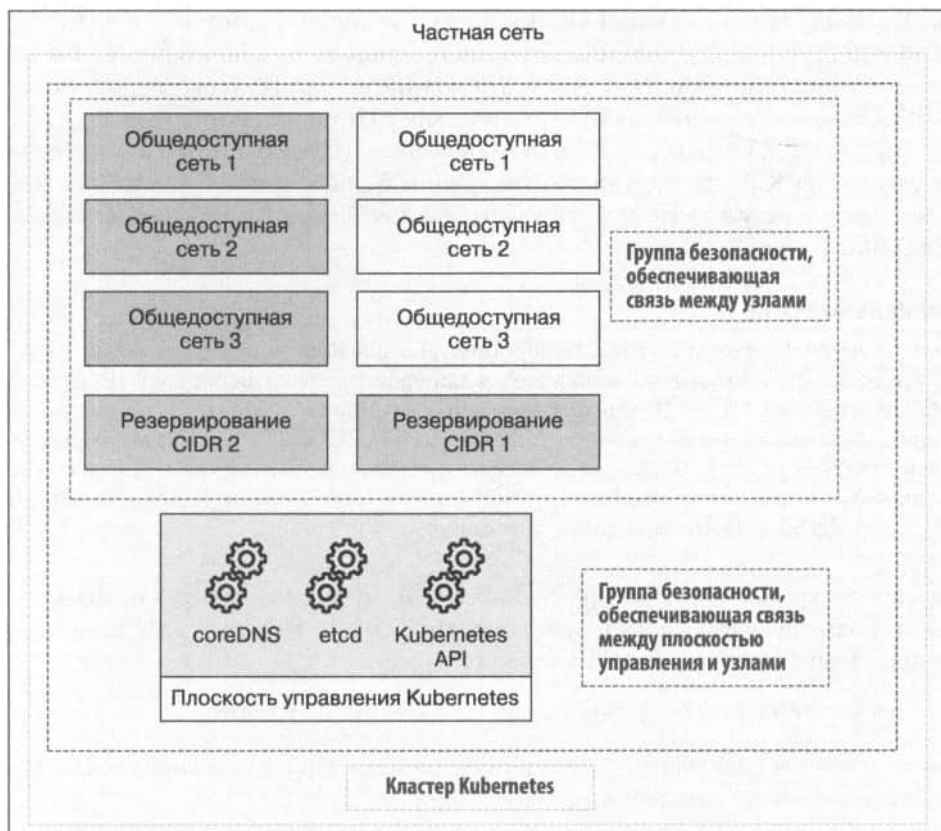
```
$ eksctl create cluster --name coffeemesh --region <aws_region> --fargate \
--alb-ingress-access
```

Процесс создания занимает около 30 минут. Рассмотрим каждый флаг в этой команде.

- `--name` — название кластера. Мы называем кластер `coffeemesh`.
- `--region` — регион AWS, в котором вы хотите развернуть кластер. Этот регион должен быть тем же, который вы использовали для создания репозитория ECR в разделе 13.4.
- `--fargate` — профиль Fargate для планирования модулей в пространствах имен `default` и `kube-system`. Профили — это политики, которые определяют, какие поды должны быть запущены Fargate.
- `--alb-ingress-access` — разрешает доступ к кластеру через балансировщик нагрузки приложения.

На рис. 14.3 показана архитектура стека, созданного `eksctl` при запуске кластера Kubernetes. По умолчанию `eksctl` создает выделенное виртуальное частное облако (virtual private cloud, VPC) для кластера.





**Рис. 14.3.** eksctl создает VPC с тремя общедоступными сетями, тремя частными сетями, двумя резервированиями CIDR и двумя группами безопасности VPC. Он также развертывает кластер Kubernetes внутри VPC

## СЕТЬ KUBERNETES

Чтобы максимально использовать возможности Kubernetes, вам необходимо понимать, как в Kubernetes работает сеть. Чтобы больше узнать о сетях Kubernetes, ознакомьтесь с книгой *Networking and Kubernetes: A Layered Approach* Джеймса Стронга и Валери Лэнси (O'Reilly, 2021).

Можно также запустить кластер в рамках существующего VPC, указав подсети, в которых следует запустить развертывание. При запуске в рамках существующего VPC необходимо убедиться, что VPC и предоставленные подсети правильно настроены для работы с кластером Kubernetes. Ознакомьтесь с документацией eksctl, чтобы узнать о сетевых требованиях кластера Kubernetes (<https://eksctl.io/usage/vpc-networking/>), и с официальной документацией AWS, чтобы узнать требования к сети VPC для кластера Kubernetes (<http://mng.bz/aPRY>).

Как вы можете видеть на рис. 14.3, `eksctl` по умолчанию создает шесть подсетей: три общедоступные и три частные, с соответствующими им шлюзами NAT и таблицами маршрутизации. *Подсеть* — это подмножество IP-адресов, доступных в VPC. Общедоступные подсети доступны через Интернет, в то время как частные — нет. `eksctl` также создает два резервирования CIDR подсети для внутреннего использования Kubernetes, а также две группы безопасности. Одна из них разрешает связь между всеми узлами в кластере, а другая — связь между плоскостью управления и узлами.

## ОПРЕДЕЛЕНИЕ CIDR

CIDR расшифровывается как Classless Inter-Domain Routing — бесклассовая маршрутизация. Это обозначение используется для представления диапазонов IP-адресов. CIDR записывается как IP-адрес, за которым следует косая черта и десятичное число, представляющее диапазон адресов. Например, `255.255.255.255/32` определяет диапазон для одного адреса. Чтобы узнать больше о нотации CIDR, смотрите статьи в «Википедии»: [https://en.wikipedia.org/wiki/Classless\\_Inter-Domain\\_Routing](https://en.wikipedia.org/wiki/Classless_Inter-Domain_Routing) и [https://ru.wikipedia.org/wiki/Бесклассовая\\_адресация](https://ru.wikipedia.org/wiki/Бесклассовая_адресация).

Создав кластер, мы можем настроить `kubectl` так, чтобы он указывал на него, что позволит нам управлять кластером из командной строки. Используйте следующую команду, чтобы указать `kubectl` на кластер:

```
$ aws eks update-kubeconfig --name coffeemesh --region <aws_region>
```

Теперь, когда мы подключены к кластеру, можем проверить его свойства. Например, получим список запущенных узлов:

```
$ kubectl get nodes
# output truncated:
NAME                                STATUS  ROLES  AGE    VERSION
fargate-ip-192-168-157-75.<aws_region>... Ready  <none> 4d16h  v1.20.7...
fargate-ip-192-168-170-234.<aws_region>... Ready  <none> 4d16h  v1.20.7...
fargate-ip-192-168-173-63.<aws_region>... Ready  <none> 4d16h  v1.20.7...
```

Чтобы получить список модулей, запущенных в кластере, выполните такую команду:

```
$ kubectl get pods -A
NAMESPACE   NAME                                READY  STATUS  RESTARTS  AGE
kube-system  coredns-647df9f975-2ns5m          1/1    Running  0          2d15h
kube-system  coredns-647df9f975-hcgjq          1/1    Running  0          2d15h
```

Есть еще много полезных команд, которые вы можете запустить, чтобы больше узнать о своем кластере. Ознакомьтесь с официальной документацией о Kubernetes CLI, чтобы выучить дополнительные команды и опции (<https://kubernetes.io/docs/>

reference/kubectl/). Можете также воспользоваться шпаргалкой `kubectl` (<https://kubernetes.io/docs/reference/kubectl/cheatsheet/>).

Теперь, когда кластер запущен, создадим роль IAM для наших учетных записей сервисов Kubernetes.

## 14.4. ИСПОЛЬЗОВАНИЕ РОЛЕЙ IAM ДЛЯ УЧЕТНЫХ ЗАПИСЕЙ СЕРВИСОВ KUBERNETES

Каждый процесс, который выполняется в вашем кластере Kubernetes, содержит информацию, идентифицирующую личность, и эта информация предоставляется *учетной записью сервиса*. Учетные записи сервисов определяют права доступа процесса внутри кластера. Иногда нашим сервисам необходимо взаимодействовать с ресурсами AWS через API AWS. Чтобы предоставить доступ к API AWS, нам нужно создать *роли IAM* — сущности, которые открывают приложениям доступ к AWS API — для наших сервисов. Чтобы связать учетную запись сервиса Kubernetes с ролью IAM, мы используем OpenID Connect (OIDC) (рис. 14.4). Благодаря этому наши модули могут получить временные учетные данные для доступа к API AWS.



**Рис. 14.4.** Поды могут пройти аутентификацию у поставщика OIDC, чтобы взять на себя роль IAM, что дает им доступ к AWS API и, следовательно, к сервисам AWS

Чтобы проверить, есть ли в вашем кластере поставщик OIDC, запустите следующую команду, заменив `<cluster_name>` названием кластера:

```
$ aws eks describe-cluster --name coffeemesh \
--query "cluster.identity.oidc.issuer" --output text
```

Вы получите результат, подобный следующему:

```
https://oidc.eks.<aws\_region>.amazonaws.com/id/BE4E5EE7DCDF9FB198D06FC9883FF1BE
```

В этом случае ID поставщика кластера — `BE4E5EE7DCDF9FB198D06FC9883FF1BE`. Теперь запустите следующую команду:

```
$ aws iam list-open-id-connect-providers | \
grep BE4E5EE7DCDF9FB198D06FC9883FF1BE
```

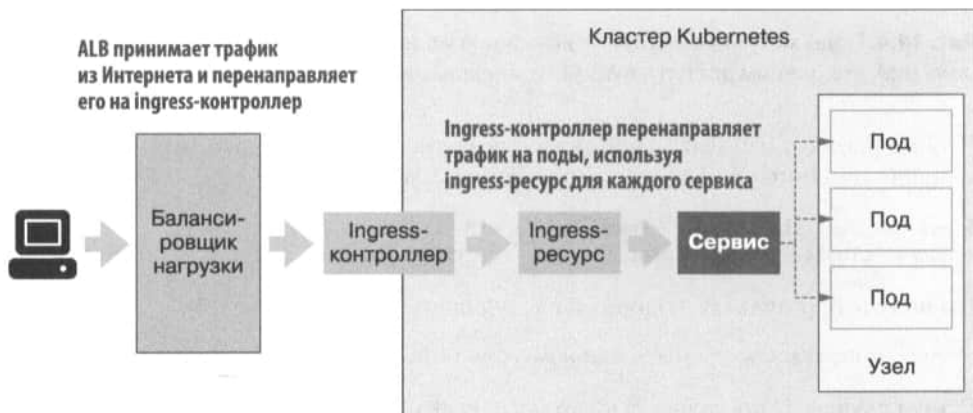
Она выводит список всех поставщиков OIDC в вашей учетной записи AWS и использует `grep` для фильтрации по ID поставщика OIDC кластера. Если вы получите пекий результат, это означает, что у вас уже есть поставщик OIDC. Если вы не получите никакого вывода, значит, у вас нет поставщика OIDC. Чтобы создать поставщик для кластера, запустите следующую команду, заменив `<cluster_name>` именем вашего кластера:

```
$ eksctl utils associate-iam-oidc-provider --cluster <cluster_name> \
--approve
```

Вот теперь можно привязать роли IAM к учетным записям сервисов. В следующем разделе мы развернем балансировщик нагрузки Kubernetes, чтобы включить в кластер внешний трафик.

## 14.5. РАЗВЕРТЫВАНИЕ БАЛАНСИРОВЩИКА НАГРУЗКИ KUBERNETES

Прямо сейчас наш кластер недоступен за пределами VPC. Если мы развернем наши приложения, они получат только внутренние IP-адреса и, следовательно, не будут доступны внешнему миру. Чтобы открыть внешний доступ к кластеру, нам нужен ingress-контроллер. Как вы можете видеть на рис. 14.5, ingress-контроллер принимает трафик за пределами кластера Kubernetes и распределяет нагрузку между подами. Чтобы перенаправить трафик на определенные поды, мы создаем *ресурсы ingress* для каждого из сервисов. Ingress-контроллер заботится об управлении ресурсами ingress.



**Рис. 14.5.** Ingress-контроллер принимает трафик за пределами кластера Kubernetes и перенаправляет его в поды в соответствии с правилами, определенными ingress-ресурсами

В этом разделе мы развернем ingress-контроллер Kubernetes в качестве контроллера балансировщика нагрузки (AWS Load Balancer Controller, LBC)<sup>1</sup>. Как вы можете видеть на рис. 14.5, контроллер балансировщика нагрузки развертывает балансировщик нагрузки уровня приложения (AWS Application Load Balancer, ALB), который находится перед нашим кластером, захватывает входящий трафик и перенаправляет его в наши сервисы. Для перенаправления трафика на сервисы ALB использует концепцию *целевых групп* (*target groups*) — правило, определяющее, как трафик должен перенаправляться из ALB на определенный ресурс. Например, у нас могут быть целевые группы, основанные на IP-адресах, ID сервисов и других показателях. Балансировщик нагрузки отслеживает работоспособность своих зарегистрированных целей и следит за тем, чтобы трафик перенаправлялся только на действительные цели.

Чтобы установить AWS Load Balancer Controller, нам нужен поставщик OIDC в кластере, поэтому, прежде чем продолжить, убедитесь, что прочли раздел 14.4. Первым шагом к развертыванию AWS Load Balancer Controller будет создание политики IAM, которая предоставляет контроллеру доступ к соответствующим API AWS. Сообщество с открытым исходным кодом, поддерживающее проект AWS Load Balancer Controller, предлагает образец необходимой нам политики, поэтому можно взять его:

```
$ curl -o alb_controller_policy.json \
https://raw.githubusercontent.com/kubernetes-sigs/aws-load-balancer-
controller/main/docs/install/iam_policy.json
```

После выполнения этой команды вы увидите в своем каталоге файл с именем `alb_controller_policy.json`. Теперь с его помощью можно создать политику IAM:

```
$ aws iam create-policy \
--policy-name ALBControllerPolicy \
--policy-document file://alb_controller_policy.json
```

Следующим шагом будет создание роли IAM, связанной с учетной записью сервиса Kubernetes для балансировщика нагрузки:

```
$ eksctl create iamserviceaccount \
--cluster=coffeemesh \
--namespace=kube-system \
--name=alb-controller \
--attach-policy-arn=arn:aws:iam::<aws_account_id>:policy/ALBControllerPolicy \
--override-existing-serviceaccounts \
--approve
```

---

<sup>1</sup> AWS Load Balancer Controller – это проект с открытым исходным кодом, размещенный на GitHub (<https://github.com/kubernetes-sigs/aws-load-balancer-controller/>). Изначально был создан Ticketmaster и CoreOS.

Эта команда создает стек CloudFormation, содержащий роль IAM, связанную с политикой, которую мы создали ранее, а также учетную запись сервиса с именем `alb-controller` в пространстве имен `kube-system`, зарезервированном для системных компонентов кластера Kubernetes.

Теперь мы можем установить контроллер балансировщика нагрузки. Для этого воспользуемся Helm — менеджером пакетов для Kubernetes. Если на вашем компьютере нет Helm, установите его. Существуют различные стратегии установки Helm в зависимости от вашей платформы, поэтому обязательно ознакомьтесь с документацией, чтобы узнать, какой вариант подходит вам лучше всего (<https://helm.sh/docs/intro/install/>).

Как только вы установите Helm на компьютере, обновите его, добавив репозиторий чартов EKS в ваш локальный helm (в Helm пакеты называются *чартами (charts)*). Чтобы добавить чарты EKS, выполните следующую команду:

```
$ helm repo add eks https://aws.github.io/eks-charts
```

Далее обновим helm, чтобы убедиться, что мы получаем самые последние обновления чартов:

```
$ helm repo update
```

Теперь, когда helm обновлен, мы можем установить AWS Load Balancer Controller. Для этого нужно получить идентификатор VPC `eksctl`, созданный при запуске кластера. Чтобы найти его, выполните следующую команду:

```
$ eksctl get cluster --name coffeemesh -o json | \
jq '[0].ResourcesVpcConfig.VpcId'
# output: "vpc-07d35ccc982a082c9"
```

Чтобы вы могли успешно выполнить эту команду, у вас должен быть установлен jq. Вернитесь к разделу 14.1, чтобы узнать, как его установить. Теперь можно установить контроллер, выполнив следующую команду:

```
$ helm install aws-load-balancer-controller eks/aws-load-balancer-
➡ controller \
  -n kube-system \
  --set clusterName=coffeemesh \
  --set serviceAccount.create=false \
  --set serviceAccount.name=alb-controller \
  --set vpcId=<vpc_id>
```

Поскольку контроллер является внутренним компонентом Kubernetes, мы устанавливаем его в пространстве имен `kube-system`. Следует убедиться, что контроллер установлен для кластера `coffeemesh`. Далее проинструктируем Helm не создавать новую учетную запись сервиса для контроллера, а вместо этого использовать учетную запись сервиса `alb-controller`, которую мы создали ранее.

Все это займет несколько минут. Чтобы убедиться, что развертывание прошло успешно, выполните следующую команду:

```
$ kubectl get deployment -n kube-system aws-load-balancer-controller
```

NAME	READY	UP-TO-DATE	AVAILABLE	AGE
alb-controller	2/2	2	2	84s

Вы поймете, что контроллер запущен, когда в столбце **READY** появится значение 2/2 — это означает, что требуемое количество ресурсов доступно. Наш кластер готов, так что пришло время внедрять сервис заказов!

## 14.6. РАЗВЕРТЫВАНИЕ МИКРОСЕРВИСОВ В КЛАСТЕРЕ KUBERNETES

Теперь, когда наш кластер Kubernetes готов, пришло время приступить к развертыванию сервисов. В этом разделе мы рассмотрим этапы развертывания сервиса заказов. Вы можете выполнить те же действия для развертывания других сервисов платформы CoffeeMesh.

