O'REILLY®

Стили API

Проектирование и внедрение



Лукаш Дыновски,
Марцин Дулак

Предисловие Жаки Рид

SPRINT book

Learning API Styles

Understanding the Trade-Offs of Common APIs and Choosing the Correct Solutions

Lukasz Dynowski, Marcin Dulak
Foreword by Jacqui Read

Beijing • Boston • Farnham • Sebastopol • Tokyo O'REILLY®

Стили API

Проектирование и внедрение

Лукаш Дыновски, Марцин Дулак
Предисловие Жаки Рид

SPRINT book 2026

ББК 32.973.2-018
УДК 004.415
Д89

Дыновски Лукаш, Дулак Марцин

Д89 Стили API. Проектирование и внедрение. — Астана: «Спринт Бук», 2026. — 416 с.: ил.

ISBN 978-601-14-1157-8

При разработке современных систем, таких как веб-приложения, микросервисы и IoT-устройства, программный интерфейс (API) необходим для обмена данными. Авторы Лукаш Дыновски и Марцин Дулак делятся с разработчиками и архитекторами ПО опытом проектирования и реализации ключевых типов API: REST, GraphQL, gRPC, веб-хуков, WebSocket и других.

В книге подробно рассматриваются наиболее популярные стили API с точки зрения сетевой модели, архитектурных паттернов и системных требований. Вы научитесь выбирать оптимальные варианты для своих приложений, оценивать компромиссы и применять практические рекомендации по реализации и интеграции API в реальных системах.

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

ББК 32.973.2-018
УДК 004.415

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

Информация, содержащаяся в данной книге, получена из источников, рассматриваемых издательством как надежные. Тем не менее, имея в виду возможные человеческие или технические ошибки, издательство не может гарантировать абсолютную точность и полноту приводимых сведений и не несет ответственности за возможные ошибки, связанные с использованием книги. Издательство не несет ответственности за доступность материалов, ссылки на которые вы можете найти в этой книге. На момент подготовки книги к изданию все ссылки на интернет-ресурсы были действующими. В книге возможны упоминания организаций, деятельность которых запрещена на территории Российской Федерации, таких как Meta Platforms Inc., Facebook, Instagram и др.

ISBN 978-1098153991 англ.

ISBN 978-601-14-1157-8

Authorized Russian translation of the English edition of Learning API Styles ISBN 9781098153991 © 2025 SRP Consultancy and Marcin Dulak.
This translation is published and sold by permission of O'Reilly Media, Inc., which owns or controls all rights to publish and sell the same.
© Перевод на русский язык ТОО «Спринт Бук», 2026
© Издание на русском языке, оформление ТОО «Спринт Бук», 2026

Краткое содержание

Глава 1. Основы API 29
Глава 2. Паттерны проектирования API 70
Глава 3. Сеть 108
Глава 4. Веб-протоколы 136
Глава 5. REST 180
Глава 6. GraphQL 230
Глава 7. Веб-фиды 259
Глава 8. gRPC 275
Глава 9. Вебхуки 316
Глава 10. WebSocket 337
Глава 11. API на основе обмена сообщениями 369

Об авторах 411
Иллюстрация на обложке 412

Оглавление

Предисловие 13

Вступление 16
- Почему мы написали эту книгу 17
- Для кого эта книга 18
- Что в книге есть 18
- Чего в книге нет 18
- Сервис прогноза погоды 19
- Условные обозначения 24
- Использование примеров кода 25

Благодарности 26
- От Лукаша Дыновски 26
- От Марцина Дулака 27

От издательства 28

Глава 1. Основы API 29
- Что такое API 29
- Сетевые API 31
- Взаимодействие через API: ключевые понятия 34
  - Сообщение 34
  - Режимы передачи 35
  - Синхронная и асинхронная коммуникация 36
- История API 38
- Зачем нужны API 41
- Что такое стили API 43
- API как продукт 47
- Жизненный цикл API 49
  - Планирование 50
  - Проектирование 53
  - Реализация 53
  - Тестирование 55
  - Развертывание 59

Сопровождение....63
Вывод из эксплуатации....64
Управление, администрирование и платформа API....66
Будущее API....67
Резюме....69

Глава 2. Паттерны проектирования API....70
Примеры паттернов проектирования API....70
Язык API....71
Наименование элементов API....71
Эволюция API....75
Версионирование API....77
Кодирование....80
Фильтрация....87
Подсчет и сортировка....88
Пагинация....88
Длительные задачи....92
Дедупликация запросов....93
Повторная отправка запроса....94
Ограничение частоты запросов....95
Кэширование....96
Удаление....97
Безопасность API....98
Проект OWASP API Security Top 10....99
Примеры паттернов безопасности API....100
Шифрование, аутентификация и авторизация....100
Очистка и валидация данных....101
Защита от скрейпинга данных....103
Лучшие практики проектирования API....103
Резюме....107

Глава 3. Сеть....108
Сетевые протоколы....108
Socket API....109
TCP/IP и модель OSI....111
Реализация службы TCP ECHO....115
Настройка сетевой лаборатории....116
Служба TCP ECHO....118
Сервер TCP ECHO....121
Клиент TCP ECHO с использованием netcat....122
Клиент TCP ECHO с использованием scapy....125

Безопасность ... 127
Клиент TCP ECHO с OpenSSL ... 129
Упражнения ... 134
Резюме ... 135

Глава 4. Веб-протоколы ... 136
Что такое гипертекст ... 136
HTTP: настройка лабораторной среды ... 138
HTTP/0.9 ... 139
HTTP и HTML в действии ... 140
HTTP/1.0 ... 143
Заголовки и тело HTTP-сообщения ... 144
Использование HTTP в браузере ... 147
Влияние DNS на соединения в браузере ... 148
Параллельные TCP-соединения в браузере ... 150
Ограничения TCP ... 152
Блокировка начала очереди TCP ... 153
Медленный старт и алгоритм предотвращения перегрузок ... 157
HTTP/1.1 ... 158
Постоянство HTTP-соединения ... 160
HTTP/2 ... 163
Фреймы и потоки в HTTP/2 ... 165
HTTP/3 ... 169
Установление соединения QUIC за 1 RTT ... 173
Упражнения ... 178
Резюме ... 179

Глава 5. REST ... 180
HTTP, ресурс и URI ... 180
Коммуникация по модели «клиент — сервер» ... 184
Происхождение стиля REST ... 187
Модели зрелости API ... 189
RESTful, RESTless или REST ... 190
Достаточно ли создать RESTless API ... 192
Реализация ... 195
Эндпоинты API ... 195
Версионирование API ... 197
CRUD ... 198
Пагинация ... 209
Ограничение частоты запросов ... 211
Кэширование API ... 212

Безопасность ... 213
JSON Web Token (JWT) ... 214
Transport Layer Security (TLS) ... 218
Документация ... 220
Спецификация OpenAPI ... 220
Три подхода к созданию спецификации API ... 224
REST API: преимущества и недостатки ... 226
Когда использовать REST ... 228
Упражнения ... 229
Резюме ... 229

Глава 6. GraphQL ... 230
Ограничения REST ... 231
Графовое мышление ... 231
Происхождение стиля GraphQL ... 233
GraphQL и REST ... 234
Ключевые элементы GraphQL ... 235
Принципы работы GraphQL ... 236
Реализация ... 237
GraphiQL ... 238
Выполнение мутаций ... 239
CRUD ... 241
Read ... 241
Create, Update, Delete ... 245
Безопасность ... 248
Атаки на GraphQL ... 248
Аутентификация ... 251
Документация ... 253
GraphQL: преимущества и недостатки ... 254
Когда использовать GraphQL ... 257
Упражнения ... 258
Резюме ... 258

Глава 7. Веб-фиды ... 259
Что такое веб-фид ... 259
Зачем нужны веб-фиды ... 260
Эволюция веб-фидов ... 261
Atom-фид ... 263
Реализация ... 266
Чтение фидов ... 269
Безопасность ... 270

Веб-фиды: преимущества и недостатки ... 272
Упражнения ... 274
Резюме ... 274

Глава 8. gRPC ... 275
Удаленный вызов процедур ... 276
Происхождение gRPC ... 277
Сериализация protobuf ... 277
Генерация кода ... 282
Сервер и клиент gRPC ECHO ... 285
Четыре типа RPC ... 290
Формат передачи данных protobuf ... 294
Реализация ... 298
Сервис обогащения Atom-ленты gRPC ... 299
Безопасность ... 309
Документация ... 310
gRPC: преимущества и недостатки ... 311
Когда использовать gRPC ... 313
Упражнения ... 314
Резюме ... 314

Глава 9. Вебхуки ... 316
Что такое вебхуки ... 317
Происхождение вебхуков ... 318
Входящие и исходящие вебхуки ... 318
Интеграция и поток данных ... 319
Реализация ... 321
Безопасность ... 324
Документация ... 328
Вебхуки: преимущества и недостатки ... 332
Когда использовать вебхуки ... 334
Упражнения ... 335
Резюме ... 335

Глава 10. WebSocket ... 337
WebSocket и WebSocket API ... 338
Установка соединения WebSocket ... 339
Реализация ... 344
Клиент и сервер WebSocket ECHO ... 344
Протокол WebSocket ... 346
Сервер оповещений о погоде на основе WebSocket ... 347
Клиент WebSocket для оповещений о погоде ... 355

Безопасность ... 356
Проверка источника ... 357
Безопасность доверенного хоста ... 357
JSON Web Token ... 357
Шифрование TLS ... 359
Документация ... 360
WebSocket: преимущества и недостатки ... 364
Когда использовать WebSocket ... 367
Упражнения ... 367
Резюме ... 368

Глава 11. API на основе обмена сообщениями ... 369
Что такое обмен сообщениями ... 370
Что такое очередь ... 373
Паттерны обмена сообщениями ... 375
Паттерн «Рабочая очередь» (Work Queue) ... 375
Паттерн «Издатель — подписчик» (pub/sub) ... 376
Паттерн «Маршрутизация» ... 377
Паттерн «Топики» ... 377
Паттерн «Запрос — ответ» ... 378
Реализация ... 379
Паттерн «Рабочая очередь» ... 379
Безопасность ... 389
Передача незашифрованных сообщений ... 390
Передача зашифрованных сообщений ... 391
Аутентификация и авторизация ... 394
Документация ... 398
Спецификация AsyncAPI ... 398
Документация AsyncAPI ... 401
Версионирование сообщений ... 402
API на основе обмена сообщениями: преимущества и недостатки ... 404
Когда использовать API на основе обмена сообщениями ... 408
Упражнения ... 409
Резюме ... 410

Об авторах ... 411

Иллюстрация на обложке ... 412

Отзывы о книге

Книга «Стили API» обязательна к прочтению всем, кто заинтересован в создании надежных и масштабируемых API. Здесь есть все необходимое для проектирования и разработки API: от базовых понятий и паттернов до всестороннего анализа плюсов и минусов разных методов взаимодействия через API.

Марк Ричардс (Mark Richards), соавтор книги Fundamentals of Software Architecture[1]

В этой книге вы найдете актуальное и глубокое исследование разработки API: от низкоуровневых протоколов до продвинутых архитектур. Книга необходима каждому специалисту, который хочет разобраться в основах API и в будущем принимать обоснованные проектные решения для создания надежных систем.

Уильям Джамир Сильва (William Jamir Silva), старший разработчик

В этой книге рассматриваются технические основы, ключевые архитектурные решения и детали реализации веб-API, которые нужно знать любому современному разработчику.

Эрик Поттер (Eric Potter), директор по техническому обучению

Эта книга содержит теоретические основы и практические рекомендации по созданию и использованию распространенных технологий API. Она будет полезна как для новичков, так и для специалистов.

Карандип Джохар (Karandeep Johar), менеджер по разработке ПО, Amplitude

---

[1] *Ричардс М., Форд Н.* Фундаментальный подход к программной архитектуре. 2-е межд. изд. — Спринт Бук, 2026.

Предисловие

Производительность программной системы зависит не только от состава ее компонентов, но и от того, как они работают вместе. Обеспечение их эффективного взаимодействия, например, через API требует продуманного проектирования и управления.

Разобраться в деталях проектирования и реализации таких систем поможет книга «Стили API. Проектирование и внедрение» — информативное и структурированное руководство, позволяющее учиться легко и эффективно.

Книга начинается с основ — в главе 1 читатель знакомится с такими ключевыми понятиями API, как жизненный цикл API. Глава 2, посвященная паттернам проектирования, обязательна для изучения всеми, кто участвует в проектировании и реализации API. Далее рассматриваются сетевые и веб-протоколы (главы 3 и 4 соответственно), после чего авторы переходят к анализу стилей API, основанных на этих протоколах (глава 5 и последующие).

Сквозная тема книги — безопасность API. В главе 1 Лукаш и Марцин справедливо отмечают: «Безопасность — это универсальный принцип, применимый ко всем этапам жизненного цикла API». Если игнорировать требования безопасности на любом из этапов, то API становится легкой мишенью для злоумышленников.

В новостях регулярно появляются сообщения о том, как государственные органы предоставили доступ к персональным документам по одному лишь номеру телефона. API часто становится узким местом, где множество компаний теряют конфиденциальные данные. Продуманные и грамотно реализованные алгоритмы безопасности — необходимое условие для того, чтобы не стать печально известным героем новостных заголовков. Тема безопасности — лейтмотив этой книги; аналогичным образом безопасность должна красной нитью проходить через все этапы жизненного цикла вашего API.

Помимо теории, в книге вы найдете полезные практические упражнения. Выполняя их, вы потренируетесь в реализации разных стилей API и, конечно же, в обеспечении их безопасности. Практика — двигатель прогресса.

Выполняя упражнения, вы получите более глубокое понимание того, как работают стили API и какие у них преимущества и недостатки.

Концепции, представленные в этой книге, необходимо освоить всем, кто собирается проектировать и реализовывать API. Вот лишь некоторые основные идеи.

Универсальных решений нет, поэтому имеются разные стили API. Описанные в книге плюсы и минусы каждого стиля позволят вам сравнить разные подходы и выбрать наиболее подходящий.

В книге *Communication Patterns*[1] я говорила о необходимости учитывать потребности вашей аудитории. У любого API тоже есть аудитория с определенными потребностями — будь то сотрудники вашей организации или сторонние пользователи. Важным этапом в эволюции представлений об API стал переход от концепции *API как связующего звена* к концепции *API как продукта*. Понимание того, что API имеет целевую аудиторию и может приносить доход, значительно повысило его ценность. Эта книга призвана помочь вам эффективно инвестировать в разработку API.

API обеспечивает инкапсуляцию: он скрывает сложные структуры данных и алгоритмы реализации функций за удобным интерфейсом. Пользователю API не нужно знать, как обрабатываются его запросы, — достаточно понимать, как их формулировать и что можно получить в ответ. Подобно тому как алгоритмы безопасности предотвращают утечки данных, проектирование API предотвращает утечки информации о деталях реализации системы. Говорите на языке ваших клиентов: имена и структуры данных должны соответствовать их ожиданиям. Уделите особое внимание паттернам именования API и другим паттернам проектирования, описанным в главе 2.

Еще одно важное понятие — жизненный цикл API. Он не последовательный, а итеративный: каждый этап цикла связан с другими, и обратная связь, полученная при реализации одного из этапов, влияет на все остальные. Нет необходимости завершать один этап, чтобы перейти к следующему. *Жизненный цикл API не предполагает детального проектирования до начала разработки*. Достаточно спроектировать лишь то, что необходимо реализовать в данный момент, а затем можно использовать такой подход снова и снова. Рассматривайте жизненный цикл API как серию вертикальных срезов — это позволит быстро реагировать на изменения. При чтении главы 1 рекомендую ознакомиться с итеративной моделью разработки ACED (https://oreil.ly/grxkb).

---

[1] *Рид Д.* Паттерны коммуникации: руководство для ИТ-разработчиков и архитекторов. — Астана, 2025.

Фундаментальные знания, полученные из книги, позволят вам создавать программные системы, компоненты которых эффективно взаимодействуют как между собой, так и с другими системами.

Желаю успехов в изучении стилей API и создании собственных API-продуктов.

Джеки Рид (Jacqui Read),
автор книги Communication Patterns,
ведущий архитектор компании The Ministry of Software Design,
соорганизатор сообщества Domain-Driven Design (DDD), Лондон,
Уинчестер, Великобритания, июнь 2025 года

Вступление

Осознаем мы это или нет, но API есть везде. Без API наши смартфоны не смогли бы получать актуальный прогноз погоды, а наземный центр управления полетами не смог бы общаться с экипажем или с активными спутниками на орбите Земли через сеть дальней космической связи NASA.

API приводят к масштабным изменениям, например, в финансовой отрасли. Сегодня, вместо того чтобы лично идти в банк, мы просто звоним туда или заходим в приложение на смартфоне.

Крупные технологические компании, такие как Amazon, Google, Microsoft, OpenAI, Revolut и Salesforce, тоже обязаны своим успехом API: без API получать большую прибыль через цифровые каналы было бы невозможно. Иными словами, API — это «русла», по которым текут данные, причем некоторое ПО, например Kubernetes, без API вообще бы не работало.

Термин *API* (application programming interface — «программный интерфейс приложения») в зависимости от контекста интерпретируется по-разному. В основном он связан с приложениями, но иногда и с «железом». В прикладном контексте API может обозначать интерфейс библиотеки или сетевой интерфейс (чаще всего).

Мы определяем API как точку взаимодействия, которая позволяет программным компонентам обмениваться данными. API дает возможность пользоваться функциями системы, не раскрывая того, что у нее «под капотом», и отделяет поведение системы от деталей ее реализации. API используют прежде всего разработчики: они интегрируют API в приложения, чтобы обеспечить взаимодействие внутри систем или между ними. Кроме того, API разделяет зоны ответственности, упрощая сопровождение: то, что находится за API, — зона ответственности предоставившей его системы.

За последние десятилетия мы наблюдали становление и расширение интернета. Он представляет собой глобальную сеть взаимосвязанных компьютерных систем, которая позволяет устройствам по всему миру обмениваться данными. Этот обмен основан на стеке протоколов TCP/IP (https://

oreil.ly/9kdrm) — «языке» сети, с помощью которого приложения и устройства передают информацию.

Эта книга посвящена сетевым API — группе API, которые используются приложениями для обмена данными по сети (обычно через интернет). API можно представить как общий язык приложений, регулируемый протоколами, правилами и соглашениями. Подобно человеку, говорящему на нескольких языках, приложение может общаться с разными API.

Почему мы написали эту книгу

В своей практике мы работали с разными API: REST, Query, RPC, брокерскими системами и др. У каждого стиля API[1] своя экосистема: сообщество, рабочие группы, а иногда и поддержка интернет-гигантов. На изучение литературы, посвященной одному конкретному стилю, потребуется много времени, однако сегодня уже нельзя ограничиваться одним стилем API. Конечно, возможности REST API велики, но не безграничны.

Эта книга заполняет пробел в ИТ-литературе, поскольку детального обзора разных архитектурных стилей современные книги по API не дают. Многие из них посвящены отдельным сложным случаям реализации API или содержат фрагменты кода без полной процедуры реализации.

За последние годы в ИТ-сфере возрос интерес к навыкам работы с API. В 2023 году компания Gartner представила отчет *API Trends* («Тренды API») (https://oreil.ly/Ody1W), в котором были озвучены ключевые проблемы организаций: дефицит API-инженеров, API-стандартов и навыков работы с API, а также проблемы безопасности. В 2024 году представители Gartner предположили, что к 2026 году спрос на API увеличится на 30 %, преимущественно благодаря популяризации ИИ и больших языковых моделей (large language models, LLM)[2].

Надеемся, что эта книга поможет решить часть озвученных проблем и закрыть некоторые из упомянутых потребностей.

---

[1] Под стилем понимается набор характеристик. См. раздел «Что такое стили API» главы 1, в котором кратко описаны все стили API, обсуждаемые в книге.

[2] Gartner: Gartner Predicts More Than 30 % of the Increase in Demand for APIs will Come From AI and Tools Using Large Language Models by 2026 («К 2026 году спрос на API возрастет более чем на 30 % благодаря бурному росту ИИ и LLM»), https://oreil.ly/whm4x, пресс-релиз, 20.03.2024.

Для кого эта книга

Книга написана для разработчиков и архитекторов ПО, которые хотят разобраться в плюсах и минусах, а также подробностях реализации разных стилей API. По каждому стилю мы рассмотрим детали, архитектуру и роль сетевых протоколов. Книга подойдет желающим:

- получить практические навыки проектирования и безопасной реализации API;
- узнать плюсы и минусы разных стилей API;
- изучить протоколы, реализующие обмен данными в API.

Что в книге есть

Вы научитесь проектировать и реализовывать синхронные и асинхронные API в разных стилях: REST API (глава 5), Query API с GraphQL (глава 6), Web Feeds API с протоколом Atom (глава 7), RPC API с фреймворком gRPC (глава 8), callback API с вебхуками (глава 9), двунаправленный API с WebSocket (глава 10) и брокерский API с RabbitMQ (глава 11).

Каждый стиль обладает важными для разработчиков и архитекторов достоинствами и недостатками. Вдобавок мы обсудим часто упускаемую из виду роль сетевых протоколов в сетевых API.

Чего в книге нет

Эта книга предназначена для технических специалистов. Если вас интересуют связанные с разработкой API вопросы лидерства, стратегий, управления, комплаенса и администрирования, то рекомендуем прочесть книги *APIs: A Strategy Guide* Дэниела Джейкобсона и др. (Daniel Jacobson et al.) (O'Reilly, 2011) и *Continuous API Management* Мехди Меджауи и др. (Mehdi Medjaoui et al., O'Reilly, 2021)[1].

Кроме того, здесь не рассматриваются вопросы, связанные с архитектурой, платформами и монетизацией API. Информацию по этим темам вы можете найти в книгах *Mastering API Architecture* Джеймса Гофа и др. (James Gough et al., O'Reilly, 2022)[2] и *API Management: An Architect's Guide to Developing and Managing APIs for Your Organization* Браджеша Де (Brajesh De, Apress, 2017).

---

[1] *Меджауи М. и др.* Непрерывное развитие API. Правильные решения в изменчивом технологическом ландшафте. 2-е изд. — СПб.: Питер, 2023.

[2] *Гоф Дж. и др.* Проектирование архитектуры API. — СПб., 2024.

Сервис прогноза погоды

Разные стили API мы будем рассматривать на интуитивно понятном примере — реализации *сервиса прогноза погоды* (Weather Forecast Service, WFS). Его архитектура представлена на рис. В.1 в виде модели C4 (https://c4model.com).