Как вы можете видеть на рис. 14.6, мы развертываем сервис заказов в новом пространстве имен под названием **orders-service**. Это позволяет нам логически сгруппировать и изолировать все ресурсы, необходимые для работы сервиса заказов. Чтобы создать новое пространство имен, запустим следующую команду:

```
$ kubectl create namespace orders-service
```



**Рис. 14.6.** Чтобы развернуть микросервис, создаем новое пространство имен и в пределах этого пространства имен развертываем все компоненты, необходимые для работы микросервиса, такие как объект **Deployment** и объект **Service**

Поскольку мы будем запускать сервис заказов в новом пространстве имен, нам также необходимо создать новый профиль Fargate, настроенный для планирования заданий в пространстве имен `orders-service`. Для этого выполните следующую команду:

```
$ eksctl create fargateprofile --namespace orders-service --cluster \
  coffeemesh --region <aws_region>
```

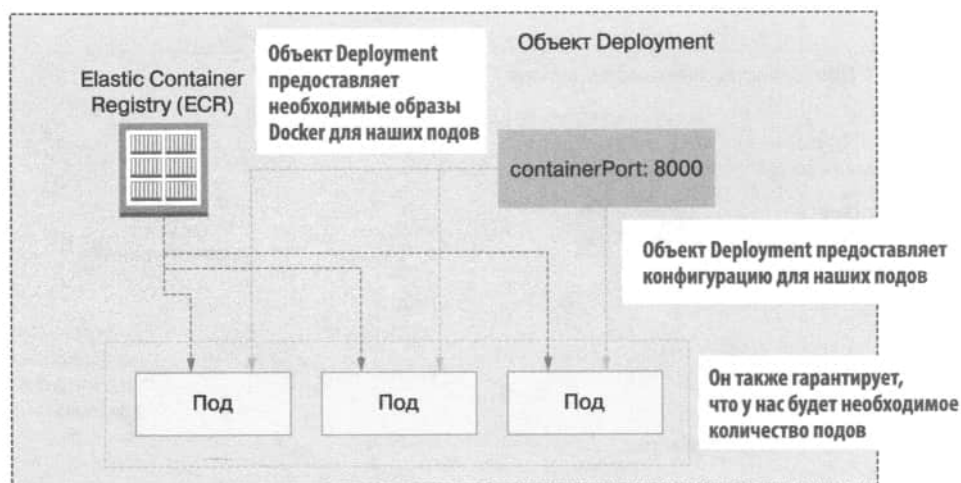
Когда пространство имен `orders-service` и профиль Fargate готовы, можно развернуть сервис заказов. Для этого сделайте следующее.

1. Создайте объект развертывания для сервиса заказов.
2. Создайте объект Service.
3. Создайте ingress-ресурс для предоставления доступа к сервису.

Далее подробно объясняется, как действовать на каждом этапе.

### 14.6.1. Создание объекта развертывания

Начнем с создания развертывания для сервиса заказов с использованием его файла манифеста. Как вы можете видеть на рис. 14.7, развертывания — это объекты Kubernetes, которые управляют нашими подами и предоставляют им все необходимое для запуска, включая образ Docker и конфигурацию порта. Создайте файл с именем `orders-service-deployment.yaml` и скопируйте в него содержимое листинга 14.1.



**Рис. 14.7.** Объект Deployment предоставляет необходимую конфигурацию для подов, такую как образ Docker и конфигурация портов, а также гарантирует, что у нас есть необходимое количество запущенных подов



Мы используем API Kubernetes версии `apps/v1` и объявляем этот объект как `Deployment`. В метаданных мы называем развертывание `orders-service`, указываем его пространство имен и добавляем метку `app: orders-service`. *Метки (labels)* — это пользовательские идентификаторы для объектов Kubernetes, и их можно использовать в том числе для мониторинга, отслеживания или планирования задач<sup>1</sup>.

В разделе `spec` мы определяем правило выбора, которое сопоставляет поды с меткой `app: orders-service`. Это означает, что в данном развертывании будут работать поды только с этой меткой. Мы также заявляем, что хотели бы запустить лишь одну реплику пода.

В разделе `spec.template` мы определяем под, управляемый этим развертыванием. Мы помечаем его парой «ключ — значение» `app: orders-service` в соответствии с правилом выбора развертывания. В разделе `spec` мы объявляем контейнеры, которые принадлежат поду. В данном случае мы хотим запустить только один контейнер в виде приложения сервиса заказов. В определении контейнера сервиса заказов мы задаем образ, который должен использоваться для запуска приложения, с указанием порта, на котором выполняется приложение.

#### Листинг 14.1. Объявление манифеста развертывания

```
# file: orders-service-deployment.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: orders-service
  namespace: orders-service
  labels:
    app: orders-service
spec:
  replicas: 1
  selector:
    matchLabels:
      app: orders-service
  template:
    metadata:
      labels:
        app: orders-service
    spec:
      containers:
        - name: orders-service
```

Версия API Kubernetes, используемая в этом манифесте

Этот манифест определяет объект развертывания

Название развертывания

Пространство имен, в пределах которого должно быть расположено развертывание

Метка для развертывания

Спецификация развертывания

Сколько подов должно быть развернуто

Селектор меток для подов

Шаблон для подов

Метка для подов

Спецификация для подов

<sup>1</sup> Чтобы узнать больше о метках и о том, как их использовать, ознакомьтесь с официальной документацией: <https://kubernetes.io/docs/concepts/overview/working-with-objects/labels/>. И прочтите статью Зейна Хичкокса под названием *matchLabels, Labels and Selectors Explained in Detail, for Beginners* («matchLabels, метки и селекторы, подробное объяснение для начинающих»), Medium (15 июля 2018 г.). <https://medium.com/@zwhitchcox/matchlabels-labels-and-selectors-explained-in-detail-for-beginners-d421bdd05362>.

```
image: <aws_account_id>.dkr.ecr.<aws_region>.amazonaws.com/
  coffeemesh-orders:1.0 ← Образ пода
ports:
  - containerPort: 8000 ← Порт, на котором выполняется API
imagePullPolicy: Always
```

Чтобы создать развертывание, мы запускаем следующую команду:

```
$ kubectl apply -f orders-service-deployment.yaml
```

Эта команда создает развертывание и запускает поды, которые мы определили в файле манифеста. Потребуется несколько секунд, чтобы поды стали доступны. Вы можете проверить их состояние с помощью такой команды:

```
$ kubectl get pods -n orders-service
```

Начальное состояние подов будет Pending, но как только они будут запущены, их состояние изменится на Running.

## ЧТО ТАКОЕ ФАЙЛ МАНИФЕСТА KUBERNETES

В Kubernetes можно создавать объекты, используя файлы манифеста. Объекты — это ресурсы, такие как пространства имен, развертывания, сервисы и т. д. Файл манифеста — это YAML-файл, который описывает свойства объекта и его желаемое состояние. Эти файлы удобно использовать, так как их можно отслеживать в системе управления версиями, что помогает нам видеть изменения в инфраструктуре.

Каждый файл манифеста содержит как минимум следующие свойства:

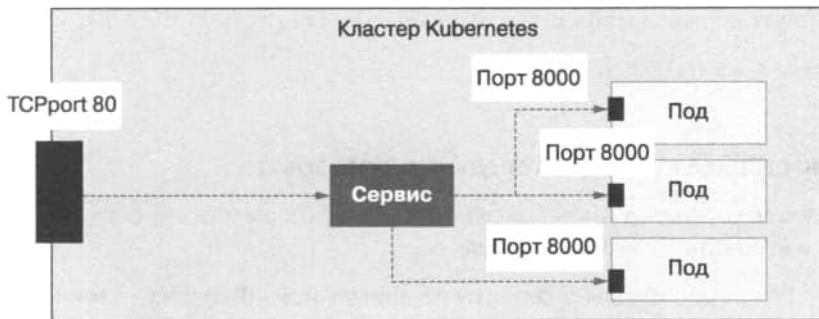
- `apiVersion` — версия API Kubernetes, которую мы хотим использовать. Каждый объект Kubernetes имеет свою стабильную версию. Вы можете проверить последнюю стабильную версию каждого объекта для вашего кластера Kubernetes, выполнив следующую команду: `kubectl api-resources`;
- `kind` — вид объекта, который мы создаем. Примеры значений: `Service`, `Ingress` и `Deployment`;
- `metadata` — набор свойств, которые предоставляют идентифицирующую информацию об объекте, например его имя, пространство имен и дополнительные метки;
- `spec` — спецификация для объекта. Например, при создании сервиса мы используем этот раздел, чтобы указать тип создаваемого сервиса (`NodePort`) и правила выбора.

Для создания объекта из файла манифеста воспользуйтесь командой `kubectl apply`. Например, если у вас есть файл манифеста с именем `deployment.yaml`, используйте его, введя следующую команду:

```
$ kubectl apply -f deployment.yaml
```

## 14.6.2. Создание объекта сервиса

Теперь, когда наше развертывание готово, создадим объект для сервиса заказов. В разделе 14.2 мы обсуждали, что сервисы — это объекты Kubernetes, которые позволяют нам предоставлять поды в качестве сетевых сервисов. Как вы можете видеть на рис. 14.8, объект сервиса предоставляет нашим приложениям доступ к веб-сервисам и перенаправляет трафик из кластера в поды по указанным портам. Создайте файл с именем `orders-service.yaml` и скопируйте в него содержимое листинга 14.2, в котором показано, как настроить простой манифест сервиса.



**Рис. 14.8.** Объект сервиса перенаправляет трафик из кластера в поды по указанным портам. Здесь трафик, входящий в кластер на порте 80, перенаправляется в поды на порте 8000

Для объявления сервиса мы используем версию v1 API Kubernetes. В метаданных указываем, что сервис называется `orders-service` и он должен быть запущен в пространстве имен `orders-service`. Добавляем метку: `app: orders-service`. В разделе `spec` сервиса задаем тип `ClusterIP`, то есть под будет доступен только изнутри кластера. В Kubernetes есть и другие типы сервисов, такие как `NodePort` и `LoadBalancer`. (Больше информации о типах сервисов и о том, когда использовать каждый из них, приводится во врезке «Какой тип сервиса Kubernetes следует использовать» ниже.)

Кроме того, создаем правило переадресации для перенаправления трафика с порта 80 на порт 8000 — именно на нем работают наши контейнеры. Наконец, указываем селектор для метки `app: orders-service`, то есть сервис будет использовать поды только с этой меткой.

### Листинг 14.2. Объявление манифеста сервиса

```
# file: orders-service.yaml
```

```
apiVersion: v1
kind: Service
metadata:
  name: orders-service
  namespace: orders-service
```

Этот манифест определяет объект Service

```

labels:
  app: orders-service
spec:
  selector:
    app: orders-service
  type: ClusterIP
  ports:
    - protocol: http
      port: 80
      targetPort: 8000

```

Это тип сервиса (Service) ClusterIP

Сервис взаимодействует по протоколу HTTP

Сервис должен быть подключен к порту 80

Сервис запускается внутри системы через порт 8000

Чтобы развернуть сервис, выполните следующую команду:

```
$ kubectl apply -f orders-service.yaml
```

### КАКОЙ ТИП СЕРВИСА KUBERNETES СЛЕДУЕТ ИСПОЛЬЗОВАТЬ

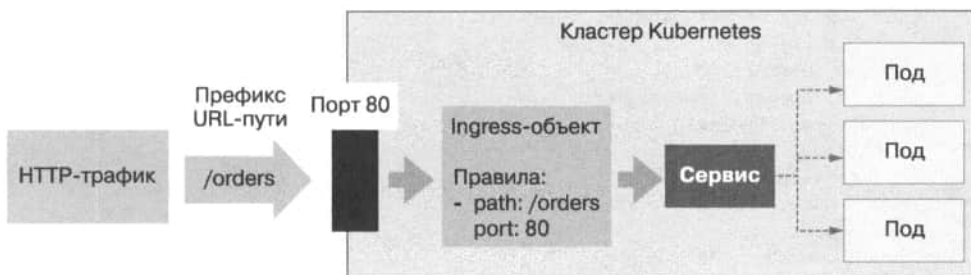
В Kubernetes есть четыре типа сервисов. Здесь мы обсудим особенности каждого типа и варианты их использования:

- **ClusterIP** — предоставляет сервисы по внутреннему IP-адресу кластера и, следовательно, делает их доступными только внутри кластера;
- **NodePort** — предоставляет сервисы по внешнему IP-адресу узла и, следовательно, делает их доступными в сети кластера;
- **LoadBalancer** — предоставляет сервис напрямую через выделенный облачный балансировщик нагрузки;
- **ExternalName** — предоставляет сервис по внутренней записи DNS внутри кластера.

Какой из этих типов вам использовать? Зависит от ваших задач. **NodePort** полезен, если нужно иметь внешний доступ к своим сервисам по IP-адресу узла, на котором они запущены. Недостаток в том, что сервис использует статический порт узла, поэтому вы можете запускать только один сервис на каждом узле. **ClusterIP** полезен, если вы предпочитаете получать доступ к сервису по IP-адресу кластера. Сервисы **ClusterIP** недоступны напрямую за пределами кластера, но вы можете предоставить к ним доступ, создав правила **ingress**, которые перенаправляют к ним трафик. **LoadBalancer** полезен, если вы хотите использовать один облачный балансировщик нагрузки для каждого сервиса. Это несколько упрощает настройку, поскольку вам не придется настраивать несколько правил **ingress**. Однако балансировщики нагрузки обычно являются самыми дорогими компонентами кластера, поэтому, если важно сэкономить, не стоит использовать этот параметр. Наконец, **ExternalName** полезен, если вы хотите получать доступ к своим сервисам из кластера через кастомные домены.

### 14.6.3. Предоставление доступа к сервисам с помощью ingress-объектов

Заключительный шаг — предоставление доступа к сервису через Интернет. Чтобы предоставить доступ к сервису, нам нужно создать ingress-ресурс, который направляет трафик к сервису. Как вы можете видеть на рис. 14.9, ingress-ресурс — это сервис, который перенаправляет HTTP-трафик на поды, запущенные в кластере Kubernetes, по указанным портам и URL-путям. Создайте файл с именем `orders-service-ingress.yaml` и скопируйте в него содержимое листинга 14.3.



**Рис. 14.9.** Ingress-объект позволяет нам перенаправлять HTTP-трафик по определенному порту и URL-пути к объекту сервиса

В ingress-манифесте мы используем версию `networking.k8s.io/v1` API Kubernetes и объявляем объект как тип `Ingress`. В метаданных даем входящему объекту имя `orders-service-ingress` и указываем, что он должен быть развернут в пространстве имен `orders-service`. Мы используем аннотации для привязки ingress-объекта к балансировщику нагрузки AWS, который развернули в разделе 14.5. В разделе `spec` мы определяем правила пересылки ingress-ресурса. Объявляем правило HTTP, которое перенаправляет весь трафик по пути `/orders` в сервис заказов, и дополнительные правила для доступа к документации API сервиса.

#### Листинг 14.3. Объявление ingress-манифеста

```
# file: orders-service-ingress.yaml
```

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: orders-service-ingress
  namespace: orders-service
  annotations:
    kubernetes.io/ingress.class: alb
    alb.ingress.kubernetes.io/target-type: ip
    alb.ingress.kubernetes.io/scheme: internet-facing
spec:
  rules:
    - http:
```

Манифест определяет ingress-объект

Конфигурация AWS для Ingress

Ingress предоставляет доступ к Application Load Balancer

Трафик направляется в поды на основе IP

Правила переадресации трафика

Ingress доступен для внешних подключений

```

paths:
- path:/orders
  pathType: Prefix
  backend:
    service:
      name: orders-service
      port:
        number: 80
- path:/docs/orders
  pathType: Prefix
  backend:
    service:
      name: orders-service
      port:
        number: 80
- path:/openapi/orders.json
  pathType: Prefix
  backend:
    service:
      name: orders-service
      port:
        number: 80

```

Правило для URL-пути /orders

Правило применяется к запросам, начинающимся с префикса /orders

Внутренний сервис, который обрабатывает этот трафик

Трафик должен быть перенаправлен в сервис orders-service

Сервис orders-service доступен на порте 80

Чтобы создать этот ingress-ресурс, мы запускаем следующую команду:

```
$ kubectl apply -f orders-service-ingress.yaml
```

Теперь доступен API сервиса заказов. Чтобы вызвать API, сначала нужно выяснить эндпоинт для правила ingress, которое мы только что создали. Выполните следующую команду, чтобы получить подробную информацию об ingress-ресурсе:

```
$ kubectl get ingress/orders-service-ingress -n orders-service
# output truncated:
NAME          CLASS    HOSTS    ADDRESS...
orders-service-ingress  <none>  *        k8s-ordersse-ordersse-3c391193...
```

Значение в поле ADDRESS — это URL-адрес балансировщика нагрузки. Вы также можете получить его, выполнив следующую команду:

```
$ kubectl get ingress/orders-service-ingress -n orders-service -o json | \
jq '.status.loadBalancer.ingress[0].hostname'
"k8s-ordersse-ordersse-3c39119336-236890178.<aws_region>.elb.amazonaws.com"
```

Мы можем использовать этот URL-адрес для вызова API сервиса заказов. Поскольку база данных еще не готова, сам API работать не будет, но мы можем получить доступ к документации API:

```
$ curl http://k8s-ordersse-ordersse-3c39119336-236890178.<aws_region>.elb.
amazonaws.com/openapi/orders.json
```

Может понадобиться некоторое время, чтобы балансировщик нагрузки стал доступен, и в это время curl не разрешит доступ к хосту. Если это произойдет,

подождите несколько минут и повторите попытку. Чтобы иметь возможность взаимодействовать с API, мы должны настроить базу данных, что и будет задачей нашего следующего раздела!

## 14.7. НАСТРОЙКА БЕССЕРВЕРНОЙ БАЗЫ ДАННЫХ С ПОМОЩЬЮ AWS AURORA

Сервис заказов почти готов: приложение запущено и мы можем получить к нему доступ через Интернет. Отсутствует только один компонент: база данных. Мы можем настроить базу данных как развертывание в нашем кластере Kubernetes с подключенным томом или выбрать один из множества сервисов управления базами данных, предлагаемых облачными провайдерами.

Чтобы не усложнять работу, в этом разделе мы настроим бессерверную базу данных Aurora от AWS — это мощный сервис баз данных, который экономичен, поскольку вы платите только за то, что используете, и очень удобен, поскольку не нужно беспокоиться об управлении базой данных или ее масштабировании.

### 14.7.1. Создание бессерверной базы данных Aurora

Запустим нашу базу данных Aurora в рамках VPC-кластера Kubernetes. Чтобы иметь возможность запускать базу данных в рамках существующего VPC, нам нужно создать *группу подсетей базы данных*: набор подсетей внутри VPC. Как вы узнали из раздела 14.3, `eksctl` делит VPC-кластер Kubernetes на шесть подсетей: три общедоступные и три частные. Шесть подсетей распределены по трем *зонам доступности* (центрам обработки данных в регионе AWS), по одной общедоступной и одной частной подсети на зону доступности.

При выборе подсетей для нашей группы подсетей базы данных необходимо учитывать следующие ограничения.

- Aurora Serverless поддерживает только одну подсеть на зону доступности.
- При создании группы подсетей все подсети должны быть либо частными, либо общедоступными<sup>1</sup>.

В целях безопасности рекомендуется использовать частные подсети, поскольку это гарантирует, что сервер базы данных недоступен за пределами VPC, то есть внешние и неавторизованные пользователи не смогут подключиться к нему напрямую.

---

<sup>1</sup> Для получения дополнительной информации по этому вопросу смотрите официальную документацию AWS: [https://docs.aws.amazon.com/AmazonRDS/latest/UserGuide/USER\\_VPC.WorkingWithRDSInstanceinaVPC.html](https://docs.aws.amazon.com/AmazonRDS/latest/UserGuide/USER_VPC.WorkingWithRDSInstanceinaVPC.html).

Чтобы найти список частных подсетей в VPC, нам сначала нужно получить ID VPC-кластера Kubernetes с помощью следующей команды:

```
$ eksctl get cluster --name coffeemesh -o json | \
jq '.[0].ResourcesVpcConfig.VpcId'
```

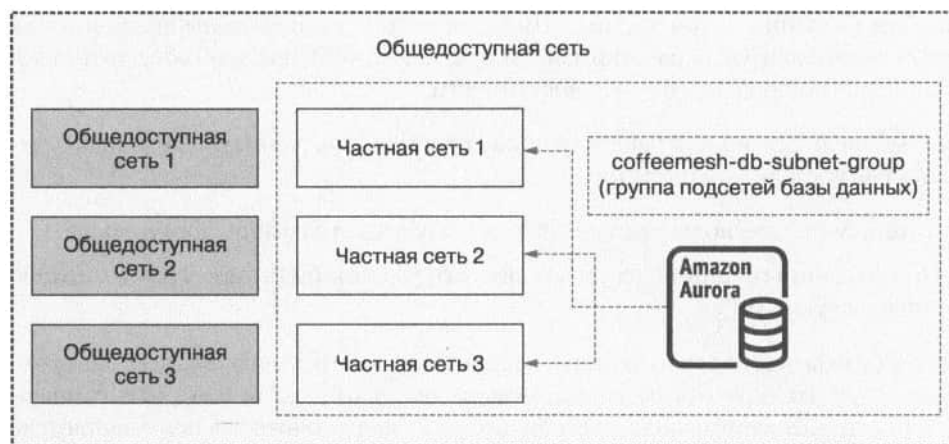
Затем используйте следующую команду, чтобы получить ID частных подсетей в VPC:

```
$ aws ec2 describe-subnets --filters Name=vpc-id,Values=<vpc_id> \
--output json | jq '.Subnets[] | select(.MapPublicIpOnLaunch == false) | \
.SubnetId'
```

Предыдущая команда перечисляет все подсети в VPC-кластере Kubernetes и использует `jq` для фильтрации общедоступных подсетей. Вооружившись всей этой информацией, мы можем создать группу подсетей базы данных, выполнив следующую команду:

```
$ aws rds create-db-subnet-group --db-subnet-group-name \
coffeemesh-db-subnet-group --db-subnet-group-description "Private subnets" \
--subnet-ids "<subnet_id>" "<subnet_id>" "<subnet_id>"
```

Как вы можете видеть на рис. 14.10, эта команда создает группу подсетей базы данных с именем `coffeemesh-db-subnet-group`. При выполнении команды убедитесь, что вы заменили `<subnet_id>` идентификаторами ваших частных подсетей. Мы развернем нашу базу данных Aurora в рамках этой группы подсетей базы данных.



**Рис. 14.10.** Мы развертываем базу данных Aurora в группе подсетей базы данных с именем `coffeemeshdb-subnet-group`. Группа подсетей создается поверх трех частных подсетей нашего VPC для предотвращения несанкционированного доступа



Далее нам нужно создать *группу безопасности VPC* — набор правил, определяющих, какой входящий и исходящий трафик разрешен с VPC, — она разрешает трафик к базе данных, чтобы наши приложения могли подключаться к ней. Следующая команда создает группу безопасности db-access:

```
$ aws ec2 create-security-group --group-name db-access --vpc-id <vpc-id> \
--description "Security group for db access"
# output:
{
  "GroupId": "sg-00b47703a4299924d"
}
```

В предыдущей команде замените <vpc-id> на ID вашего VPC-кластера Kubernetes. Результатом предыдущей команды станет ID группы безопасности, которую мы только что создали. Мы разрешим трафик со всех IP-адресов на порт PostgreSQL по умолчанию — это порт 5432. Поскольку мы собираемся развернуть базу данных в частных подсетях, можно прослушивать все IP-адреса, но для дополнительной безопасности вы можете ограничить диапазон адресами ваших модулей. Создадим ingress-правило для трафика в отношении нашей группы безопасности, используя следующую команду:

```
$ aws ec2 authorize-security-group-ingress --group-id \
<db-security-group-id> --ip-permissions \
'FromPort=5432,IpProtocol=TCP,IpRanges=0.0.0.0/0'
```

В этой команде замените <db-security-group-id> на ID вашей группы безопасности, определяющей доступ к базе данных.

Теперь, когда у нас есть группа подсетей базы данных и группа безопасности, которая позволяет нашим подам подключаться к ней, мы можем использовать первую для запуска бессерверного кластера Aurora внутри нашего VPC. Выполните следующую команду, чтобы запустить бессерверный кластер Aurora:

```
$ aws rds create-db-cluster --db-cluster-identifier coffeemesh-orders-db \
--engine aurora-postgresql --engine-version 10.14 \
--engine-mode serverless \
--scaling-configuration MinCapacity=8,MaxCapacity=64,
➤ SecondsUntilAutoPause=1000,AutoPause=true \
--master-username <username> \
--master-user-password <password> \
--vpc-security-group-ids <security_group_id> \
--db-subnet-group <db_subnet_group_name>
```

Рассмотрим параметры команды.

- `--db-cluster-identifier` — имя кластера базы данных, у нас это `coffeemesh-orders-db`.

- `--engine` — движок (или ядро) базы данных, который вы хотите использовать. Мы используем движок, совместимый с PostgreSQL, но вы можете выбрать движок, совместимый с MySQL.
- `--engine-version` — версия движка СУБД, которую вы хотите использовать. Мы выбираем версию 10.14, которая на данный момент является единственной версией, доступной для бессерверной Aurora PostgreSQL. Ознакомьтесь с документацией AWS, чтобы быть в курсе новых версий (<http://mng.bz/gRyn>).
- `--engine-mode` — режим движка базы данных. Мы выбираем бессерверный, чтобы сделать пример простым и экономичным.
- `--scaling-configuration` — конфигурация автоматического масштабирования для кластера Aurora. Мы настраиваем кластер с минимальным количеством 8 единиц пропускной способности Aurora (Aurora Capacity Units, ACU) и максимальным — 64. Каждый ACU обеспечивает примерно 2 Гбайт оперативной памяти. Мы также настраиваем кластер на автоматическое уменьшение масштаба до 0 ACU после 1000 секунд бездействия<sup>1</sup>.
- `--master-username` — имя главного пользователя базы данных.
- `--master-user-password` — пароль главного пользователя базы данных.
- `--vpc-security-group-ids` — ID группы безопасности, определяющей доступ к базе данных, которую мы создали на предыдущем шаге.
- `--db-subnet-group` — название группы безопасности базы данных, которую мы создали ранее.

После выполнения этой команды вы получите большую полезную нагрузку в формате JSON с подробной информацией о базе данных. Чтобы подключиться к базе данных, нам нужно значение свойства `DBCluster.Endpoint` полезной нагрузки, которое содержит имя хоста базы данных. Мы будем использовать это значение в следующих подразделах для подключения к базе данных.

## 14.7.2. Управление секретами в Kubernetes

Чтобы подключить сервисы к базе данных, нам нужен безопасный способ передачи учетных данных для подключения. Основной способ управления конфиденциальной информацией в Kubernetes — использование секретов. Так мы избегаем необходимости раскрывать конфиденциальную информацию в коде или через наши образы. В этом подразделе вы узнаете, как безопасно управлять секретами Kubernetes.

---

<sup>1</sup> Смотрите официальную документацию для получения дополнительной информации о том, как работает Aurora Serverless, и параметрах конфигурации автоматического масштабирования: <https://docs.aws.amazon.com/AmazonRDS/latest/AuroraUserGuide/aurora-serverless-v2.html>.

AWS EKS предлагает два безопасных способа управления секретами: мы можем использовать AWS Secrets & Configuration Provider для Kubernetes<sup>1</sup> или AWS Key Management Service (KMS) для защиты секретов методом *envelope encryption*. В этом подразделе мы будем использовать *envelope encryption*<sup>2</sup>.

*Envelope encryption* — это метод шифрования данных с помощью ключа шифрования данных (DEK) и шифрования DEK с помощью ключа шифрования ключа (KEK)<sup>3</sup> (рис. 14.11). Это звучит сложно, но этот метод прост в использовании, поскольку AWS выполняет за нас всю тяжелую работу.



**Рис. 14.11.** *Envelope encryption* — это метод шифрования данных с помощью ключа шифрования данных (DEK) и шифрования DEK с помощью ключа шифрования ключа

Чтобы использовать *envelope encryption*, сначала нужно сгенерировать ключ AWS KMS. Вы можете использовать для этого следующую команду:

```
$ aws kms create-key
```

Результатом станет полезная нагрузка с метаданными о вновь созданном ключе. Из этой полезной нагрузки нам нужно взять свойство `KeyMetadata.Arn`, которое представляет ARN ключа или имя ресурса Amazon. Следующий шаг — включить шифрование секретов в нашем кластере Kubernetes с помощью `eksctl`:

```
$ eksctl utils enable-secrets-encryption --cluster coffeemesh \
--key-arn=<key_arn> --region <aws_region>
```

<sup>1</sup> Вы можете больше узнать об этом из статьи: *Pierce T. How to use AWS Secrets & Configuration Provider with Your Kubernetes Secrets Store CSI driver* («Как использовать AWS Secrets & Configuration Provider с вашим CSI-драйвером хранилища секретов Kubernetes»), <https://aws.amazon.com/blogs/security/how-to-use-aws-secrets-configuration-provider-with-kubernetes-secrets-store-csi-driver/>.

<sup>2</sup> Безопасное управление Kubernetes — важная тема, и чтобы узнать о ней больше, см. книгу: *Bueno A. S., Block A. Securing Kubernetes Secrets* (Manning, 2022). <https://livebook.manning.com/book/securing-kubernetes-secrets/chapter-4/v-3/point-13495-119-134-1>.

<sup>3</sup> Там же.

Убедитесь, что вы заменили `<key_arn>` на ARN вашего ключа KMS, а `<aws_region>` — на регион, в котором вы развернули кластер Kubernetes. Выполнение операции, инициированной предыдущей командой, может занять до 45 минут. Команда выполняется до тех пор, пока не будет создан кластер, поэтому просто подождите, пока она завершится. Как только все будет готово, мы сможем создать секреты Kubernetes.

Создадим секрет, который представляет собой строку подключения к базе данных. Она имеет такую структуру:

```
<engine>://<username>:<password>@<hostname>:<port>/<database_name>
```

Рассмотрим каждый компонент строки подключения:

- `engine` — движок базы данных, например `postgresql`;
- `username` — имя пользователя, которое мы выбрали ранее при создании базы данных;
- `password` — пароль, который мы выбрали при создании базы данных;
- `hostname` — имя хоста базы данных, которое мы получили в предыдущем разделе из DBCluster. Свойство `Endpoint` полезной нагрузки возвращается командой `aws rds create-db-cluster`;
- `port` — порт, на котором запущена база данных. Каждая база данных имеет свой порт по умолчанию, например 5432 для PostgreSQL и 3306 для MySQL;
- `database_name` — имя базы данных, к которой мы подключаемся. В PostgreSQL база данных по умолчанию называется `postgres`.

Например, для базы данных PostgreSQL типичная строка подключения выглядит так:

```
postgresql://username:password@localhost:5432/postgres
```

Чтобы сохранить строку подключения к базе данных в качестве секрета Kubernetes, мы запускаем следующую команду:

```
$ kubectl create secret generic -n orders-service db-credentials \
--from-literal=DB_URL=<connection_string>
```

Она создает секретный объект с именем `db-credentials` в пространстве имен `orders-service`. Чтобы получить подробную информацию об этом секретном объекте, вы можете выполнить такую команду:

```
$ kubectl get secret db-credentials -n orders-service -o json
# output:
{
  "apiVersion": "v1",
  "data": {
    "DB_URL": "cG9zdGdyZXNxbDovL3VzZXJ0YW11OnBhc3N3b3JkQGNvZmZlZW1lc2gtZG
```

```

    IuY2x1c3Rlci1jYn
➔ Y0YWhncC2JjZWcuZXUtd2VzdC0xLnJkcy5hbWF6b25hd3MuY29t0jU0MzIvcG9zdGdyZXM="
  },
  "kind": "Secret",
  "metadata": {
    "creationTimestamp": "2021-11-19T15:21:42Z",
    "name": "db-credentials",
    "namespace": "orders-service",
    "resourceVersion": "599258",
    "uid": "d2c210e7-c61c-46b7-9f43-9407766e147c"
  },
  "type": "Opaque"
}

```

Секреты перечислены в свойстве `data` полезной нагрузки, и они закодированы в Base64. Чтобы получить их значения, выполните команду:

```
$ echo <DB_URL> | base64 --decode
```

где `<DB_URL>` — значение ключа `DB_URL` в кодировке Base64.

Чтобы сделать секрет доступным для сервиса заказов, нам нужно обновить развертывание этого сервиса. Так мы сможем использовать секрет и предоставить его в качестве переменной окружения (листинг 14.4).

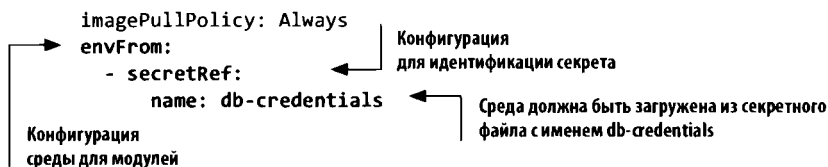
#### Листинг 14.4. Использование секретов в качестве переменных среды при развертывании

```
# file: orders-service-deployment.yaml
```

```

apiVersion: apps/v1
kind: Deployment
metadata:
  name: orders-service
  namespace: orders-service
  labels:
    app: orders-service
spec:
  replicas:
  selector:
    matchLabels:
      app: orders-service
  template:
    metadata:
      labels:
        app: orders-service
    spec:
      containers:
        - name: orders-service
          image:
➔ <aws_account_id>.dkr.ecr.<aws_region>.amazonaws.com/coffeemesh-orders:1.0
          ports:
            - containerPort: 8000

```



Применим изменения, выполнив следующую команду:

```
$ kubectl apply -f orders-service-deployment.yaml
```

Теперь наш сервис может подключаться к базе данных! Заключительным шагом будет применение миграции базы данных, чем мы и займемся в следующем подразделе.

### 14.7.3. Запуск миграции базы данных и подключение сервиса к базе данных

Наша база данных запущена, и теперь мы можем подключить к ней сервис заказов. Однако прежде, чем создавать записи и выполнять запросы, мы должны убедиться, что база данных содержит ожидаемые схемы. Как вы помните из главы 7, процесс создания схем базы данных называется *миграцией*. Миграции нашего приложения доступны в папке `migrations`. В этом подразделе мы выполним миграцию с использованием бессерверной базы данных `Aurora`.

Ранее мы развернули базу данных `Aurora` в наших частных подсетях, а это означает, что мы не можем получить прямой доступ к базе данных для запуска миграций. Возможны два основных варианта подключения к базе данных: подключиться через сервер-бастион или создать задание `Kubernetes`, которое использует миграции. Поскольку мы работаем с `Kubernetes` и наш кластер уже запущен, нам больше подойдет второй вариант.

#### ОПРЕДЕЛЕНИЕ

*Сервер-бастион* — это сервер, который позволяет установить защищенное соединение с частной сетью. Подключившись к серверу-бастиону, вы сможете получить доступ к другим серверам в частной сети.

Чтобы создать задание `Kubernetes`, сначала нужно создать образ `Docker` для запуска миграции базы данных. Создайте файл с именем `migrations.dockerfile` и скопируйте в него содержимое листинга 14.5. Этот `Dockerfile` устанавливает зависимости как для продакшена, так и для разработки и копирует их поверх миграции и конфигурации `Alembic` в контейнер. Как вы помните из главы 7, `Alembic` предназначен для управления миграциями баз данных. Команда для контейнера — `alembic upgrade`.

**Листинг 14.5.** Файл Dockerfile для задания миграции базы данных

```
# file: migrations.dockerfile

FROM python:3.9-slim

RUN mkdir -p /orders/orders

WORKDIR/orders

RUN pip install -U pip && pip install pipenv

COPY Pipfile Pipfile.lock /orders/

RUN pipenv install --dev --system --deploy

COPY orders/repository /orders/orders/repository/
COPY migrations /orders/migrations
COPY alembic.ini /orders/alembic.ini

ENV PYTHONPATH=/orders ← Устанавливаем переменную
                        окружения PYTHONPATH

CMD ["alembic", "upgrade", "heads"]
```

Чтобы создать образ Docker, выполните следующую команду:

```
$ docker build -t
➔ <aws_account_number>.dkr.ecr.<aws_region>.amazonaws.com/coffeemesh-
➔ orders-migrations:1.0 -f migrations.dockerfile .
```

Мы присвоим образу имя `coffeemesh-orders-migrations` и назовем его версией 1.0. Убедитесь, что вы заменили `<aws_account_id>` на свой ID учетной записи AWS, а `<aws_region>` — на регион, в котором хотите хранить свои Docker-образы. Прежде чем мы поместим образ в реестр контейнеров, нам нужно создать репозиторий:

```
$ aws ecr create-repository --repository-name coffeemesh-orders-migrations
```

Теперь поместим образ в реестр контейнеров:

```
$ docker push
➔ <aws_account_id>.dkr.ecr.<aws_region>.amazonaws.com/coffeemesh-orders-
➔ migrations:1.0
```

Если срок действия ваших учетных данных ECR истек, вы можете обновить их, повторно выполнив следующую команду:

```
$ aws ecr get-login-password --region <aws_region> | docker login \
--username AWS --password-stdin \
<aws_account_id>.dkr.ecr.<aws_region>.amazonaws.com
```

Теперь, когда наш образ готов, нужно создать объект задания Kubernetes. Для того воспользуемся файлом манифеста. Создайте файл с именем `orders-migrations-job.yaml`

и скопируйте в него содержимое листинга 14.6. В этом листинге определен объект Kubernetes типа `Job` с использованием API `batch/v1`. Точно так же, как мы делали в предыдущем подразделе для сервиса заказов, мы указываем строку подключения к базе данных в среде, загружая секретные данные `db-credentials`, используя свойство `envFrom` определения контейнера. Устанавливаем для параметра `ttlSecondsAfterFinished` значение 30 секунд — оно определяет, как долго под будет находиться в пространстве имен `orders-service` после завершения задания.

#### Листинг 14.6. Создание задания по миграции базы данных

```
# file: orders-migrations-job.yaml

apiVersion: batch/v1
kind: Job
metadata:
  name: orders-service-migrations
  namespace: orders-service
  labels:
    app: orders-service
spec:
  ttlSecondsAfterFinished: 30
  template:
    spec:
      containers:
        - name: orders-service-migrations
          image:
            ➔ <aws_account_id>.dkr.ecr.<aws_region>.amazonaws.com/coffeemesh-orders-
            ➔ migrations:1.0
          imagePullPolicy: Always
          envFrom:
            - secretRef:
                name: db-credentials
      restartPolicy: Never
```

Под должен быть удален  
через 30 секунд после завершения

Создадим задание, выполнив следующую команду:

```
$ kubectl apply -f orders-migrations-job.yaml
```

Под задания будет запущен в течение нескольких секунд. Можете проверить его состояние, выполнив следующую команду:

```
$ kubectl get pods -n orders-service
```

Как только под получит статус `Running` или `Completed`, вы можете проверить журналы задания, выполнив такую команду:

```
$ kubectl logs -f jobs/orders-service-migrations -n orders-service
```

Стоит просматривать журналы пода таким образом, чтобы проверять, как проходит процесс, и выявлять любые проблемы, возникающие при его выполнении. Поскольку задание миграции является временным и будет удалено после завершения,



убедитесь, что проверяете журналы во время работы процесса. Как только задание по миграции завершено, база данных готова к использованию! Наконец-то мы можем взаимодействовать с сервисом заказов — момент, которого мы так долго ждали!

В следующем разделе рассказывается, какое еще изменение необходимо внести для завершения развертывания.

## 14.8. ОБНОВЛЕНИЕ СПЕЦИФИКАЦИИ OPENAPI С УКАЗАНИЕМ ИМЕНИ ХОСТА ALB

Теперь, когда наш сервис готов, а база данных развернута и настроена, пришло время проверить работу приложения. В главах 2 и 6 вы научились взаимодействовать с API, используя Swagger UI. Чтобы использовать Swagger UI в развертывании, нужно обновить спецификацию API, указав имя хоста ALB кластера Kubernetes (листинг 14.7). В этом разделе мы обновим спецификацию API сервиса заказов, создадим новое развертывание и протестируем его.

**Листинг 14.7.** Добавление имени хоста ALB в качестве сервера

```
# file: oas.yaml

openapi: 3.0.0

info:
  title: Orders API
  description: API that allows you to manage orders for CoffeeMesh
  version: 1.0.0

servers:
  - url: <alb-hostname>
    description: ALB's hostname
  - url: https://coffeemesh.com
    description: main production server
  - url: https://coffeemesh-staging.com
    description: staging server for testing purposes only
  - url: http://localhost:8000
    description: URL for local testing
...
```

В листинге 14.7 замените `<alb-hostname>` именем хоста вашего собственного ALB. Как говорилось в разделе 14.6, имя хоста ALB можно получить, выполнив следующую команду:

```
$ kubectl get ingress/orders-service-ingress -n orders-service -o json | \
jq '.status.loadBalancer.ingress[0].hostname'
# output:
# "k8s-ordersse-orderssse-8cf837ce7a-1036161040.<aws_region>.elb.amazonaws.com"
```

Теперь нам нужно перестроить образ Docker:

```
$ docker build -t
<aws_account_number>.dkr.ecr.<aws_region>.amazonaws.com/coffeemesh-orders:1.1 .
```

Затем мы публикуем новый образ в AWS ECR:

```
$ docker push
<aws_account_number>.dkr.ecr.<aws_region>.amazonaws.com/coffeemesh-orders:1.1
```

Далее нужно обновить манифест развертывания сервиса заказов (листинг 14.8).

#### **Листинг 14.8.** Объявление манифеста развертывания

```
# file: orders-service-deployment.yaml
```

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: orders-service
  namespace: orders-service
  labels:
    app: orders-service
spec:
  replicas: 1
  selector:
    matchLabels:
      app: orders-service
  template:
    metadata:
      labels:
        app: orders-service
    spec:
      containers:
        - name: orders-service
          image:
➔ <aws_account_id>.dkr.ecr.<aws_region>.amazonaws.com/coffeemesh-orders:1.1
          ports:
            - containerPort: 8000
          imagePullPolicy: Always
```

Наконец, применим новую конфигурацию развертывания, выполнив следующую команду:

```
$ kubectl apply -f orders-service-deployment.yaml
```

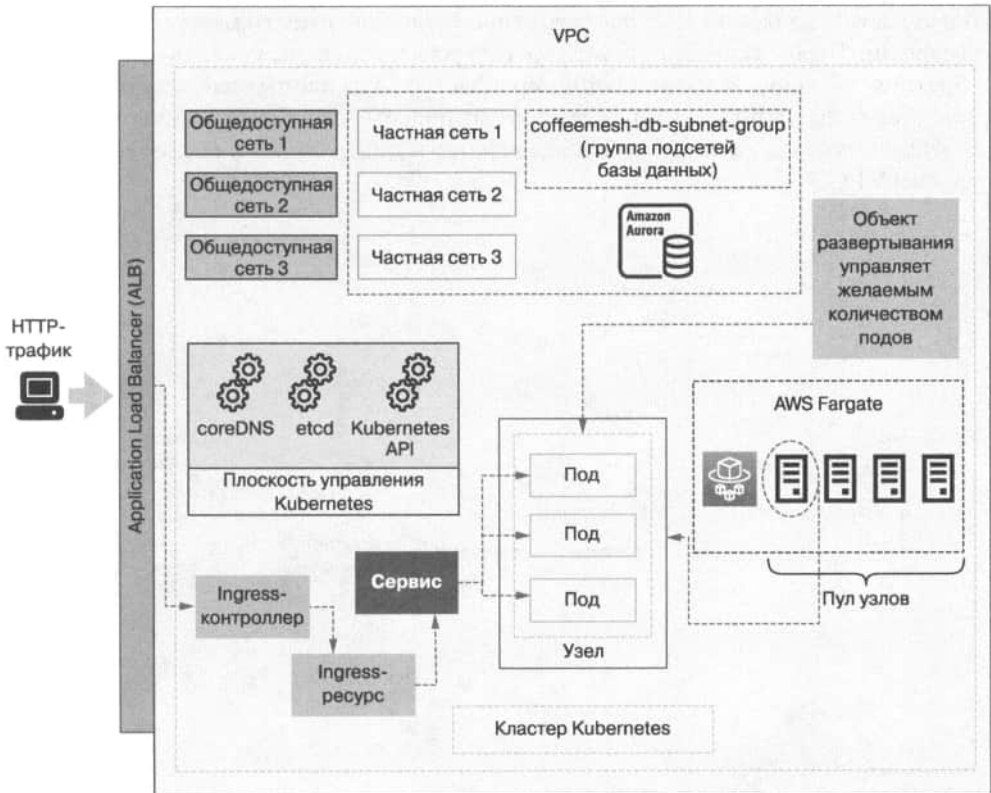
Следить за развертыванием можно, выполнив такую команду:

```
kubectl get pods -n orders-service
```

Как только старый под завершит работу, а новый запустится, загрузите пользовательский интерфейс сервиса заказов, вставив имя хоста ALB в браузер и посетив страницу `/docs/orders`. Вы можете поэкспериментировать с API, используя подход, описанный в главах 2 и 6: создание заказов, их изменение и извлечение их данных с сервера.

Наше путешествие наконец завершено! Если вы смогли довести дело до конца и вам удалось запустить свой кластер Kubernetes, примите мои самые искренние поздравления! Вы сделали это!

На рис. 14.12 приведен обзор архитектуры, которую мы развернули в этой главе.



**Рис. 14.12.** Краткий обзор архитектуры, развернутой в этой главе

В этой главе приведен краткий обзор Kubernetes, но этого достаточно, чтобы получить представление о том, как работает Kubernetes, и чтобы запустить кластер в производственной среде. Если вы работаете или намерены работать с Kubernetes, я настоятельно рекомендую вам продолжить изучение технологии. Можете воспользоваться теми ресурсами, которые я привел в этой главе, а также прочтите фундаментальную книгу «Kubernetes в действии»<sup>1</sup>.

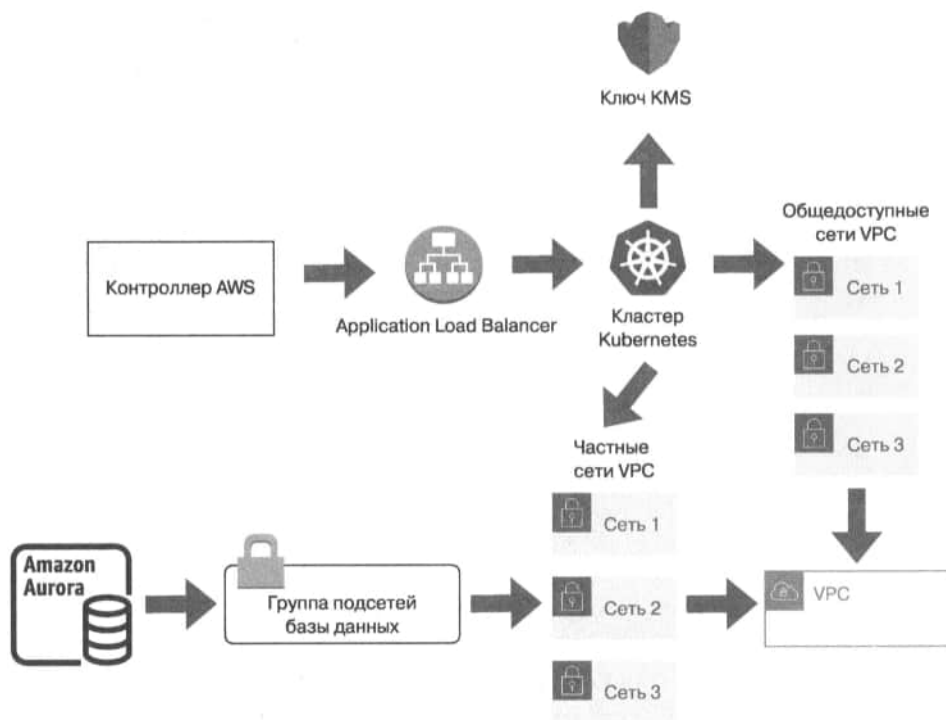
В следующем разделе мы удалим все ресурсы, которые создали в этой главе. Не пропускайте ее, если не хотите платить больше, чем положено!

<sup>1</sup> *Лукуша М. Kubernetes в действии.*

## 14.9. УДАЛЕНИЕ КЛАСТЕРА KUBERNETES

В этом разделе объясняется, как удалить все ресурсы, которые мы создали ранее в главе. Это крайне важный шаг — так вы сможете быть уверены, что вам не выставят счет за кластер Kubernetes после того, как вы закончите работу с примерами.

Как показано на рис. 14.13, у нас есть отношения зависимости между некоторыми ресурсами. Чтобы успешно удалить все ресурсы, мы должны удалить их в порядке, обратном их зависимостям. Например, кластер базы данных зависит от группы подсетей базы данных, которая зависит от подсетей VPC, а те зависят от VPC. В этом случае мы начнем с удаления кластера базы данных, а последним делом удалим VPC.



**Рис. 14.13.** Ресурсы в нашем стеке имеют отношения зависимости. Направление зависимости указано стрелками. Удаляя ресурсы, начинаем с тех, на которые стрелки не указывают

Удалим кластер базы данных:

```
$ aws rds delete-db-cluster --db-cluster-identifier coffeemesh-db \
--skip-final-snapshot
```

Флаг `--skip-final-snapshot` указывает команде не создавать моментальный снимок базы данных перед удалением. Удаление базы данных занимает несколько минут.

Как только он будет удален, можем удалить группу подсетей базы данных:

```
$ aws rds delete-db-subnet-group --db-subnet-group-name \
coffeemesh-db-subnet-group
```

Далее удалим AWS Load Balancer Controller. Это двухэтапный процесс: сначала мы удаляем контроллер с помощью `helm`, а затем удаляем ALB, который был создан при установке контроллера. Чтобы удалить ALB, нам нужен его URL-адрес, поэтому сначала извлечем это значение (убедитесь, что выполнили этот шаг перед удалением с помощью `helm`):

```
$ kubectl get ingress/orders-service-ingress -n orders-service -o json | \
jq '.status.loadBalancer.ingress[0].hostname'
# output: "k8s-ordersse-ordersse-8cf837ce7a-
➔ 1036161040.<aws_region>.elb.amazonaws.com"
```

Теперь удалим контроллер:

```
$ helm uninstall aws-load-balancer-controller -n kube-system
```

После выполнения этой команды нам нужно удалить ALB, а для этого следует найти его ARN. Воспользуемся AWS CLI, чтобы перечислить балансировщики нагрузки в нашей учетной записи и отфильтровать их по DNS-именам. Следующая команда извлекает ARN балансировщика нагрузки, DNS-имя которого совпадает с URL-адресом ALB, полученным ранее:

```
$ aws elbv2 describe-load-balancers | jq '.LoadBalancers[] | \
select(.DNSName == "<load_balancer_url>") | .LoadBalancerArn'
# output: "arn:aws:elasticloadbalancing:<aws_region>:<aws_account_id>:
➔ loadbalancer/app/k8s-ordersse-ordersse-8cf837ce7a/cf708f97c2485719"
```

Убедитесь, что заменили `<load_balancer_url>` URL-адресом балансировщика нагрузки, который получили на предыдущем шаге. Эта команда дает нам ARN балансировщика нагрузки, который мы можем использовать для его удаления:

```
$ aws elbv2 delete-load-balancer --load-balancer-arn "<load_balancer_arn>"
```

Теперь мы можем удалить кластер Kubernetes:

```
$ eksctl delete cluster coffeemesh
```

Наконец, удалим ключ KMS, который создали ранее для шифрования наших секретов Kubernetes. Для этого выполняем следующую команду:

```
$ aws kms schedule-key-deletion --key-id <key_id>
```

где `<key_id>` — это ID ключа, который мы создали ранее.

## РЕЗЮМЕ

- Kubernetes — это инструмент для оркестрации контейнеров, который уже стал общепринятым при масштабном развертывании микросервисов. Использование Kubernetes помогает перемещаться между облачными провайдерами, сохраняя при этом единообразный интерфейс сервисов.
- Тремя основными управляемыми сервисами Kubernetes являются Google Kubernetes Engine (GKE), Azure Kubernetes Service (AKS) и AWS Elastic Kubernetes Service (EKS). В этой главе вы научились развертывать кластер Kubernetes с помощью EKS, который является наиболее популярным управляемым сервисом Kubernetes.
- Мы можем развернуть кластер Kubernetes в AWS с помощью консоли, CloudFormation или инструмента командной строки eksctl. В этой главе мы использовали eksctl CLI, поскольку это рекомендуемый AWS способ управления кластером Kubernetes.
- Чтобы сделать кластер Kubernetes доступным из Интернета, мы используем ingress-контроллер, такой как AWS Load Balancer Controller.
- Чтобы развернуть микросервис в кластере Kubernetes, мы создаем следующие ресурсы:
  - **Deployment**, которое управляет желаемым состоянием подов, процессов, запускающих Docker-образ;
  - **Service**, который позволяет нам предоставлять приложение в качестве веб-сервиса;
  - **Ingress** — объект, привязанный к ingress-контроллеру (AWS Load Balancer Controller), который перенаправляет трафик в сервис.
- Aurora Serverless — мощный движок базы данных. Это отличный выбор при работе с микросервисами. С Aurora Serverless вы платите только за то, чем пользуетесь, и вам не нужно беспокоиться о масштабировании базы данных, тем самым сокращая свои затраты и время на управление ею.
- Чтобы безопасно передавать конфиденциальные сведения о конфигурации приложениям в Kubernetes, мы используем секреты Kubernetes. С EKS возможны две стратегии безопасного управления секретами:
  - использование AWS Secrets & Configuration Provider для Kubernetes;
  - использование секретов Kubernetes в сочетании с сервисом AWS Key Managed Service.

## *Приложения*

# *Типы веб-API и протоколов*

---

В этом приложении мы рассмотрим протоколы API, которые можно использовать для реализации интерфейсов приложений. Каждый из них разрабатывался для решения конкретных проблем интеграции между потребителями и производителями API. Мы обсудим преимущества и ограничения каждого протокола, чтобы вы могли сделать наилучший выбор при проектировании и создании собственных API. Разберем следующие протоколы:

- RPC и его варианты, JSON-RPC и XML-RPC;
- SOAP;
- gRPC;
- REST;
- GraphQL.

Выбор правильного типа API имеет основополагающее значение для производительности и стратегии интеграции микросервисов. Вот какие факторы определяют выбор протокола API:

- является ли API публичным или приватным;
- тип потребителя API: небольшие устройства, мобильные приложения, браузеры или другие микросервисы;

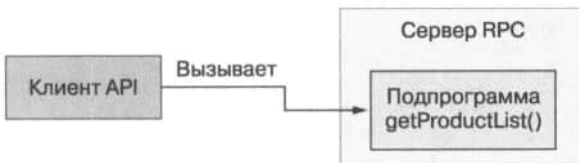


- возможности и ресурсы, которые мы хотим предоставить; например, мы разрабатываем иерархическую модель данных, которая может быть организована вокруг эндпоинтов, или сильно взаимосвязанную сеть ресурсов с перекрестными ссылками.

Мы будем принимать во внимание эти факторы в следующих разделах, при обсуждении преимуществ и ограничений каждого протокола, чтобы оценить его пригодность для различных сценариев.

## A.1. РАССВЕТ API: RPC, XML-RPC И JSON-RPC

Начнем с рассмотрения удаленного вызова процедуры и двух наиболее распространенных реализаций, а именно XML-RPC и JSON-RPC. Как вы можете видеть на рис. А.1, *удаленный вызов процедуры* (remote procedure call, RPC) — это протокол, который позволяет клиенту вызывать процедуру или подпрограмму на другом компьютере. Эта форма коммуникации возникла в 1980-е годы, с появлением распределенных вычислительных систем, и со временем стала стандартной реализацией<sup>1</sup>.



**Рис. А.1.** Используя RPC, программа вызывает функцию или подпрограмму с сервера API

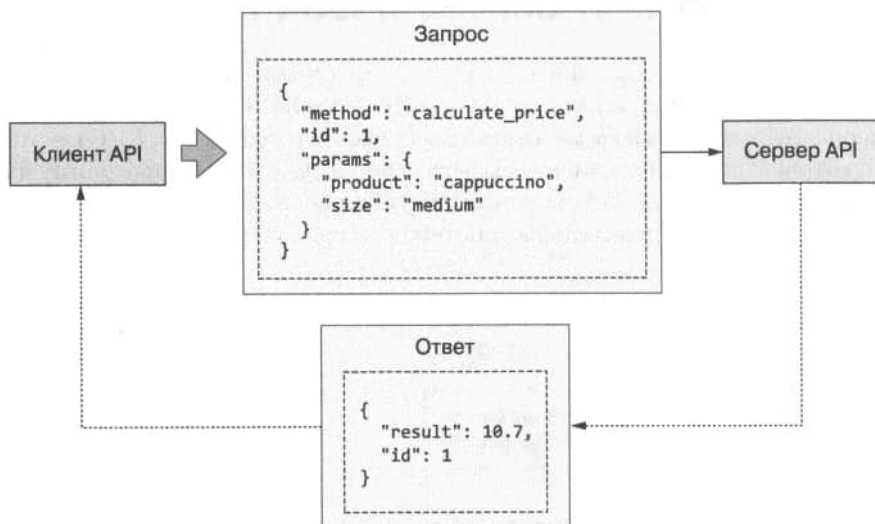
XML-RPC — это протокол RPC, который использует расширяемый язык разметки (Extensible Markup Language, XML) через протокол HTTP для обмена данными между клиентом и сервером. Был создан Дэйвом Уайнером в 1998 году и в конечном счете превратился в то, что позже стало известно как SOAP (см. раздел А.2).