WFS реализован на языке программирования Python с использованием фреймворков Django (https://www.djangoproject.com) и Starlette (https://www.starlette.io). Сервис предоставляет два типа информации: прогноз погоды на ближайшие семь дней и архивные записи. Получать данные можно через REST, GraphQL и Atom-ленты. Кроме того, в сервисе есть система оповещений, настраиваемая через WebSocket, и функция обогащения данных через LLM, реализуемая с помощью gRPC. Однако для реализации WFS подходят не все стили API, описанные в книге, поэтому для вебхуков и RabbitMQ мы приводим другие примеры.

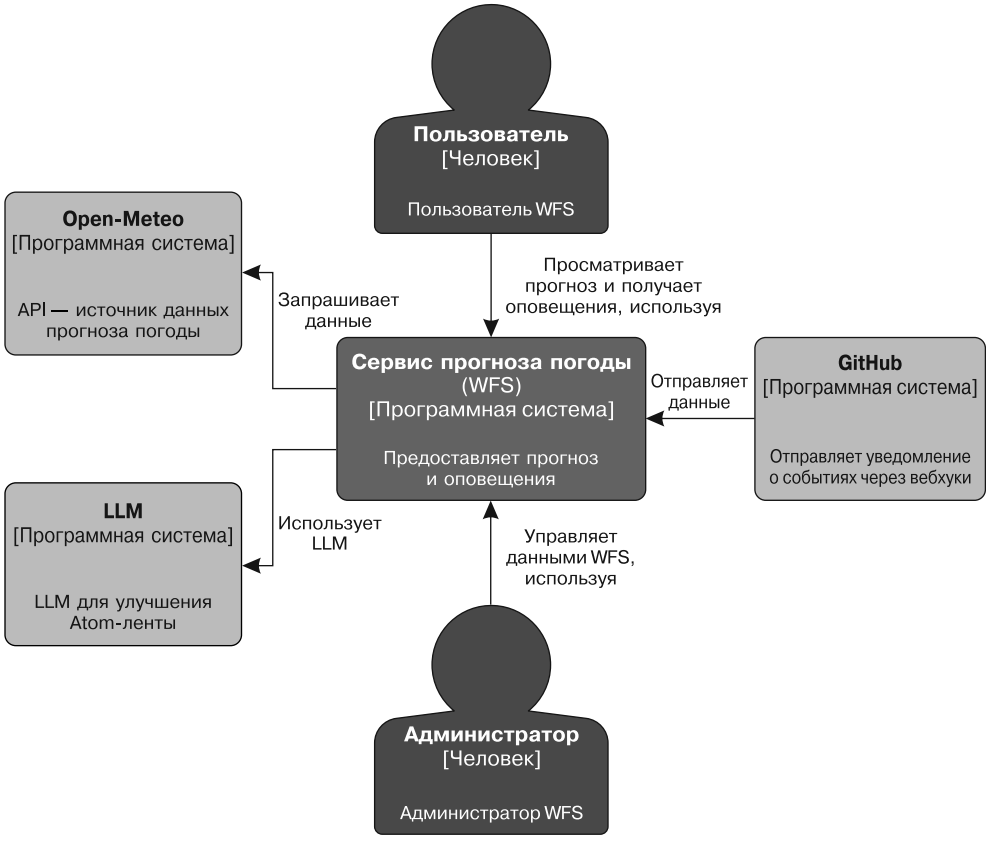


Рис. В.1. Архитектура сервиса прогноза погоды в виде модели C4

В 2021 году GitHub выпустил Codespaces — облачную среду разработки, бесплатную для всех пользователей GitHub. На ее основе мы создали среду для запуска примеров из этой книги. Поскольку она основана на Docker, то код можно выполнять локально, если платформа поддерживает контейнеры Docker.

С кодом проще всего работать через Codespaces, но можно запускать его и локально, как показано далее в примере B.1.



Среда Codespaces бесплатна, время использования ограничено (на момент публикации книги — до 60 часов в месяц). Чтобы не выходить за рамки лимита, останавливайте среду после завершения работы.

Для работы с GitHub Codespaces нужна учетная запись GitHub, которую можно создать через форму регистрации (https://oreil.ly/b1dhe).

Далее создайте форк репозитория с кодом из книги. Перейдите по адресу https://github.com/ldynia/learning-api-styles и нажмите кнопку Fork (рис. B.2).

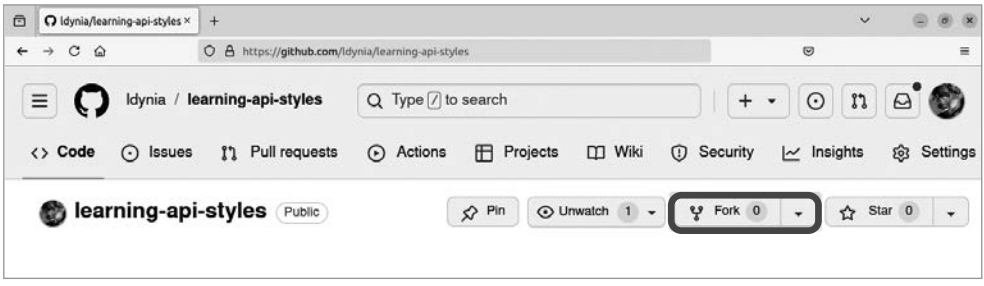


Рис. B.2. Репозиторий с кодом книги и расположение кнопки Fork

Запустить среду можно тремя способами: через браузер в GitHub Codespaces, в локальной *интегрированной среде разработки* (IDE), поддерживающей CodeSpaces, или локально с помощью Docker.

Самый простой вариант — использовать браузер. В форке репозитория нажмите зеленую кнопку Code, перейдите на вкладку Codespaces и выберите Create codespace on main.

В браузере откроется Visual Studio Code (https://code.visualstudio.com) (рис. B.3). Настройка завершена, можно запускать примеры.



Для наиболее эффективной работы с GitHub Codespaces через браузер рекомендуем Google Chrome (https://www.google.com/chrome).

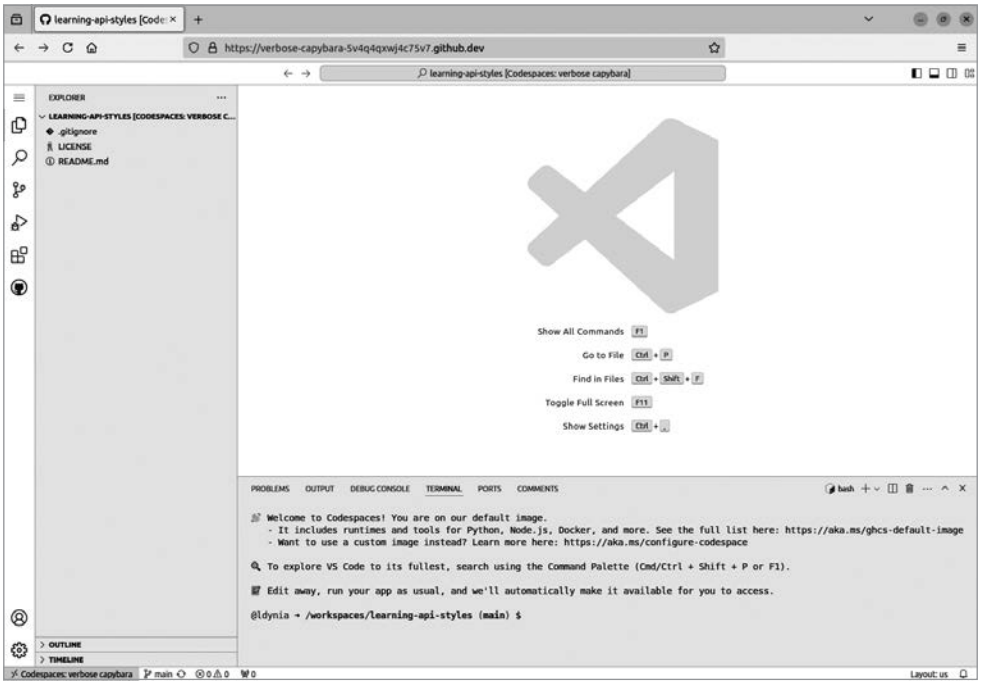


Рис. B.3. Форк репозитория с кодом книги, открытый удаленно в GitHub Codespaces

Другой способ — использовать GitHub Codespaces через локальную IDE. Большинство сред разработки поддерживают эту функцию. Например, в локально установленном VS Code нужно нажать кнопку Open a Remote Window (в левом нижнем углу), выбрать Connect to Codespace и подключиться к нужному Codespace. Этот процесс показан на рис. B.4. После подключения среда готова к работе.

Для локальной работы можно использовать копию репозитория с кодом книги или форк. В этом случае потребуется вручную настроить окружение: установить IDE, Docker (https://www.docker.com) и Docker Compose (https://oreil.ly/aKCLA). В примере B.1 показано, как клонировать репозиторий.

Пример B.1. Локальное клонирование репозитория с кодом книги

```
git clone https://github.com/ldynia/learning-api-styles  ❶

code learning-api-styles  ❷
```

❶ Клонируем репозиторий с кодом книги (можно использовать форк).

❷ Открываем репозиторий в VS Code.

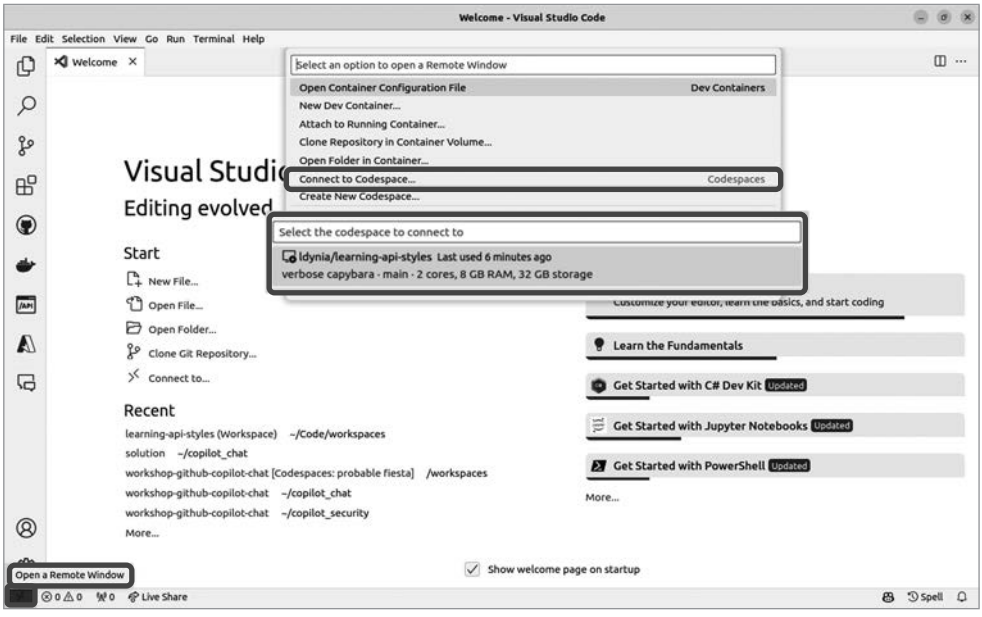


Рис. B.4. Подключение к GitHub Codespaces через VS Code

Далее запускаем контейнер разработки[1] (рис. B.5): нажимаем кнопку Open a Remote Window (в левом нижнем углу), выбираем Reopen in Container. Система соберет Docker-образ и подготовит dev container. Локальная среда готова к работе.

Для работы с примерами часто требуется несколько терминальных окон или панелей, поэтому рекомендуем установить программу `tmux` (https://oreil.ly/6N4CI). Команда из примера B.2 создает сессию `tmux` с тремя панелями.

Пример B.2. Сессия tmux с тремя панелями

```
tmux new-session \; split-window -v \; split-window -h \; select-pane -U \;  ❶
```

❶ Команда `tmux new-session` создает рабочее пространство с несколькими панелями. `split-window -v` делит панель по вертикали, `split-window -h` — по горизонтали, а `select-pane -U` выбирает верхнюю панель. Обратные слеши `\;` используются для разделения команд `tmux` и последовательного их выполнения. Это гарантирует, что они будут выполняться в правильном порядке, в результате чего создается сессия `tmux` с нужной структурой панели.

[1] Контейнеры позволяют настроить среду разработки и запустить ее локально или удаленно. Контейнеры разработки поддерживаются в разных IDE, например VS Code, в котором есть руководство для пользователя (https://oreil.ly/uWysc).

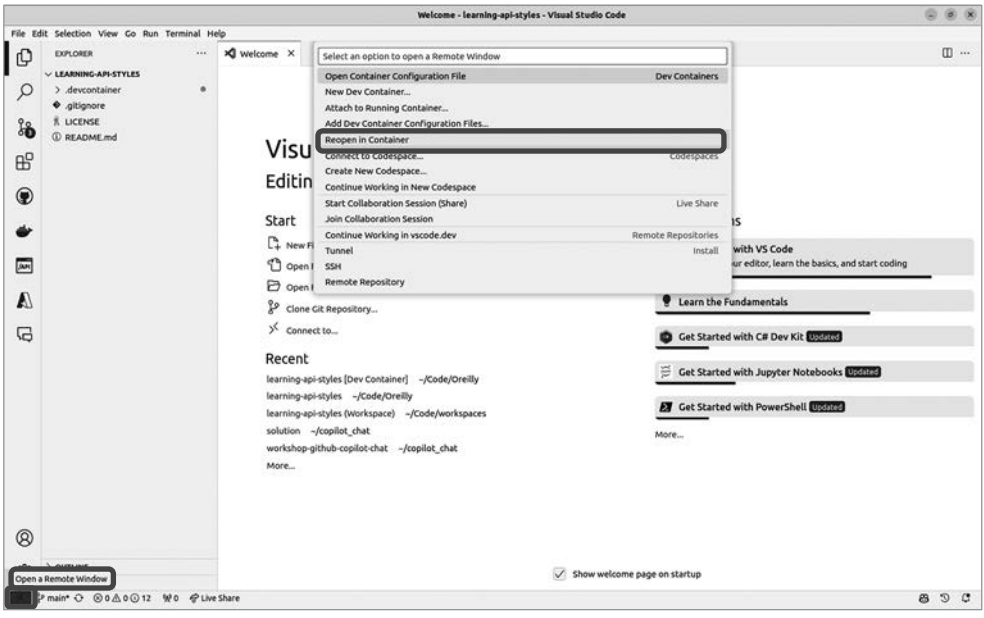


Рис. B.5. Локальная копия репозитория с кодом книги в VS Code

На рис. B.6 представлен терминал, разделенный на три панели.

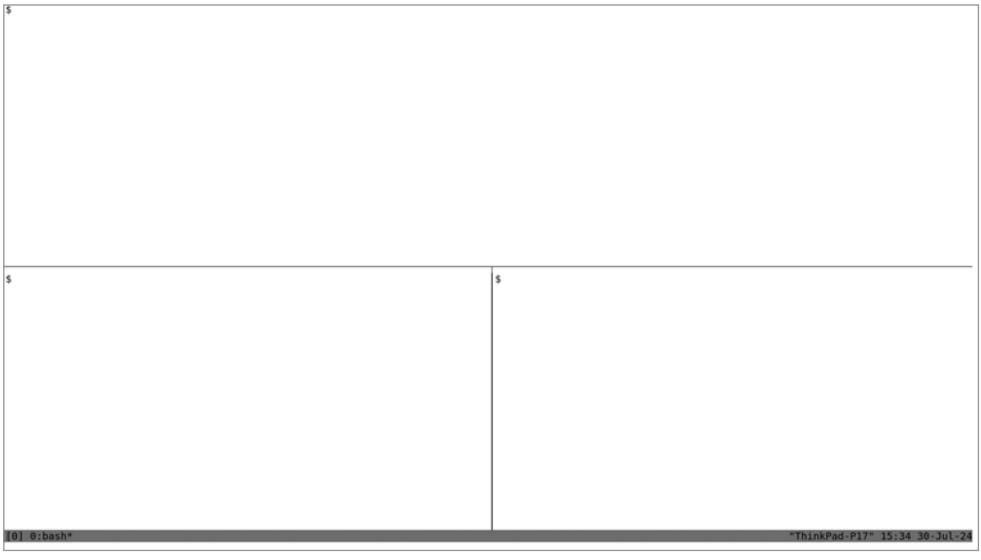


Рис. B.6. Сессия tmux с тремя панелями

Для управления панелями используются следующие горячие клавиши `tmux`.

Ctrl+B Shift+"
: Разделить панель по горизонтали.

Ctrl+B Shift+%
: Разделить панель по вертикали.

Ctrl+B+↑ ↓ ← →
: Переключиться между панелями (вверх, вниз, влево, вправо).

Ctrl+B Ctrl+↑ ↓ ← →
: Изменить размер панели (верхней, нижней, левой или правой). Можно также использовать мышь.

Ctrl+B X Y
: Закрыть панель. (Или введите команду `exit`.)

Если вы хотите глубже разобраться в `tmux`, рекомендуем посмотреть обучающее видео на YouTube (https://oreil.ly/upurq) или воспользоваться памяткой по горячим клавишам (https://tmuxcheatsheet.com).

Условные обозначения

В книге используются следующие условные обозначения.

Курсив
: Курсивом выделены новые термины и важные понятия.

`Моноширинный шрифт`
: Используется для листингов программ, а также внутри абзацев для обозначения таких элементов, как переменные и функции, базы данных, типы данных, переменные окружения, операторы и ключевые слова, пути, имена файлов и их расширений, названия папок и каталогов.

`Жирный моноширинный шрифт`
: Показывает команды или другой текст, который пользователь должен ввести самостоятельно.

`Курсивный моноширинный шрифт`
: Используется для оформления текста, который должен быть заменен значениями, предоставляемыми пользователем или определяемыми контекстом.

Шрифт без засечек
: Используется для обозначения элементов интерфейса, URL и адресов электронной почты, названий клавиш.



Так оформляется совет или предложение.



Так оформляется примечание.



Так оформляется предупреждение.

Использование примеров кода

Примеры кода, упражнения и другие дополнительные материалы доступны для скачивания по адресу https://learningapistyles.com.

При технических вопросах или проблемах с использованием примеров кода пишите на support@oreilly.com.

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

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

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

Благодарности

Написание книги похоже на выращивание дерева. Дерево — прекрасная метафора процесса формирования нового знания: если садовник терпеливо ухаживает за деревом, то оно вырастает и его плоды приносят пользу окружающим. Мы благодарим всех, кто участвовал в создании этой книги и помогал дереву расти.

В первую очередь авторы выражают благодарность людям, которые сделали их встречу возможной: Троэльсу Равену Бэрентцену (Troels Ravn Bærentzen) и Александру Зафирову (Alexander Zafirov).

Особая благодарность техническим рецензентам, чьи замечания помогли дереву обрести новые ветви: Эрику Поттеру (Eric Potter), Карамдипу Джохару (Karamdeep Johar) и другим, чьи инициалы мы не смогли расшифровать. Отдельное спасибо Маркосу Фуэнтесу (Marcos Fuentes) и Уильяму Джамиру Силве (William Jamir Silva): ваши правки потребовали серьезной работы, но именно благодаря им дерево расцвело.

Благодарим Мишель Кронин (Michele Cronin), которая терпела нас эти два года, помогала преодолевать трудности и связывала нас с нужными людьми. Спасибо Клэр Лейлок (Clare Laylock) за редактуру, которая началась с 3345 комментариев. Бен Херст (Ben Hurst), спасибо за корректуру (обрезку дерева)! И наконец, мы благодарим команду O'Reilly, работавшую над книгой, в том числе художников и других специалистов, с которыми мы никогда не пересекались.

От Лукаша Дыновски

Хочу поблагодарить жену Джульетту и детей Камиллу и Филиппа за поддержку, которая помогла мне полностью сосредоточиться на книге. Я никогда не написал бы ее без вашей помощи. Я должен попросить у вас прощения за то, что уделял вам так мало внимания и на время написания книги стал настоящим отшельником. Прошу прощения у детей за то, что последними забирал их из школы и детского сада, и у друзей за редкие встречи. Я обещаю вернуть вам это упущенное время.

Благодарю Луизу Корриган (Louise Corrigan), которая неожиданно отправила мне письмо, ставшее семенем нашего будущего дерева.

За вступительное слово благодарю Джеки Рид — мою соперницу в прошлом и союзницу в настоящем. Увидев Джеки на архитектурных ката от O'Reilly, я понял, что мир программной архитектуры еще не раз услышит о ней. Недавно Джеки выпустила книгу «Паттерны коммуникации» (https://communicationpatternsbook.com), в которой среди прочего учит читателей созданию диаграмм — навыку, который необходимо освоить многим из нас.

Дереву нужна почва для роста. Благодарю Марка Ричардса (Mark Richards) за поддержку и регулярную публикацию полезного и вдохновляющего контента — например, за выходящие по понедельникам уроки по архитектуре ПО (https://www.developertoarchitect.com/lessons).

Спасибо «Монти Пайтон», Тиму Бертону, Pink Floyd, Жану Ван Хамме (Jean Van Hamme) и Гжегожу Росински (Grzegorz Rosiński) за «Торгала» и за «лучших людей» из «Индианы Джонса». Я не совсем понимаю, почему упомянул их, но мне кажется, что это правильно.

Последняя благодарность — вам, читатели. Если вам знакомо выражение «одержимость клиентами», то знайте: два года нашей жизни были посвящены «одержимости читателем». Живите долго и создавайте пул-реквесты.

От Марцина Дулака

Благодарю Алекса Шуляйта (Alex Schuleit) за возможность поработать над этой книгой. Спасибо Руфусу и другим меньшим братьям, которые, оккупируя клавиатуру или прося еду, вынуждали меня делать перерывы.

От издательства

Мы выражаем огромную благодарность клубу рецензентов ИТ-литературы ReadIT Club за помощь в работе над русскоязычным изданием книги и их вклад в повышение качества переводной литературы.

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

Мы будем рады узнать ваше мнение!

ГЛАВА 1

Основы API

Термин *API* в английском языке расшифровывается как *application programming interface* и означает программный интерфейс приложения. В этой книге речь пойдет о сетевых API. В главе 1 мы представим историю API и дадим краткий обзор всех видов API, которые рассмотрим в книге. Далее сравним их по критериям, определяющим стиль API.

Кроме того, мы изучим концепции, общие для всех сетевых API, и цели создания API, в том числе их использование в качестве продуктов и монетизацию. Мы разберем все этапы жизненного цикла разработки API, а также принципы управления и администрирования. Понимание этих этапов поможет вам выбрать API, соответствующий вашим потребностям.

Прежде чем перейти к конкретным видам API, обсудим ключевые понятия и историю создания таких интерфейсов.