С ростом популярности JavaScript Object Notation (JSON) в качестве формата сериализации данных стала использоваться альтернативная реализация RPC в виде

<sup>1</sup> Считается, что впервые термин «удаленный вызов процедуры» был использован Брюсом Джемом Нельсоном в его докторской диссертации (технический отчет CSL-81-9, исследовательский центр Xerox Palo Alto, Пало-Альто, Калифорния, 1981). Более формальное описание требований к реализации RPC вы найдете в статье: Birrell A. B., Nelson B. J. Implementing Remote Procedure Calls («Реализация удаленных вызовов процедур») // ACM Transactions on Computer Systems, 1984. Том 2. — № 1. — С. 39–59.

JSON-RPC. Он был представлен в 2005 году. JSON-RPC предлагает упрощенный способ обмена данными между клиентом и сервером API. Как вы можете видеть на рис. А.2, полезная нагрузка JSON-RPC обычно содержит три свойства:

- **method** — метод или функция, которые клиент желает вызвать в удаленном сервисе;
- **params** — параметры, которые должны быть переданы методу или функции при вызове;
- **id** — значение для идентификации запроса.



**Рис. А.2.** Используя JSON-RPC, клиент API отправляет запрос на сервер API, вызывая функцию `calculate_price()`, чтобы узнать цену средней чашки капучино. Сервер выдает результат запроса: \$10,70

В свою очередь, полезная нагрузка ответов JSON-RPC включает следующие параметры:

- **result** — значение, возвращаемое вызванным методом или функцией;
- **error** — код ошибки, возникшей во время вызова, если таковой имеется;
- **id** — ID обрабатываемого запроса.

RPC — это облегченный протокол, который позволяет управлять интеграцией с помощью API без необходимости реализации сложных интерфейсов. Клиенту RPC нужно знать только имя функции, которую он должен вызвать на удаленном сервере, и ее подпись. Ему не нужно искать разные эндпоинты и соответствовать

их схемам, как в REST. Однако отсутствие надлежащего интерфейса между потребителем и производителем API неизбежно приведет к созданию сильной связанности между клиентом и деталями реализации сервера. Как следствие, небольшое изменение в деталях реализации может нарушить интеграцию. По этой причине RPC рекомендуется в основном для внутренней интеграции с помощью API, где вы полностью контролируете как клиент, так и сервер.

## A.2. SOAP И ПОЯВЛЕНИЕ СТАНДАРТОВ API

В этом разделе обсуждается протокол простого объектного доступа (Simple Object Access Protocol, SOAP). SOAP обеспечивает взаимодействие с веб-сервисами путем обмена полезными данными в формате XML. Он был представлен в 1998 году Дэйвом Уайнером, Доном Боксом, Бобом Аткинсоном и Мохсеном Аль-Госейном для Microsoft и после ряда итераций в 2003 году стал стандартным протоколом для веб-приложений. SOAP был задуман как протокол обмена сообщениями и работает поверх уровня передачи данных (data transport layer), такого как HTTP.

SOAP был разработан для достижения трех основных целей:

- *расширяемости (extensibility)* — SOAP может быть расширен за счет возможностей, имеющихся в других системах обмена сообщениями;
- *нейтральности (neutrality)* — может работать по любому выбранному протоколу передачи данных, включая HTTP, или непосредственно по TCP или UDP;
- *автономности (independence)* — обеспечивает взаимодействие между веб-приложениями независимо от их моделей разработки.

Полезные данные, которыми обменивается эндпоинт SOAP, представлены в формате XML и включают следующие свойства (рис. А.3):

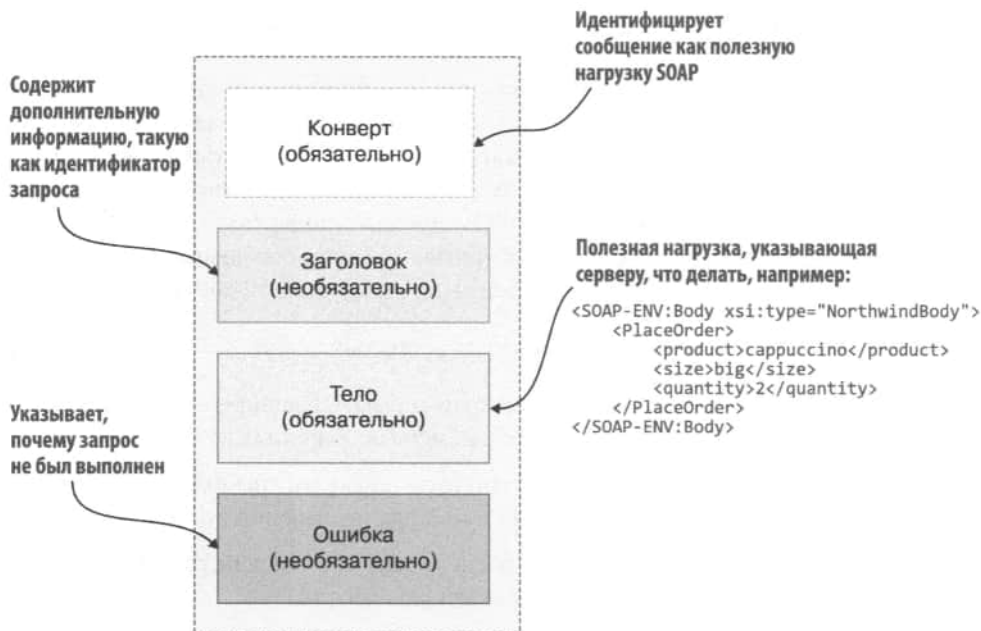
- **Envelope** (конверт, обязательно) — идентифицирует XML-документ как полезную нагрузку SOAP;
- **Header** (заголовок, необязательно) — включает дополнительную информацию о данных, содержащихся в сообщении, например тип кодировки;
- **Body** (тело, обязательно) — содержит полезную нагрузку (фактическое сообщение для обмена) запроса/ответа;
- **Fault** (ошибка, необязательно) — содержит ошибки, возникшие при обработке запроса.

SOAP стал важным вкладом в сферу разработки API. Доступность стандартного протокола для обмена данными между веб-приложениями привела к появлению

поставщиков (vendor) API. Внезапно стало возможным продавать цифровые сервисы, просто предоставив API, который каждый мог понять и использовать.

В последние годы SOAP был вытеснен более новыми протоколами и архитектурами. Снижению использования SOAP способствовали следующие причины.

- Полезные данные, которыми обмениваются через SOAP, содержат объемные XML-документы, требующие большой пропускной способности.



**Рис. А.3.** В верхней части SOAP-сообщения мы видим раздел Envelope, который сообщает нам, что это полезная нагрузка SOAP. Необязательный раздел Header содержит метаданные о сообщении, такие как тип кодировки. Раздел Body включает фактическую полезную нагрузку сообщения: данные, которыми обмениваются клиент и сервер. Наконец, раздел Fault содержит подробную информацию о любых ошибках, возникших при обработке полезной нагрузки

- XML сложен для чтения и поддержки и требует тщательного парсинга, что делает обмен сообщениями, структурированными в XML, менее удобным.
- SOAP не предоставляет четкой структуры для организации данных и не позволяет реализовать все возможности API. Он обеспечивает способ обмена сообщениями, а агенты, задействованные по обе стороны API, должны решать, как понимать такие сообщения.

### А.3. RPC С НОВА НАНОСИТ УДАР: БЫСТРЫЙ ОБМЕН ДАННЫМИ ЧЕРЕЗ gRPC

В этом разделе обсуждается конкретная реализация протокола RPC под названием gRPC<sup>1</sup>, который был разработан Google в 2015 году. Этот протокол использует HTTP/2 в качестве транспортного слоя и обменивается полезной нагрузкой, закодированной с помощью Protocol Buffers (Protobuf) — метода сериализации структурированных данных. Как я объяснял в главе 2, сериализация — это процесс преобразования данных в формат, который можно сохранить или передать по сети. Другой процесс должен иметь возможность извлекать сохраненные данные и восстанавливать их в исходном формате. Процесс восстановления сериализованных данных также известен как *демаршаллинг*.

Одни методы сериализации зависят от языка, например pickle для Python. Другие, такие как популярный формат JSON, не зависят от языка и могут быть переведены в собственные структуры данных других языков.

Очевидным недостатком JSON является то, что он допускает сериализацию только простых представлений данных, состоящих из строк, логических значений, массивов, ассоциативных массивов и нулевых значений. Поскольку JSON не зависит от языка и должен быть строго переносим между языками и средами, он не допускает сериализацию специфичных для языка функций, таких как NaN в JavaScript, кортежи или множества в Python или классы в объектно-ориентированных языках.

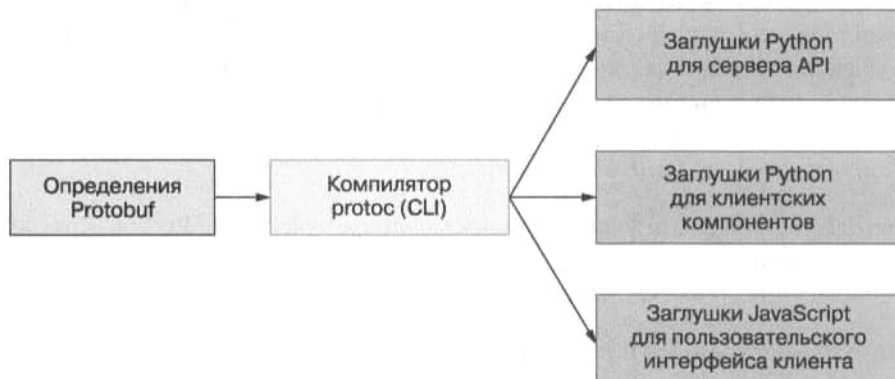
Формат pickle в Python позволяет сериализовать любой тип структуры данных, выполняемой в ваших программах на Python, включая кастомные объекты. Недостатком, однако, является то, что сериализованные данные очень специфичны для версии Python. Из-за небольших изменений во внутренней реализации Python между различными версиями нет гарантий, что другой процесс сможет надежно парсировать выбранный файл.

Protobuf находится где-то посередине: он позволяет определять более сложные структуры данных, чем JSON, включая перечисления, и способен генерировать

---

<sup>1</sup> Вам наверняка интересно, что означает буква g в gRPC. Согласно официальной документации, в каждом выпуске оно обозначает разное слово. Например, в версии 1.1 оно означает «хороший», в то время как в версии 1.2 — «зеленый» и т. д. ([https://grpc.github.io/grpc/core/md\\_doc\\_g\\_stands\\_for.html](https://grpc.github.io/grpc/core/md_doc_g_stands_for.html)). Некоторые люди считают, что буква g означает Google, поскольку этот протокол был изобретен компанией Google (см. статью: Is gRPC the Future of Client-Server Communication? («Станет ли gRPC будущим клиент-серверной коммуникации?») // Bleeding Edge Press, Medium, 19 июля 2018 года. <https://medium.com/@EdgePress/is-grpc-the-future-of-client-server-communication-b112acf9f365>).

собственные классы из сериализованных данных, которые можно расширить, чтобы добавить пользовательскую функциональность. Как вы можете видеть на рис. А.4, в gRPC сначала следует определить схему для структур данных, которыми вы хотите обмениваться через API, используя формат спецификации Protobuf, а затем использовать Protobuf CLI для автоматической генерации кода как для клиента, так и для сервера API.



**Рис. А.4.** gRPC использует Protobuf для кодирования данных, которыми обмениваются через API. Используя protoc CLI, мы можем генерировать код заглушки как для клиента, так и для сервера на основе спецификации Protobuf

Структуры данных, сгенерированные на основе спецификаций Protobuf, называются *заглушками* (*stubs*). Заглушки реализованы в коде на том языке, который мы используем для создания клиента и сервера API. Как вы можете видеть на рис. А.5, заглушки выполняют парсинг и проверку данных, которыми обмениваются клиент и сервер.



**Рис. А.5.** Заглушки, сгенерированные с помощью Protobuf, выполняют парсинг полезных данных, которыми обмениваются клиент и сервер API, и переводят их в нативный код

gRPC предлагает более надежный подход к интеграции с помощью API, чем обычный RPC. Использование Protobuf служит механизмом принудительного исполнения, гарантирующим, что данные, которыми обмениваются клиент и сервер, поступают в ожидаемом формате. Это также помогает обеспечить высокую оптимизацию взаимодействия через API, поскольку обмен данными осуществляется непосредственно в двоичном формате. По этой причине gRPC — идеальный кандидат для реализации внутренних интеграций с помощью API, где производительность является важным фактором<sup>1</sup>.

## A.4. API HTTP И REST

В этом разделе объясняется передача репрезентативного состояния (REST) и ее основные особенности. REST — это архитектурный стиль для проектирования веб-сервисов и их интерфейсов. Как вы видели в главе 4, REST API структурированы вокруг ресурсов. Различают два типа ресурсов: коллекции и синглтоны, и для их представления используются разные URL-адреса. Например, на рис. A.6 `/orders` представляет собой набор заказов, в то время как `/orders/{order_id}` представляет URI отдельного заказа. Мы используем `/orders` для получения списка заказов и размещения новых заказов и `/orders/{order_id}` — для выполнения действий с одним заказом.

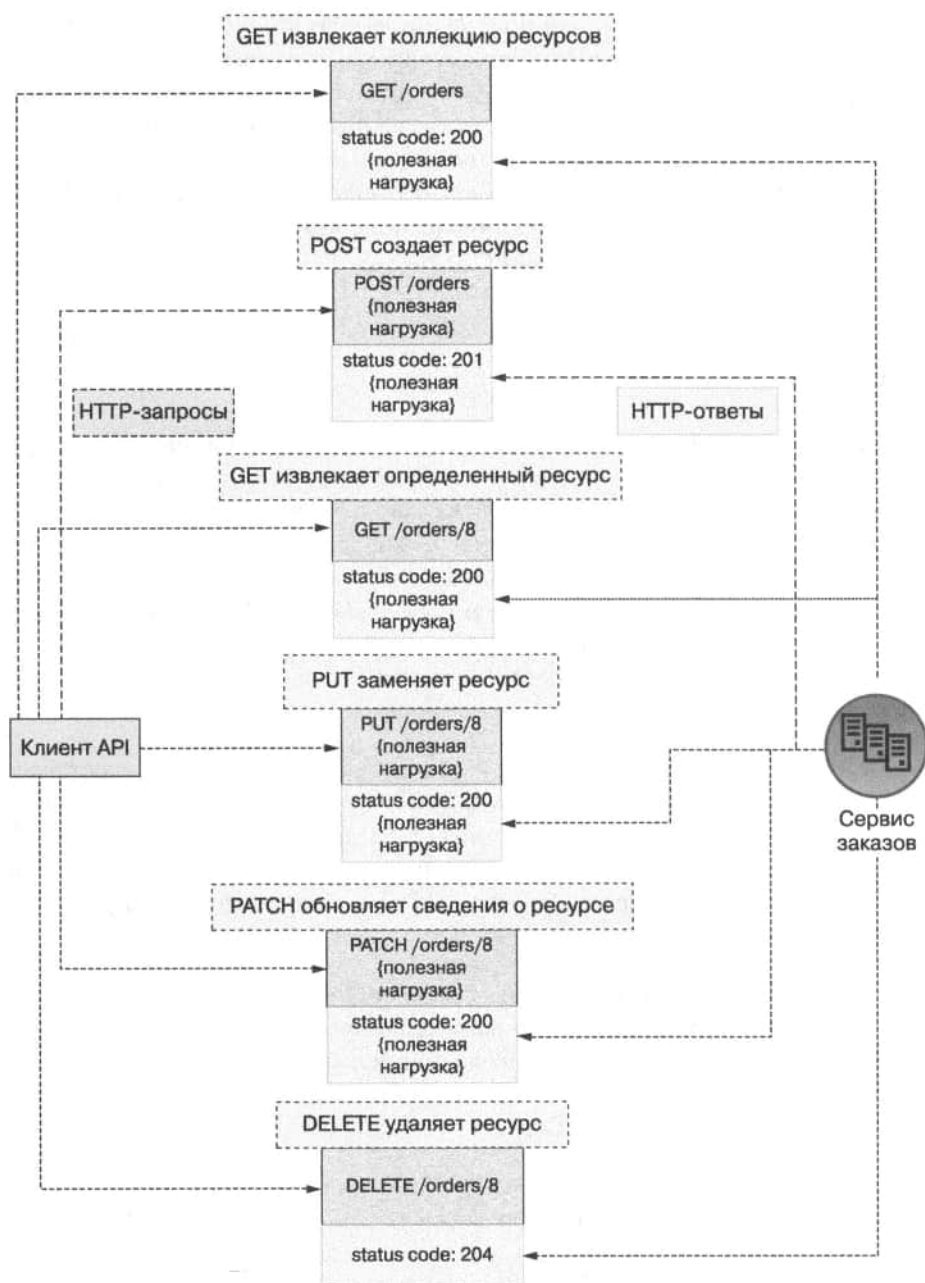
Хороший REST API использует возможности протокола HTTP для предоставления выразительных API. Например, методы HTTP применяются для определения эндпоинтов API и выражения их намерений (POST для создания ресурсов и GET для извлечения ресурсов); статус-коды HTTP используются для описания результата обработки запроса; а полезная нагрузка HTTP — для обмена данными между клиентом и сервером.

Мы документируем REST API, используя стандарт OpenAPI, который был создан в 2010 году Тони Тэмом и изначально назывался Swagger API. Проект набирал популярность, и в 2015 году была запущена инициатива OpenAPI для поддержания спецификации. В 2016 году спецификация была официально выпущена под названием OpenAPI Specification (OAS).

Данные, которыми обмениваются через REST API, передаются в теле запроса/ответа HTTP. Они могут быть закодированы в любом формате, который предпочитает производитель API, но чаще всего используется JSON.

---

<sup>1</sup> Согласно отчету Postman о состоянии API за 2022 год, gRPC используют 11 % опрошенных разработчиков (<https://www.postman.com/state-of-api/api-technologies/#api-technologies>).



**Рис. А.6.** REST API структурированы вокруг эндпоинтов. Мы различаем одноэлементные эндпоинты, такие как `GET /orders/8`, и эндпоинты коллекций, такие как `GET /orders`. Ответы REST API включают статус-коды HTTP, которые описывают результат обработки запроса



Благодаря возможности создания документации по API с высоким уровнем детализации в формате стандартной спецификации REST стал идеальным кандидатом для интеграции с помощью корпоративных API и создания публичных API с большим и разнообразным кругом потребителей.

## A.5. ДЕТАЛИЗИРОВАННЫЕ ЗАПРОСЫ С ИСПОЛЬЗОВАНИЕМ GRAPHQL

В этом разделе мы рассмотрим GraphQL и сравним его с REST. GraphQL — это язык запросов, основанный на графах и узлах. На момент написания книги это один из самых популярных вариантов реализации веб-API<sup>1</sup>. Он был разработан в 2012 году и стал открытым в 2015-м.