Что такое API

В термине *application programming interface* слово *interface* означает точку взаимодействия, где сущности обмениваются данными. Сущностью может быть пользователь, система или организация.

На рис. 1.1 показан пример взаимодействия между сущностями. На электронной микрофотографии игла видно, как взаимодействует с поверхностью виниловой пластинки. Дорожки — это физическое воплощение звуковой волны, выгравированной на виниле. Сила воздействует на иглу и удерживает ее в дорожке, а при вращении пластинки мы слышим результат этого взаимодействия.

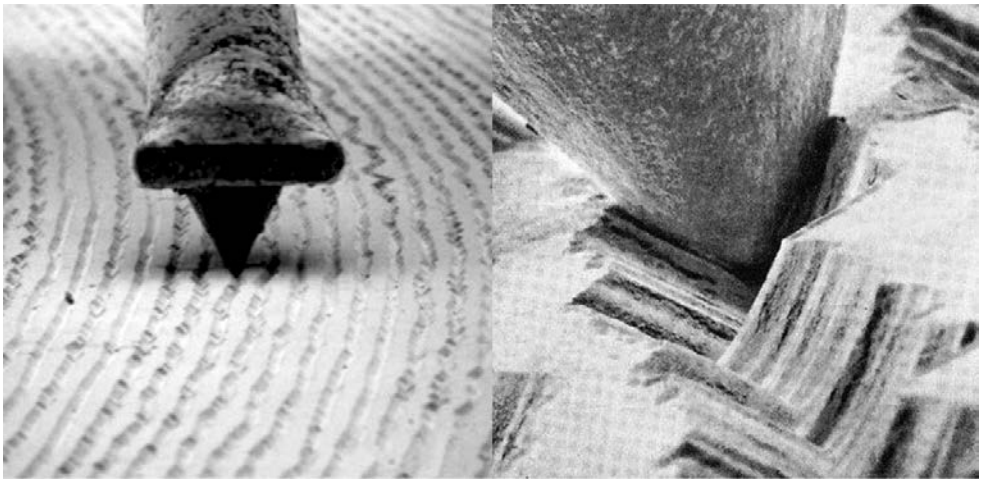


Рис. 1.1. Электронная микрофотография иглы и виниловой пластинки[1]

Вот другие примеры интерфейсов, с которыми мы взаимодействуем ежедневно.

Устройство для взаимодействия человека с компьютерной системой (human interface device, HID)

Устройство, позволяющее человеку взаимодействовать с другим устройством и управлять им, например пульт от телевизора или компьютерная мышь.

Интерфейс командной строки (command-line interface, CLI)

Интерфейс, получающий команды от пользователя в виде текстовых строк и передающий их в компьютерную систему.

Графический пользовательский интерфейс (graphical user interface, GUI)

Интерфейс, позволяющий пользователям взаимодействовать с компьютерными системами через графические элементы.

Чат-интерфейс

Интерфейс приложения для общения через чат.

Интерфейс «мозг — компьютер» (brain-computer interface, BCI)

Интерфейс, обеспечивающий прямое взаимодействие между мозгом и внешним устройством. Известен также как интерфейс «мозг — машина».

---

[1] Изображение взято из поста Electron Microscope Slow-Motion Video of Vinyl LP («Виниловая пластинка: замедленное видео с электронного микроскопа») в блоге Бена Краснова (Ben Krasnow, https://oreil.ly/uxWIh).

API встречаются в разных компьютерных системах.

Библиотеки или фреймворки

Java Platform, Standard Edition и Java Development Kit (JDK, https://oreil.ly/aMYLg) — пример библиотеки, которая предоставляет API для взаимодействия с операционной системой, сетью и другими компонентами; Flask API (https://oreil.ly/OiLFi) — пример веб-фреймворка на Python.

Операционные системы

Portable Operating System Interface (POSIX) — набор стандартов, разработанных IEEE Computer Society (IEEE CS). POSIX унифицирует API для обеспечения совместимости между операционными системами (Unix-подобными и др.).

Сетевые приложения

API этой категории предназначены для приложений, взаимодействующих по компьютерной сети. Типичные интерфейсы здесь — *веб-API* (web API), предоставляемые веб-сервером и используемые браузером. Для взаимодействия с сервером браузер чаще всего задействует веб-API JavaScript[1]. Однако есть приложения с сетевыми API, которые работают и без браузера, например инструменты командной строки, мобильные и настольные приложения.

Таким образом, API используются в разных типах компьютерных систем. Они предоставляют конкретные функции, чтобы пользователям не приходилось реализовывать их с нуля, и скрывают сложность системы за интерфейсом. Для пользователя API — что-то вроде «черного ящика». Следуя документации, мы подаем данные на вход и получаем результат на выходе (иногда вместе с «побочными продуктами» взаимодействия).

Сетевые API

Сетевые API, которым посвящена книга, — это точки взаимодействия, позволяющие программам обмениваться данными по сети (обычно через интернет).



Большинство API, описанных в книге, относятся к категории веб-API. API основаны на веб-технологиях — браузерах и протоколах наподобие HTTP (подробнее об HTTP см. в главе 4). Однако термин «веб-API» обозначает лишь те API, которые построены на веб-архитектуре и веб-технологиях, поэтому мы будем пользоваться более широким термином — «сетевые API». Каждый раз, когда в книге встречается термин API, речь идет именно о сетевых API, если не указано иное.

[1] Спецификация веб-API JavaScript доступна на https://oreil.ly/LVCGP.

Теперь разберем аббревиатуру API в контексте сетевых приложений. На рис. 1.2 показаны компоненты стандартного веб-приложения с архитектурой «клиент — сервер»[1].

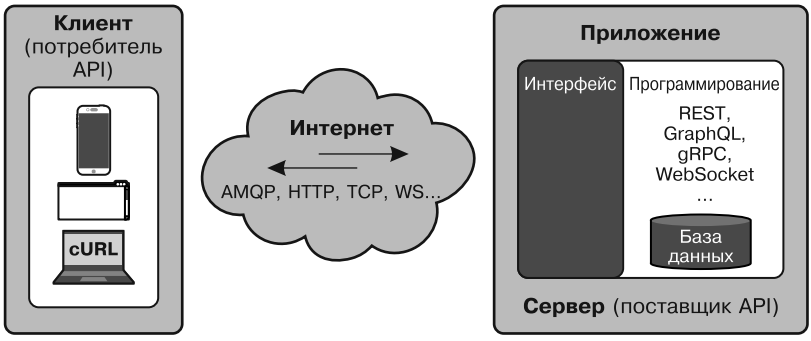


Рис. 1.2. Сетевые API

Проанализируем слова, из которых складывается само название — API.

Приложение (application)

Хотя слово «приложение» на рис. 1.2 появляется только на стороне сервера, оно есть и на стороне клиента. Серверное приложение — это «черный ящик», скрывающий детали реализации. Мы знаем, что входные данные попадают в него, подвергаются преобразованиям (скрытым от внешнего наблюдателя), а на выходе мы получаем желаемый результат. В контексте веб-API серверное приложение называют *бэкендом* (backend), а клиентское — *фронтендом* (frontend). Бэкенд предоставляет данные, а фронтенд их потребляет.

Программирование (programming)

Программирование — это передача компьютеру инструкций для выполнения задач. Как правило, функциональность серверного приложения программируется клиентским приложением. Поэтому программирование происходит и на стороне клиента, и на стороне сервера, хотя на рис. 1.2 термин появляется только на стороне сервера. Программирование (реализацию) функциональности на фронтенде и бэкенде выполняет разработчик, используя соответствующие инструменты и библиотеки.

Интерфейс (interface)

Интерфейс позволяет системам взаимодействовать и обмениваться данными. Это точка взаимодействия, где клиентское приложение начинает использовать функциональность, предоставленную серверным приложением.

---

[1] Определения терминов «клиент» и «сервер» даны в разделе «Socket API» главы 3.

Наглядно представить сетевой API поможет такая аналогия: многополосный платный мост, соединяющий два города, по которому едут грузовики (рис. 1.3).

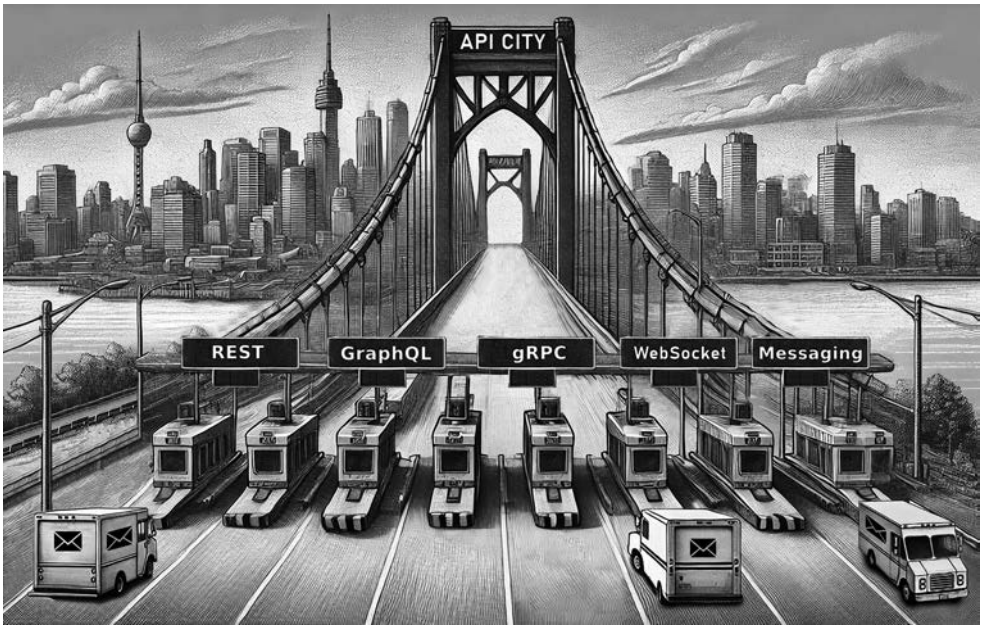


Рис. 1.3. Визуальная аналогия сетевых API[1]

Города по обе стороны моста представляют собой клиентское и серверное приложения. Мост, соединяющий города, — это сеть, по которой передаются данные.

Пункт оплаты, через который грузовик въезжает на мост, — стиль API. Грузовик с грузом (сообщением), въезжающий на мост или покидающий его, — это запрос или ответ. Адрес на грузе соответствует эндпоинту API. Грузовик и его груз проходят проверку в пункте оплаты: это гарантирует, что запрос и ответ корректно отформатированы и авторизованы. После проверки грузовик въезжает на мост и двигается в соответствии с правилами дорожного движения (они определяются сетевыми протоколами, о которых речь пойдет в главе 3).

Эта аналогия подойдет также для описания синхронного и асинхронного типов связи (см. подраздел «Синхронная и асинхронная коммуникация» далее в этой главе). В качестве примера рассмотрим процесс доставки груза. При синхронной доставке сообщений грузовик доставляет груз и ожидает квитанцию

[1] Изображение сгенерировано моделью DALL·E от OpenAI.

о получении (подтверждение того, что груз был получен). При асинхронной доставке грузовик привозит груз и уезжает. Квитанция о получении отправляется позже на другом грузовике.

Взаимодействие через API: ключевые понятия

Прежде чем переходить к собственно сетевым API, рассмотрим понятия, связанные с обменом данными по сети: сообщение, режимы передачи, а также синхронный и асинхронный типы связи.

Сообщение

Сообщение — это дискретная единица данных (то есть самостоятельная запись), которую отправитель передает одному или нескольким получателям. Оно обычно содержит полезную нагрузку (payload) и метаданные.

В модели взаимодействия открытых систем (Open Systems Interconnection, OSI) термин «сообщение» связывают с протоколом прикладного уровня (см. раздел «TCP/IP и модель OSI» главы 3). Рассмотрим несколько примеров сообщений прикладного уровня.

HTTP

В примере 1.1 показано стандартное сообщение HTTP/1.1 `POST` (подробнее об HTTP см. в главе 4). Оно содержит строку запроса `POST`, за которой следуют три заголовка и тело сообщения (`comment=Hello`).

Пример 1.1. HTTP-сообщение, передаваемое запросом POST

```
POST /submit HTTP/1.1
Host: example.com
Content-Type: application/x-www-form-urlencoded
Content-Length: 13

comment=Hello
```

XMPP

В примере 1.2 приводится сообщение в формате XML для протокола XMPP (Extensible Messaging and Presence Protocol) (https://oreil.ly/dCqrv), которое входит в XMPP-поток. Обратите внимание, как XML-атрибуты используются для описания метаданных сообщения.

Пример 1.2. Сообщение в XMPP-потоке

```
<?xml version='1.0'?>
<stream:stream ...>
  <message from='sender@example.com' to='receiver@example.net' xml:lang='en'>
```

```
    <body>Hello</body>
  </message>
</stream:stream>
```

MQTT

В примере 1.3 показано двоичное сообщение, отправленное по протоколу MQTT (Message Queuing Telemetry Transport) (https://oreil.ly/5K2et). В MQTT сообщение называется пакетом и состоит из фиксированного заголовка, необязательного переменного заголовка и полезной нагрузки. Сообщение в примере — это управляющий пакет (команда `PUBLISH`). Переменный заголовок содержит имя топика `sensor/temperature`, а полезная нагрузка — данные `25.5 C`.

Пример 1.3. Структура пакета PUBLISH в MQTT

```
Fixed Header:     PUBLISH ...
Variable Header:  sensor/temperature
Payload:          25.5 C
```

Режимы передачи

Термин «режим передачи» (transmission mode или data transmission mode) пришел из области телекоммуникаций и описывает, как взаимодействующие стороны обмениваются данными. На рис. 1.4 показаны три режима, которые используются для передачи данных между взаимодействующими сервисами или устройствами. Каждая сторона может находиться в одном из трех состояний: *передача*, *прием* или *передача* и *прием* одновременно.

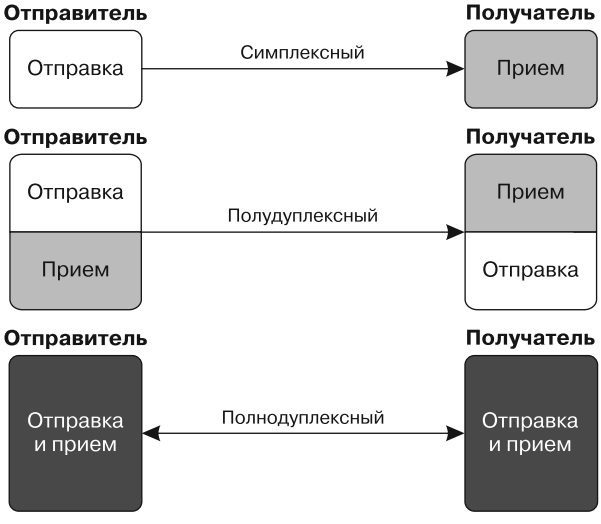


Рис. 1.4. Режимы передачи



Режимы передачи есть не только в API. В любой системе, обменивающейся информацией, — будь то аппаратное устройство или программный компонент — реализуется подходящий режим передачи данных.

Рассмотрим различные режимы передачи.

Симплексный (simplex)

Односторонний режим передачи. Данные передаются только в одном направлении — от отправителя к получателю. Например, радио и телевизор работают в симплексном режиме: данные передаются от устройства к зрителю или слушателю.

Полудуплексный (half-duplex/semi-duplex)

Двусторонний режим передачи, в котором данные передаются в обоих направлениях попеременно: от отправителя к получателю. Пример устройства с полудуплексным режимом — рация: при общении обе стороны соблюдают протокол связи (набор команд наподобие «прием» и «конец связи»).

Полнодуплексный (full-duplex/duplex)

Двусторонний режим передачи, при котором данные передаются одновременно в обоих направлениях[1]. Пример устройства, работающего в таком режиме, — телефон: при разговоре мы можем одновременно говорить и слышать собеседника.

Синхронная и асинхронная коммуникация

Термин «синхронный» (synchronous) происходит от греческих слов *syn* — «вместе» и *chronos* — «время». Поэтому *синхронная коммуникация*, в отличие от *асинхронной*, предполагает синхронизацию по времени. Например, в телекоммуникациях при синхронной передаче данных используется общий тактовый сигнал. Он гарантирует, что получатель может принимать и правильно интерпретировать сигнал с частотой, заданной отправителем. Из этого следует, что понятия синхронной и асинхронной коммуникации применимы не только к модели «запрос — ответ».

К синхронной коммуникации относятся телефонные и видеозвонки, где участники взаимодействуют одновременно. Электронная почта и сообщения на форумах — это примеры асинхронной коммуникации, при которой пользователи отправляют и получают сообщения в разное время. При асинхронной коммуникации участники после отправки или получения сообщения могут

[1] Одновременная двунаправленная передача данных также может осуществляться в двойственно-симплексном режиме. В этом режиме используется два симплексных канала связи, в отличие от полнодуплексного режима, в котором используется только один канал.

выполнять другие задачи. В технических терминах это означает, что они могут выполнять *неблокирующую обработку* (nonblocking processing). На рис. 1.5 показано различие между синхронной и асинхронной коммуникацией на примере модели взаимодействия «запрос — ответ».

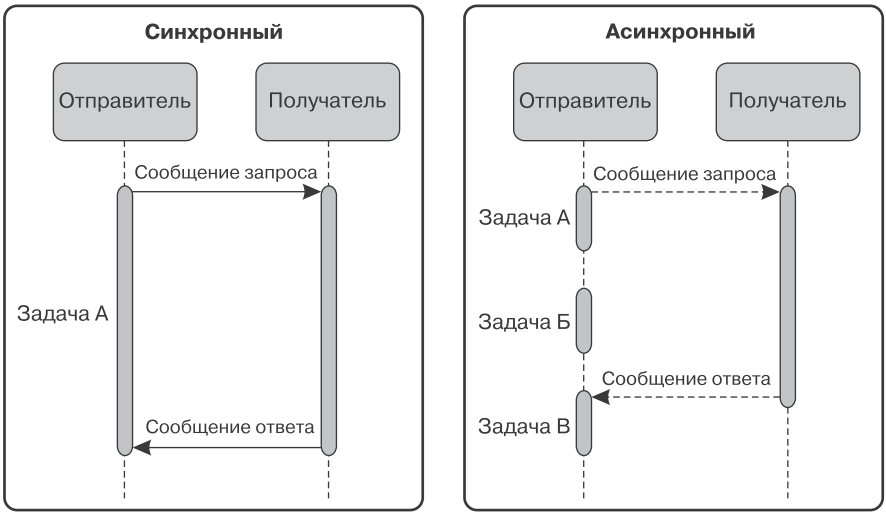


Рис. 1.5. Сравнение синхронного и асинхронного типов коммуникации

При синхронной коммуникации отправитель передает сообщение получателю и остается в заблокированном состоянии до получения ответа. Без ответа отправитель не может продолжить работу или начать выполнять новую задачу. Успешная синхронная коммуникация требует, чтобы обе стороны были доступны и активны во время обмена сообщениями.

При асинхронной коммуникации отправитель передает сообщение получателю и, ожидая ответа, может выполнять другие задачи. В этом случае отправителю и получателю не нужно быть доступными одновременно при обмене сообщениями.

Сетевой протокол может определять, как будет действовать отправитель — синхронно или асинхронно. Если же протокол не содержит таких требований, то отправитель сам решает, какой стиль коммуникации использовать. Например, в HTTP применяется модель «запрос — ответ», где каждому запросу соответствует определенный ответ, поэтому можно предположить, что по умолчанию клиенты, использующие HTTP, общаются синхронно. Однако клиент JavaScript, отправляющий HTTP-запрос с помощью метода `fetch()`, взаимодействует асинхронно, а клиент `curl`, отправляющий такой же HTTP-запрос, — синхронно.

История API

Концепция API появилась задолго до соответствующего термина. Сам термин был введен в 1940-х годах, однако широкое распространение получил лишь в 1960–1970-х[1].

В 1940-х годах британские специалисты в области компьютерных наук Морис Уилкс (Maurice Wilkes), Дэвид Уилер (David Wheeler) и Стэнли Гилл (Stanley Gill) работали над модульной программной библиотекой для одного из первых компьютеров — EDSAC (Electronic Delay Storage Automatic Calculator). Тогда же был создан документ — первая книга по программированию[2], — в котором прозвучала идея использовать подпрограммы в качестве стандартизированных интерфейсов к вычислительным машинам. Понятие «подпрограмма» соотносимо с современным понятием API.

> Чтобы пользоваться подпрограммой, не нужно точно знать, как она устроена и на каких вычислительных процессах основана. Главное условие ее использования — наличие краткой спецификации ее функциональности и способа применения... Чтобы стать частью библиотеки, подпрограмма должна быть разработана в достаточно общей форме, что позволит применять ее в разных контекстах без необходимости внесения внутренних изменений.
>
> *The Preparation of Programs for an Electronic Digital Computer*
> («Подготовка программ для электронного цифрового компьютера», 1957)

На рис. 1.6, *вверху*, — компьютер EDSAC, состоящий из стоек для хранения данных и блоков ввода-вывода. В прямоугольнике справа — перфолента с программой, вставленная в считыватель. Считыватель подключен к телетайпу для вывода результатов. На нижнем изображении оператор (программист) с помощью специальной клавиатуры вбивает команду в перфоленту. Для копирования стандартных подпрограмм из библиотеки используется считыватель (в центре изображения), который управляет клавиатурой и позволяет воспроизводить команды на новой ленте. В прямоугольнике слева — стальной шкаф с библиотекой лент, на которых были вбиты подпрограммы.

Термин *application program interface* без суффикса *-ing* в слове *program* впервые использовали Айра У. Коттон (Ira W. Cotton) и Фрэнк С. Грейторекс (Frank S. Greatorex) в работе *Data Structures and Techniques for Remote Computer*

---

[1] Истории API посвящен доклад Джошуа Блоха (Joshua Bloch) A Brief, Opinionated History of the API («Краткая история API: субъективный взгляд») (https://oreil.ly/hLiqt), представленный на конференции QCon в Нью-Йорке в 2018 году.

[2] *Wilkes M. V., Wheeler D. J., Gill S.* The Preparation of Programs for an Electronic Digital Computer. — Addison-Wesley Press, 1951/1957. https://oreil.ly/49YYe.

Graphics[1] (1968). Термин обозначал взаимодействие между приложением и компьютерной системой. При этом авторы уточняют, что система является аппаратно независимой и может адаптироваться к разным типам устройств при сохранении неизменного программного интерфейса приложения.