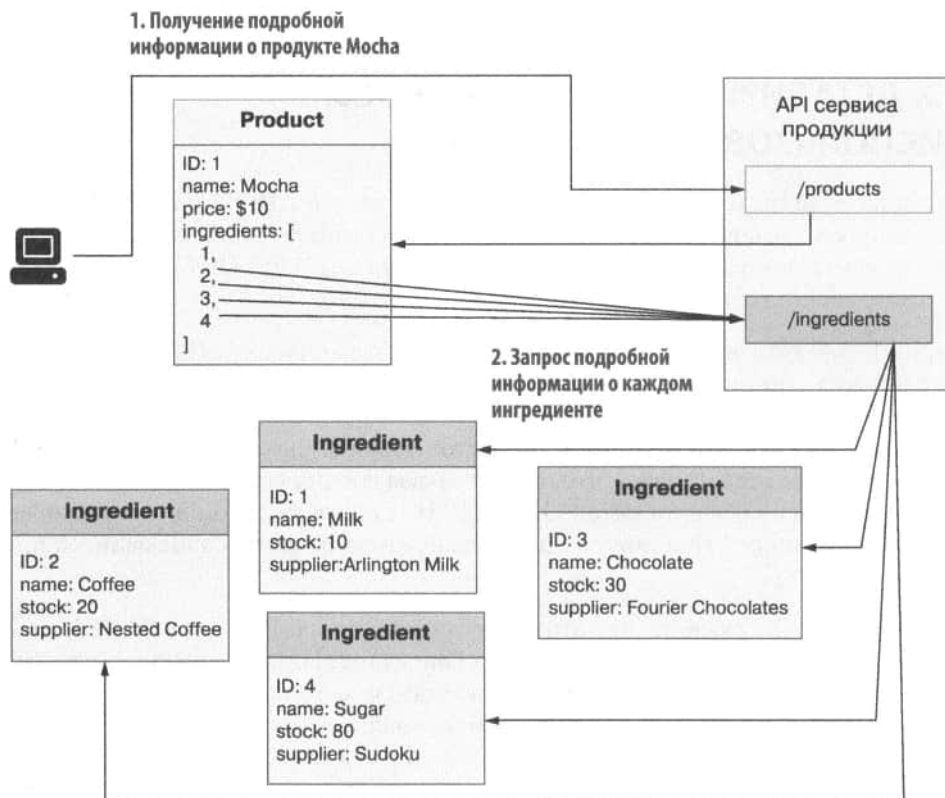
GraphQL создавался для устранения некоторых ограничений REST API, таких как сложность представления определенных операций через эндпоинты HTTP. Допустим, вы заказали чашку кофе через сайт CoffeeMesh, а позже передумали и решили отменить заказ. Какой HTTP-метод подойдет для представления этого действия? Вы можете сказать, что отмена заказа сродни его удалению, поэтому вы могли бы использовать метод DELETE. Но действительно ли отмена то же самое, что удаление? Планируете ли вы вычеркнуть заказ из своих записей после отмены?

Вероятно, нет. Вы скажете, что это должен быть запрос PUT или PATCH, поскольку вы меняете состояние заказа на `cancelled`. Или вы считаете, что это должен быть запрос POST, поскольку пользователь запускает операцию, которая включает в себя нечто большее, чем просто обновление записи. Как бы вы на это ни смотрели, HTTP действительно накладывает некоторые ограничения, когда дело доходит до моделирования действий пользователя, и GraphQL обходит эту проблему, не ограничивая себя исключительно использованием элементов протокола HTTP.

Другим ограничением REST является невозможность для клиентов выполнять детализированные запросы данных, технически известные как *избыточная выборка* (*overfetching*). Например, API предоставляет доступ к ресурсам `/products` и `/ingredients`. Как вы можете видеть на рис. A.7, с помощью `/products` мы можем получить список товаров, включая ID их ингредиентов. Однако если мы хотим получить название каждого ингредиента, то должны запросить подробную информацию о нем в API `/ingredients`. В результате клиенту необходимо отправлять различные запросы в API, чтобы получить простое представление товара. Кроме того, клиент API получает больше информации, чем ему нужно: в каждом запросе к API `/ingredients` он получает полное описание ингредиента, а ему нужно только

<sup>1</sup> Согласно тому же отчету, GraphQL используют 28 % опрошенных разработчиков (<https://www.postman.com/state-of-api/api-technologies/#api-technologies>).

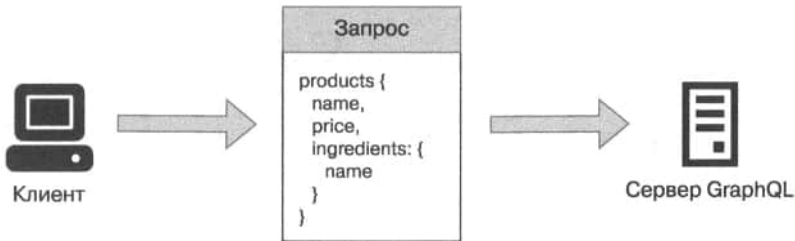
название. Избыточная выборка становится проблемой для небольших устройств, таких как мобильные телефоны, которые не всегда способны обрабатывать и хранить большие объемы данных и могут иметь более ограниченный доступ к сети.



**Рис. А.7.** Ограничением REST API является неспособность клиентов API выполнять детализированные запросы данных, иначе известные как избыточная выборка. Здесь эндпоинт `/products` возвращает список товаров с идентификаторами их ингредиентов. Чтобы получить названия ингредиентов, клиент должен запросить подробную информацию о каждом в эндпоинте `/ingredients`. В результате клиент API делает слишком много запросов к серверу и получает больше данных, чем нужно

GraphQL дает возможность избежать этих проблем, позволяя клиентам выполнять детализированные запросы на сервере. С помощью GraphQL мы можем создавать связи между различными моделями данных, и клиенты API будут извлекать данные из связанных объектов. Например, на рис. А.8 клиент API может запросить список товаров и названия их ингредиентов в одном запросе. Позволяя клиентам извлекать необходимые данные с сервера одним запросом, GraphQL становится идеальным кандидатом для API, которые используются клиентами с ограниченным доступом к сети или ограниченными возможностями хранения. GraphQL также

станет хорошим выбором для API с сильно взаимосвязанными ресурсами, в которых пользователи, скорее всего, будут извлекать данные из связанных объектов, таких как товары и ингредиенты (рис. А.8).



**Рис. А.8.** Используя GraphQL API, мы можем запрашивать данные из связанных объектов, таких как товары и ингредиенты. Здесь клиент API запрашивает список товаров с названиями их ингредиентов

Надо отметить, что GraphQL также имеет ограничения. Самое главное — он не обеспечивает большой поддержки кастомных скалярных типов. GraphQL поставляется с базовым набором встроенных скаляров, таких как целое число и строка. GraphQL позволяет объявлять кастомные скаляры, но нельзя документировать их форму или то, как они проверяются с помощью SDL. Согласно официальной документации GraphQL, «Реализация должна определить, как этот тип должен быть сериализован, десериализован и проверен» (<https://graphql.org/learn/schema/>). Поскольку одним из краеугольных камней надежной интеграции с помощью API является отличная документация, GraphQL оказывается сложным выбором для публичных API, которые должны надежно использоваться внешними клиентами.

Другое ограничение GraphQL заключается в том, что все запросы обычно выполняются с помощью POST-запросов, что затрудняет кэширование ответов. По моему опыту, большинству разработчиков также сложнее взаимодействовать с GraphQL API. Фактически отчет Postman о состоянии API за 2022 год показал, что только 28 % опрошенных разработчиков используют GraphQL и до 14 % даже не слышали о нем. В то время как взаимодействие с REST API может быть весьма простым, с GraphQL вы должны знать, как создавать документы запроса и как отправлять их на сервер. Поскольку разработчики не очень хорошо знакомы с GraphQL, вы можете снизить вероятность использования ваших API, выбрав эту технологию.

# Управление жизненным циклом API

---

API очень редко бывают статичными. По мере развития вашего продукта вам придется предоставлять через API новые возможности и функции, а это означает, что нужно будет создавать новые эндпоинты или изменять схемы, чтобы вводить новые сущности или поля. Часто изменения API не имеют обратной совместимости, то есть запросы клиентов, которые не знают о новых изменениях, не будут выполнены. Поддержание работы API подразумевает в том числе обеспечение того, чтобы любые вносимые вами изменения не нарушали уже существующую интеграцию с другими приложениями, и этой цели служит управление версиями API. В этом приложении мы рассмотрим стратегии управления версиями API для контроля за изменениями интерфейсов.

В какой-то момент API может оказаться ненужным. Возможно, вы переносите REST API на GraphQL или вообще прекращаете работу с продуктом. Если вы планируете отказаться от использования API, то должны сообщить своим клиентам, когда и как это произойдет, и во втором разделе приложения я расскажу вам, как передавать эту информацию своим пользователям.

## Б.1. СТРАТЕГИИ УПРАВЛЕНИЯ ВЕРСИЯМИ ДЛЯ РАЗВИВАЮЩИХСЯ API

Для управления изменениями API используют два основных типа систем версионирования.

- *Семантическое управление версиями (SemVer, <https://semver.org/>)* — это наиболее распространенный тип управления версиями, он широко используется для управления релизами программного обеспечения. SemVer имеет следующий формат: MAJOR.MINOR.PATCH, например 1.1.0. Первая цифра указывает на основную версию релиза, вторая цифра — на второстепенную версию, а третья цифра — на версию исправления.

Основная версия (MAJOR) меняется всякий раз, когда вы вносите критические изменения в API, например, когда добавляется новое поле в полезной нагрузке запроса. Второстепенные версии (MINOR) соответствуют изменениям, не нарушающим обратную совместимость. Это, например, введение нового необязательного параметра запроса. Пользователи вашего API продолжают вызывать эндпоинты одним и тем же способом в разных второстепенных версиях и получать ответы. Версии исправлений (PATCH) указывают на исправления ошибок.

В контексте API обычно используют только основную версию, поэтому у нас могут быть версии v1 и v2. Незначительные изменения и исправления, улучшающие API, как правило, можно внедрить без нарушения существующих интеграций.

- *Календарное управление версиями (CalVer, <https://calver.org/>)* — вариант, когда для описания новых версий используются календарные даты. Подходит, если ваши API очень часто меняются или если ваши релизы зависят от времени. Все больше программных продуктов применяют календарное управление версиями (например, Ubuntu (<https://ubuntu.com/>)). AWS также использует календарное управление версиями в таких продуктах, как CloudFormation (<http://mng.bz/epQZ>) и S3 API (<http://mng.bz/p6B0>).

CalVer не предоставляет полной спецификации о том, как форматировать версии; подразумевается только использование дат. Можно указывать версию в формате YYYY.MM.DD или YY.MM. Если вы выпускаете несколько релизов в день, можете применять дополнительный счетчик для отслеживания каждого релиза. Например, 2022.12.01.3 означает, что это третий релиз, выпущенный 12 декабря 2022 года. (Более подробную информацию о календарном управлении версиями смотрите в статье <http://mng.bz/O6MO>.)

Какой вариант системы версионирования лучше? Это зависит от ваших потребностей и общей стратегии управления API. Чаще используется SemVer, поскольку он более понятен. Однако, если внедрение вашего продукта зависит от времени, лучше подойдет CalVer. На выбор системы управления версиями также повлияет ваша стратегия версионирования, поэтому давайте рассмотрим различные методы, которые доступны для указания версии API.

- *Управление версиями с использованием URL-адреса* — вы можете встроить версию API в URL-адрес, например, так: <https://coffeemesh.com/api/v1/coffee>. Это очень удобно, потому что потребители вашего API будут знать, что они всегда смогут вызвать один и тот же эндпоинт и получить одни и те же результаты. Если вы выпустите новую версию своего API, эта версия перейдет по другому URL-адресу (/api/v2) и, следовательно, не будет конфликтовать с вашими предыдущими версиями. Это также упрощает изучение вашего API, поскольку для обнаружения и тестирования различных его версий пользователям нужно всего лишь изменить поле версии в URL. С другой стороны, при работе с REST API использование URL-адреса для управления версиями считается нарушением принципов REST, поскольку каждый ресурс должен быть представлен только одним URI.
- *Управление версиями с использованием поля заголовка Accept* — HTTP-заголовок запроса `Accept` указывает, какие типы контента клиент может парсировать. В контексте API типичное значение заголовка `Accept` — `application/json`. Это означает, что клиент принимает данные только в формате JSON. Поскольку версия API также влияет на тип контента, который мы получаем с сервера, можно использовать поле `Header`, чтобы указать, какую версию API нужно задействовать. Вот пример поля заголовка, указывающего тип контента и версию API, является: `Accept application/json;v1`.

Этот подход лучше сочетается с принципами REST, поскольку в данном случае не изменяются эндпоинты ресурсов, но требуется тщательный парсинг. Введение дополнительных символов в поле заголовка, как в следующем фрагменте, может привести к ошибкам во время выполнения:

```
Accept: application/json; v1
# обратите внимание на пробел после точки с запятой
```

Поскольку мы используем заголовок `Accept`, то отвечаем 415 (Unsupported Media Type) на любые ошибки в объявлении версии API или когда клиент запрашивает недоступную версию API.

- *Управление версиями с использованием кастомных полей заголовка запроса* — при таком подходе вы определяете кастомное поле заголовка запроса, например `Accept-version`, чтобы указать версию API, которую хотите использовать. Этот подход является наименее предпочтительным, поскольку некоторые фреймворки могут не воспринимать нестандартные поля заголовка, а это чревато проблемами интеграции с вашими клиентами.

Каждая стратегия управления версиями имеет свои преимущества и недостатки. Управление версиями через URL — наиболее распространенный вариант, поскольку он интуитивно понятен и прост в использовании. Однако указание версии API в URL-адресе также означает, что URI ресурсов будут меняться в зависимости от версии API, что может сбивать с толку некоторых клиентов.

Использование заголовка `Accept` — еще один популярный вариант, но он объединяет логику обработки медиатипов с логикой обработки версий API. Кроме того, использование одного и того же статус-кода ошибки для медиатипов и версий API может запутать клиентов API. Лучше всего тщательно изучить потребности приложения и согласовать с клиентами API наиболее предпочтительное решение.

## Б.2. ВЫВОД API ИЗ ЭКСПЛУАТАЦИИ

В этом разделе мы изучим стратегии, позволяющие аккуратно вывести API из эксплуатации (объявить их устаревшими — *deprecated*). API не вечны; по мере развития и изменения товаров и услуг, которые вы предлагаете с помощью API, некоторые из них станут устаревшими, и вы в конечном счете откажетесь от них. Однако у вас могут быть внешние потребители, чьи системы зависят от ваших API, поэтому нельзя просто отключить их, не вызвав сбоев в работе ваших клиентов. Вы должны организовать процесс вывода вашего API из эксплуатации, и для уведомления об этом используются определенные HTTP-заголовки. Давайте посмотрим, как это работает.

Прежде чем вы удалите API, вам следует сначала объявить его устаревшим. Такой API все еще находится в эксплуатации, но его не обслуживают, не улучшают и не исправляют. Как только вы объявите свои API устаревшими, ваши пользователи не будут ожидать дальнейших изменений в них. За это время пользователи смогут перенести свои системы на новый API без нарушения их работы.

Как только вы решите отказаться от своего API, вам следует сообщить об этом потребителям по привычному каналу связи, например по электронной почте или в рассылке новостей. В то же время вы должны добавить в своих ответах заголовок `Deprecation` (Устаревание)<sup>1</sup>. Если API планируется признать устаревшим в будущем, указывайте в заголовке ту дату, когда API будет признан устаревшим:

```
Deprecation: Friday, 22nd March 2025 23:59:59 GMT
```

Как только API становится устаревшим, мы устанавливаем для заголовка `Deprecation` значение `true`:

```
Deprecation: true
```

<sup>1</sup> Dalal S., Wilde E. The Deprecation HTTP Header Field («Поле Deprecation HTTP-заголовка»). <https://datatracker.ietf.org/doc/html/draft-ietf-httpapi-deprecation-header-02>.

Вы также можете использовать заголовок `Link`, чтобы предоставить дополнительную информацию о процессе устаревания вашего API. Например, вы можете предоставить ссылку на свою политику устаревания:

```
Link: <https://coffeemesh.com/deprecation>; rel="deprecation";  
➡ type="text/html"
```

В этом случае мы сообщаем пользователю, что он может перейти по ссылке <https://coffeemesh.com/deprecation>, чтобы найти дополнительную информацию об устаревании API.

Если вы выводите из эксплуатации старую версию API, можете использовать заголовок `Link`, чтобы указать URL-адрес к API новой версии:

```
Link: <https://coffeemesh.com/v2.0.0/coffee>; rel="successor-version"
```

Помимо информации об устаревании API, вы должны объявить, когда API будет удален. Для этого используется заголовок `Sunset`<sup>1</sup>:

```
Sunset: Friday, 22nd June 2025 23:59:59 GMT
```

Дата в заголовке `Sunset` должна или совпадать с датой, указанной в заголовке `Deprecation`, или быть более поздней. Как только вы удалили API, сообщите своим клиентам API, что старые эндпоинты больше недоступны. Вы можете использовать любую комбинацию статус-кодов 3xx и 4xx для тех случаев, когда пользователь вызывает старый API. Хорошим вариантом является статус-код 410 (`Gone`). Мы используем его для уведомления о том, что запрошенный ресурс больше не существует по известной причине. В некоторых обстоятельствах может оказаться полезным код 301 (`Moved Permanently`). Его используют для уведомления о том, что запрошенному ресурсу присвоен новый URI, поэтому он подходит при переносе API на новый эндпоинт.

Правильное управление изменениями и устареваниями API часто упускают из виду, хотя это очень важный этап, необходимый для обеспечения высококачественной и надежной интеграции с помощью API. Учитывайте рекомендации из этого приложения и вы сможете уверенно развивать свои API, не нарушая интеграции с клиентами.

---

<sup>1</sup> Wilde E. The Sunset HTTP Header Field // RFC 8594. <https://tools.ietf.org/html/rfc8594>.



# Авторизация API с использованием поставщика удостоверений

---

В главе 11 вы узнали, как работают протоколы Open Authorization (OAuth) и OpenID Connect (OIDC). Вы также научились создавать, проверять и валидировать веб-токены JSON (JWT). Наконец, вы изучили паттерн для добавления промежуточного программного обеспечения авторизации в ваши API. Вопрос, на который нам все еще нужно ответить: как создать комплексную систему аутентификации и авторизации?