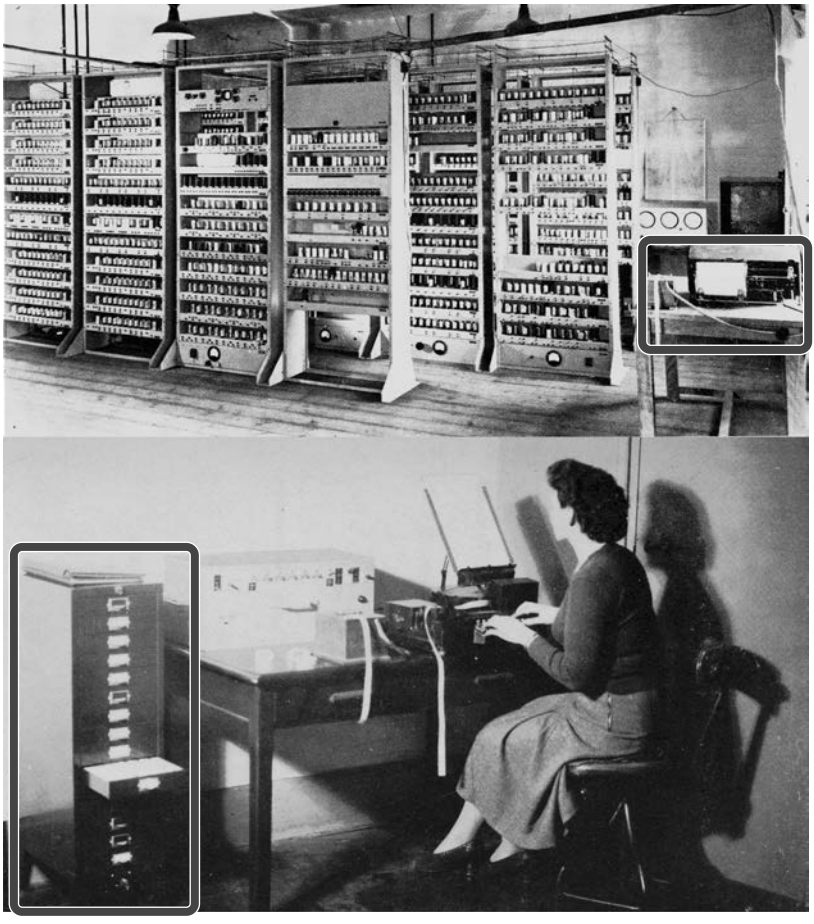


Рис. 1.6. Компьютер EDSAC *(вверху)* и оператор, использующий клавиатуру для перфорации программы на ленту *(внизу)* (1950-е годы)[2]

[1] *Cotton I. W., Greatorex F. S.* Data Structures and Techniques for Remote Computer Graphics («Техники и структуры данных в удаленной графической обработке») (https://oreil.ly/k4fek): материалы конференции AFIPS '68: Fall Joint Computer Conference, 9–11 декабря 1968 года.

[2] Описано в книге The Preparation of Programs for an Electronic Digital Computer (https://oreil.ly/Iw6Do).

По мере распространения компьютерных сетей в 1970-х и 1980-х годах программисты начали обращаться к библиотекам не только на локальных компьютерах, но и на удаленных. Понятие API расширилось с появлением *удаленного вызова процедур* (Remote Procedure Call, RPC) (подробнее см. в разделе «Удаленный вызов процедур» главы 8).

В 1990-х годах компьютеры продолжали работать в разных средах, операционных системах и на разных языках программирования. Тогда же консорциум Object Management Group (OMG) (https://www.omg.org) определил стандарт Common Object Request Broker Architecture (CORBA) (https://www.corba.org). CORBA не зависел от языка программирования и позволял удаленно вызывать объекты и выполнять над ними операции, несмотря на их распределенность по разным системам. Связи CORBA обеспечивали межъязыковое взаимодействие объектов, а сами объекты были описаны на языке определения интерфейсов (Interface Definition Language, IDL) (https://oreil.ly/Akrew).

В 1993 году Microsoft создала *модель компонентных объектов* (Component Object Model, COM) (https://oreil.ly/PizNK) для решения аналогичных задач, но в двоичных программах с использованием *двоичного интерфейса приложений* (Application Binary Interface, ABI) (https://oreil.ly/qm08x). В отличие от API, определяющих доступ к программам на уровне исходного (высокоуровневого) кода, ABI зависел от оборудования. Он определял, как обращаться к подпрограммам (функциям) и структурам данных на уровне машинного (низкоуровневого) кода.

В 1998 году был создан протокол SOAP (Simple Object Access Protocol) (https://oreil.ly/h5EGt) для обмена XML-сообщениями между веб-сервисами. В нем использовались протоколы прикладного уровня: в основном HTTP (Hypertext Transfer Protocol) и реже SMTP (Simple Mail Transfer Protocol)[1]. Протокол критиковали за «многословность» (слишком большие сообщения), медленный парсинг (использовались не все возможности HTTP), отсутствие стандартной модели взаимодействия между клиентом и сервером.

В 2000 году Рой Томас Филдинг (Roy Thomas Fielding) представил докторскую диссертацию *Architectural Styles and the Design of Network-Based Software Architectures* («Архитектурные стили и проектирование сетевых программных архитектур») (https://oreil.ly/O1rcQ). Он описал архитектурный стиль REST (Representational State Transfer) и сформулировал концепцию сетевого API.

---

[1] Согласно отчету Postman о рынке API за 2023 год, SOAP занимает четвертое место в рейтинге самых используемых API (https://oreil.ly/knMeq).

API-МАНДАТ

Архитектурный стиль REST, представленный Филдингом в 2000 году, способствовал внедрению сетевых API в публичном интернете. Вскоре после этого был выпущен документ, известный как API-мандат, который показал, что сетевые API распространились и в корпоративной среде.

В 2002 году генеральный директор Amazon Джефф Безос (Jeff Bezos) выпустил меморандум для сотрудников (API-мандат) (https://oreil.ly/YlZ7C). Мандат относился в первую очередь к сервисно ориентированной архитектуре (SOA). На современном языке его основные принципы звучат так.

1. Данные и функциональность сервисов предоставляются через API.
2. Сервисы взаимодействуют только через API.
3. Запрещены любые другие способы связи, кроме сетевых API.
4. API реализуются с помощью технологий, подходящих для конкретной задачи.
5. API проектируются так, чтобы их можно было открывать внешним клиентам.

В 2010-х годах наблюдался рост использования API-технологий — этот период часто называют эпохой API. Вот хронология некоторых API-технологий, появившихся в это время: REST (2000), вебхуки (2007), WebSocket (2011), GraphQL (2015) и gRPC (2015).

Зачем нужны API

Рассмотрим ценность API для специалистов, разработчиков и организаций.

Часто говорят, что нужно развивать так называемые *мягкие навыки* (soft skills) — универсальные профессиональные качества, такие как тайм-менеджмент, коммуникабельность или работа в команде. Благодаря мягким навыкам соискателя могут пригласить на собеседование, но в конечном счете компании ищут кандидатов, которые помогут им достичь их целей. Организациям нужны специалисты с конкретными навыками — это могут быть как *мягкие* навыки (межличностное общение), так и *жесткие*. Если кандидат обладает навыками, которые нужны компании, то получает преимущество. Поэтому, независимо от того, являетесь ли вы разработчиком ПО или архитектором сетевых приложений, необходимо развивать навыки работы с API, поскольку они востребованы на разных позициях в разных отраслях.



Согласно отчету Gartner API Trends («Тренды API») за 2023 год, 38 % организаций, участвовавших в опросе, планировали создать должности разработчиков API в следующем году (https://oreil.ly/Ody1W).

API позволяют вести разработку на основе модульной архитектуры, где создание новых компонентов и управление ими не зависит от других компонентов. Вдобавок API обеспечивают взаимодействие, гарантируя, что различные системы общаются через стандартизированный интерфейс. При необходимости можно перейти к сервисно ориентированной архитектуре, например микросервисной.

Любая организация стремится к получению прибыли. В условиях растущей цифровизации, где действует принцип «каждый бизнес — это программный бизнес»[1], прибыль напрямую зависит от того, представлена ли организация в интернете. API помогают организациям расти, подключаться к внутренним и внешним сервисам и упрощают создание программного обеспечения описанным ниже образом.

Создание новых источников дохода

Сайты наподобие Skyscanner (https://www.skyscanner.com) выступают в роли API-агрегаторов. Они запрашивают предложения авиакомпаний и отелей через API и делают данные доступными для поиска. Далее пользователи могут выбирать подходящие авиакомпании или отели, что создает дополнительный доход для перевозчиков и гостиниц. Доход получают и провайдеры API. По данным отчетов, более половины выручки компаний Salesforce, eBay и Expedia приходится на API[2].

Расширение рынка

Платформа для видеоконференций Zoom (https://zoom.com), помимо сайта, доступна на смартфонах, планшетах, компьютерах и смарт-телевизорах. Компании, предоставляющие API для мобильных устройств, потенциально могут охватить значительную часть населения мира[3].

Доступ к новым технологиям

В 2021 году GitHub выпустил инструмент GitHub Copilot, основанный на модели Generative Pre-trained Transformer (GPT), которая обучена на

---

[1] *Humphrey W. S.* Winning with Software: An Executive Strategy. — Addison-Wesley Professional, 2001.

[2] *Iyer B., Subramaniam M.* The Strategic Value of APIs («Стратегическая ценность API») // Harvard Business Review, 07.01.2015. https://oreil.ly/k3KXU.

[3] В 2024 году численность населения мира составляла 8,1 млрд человек (https://oreil.ly/FBvVV), а количество подписок на смартфоны — 7,2 млрд (https://oreil.ly/EHMBT).

миллионах репозиториев. Со временем GitHub Copilot превратился из инструмента автодополнения кода в полноценного ассистента для программиста. Когда OpenAI сделал API моделей публичными, GitHub (и другие компании) получили доступ к обновленным версиям моделей.

ЧАСТНЫЕ, ПУБЛИЧНЫЕ И ПАРТНЕРСКИЕ API

API классифицируют разными способами, в том числе по типу доступа.

Частные

Частные API также известны как *внутренние* и разрабатываются внутри организации. Их цель — обмен данными между подразделениями.

Публичные

Публичные API предоставляются организациями для общего пользования. Доступ к сервисам и данным через публичные API может быть как бесплатным, так и платным. Например, сервис прогноза погоды Open-Meteo (https://open-meteo.com) предоставляет API бесплатно для некоммерческого использования и платно для коммерческого.

Партнерские

Партнерские API доступны только организации-разработчику и ее партнерам.

Что такое стили API

Чтобы вы могли понять, что такое стили API, сравним их с архитектурными стилями в гражданском строительстве. Архитектурный стиль — это набор характеристик, особенностей, методов, материалов, конструкций, а также региональных и культурных факторов, влияющих на проектирование, функциональность и внешний вид здания. Архитектурный стиль часто ассоциируют с определенным историческим периодом. Кроме того, стилям даются названия, отражающие их особенности: вы наверняка слышали о римских и греческих постройках, а также о постройках эпохи Ренессанса, барокко, зданиях Викторианской эпохи и памятниках народной архитектуры.

Приведенное выше определение архитектурных стилей применимо к программному обеспечению и API. В этой книге под *стилем API* мы понимаем парадигму, которая включает в себя паттерны, практики и протоколы для проектирования, реализации и предоставления API. Стили API, как и архитектурные, имеют свое назначение и функцию, а также могут появляться и исчезать с течением времени.