Для аутентификации и авторизации можно использовать различные стратегии. Вы можете создать свой собственный сервис аутентификации или задействовать сервис поставщика удостоверений (identity-as-a-service provider, англ. «поставщик идентификации как услуги»), такой как Auth0, Okta, Azure Active Directory или AWS Cognito. Если вы не являетесь экспертом в области веб-безопасности и протоколов аутентификации и не располагаете достаточными ресурсами для правильной конфигурации системы, рекомендую вам воспользоваться услугами поставщика удостоверений. В этом приложении вы узнаете, как добавить аутентификацию в свои API с помощью Auth0, которая является одной из самых популярных систем управления идентификационными данными.

Мы будем использовать бесплатный тарифный план Auth0. Он заботится об управлении учетными записями пользователей и выдаче защищенных токенов, а также обеспечивает простую интеграцию для входа в социальные сети с такими

поставщиками удостоверений, как Google, Facebook, Twitter и др. Система Auth0 построена на стандартах, поэтому все, что вы узнаете об аутентификации с помощью Auth0, применимо к любому другому поставщику. Если даже вы используете другую систему аутентификации в своих проектах или на работе, то все равно почерпнете полезную информацию из этого приложения и сможете применить ее к любой системе.

Код для этого приложения доступен в папке `appendix_c` в репозитории GitHub для книги. Я рекомендую вам использовать этот код, в частности папку с именем `appendix_c/ui`, поскольку она понадобится вам для запуска примеров в разделе В.2.

## В.1. ИСПОЛЬЗОВАНИЕ ПОСТАВЩИКА УДОСТОВЕРЕНИЙ КАК УСЛУГИ

В этом разделе объясняется, как интегрировать наш код с помощью поставщика удостоверений как услуги (identity-as-a-service, IDaaS). Поставщик IDaaS — это сервис, который заботится об аутентификации пользователей и выдаче токенов доступа для них. Использование IDaaS-провайдера удобно, поскольку это означает, что мы можем сосредоточиться на разработке наших API. Хорошие провайдеры IDaaS основаны на стандартах и надежных протоколах безопасности, что также снижает риски для безопасности наших серверов. В этом разделе вы узнаете, как создать интеграцию с помощью Auth0, который является одним из самых популярных поставщиков IDaaS.

Чтобы работать с Auth0, сначала создайте учетную запись, а затем создайте клиент, следуя документации Auth0 (<https://auth0.com/docs/get-started>). В первую очередь перейдите на свою панель мониторинга (dashboard) и создайте API для представления сервиса заказов. Сконфигурируйте его, как показано на рис. В.1, назначив ему `http://127.0.0.1:8000/orders` в качестве значения идентификатора и назовите алгоритм подписи RS256.

Как только вы создадите API, перейдите в раздел Permissions (Разрешения) и добавьте область разрешений (permission scope) в API, как показано на рис. В.2.

Затем нажмите Settings (Настройки) на панели слева и перейдите на вкладку Custom Domains (Пользовательские домены) (рис. В.3).

Вы можете добавить кастомный домен, если хотите, или использовать домен Auth0 по умолчанию для вашего арендатора. Мы используем этот домен для создания хорошо известного URL-адреса нашего сервиса аутентификации:

`https://<tenant>.<region>.auth0.com/.well-known/openid-configuration`

**New API**

**Name \***

Orders API

A friendly name for the API.

**Identifier \***

http://127.0.0.1:8000/orders

A logical identifier for this API. We recommend using a URL but note that this doesn't have to be a publicly available URL, Auth0 will not call your API at all. **This field cannot be modified.**

**Signing Algorithm \***

RS256

Algorithm to sign the tokens with. When selecting RS256 the token will be signed with Auth0's private key.

Cancel Create

**Рис. В.1.** Чтобы создать новый API, нажмите кнопку Create API (Создать API) и заполните форму, указав название API, его URL-идентификатор и алгоритм подписи, который вы хотите использовать для его токенов доступа

Auth0 Dashboard

Getting Started Activity Applications Applications API SSO integrations Authentication Organizations User Management Branding Security Actions Auth Pipeline Monitoring

Back to APIs

**Orders API**

Custom API Identifier: http://127.0.0.1:8000/orders

Quick Start Settings Permissions Machine to Machine Applications Test

**Add a Permission (Scope)**

Define the permissions (scopes) that this API uses.

Permission (Scope) *	Description *
access:orders	Gives access to the orders API

+ Add

**List of Permissions (Scopes)**

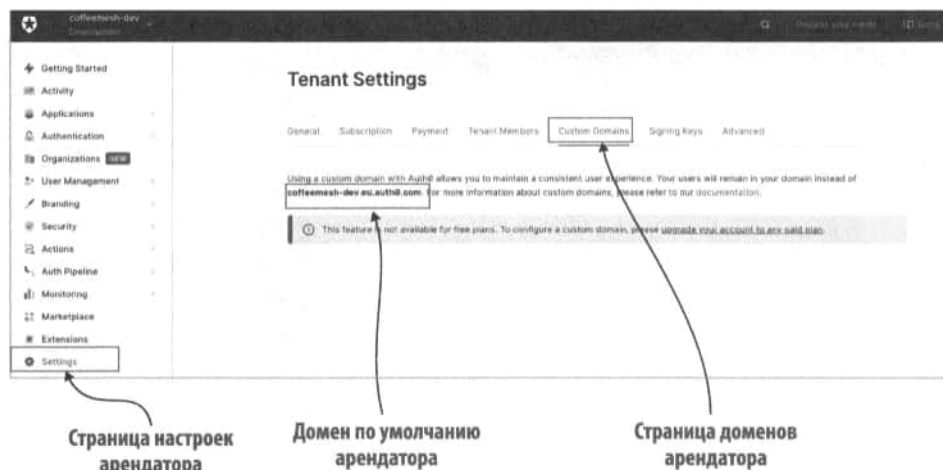
These are all the permission(s) (scopes) that this API uses.

Ваша страница API

Вкладка разрешений  
этого API

Добавьте области разрешений  
в API, используя эту форму

**Рис. В.2.** Чтобы добавить области разрешений в API, перейдите на вкладку Permissions (Разрешения) и заполните форму Add a Permission (Scope) (Добавить разрешение)



**Рис. В.3.** Чтобы узнать домен по умолчанию вашего арендатора, откройте страницу настроек арендатора и перейдите на вкладку Custom Domains

Например, для CoffeeMesh доменом клиента является <https://coffeemesh-dev.eu.auth0.com/.well-known/openid-configuration>.

Теперь вызовите этот URL-адрес и найдите свойство `jwtks_uri`, представляющее URL-адрес, возвращающий открытые ключи, которые мы можем использовать для проверки токенов Auth0. Вот пример:

```
$ curl https://coffeemesh.eu.auth0.com/.well-known/openid-configuration \
| jq .jwtks_uri
# output:
"https://coffeemesh-dev.eu.auth0.com/.well-known/jwks.json"
```

Если вы вызовете этот URL-адрес, то получите массив объектов, каждый из которых содержит информацию об открытых ключах вашего арендатора. Каждый объект выглядит следующим образом:

```
{
  "alg": "RS256",
  "kty": "RSA",
  "use": "sig",
  "n": "sV2z9AApyKK-
➔ Zo9vrzHbonNsHTgYiIOx1dHx3U102fUhfPFzUcdnjb7li960iTKyTbFlMRbsN2fFZOHaS_4Q
➔ 3C7UzjkVw__jK3AcPZ-0cCiLBS-HQzE_6ii-OPo84-
➔ W9Pp2ScKdAlJlQBiMDtNv8vu0EMr5c5YbJz1H1ppFY_hA7ldgc101SHp0n9GZYqP5HV713m
➔ 6smE5b7abHLqrUSz9eVbS0rTU0cSd5_LUHvQqFb5Wt7kRaIIiHnQFob-
➔ cyM1AmxDNsX1qR2cX_jqjWCR02iK5DTG--ure8GQUTCMPZ0LKBKSDelTwHuEn_r4z-
➔ x30wf-21A0yzMSlxcxCIojpQ",
  "e": "AQAB",
```

```

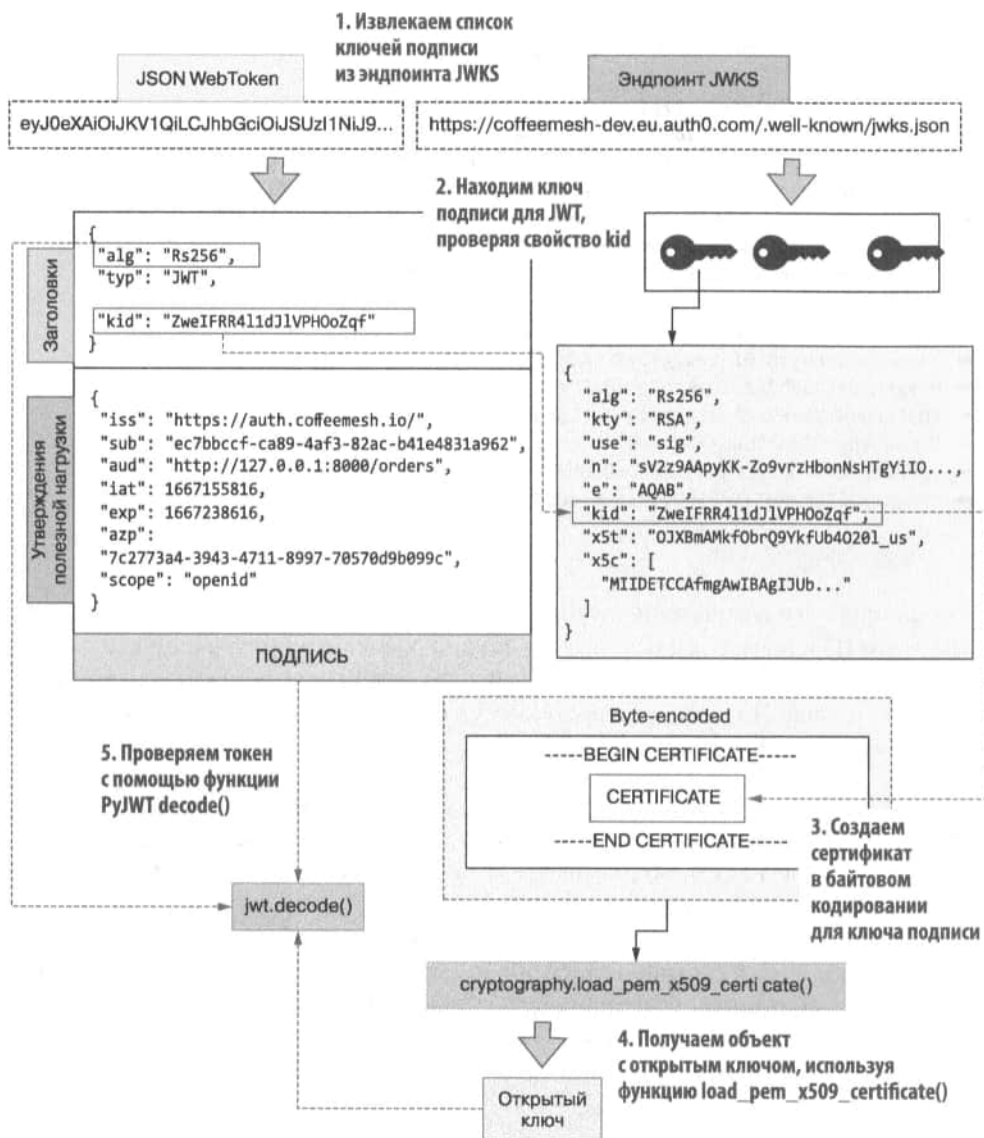
"kid": "ZweIFRR411dJlVPHo0Zqf",
"x5t": "OJXBmAMkfObrQ9YkfUb40201_us",
"x5c": [
  "MIIDETCCAfmGAWIBAgIJUBXpEmZ8n1mXMA0GCSqGSIb3DQEBCwUAMCYxJDAiBgNVBAMTG2NvZm
  ➔ ZlZW1lc2gtZGV2LmV1LmF1dGgwLmNvbTAeFw0yMTEwMjkyMjQ0MjBaFw0zNTA3MDgyMjQ0Mj
  ➔ jBAMCYxJDAiBgNVBAMTG2NvZmZlZW1lc2gtZGV2LmV1LmF1dGgwLmNvbTCCASIDQYJKoZI
  ➔ hvCNAQEBAQADgEPPADCCAQoCggEBALFds/QAKciivmaPb68x26JzbB04GiDsXR8d1NdNn
  ➔ 1ITxc1HHZ42+5YvetIkysk2xZTEW7DdnxWTh2uf+ENwu1M45FcP/4ytWHD2fTHAoiwUvh0M
  ➔ xP+oovjj6POP1vt6dKnCnQJSSKgYpg7Tb/L7jhDK+X0Wgyc9R5aaRWP4Q09XYHNdNUh6dJ/
  ➔ RmWKj+R1e9dSurJh0W+2mxy6q1Es/X1W0jq01DnEnefy1B70KhW+Vre5EWpSIh50BaG/nMj
  ➔ NQJsqZbF9akdnF/46o1gkTtoiUQ0xvvrq3vBkFEWjD2dC5ASkg3pU8B7hJ/6+M/sd9MH/tp
  ➔ QNMsZEpXMXCSKI6UCAwEAANCMCAAwDwYDVR0TAQH/BAUwAwEB/zAdBgNVHQ4EFgQUWr1+q/
  ➔ l4wp/MWDDYrhjxns0ip2wWdgyDVR0PAQH/BAQDAgKEMA0GCSqGSIb3DQEBCwUAA4IBAQA+Y
  ➔ H+sxCM1BzEOJ5HjGZw1upRroCGmeQzEh+Cx73sTKw+vi8u70bdkDt9sBLK1KG9xbPjt3+QW
  ➔ ZDJF9rwx4vXbFfvxZD+dtHivn4NH4/sLQXG20JN/b6GtHdV11bJIGUewb8DBsx94wXYMwag
  ➔ 0gXUk5spgaGGdoc16uSrrbxt/rmzFk3VMQ8qG5i8E33N/DZb88P4u3WJMNMsujw9Q8meg4
  ➔ ygEFadXBcfJPHuirLiW0j1Gm+m6DZQM510tpQ/cvcZXRNPogqj7wsZXH4za9DjJnQf8ZOK
  ➔ Q86WK1/9CE5AvHBTtTr810DviIqvs8sqC866+2t2euxcfOYMIw5E42o"
]
}

```

Двумя наиболее важными полями в этой полезной нагрузке являются `kid` и `x5c`. `kid` — это ID ключа, и мы используем его для сопоставления с полем `kid` заголовка JWT. Он сообщает, какой ключ нам нужно использовать для проверки подписи токена. Поле `x5c` содержит массив открытых ключей в виде сертификатов X.509, первый из которых мы используем для проверки подписи JWT.

Это вся информация, которая нам нужна для интеграции нашего кода с Auth0. Мы внедрим нашу Auth0-интеграцию в под `orders/web/api/auth.py`, который создали в главе 11 (см. подраздел 11.4.1) для инкапсуляции нашего кода авторизации. Удалите содержимое `orders/web/api/auth.py` и замените его содержимым листинга В.1. Сначала мы импортируем необходимые зависимости, создаем шаблон для сертификата X.509 и загружаем открытые ключи из хорошо известного эндпоинта. Сертификаты X.509 заключены между инструкциями -----BEGIN CERTIFICATE----- и -----END CERTIFICATE-----, поэтому наш шаблон включает в себя обе инструкции с переменной шаблона `key`, которую мы заменим фактическим ключом.

Поскольку Auth0 использует несколько ключей для подписи токенов, мы загружаем открытые ключи, вызывая эндпоинт JWKS, и динамически загружаем правильный ключ для данного токена. Как вы можете видеть на рис. В.4, свойство `kid` в заголовках токена сообщает нам, какой ключ нужно использовать, и наша функция `_get_certificate_for_kid()` находит сертификат X.509 для дочернего элемента токена. Чтобы загрузить ключ, мы используем функцию `load_pem_x509_certificate()`, передавая открытый ключ, отформатированный в нашем сертификате X.509.



**Рис. В.4.** Чтобы проверить JWT, мы проверяем его подпись, используя соответствующий ключ подписи, который доступен в эндпоинте JWKS

Поскольку токены могут быть подписаны с помощью различных алгоритмов, мы извлекаем алгоритмы непосредственно из заголовков токена. Auth0 выдает токены, которые могут получить доступ как к нашему API, так и к пользовательскому информационному API, поэтому мы включаем оба сервиса в аудиторию токена (листинг В.1).

**Листинг В.1.** Добавление модуля авторизации в API

```
# file: orders/web/api/auth.py
```

```
import jwt
import requests
from cryptography.x509 import load_pem_x509_certificate

X509_CERT_TEMPLATE = (
    "-----BEGIN CERTIFICATE-----\n{key}\n-----END CERTIFICATE-----"
)

public_keys = requests.get(
    "https://coffeemesh-dev.eu.auth0.com/.well-known/jwks.json"
).json()["keys"]

def _get_certificate_for_kid(kid):
    """
    Return the public key whose ID matches the provided kid.
    If no match is found, an exception
    """
    for key in public_keys:
        if key["kid"] == kid:
            return key["x5c"][0]
    raise Exception(f"Not matching key is raised. found for kid {kid}")

def load_public_key_from_x509_cert(certificat):
    """
    Loads the public signing key into a RSAPublicKey object. To do that,
    we first need to format the key into a PEM certificate and make sure
    it's utf-8 encoded. We can then load the key using cryptography's
    convenient `load_pem_x509_certificate` function.
    """
    return load_pem_x509_certificate(certificat).public_key()

def decode_and_validate_token(access_token):
    """
    Validates an access token. If the token is valid, it returns the token
    payload.
    """
    unverified_headers = jwt.get_unverified_header(access_token)
    x509_certificate = _get_certificate_for_kid(
        unverified_headers["kid"]
    )
    public_key = load_public_key_from_x509_cert(
        X509_CERT_TEMPLATE.format(key=x509_certificate).encode("utf-8")
    )
    return jwt.decode(
        access_token,
        key=public_key,
        algorithms=unverified_headers["alg"],
    )
```

Шаблон для сертификата X509

Извлекаем список ключей подписи из хорошо известного эндпоинта арендатора

Функция, которая возвращает сертификат для заданного ID ключа

Ищем сертификат, который соответствует указанному ID ключа

Если совпадение не найдено, вызываем исключение

Функция, которая загружает объект открытого ключа для данного сертификата

Загружаем открытый ключ

Функция, которая декодирует и проверяет JWT

Извлекаем заголовок токена без проверки

Извлекаем сертификат, соответствующий ID ключа токена

Загружаем объект открытого ключа сертификата

Проверяем и декодируем токен

Проверяем подпись токена, используя алгоритм, указанный в заголовке токена

```

    audience=[
      "http://127.0.0.1:8000/orders",
      "https://coffeemesh-dev.eu.auth0.com/userinfo",
    ],
  )

```

Передаем список ожидаемых аудиторий для токена

Мы готовы к работе! Сервис заказов теперь может проверять токены, выданные Auth0. В следующих разделах показано, как использовать эту интеграцию, чтобы сделать сервер, предоставляющий API, доступным для одностраничного приложения (single-page application, SPA) и другого микросервиса.

## V.2. ИСПОЛЬЗОВАНИЕ СХЕМЫ АВТОРИЗАЦИИ РКСЕ

В схеме РКСЕ клиент API запрашивает идентификационный токен и токен доступа непосредственно с сервера авторизации. Как я объяснял в главе 11, мы должны использовать токен доступа для взаимодействия с сервером API. ID токен можно использовать в UI для отображения сведений о пользователе, но он никогда не должен отправляться на наш сервер, предоставляющий API.

Чтобы проиллюстрировать, как работает эта схема, я включил SPA в каталог `appendix_c/ui` в репозитории GitHub для этой книги. The SPA — простое приложение, созданное с использованием Vue.js, который взаимодействует с API сервиса заказов и настроен на аутентификацию с помощью сервера Auth0.

Сначала мы настроим приложение. Перейдите в свою учетную запись Auth0 и создайте новое приложение. Выберите Single Page Web Applications (Одностраничное веб-приложение) и дайте ему название, затем нажмите кнопку Create (Создать). На странице настроек приложения в разделе Application URI укажите значение `http://localhost:8000` для полей Allowed Callback URLs (Разрешенные URL-адреса обратного вызова), Allowed Logout URLs (Разрешенные URL-адреса выхода из системы), Allowed Web Origins (Разрешенные веб-источники) и Allowed Origins (CORS) (Разрешенные источники происхождения). Из настроек приложения нам нужны два значения: домен и ID клиента. Откройте файл `ui/.env.local` и замените значение для `VUE_APP_AUTH_CLIENT_ID` на ID клиента, а для `VUE_APP_AUTH_DOMAIN` — на домен со страницы настроек вашего приложения в Auth0.

Чтобы запустить пользовательский интерфейс, вам нужна обновленная версия Node.js и npm, который вы можете загрузить с сайта node.js (<https://nodejs.org/en/>). Как только вы их установите, нужно установить yarn:

```
$ npm install -g yarn
```



Затем перейдите в папку `ui/` и установите зависимости, выполнив следующую команду:

```
$ yarn
```

Как только приложение настроено, вы можете запустить его, выполнив такую команду:

```
$ yarn serve --mode local
```

Приложение станет доступно по адресу `http://localhost:8080`. Убедитесь, что API сервиса заказов также запущен, так как Vue.js приложение общается с ним. Чтобы запустить этот API, выполните следующую команду из папки `orders`:

```
$ AUTH_ON=True uvicorn orders.web.app:app --reload
```

Как только вы зарегистрируете пользователя через пользовательский интерфейс, вы сможете увидеть свой токен авторизации. Можете использовать его для вызова API непосредственно из терминала. Например, вы можете получить список заказов для вашего пользователя, вызвав API с помощью следующей команды:

```
$ curl http://localhost:8080/orders \
-H 'Authorization: Bearer <ACCESS_TOKEN>'
```

Через Vue.js в приложении вы можете создавать новые заказы и отображать заказы, размещенные пользователем, нажав кнопку `Show My Orders` (Показать мои заказы).

Схема РКСЕ работает для пользователей, получающих доступ к вашим API через браузер. Однако эта схема неудобна для межмашинного взаимодействия. Чтобы обеспечить более программный доступ к вашим API, вам необходимо использовать схему учетных данных клиента. В следующем разделе я объясню, как включить ее.

## В.3. ИСПОЛЬЗОВАНИЕ СХЕМЫ УЧЕТНЫХ ДАННЫХ КЛИЕНТА

В этом разделе объясняется, как реализовать схему учетных данных клиента для обмена данными между серверами. Мы используем схему от сервера к серверу, когда необходимо аутентифицировать наши собственные сервисы для доступа к другим API или когда мы хотим разрешить программный доступ к нашим API. В схеме учетных данных клиента наши сервисы запрашивают токен доступа у сервиса аутентификации, предоставляя общий секрет с ID клиента и желаемой аудиторией. Затем мы можем использовать этот токен доступа для доступа к API целевой аудитории.

Чтобы использовать эту схему авторизации, вам необходимо зарегистрировать межсерверный клиент у поставщика IDaaS. На странице приложений панели управления Auth0 нажмите **Create Application** (Создать приложение) и выберите **Machine to Machine Applications** (Приложения межмашинного взаимодействия). Назовите приложение и нажмите кнопку **Create** (Создать). На следующем экране, где вас попросят выбрать API, для которого вы хотите авторизовать этот клиент, выберите API сервиса заказов, а затем укажите разрешение, которое мы создали в главе 11 (см. раздел 11.6). Как только вы зарегистрируете клиент, вы получите ID и секрет клиента, которые сможете использовать для получения токенов доступа.

В листинге B.2 показано, как реализовать авторизацию от сервера к серверу для получения токена доступа и выполнения вызова API сервиса заказов. Код, приведенный в листинге B.2, доступен в репозитории книги на GitHub, в файле `machine_to_machine_test.py`. Мы создаем функцию для получения токена доступа с сервера авторизации, выполняя запрос POST `https://coffeemesh-dev.eu.auth0.com/oauth/token`. В полезной нагрузке запроса указываем ID и секрет клиента, а также аудиторию, для которой хотим сгенерировать токен доступа. Мы также заявляем, что хотим использовать схему учетных данных клиента, в свойстве `grant_type`. Если клиент прошел правильную аутентификацию, мы получаем обратно токен доступа, который затем используем для вызова API сервиса заказов.

### Листинг B.2. Авторизация клиента для межмашинного доступа к API сервиса заказов

```
# file: machine_to_machine_test.py

import requests

def get_access_token():
    payload = {
        "client_id": "<client_id>",
        "client_secret": "<client_secret>",
        "audience": "http://127.0.0.1:8000/orders",
        "grant_type": "client_credentials"
    }

    response = requests.post(
        "https://coffeemesh-dev.eu.auth0.com/oauth/token",
        json=payload,
        headers={'content-type': "application/json"}
    )

    return response.json()['access_token']

def create_order(token):
    order_payload = {
        'order': [{
            'product': 'cappuccino',
            'size': 'small',
```

```
        'quantity': 1
    }}
}

order = requests.post(
    'http://127.0.0.1:8000/orders',
    json=order_payload,
    headers={
        "content-type": "application/json",
        "authorization": f"Bearer {token}",
    }
)

return order.json()

access_token = get_access_token()
print(access_token)
order = create_order(access_token)
print(order)
```

Это все, что требуется для использования схемы учетных данных клиента! В следующем разделе вы научитесь аутентифицировать свои запросы с помощью пользовательского интерфейса Swagger, чтобы вам было легче тестировать API.

## В.4. АВТОРИЗАЦИЯ ЗАПРОСОВ В ПОЛЬЗОВАТЕЛЬСКОМ ИНТЕРФЕЙСЕ SWAGGER

На протяжении этой книги вы научились тестировать свои API с помощью Swagger UI. Вы также можете использовать его для проверки авторизации API, и в этом разделе вы узнаете, как это сделать. Сначала зайдите в `appendix_c/orders` и запустите сервер API с авторизацией:

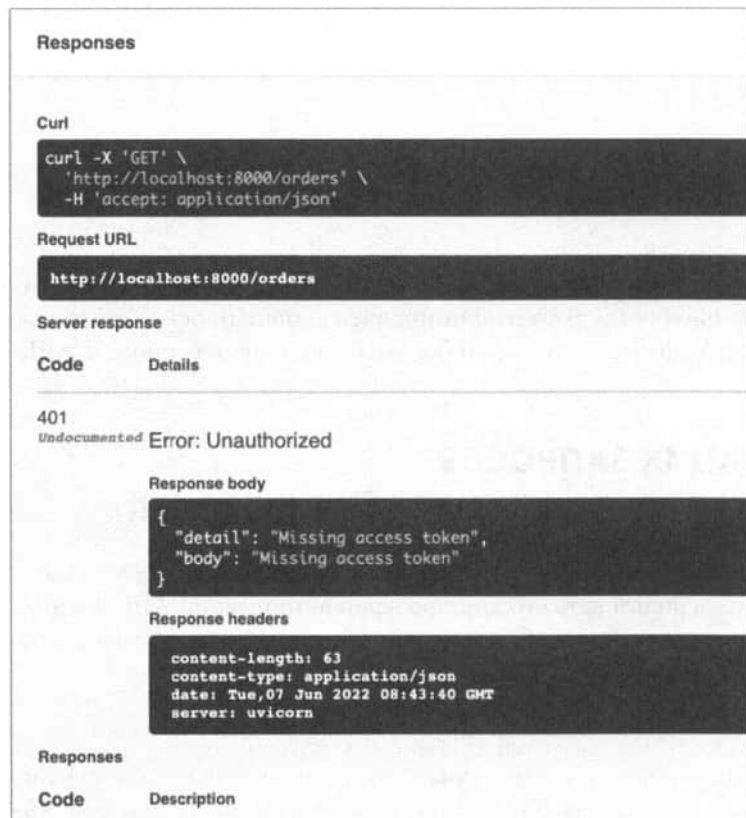
```
$ AUTH_ON=True uvicorn orders.web.app:app --reload
```

Теперь вы можете получить доступ к пользовательскому интерфейсу Swagger по адресу `http://localhost:8000/docs/orders`. Как вы можете видеть на рис. В.5, если вы выберете любой из эндпоинтов, то получите ответ 401, поскольку мы не авторизовали наши запросы.

Чтобы авторизовать запрос, нажмите кнопку **Authorize** (Авторизовать) в правом верхнем углу экрана. Вы увидите всплывающее меню со схемами безопасности, задокументированными в спецификации API: `openId` (схемы кода авторизации и PKCE), `oauth2` (схема учетных данных клиента) и `bearerAuth`. Самый простой способ протестировать слой авторизации API — использовать схему безопасности `bearerAuth`, поскольку для этого требуется только ввести токен авторизации. Вы можете создать токен с помощью приложения Vue.js в `appendix_c/ui` или

с использованием сценария `machine_to_machine_test.py`. Например, если вы запустите `machine_to_machine_test.py`, то получите токен и результат создания заказа:

```
$ python machine_to_machine_test.py
# output:
eyJhbGciOiJSUzI1NiIsInR5cCI6IkpXVCIsImtpZCI6Imlp3ZULGU1I0bDFkSmxWUEhPb1px...
➔ {'order': [{'product': 'latte', 'size': 'small', 'quantity': 1}], 'id':
➔ '6e420d2e-b213-4d15-bc46-0c680e590154', 'created': '2022-06-
➔ 07T09:01:47.757223', 'status': 'created'}
```



**Рис. В.5.** Если мы сделаем недопустимый запрос с помощью пользовательского интерфейса Swagger, то получим ответ 401

Скопируйте токен и вставьте его в поле ввода значения схемы безопасности `bearerAuth`, как показано на рис. В.6, а затем нажмите кнопку **Authorize** (Авторизовать). Если вы отправите запрос в эндпоинт `GET /orders` сейчас, то получите успешный ответ. Пока токен действителен, вы можете попробовать любой другой эндпоинт, и ваши запросы будут успешно обработаны.

**Available authorizations** ✕

Each API may declare one or more scopes.  
API requires the following scopes. Select which ones you want to grant to Swagger UI.

**oauth2 (OAuth2, clientCredentials)**

Token URL: `https://coffeemesh-dev.eu.auth0.com/oauth/token`  
Flow: `clientCredentials`

client\_id:

client\_secret:

**bearerAuth (http, Bearer)**

Value:

**Рис. В.6.** Чтобы авторизовать запрос, укажите токен авторизации в поле Value (Значение) из формы bearerAuth

Это все, что требуется, чтобы протестировать авторизацию вашего API с помощью Swagger UI.

Вы только что узнали, как добавить надежный слой аутентификации и авторизации путем интеграции с внешним поставщиком удостоверений, как протестировать схемы PKCE и учетных данных клиента, а также как протестировать реализацию авторизации API с помощью Swagger. Вы готовы приступить к разработке защищенных API!

# КРОК

СОЗДАЕМ НАСТОЯЩЕЕ,  
ИНТЕГРИРУЕМ БУДУЩЕЕ



[croc.ru](http://croc.ru)

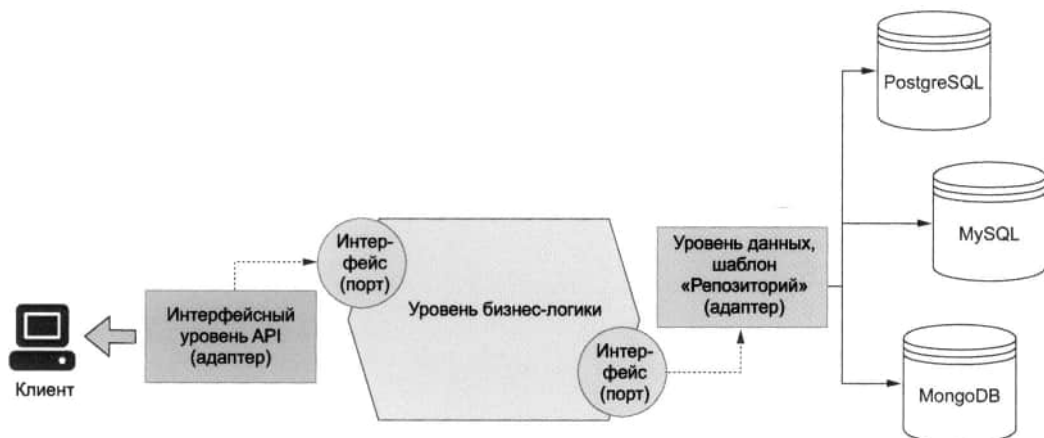
---

КРОК — технологический партнер с комплексной экспертизой в области построения и развития инфраструктуры, внедрения информационных систем, разработки программных решений и сервисной поддержки.

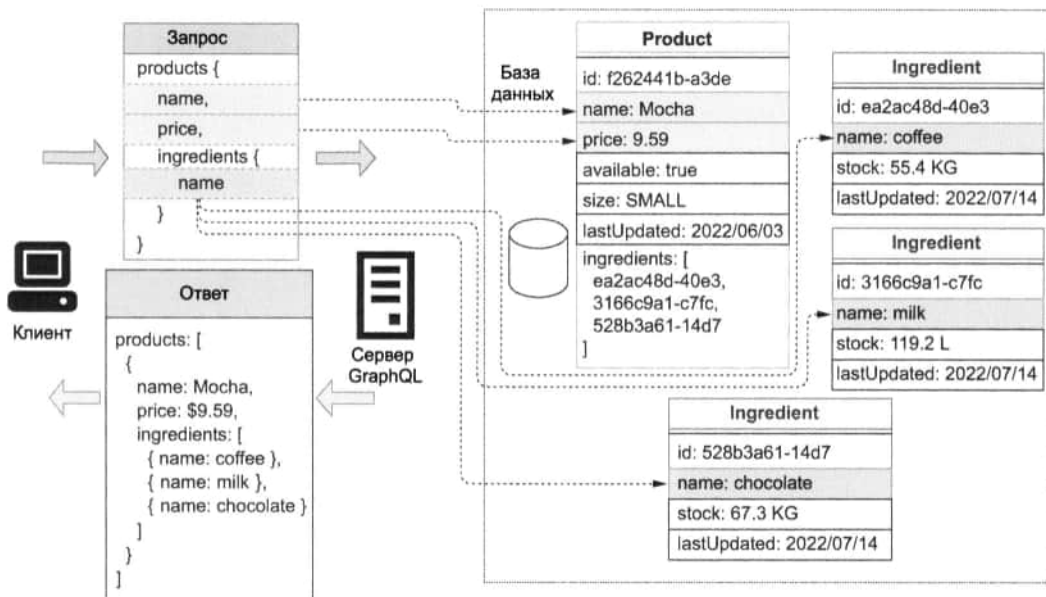
Центры компетенций КРОК фокусируются на ключевых отраслевых кластерах — промышленность, финансовый сектор, розничные продажи, муниципальное управление, спорт и культура.

Ежегодно сотни проектов КРОК становятся системообразующими для экономики и социально-культурной сферы.

---



Гексагональная архитектура помогает создавать сервисы со слабосвязанными компонентами. Мы различаем основной уровень бизнес-логики, представляющий ядро архитектуры, и набор портов и адаптеров, связывающих уровень бизнес-логики с внешними компонентами, такими как базы данных и соединения со сторонними API через HTTP. Мы используем шаблон «Репозиторий», чтобы инкапсулировать сложность доступа к источникам данных и исключить возможность просачивания деталей их реализации на уровень бизнес-логики.



GraphQL позволяет клиентам API запрашивать данные с сервера и получать информацию о связанных ресурсах. В этом примере API сервиса продукции имеет два типа ресурсов: товары и ингредиенты, связанные через поле со списком ингредиентов товара. Используя эту связь, клиент может запросить название и цену каждого товара, а также названия ингредиентов, составляющих любой товар.

# Микросервисы и API

Хосе Аро Перальта

**П**ростые и понятные API — необходимое условие успеха микросервисных приложений. Хорошо продуманные API гарантируют надежную интеграцию сервисов и помогают упростить сопровождение, масштабирование и дальнейшее совершенствование. Познакомьтесь с паттернами, протоколами и стратегиями, которые помогут вам проектировать, реализовывать и разворачивать эффективные микросервисы с REST и GraphQL API.

Книга наполнена проверенными советами и примерами кода на языке Python. Авторы фокусируются на реализации, а не на философии. Изучите проверенные методы проектирования простых в использовании API для микросервисных приложений. Создавайте надежные API микросервисов, тестируйте, защищайте и разворачивайте их в облаке, следуя принципам и шаблонам, применимым в любом языке программирования.

- стратегии декомпозиции сервисов;
- лучшие практики проектирования и реализации REST и GraphQL API;
- паттерны реализации сервисов из слабосвязанных компонентов;
- авторизация API с помощью OAuth и OIDC;
- разворачивание с помощью AWS и Kubernetes.

Для разработчиков, знакомых с основами веб-разработки.

Хосе Аро Перальта — консультант, автор и преподаватель. Является основателем microapis.io.

«Содержательное руководство по созданию REST и GraphQL API с понятными примерами на основе FastAPI и Flask. А главы о паттернах реализации сервисов должен прочитать каждый разработчик».

— Уильям Джамир Сильва, Adjust

«Замечательное введение в веб-API микросервисов на Python».

— Стюарт Вудворд, генеральный директор Hanamatu

«Хорошо спроектированный API обеспечит успех вашему следующему проекту. Эта книга снабдит вас всей необходимой информацией. Отличное руководство!»

— Ален Ломпо, ISO-Gruppe

«Превосходная книга с практическими примерами».

— Самбасива Андалури, IBM

 ПИТЕР®



WWW.PITER.COM  
интернет-магазин

Заказ книг:  
(812) 703-73-74  
books@piter.com



PiterBooks  
PiterForPeople  
ThePiterBooks  
Company/piter

Выпущено  
при поддержке

**КРОК**

ISBN: 978-5-4461-2094-9



 MANNING