Каждый такой стиль представляет собой совокупность характеристик (протоколов, ограничений проектирования и т. д.), причем некоторые характеристики могут доминировать. Проведем параллель с музыкальной группой. В любой музыкальной группе есть ударные, басовые, струнные, клавишные инструменты и вокал. Используя все это, группа создает музыку в разных стилях: рок, поп, регги и многих других. Часто группа смешивает стили, расширяя спектр музыкальных направлений: поп-рок, фанк-рок или джаз-фанк. Однако группу определяет стиль исполняемой музыки, а не набор используемых инструментов. То же самое справедливо и по отношению к стилям API. Разные стили могут использовать одинаковые «инструменты»: протоколы, типы коммуникации, принципы или ограничения проектирования — однако определяющими для стиля становятся его доминирующие характеристики. Стили API можно классифицировать следующим образом.

REST/RESTful

RESTful API соответствуют шести ограничениям REST, изложенным в диссертации Роя Филдинга. Хотя многие API позиционируются как RESTful, в них часто отсутствуют гипермедиассылки, обязательные для проектирования RESTful API[1]. Подробно REST описан в главе 5.

API на основе запросов (query-based API)

Ключевая характеристика таких API — возможность формировать запросы к API, чтобы данные возвращались в определенной структуре. Наиболее известный API этого стиля — GraphQL (https://graphql.org), описанный в главе 6. GraphQL API предоставляет эндпоинт, куда клиенты отправляют запросы с указанием полей, которые необходимо извлечь.

Веб-фид (web feed)

Ключевая характеристика этого стиля — возможность доставлять постоянно обновляемый контент в структурированном формате. Известные примеры такого подхода — стандарты RSS (Really Simple Syndication, также известный как Rich Site Summary) (https://oreil.ly/wfbrT) и Atom (https://oreil.ly/3X1ib). В обоих стандартах данные структурируются в формате XML, а в качестве прикладного протокола используется HTTP. Этот стиль описан в главе 7.

Удаленный вызов процедур (Remote Procedure Call, RPC)

Основная характеристика стиля — возможность вызывать функцию (процедуру) на удаленном сервере так, как если бы она выполнялась локально.

---

[1] *Fielding R.* Rest APIs Must Be Hypertext-Driven («REST API должен быть основан на гипертексте») // Untangled, 20.10.2008. https://oreil.ly/NJLH1.

Клиент RPC выполняет вызов (передает данные по сети) на сервер RPC, который предоставляет процедуру, и получает в ответ результаты вызова (вычисления). Лидирующая технология в этом стиле API — gRPC (https://grpc.io), описанная в главе 8.

Обратный вызов (callback)

Основная характеристика стиля — механизм обратного вызова. В этом стиле задействованы две стороны: система-получатель, которая создает и обрабатывает обратный вызов (получает сообщения), и система-источник, которая запрашивает обратный вызов (отправляет сообщения). Каждое сообщение, отправленное от источника к получателю, требует подтверждения. Наиболее известная технология, основанная на этом стиле, — вебхуки (см. главу 9). Они напоминают событийно-ориентированный стиль API, поскольку механизм, запускающий обратный вызов, — это событие. Отличие лишь в том, что вебхуки не проходят через посредника наподобие брокера сообщений. Обычно вебхуки ассоциируют с *callback-запросами HTTP*, поскольку при возникновении события система-источник уведомляет систему-получателя, отправляя сообщение с помощью HTTP `POST`-запроса, который действует как механизм обратного вызова.

Двунаправленные API (bidirectional)

API этого стиля позволяют обмениваться данными между клиентом и сервером одновременно в обоих направлениях (см. полнодуплексную передачу, показанную на рис. 1.4). К этой категории относятся API, построенные на протоколе WebSocket (https://oreil.ly/XzJeg) или фреймворке gRPC (https://grpc.io), поскольку оба поддерживают обмен данными между клиентом и сервером в полнодуплексном режиме. (WebSocket рассматривается в главе 10, а gRPC — в главе 8.)

Брокерские API (broker-based)

В API, реализованных в этом стиле, сообщения передаются через посредника — *брокера*. Такие API доставляют сообщения, используя разные протоколы. К брокерским системам обмена сообщениями относятся Apache ActiveMQ (https://activemq.apache.org), Apache Kafka (https://kafka.apache.org) и RabbitMQ (https://www.rabbitmq.com), которые подробно описаны в главе 11.



Есть и другие варианты классификации сетевых API. Например, авторы книги Continuous API Management («Непрерывное развитие API») (O'Reilly, 2021) выделяют пять стилей: туннельные (tunnel), ресурсные (resource), гипермедиа (hypermedia), основанные на запросах (query) и событийные (event-based).

В табл. 1.1 приведены характеристики стилей API. В столбцах представлены стили, а в строках — их характеристики.

Таблица 1.1. Ключевые характеристики стилей API

	RESTful	На основе запросов	Веб-фид	RPC	Обратный вызов	Двунаправленный	Брокерский
Технология API	REST	GraphQL	Atom	gRPC	Вебхуки	WebSocket	RabbitMQ
Протокол	HTTP	HTTP*	HTTP	HTTP/2	HTTP	WebSocket	AMQP
Тип коммуникации	Синхронный	Синхронный	Асинхронный	Синхронный	Асинхронный	Асинхронный	Асинхронный
Поддержка двоичных данных	Есть	Частично	Частично	Есть	Есть	Есть	Есть
Отзывчивость системы	Средняя	Средняя	Средняя	Высокая	Средняя	Высокая	Высокая
Трудозатраты на разработку	Средние	Средние	Низкие	Высокие	Низкие	Средние	Высокие

* В HTTP и GraphQL также может использоваться протокол WebSocket.

Факторы, описанные в табл. 1.1, определяются следующим образом.

Технология API

Пример технологии, реализующей стиль API.

Протокол

Сетевой протокол, используемый данной технологией.

Тип коммуникации

Тип взаимодействия, характерный для стиля: синхронный или асинхронный (см. подраздел «Синхронная и асинхронная коммуникация» ранее в этой главе).

Поддержка двоичных данных

Возможность передачи двоичных данных. «Есть» означает поддержку по умолчанию, «Нет» — отсутствие поддержки, «Частично» — двоичное кодирование поддерживается, но требует реализации на уровне приложения, например кодирование изображения в строку Base64 (https://oreil.ly/miCop).

Отзывчивость системы

Способность системы реагировать своевременно, согласно *The Reactive Manifesto* («Манифесту реактивных систем») (https://oreil.ly/zAaMk). Возможные значения параметра: низкий, средний и высокий уровень.

Трудозатраты на разработку

Субъективная оценка усилий, необходимых для создания минимально жизнеспособного продукта API низкого уровня сложности. Возможные значения параметра: низкий, средний и высокий уровень.

API как продукт

После выхода API-мандата (см. одноименную врезку выше в этой главе) организации стали воспринимать API не только как связующее звено между сервисами, но и как потенциальный источник дохода. Иными словами, на API стали смотреть как на продукт.

Чтобы увидеть в API *продукт*, нужно вспомнить о том, что большинство организаций рассматривают API как технологию для интеграции. Такой подход приводит к созданию API низкого качества: они, как правило, недолговечны, не позволяют увеличить охваты и не поддаются масштабированию. Эти API плохо документированы, не имеют стандартов проектирования, версионирования и безопасности, и никто всерьез не задумывается об их расширении.

Относиться к API как к продукту означает уделять ему такое же внимание, как и любому другому активу. Успех API-продуктов зависит не только от маркетинга или дизайна, но и от сторонних разработчиков. Если учитывать их потребности, то можно существенно повысить эффективность внедрения вашего API. В последние годы стали набирать популярность порталы для разработчиков, такие как Backstage (https://backstage.io), Clutch (https://clutch.sh) и Cortex (https://www.cortex.io). Эти ресурсы помогают по-новому взглянуть на внутренние API организации и на процесс управления ими.

Отношение к API как к продукту предполагает анализ технических деталей его реализации и опыта его использования. Владелец такого продукта должен знать ответы на следующие вопросы.

- Кто основные пользователи API?
- Из каких регионов поступают запросы?
- Какие ошибки возникают в API?
- Каков процент этих ошибок?
- Каково время отклика?

Поэтому необходимо внедрять инструменты для мониторинга API и формирования отчетов об использовании.

МОНЕТИЗАЦИЯ API

Выступая в качестве цифрового канала для реализации продуктов и услуг организации, API также могут приносить прямой или косвенный доход. Получение дохода от использования API называют *монетизацией API*. Есть несколько моделей монетизации.

Плата за использование (pay-per-use)

Пользователь платит за использование API, которое измеряется количеством вызовов, объемом передаваемых данных или другими метриками. Эта модель позволяет провайдеру выставлять точные счета, а пользователю — контролировать и прогнозировать свои затраты на API. Такая модель подходит пользователям, которые не до конца понимают свои потребности. Основная сложность — для ее реализации нужны инструменты точного отслеживания вызовов.

Подписка

Пользователь платит фиксированную сумму за использование API (ежемесячно или ежегодно). Такая модель обеспечивает предсказуемый доход от API и предполагает наличие специальных предложений, например увеличение количества вызовов в месяц или дополнительные функции.

Freemium

Пользователь получает бесплатный доступ к базовым функциям API, но платит за дополнительные функции, превышение лимита вызовов или премиум-поддержку. Модель ориентирована на увеличение клиентской базы и конвертацию клиентов, использующих бесплатные функции, в клиентов, которые оплачивают пользование API.

Комиссия за транзакцию

Комиссию взимают за каждую платежную операцию, совершенную через API. Такая модель распространена в API электронной коммерции или финансовых услуг. Например, процент может взиматься с каждой транзакции, обработанной платежным API.

Разделение доходов

Доход от API распределяется между сторонним разработчиком и провайдером API. Такая модель распространена в партнерском маркетинге или на маркетплейсах, где сторонние разработчики получают доход через API-провайдеров. Например, сайт для бронирования путешествий интегрирует API бронирования отелей. Каждый раз, когда пользователь бронирует номер через сайт, разработчик получает часть комиссии от платы за бронирование.



Если вам интересна концепция «API как продукт», то рекомендуем книгу *Building an API Product* Бруно Педро (Bruno Pedro) (Packt Publishing, 2024).

В API, как и в любом цифровом продукте, важно обеспечить безопасность. Нужно предусмотреть процедуры аутентификации и авторизации (см. подраздел «Шифрование, аутентификация и авторизация» в главе 2), создать соответствующие политики, установить лимит на количество запросов в единицу времени и настроить оповещения о подозрительном трафике.



Если API становится публичным, то разработчик берет на себя обязательства сопровождать его. Грамотное управление API требуется как на этапе его выпуска, так и на этапе устаревания и подразумевает коммуникацию с пользователями и заблаговременное уведомление о прекращении поддержки. Если эти принципы не учтены изначально, то исправить ошибки после выпуска API будет сложно[1]. Практический пример грамотного управления API на этапе устаревания — Facebook. Разработчики заранее запланировали прекращение доступа к Groups API и уведомили об этом сторонних разработчиков.

Жизненный цикл API

Жизненный цикл разработки ПО (Software Development Life Cycle, SDLC) — это процесс управления программным обеспечением на протяжении всего времени его существования, от планирования до вывода из эксплуатации. API относится к программному обеспечению, поэтому термин «жизненный цикл API» синонимичен SDLC. Даже если ваш API не пройдет полный цикл, понимание того, из каких этапов тот состоит, поможет вам выработать подход к разработке.

Жизненный цикл API состоит из таких этапов, как: планирование, проектирование, реализация, тестирование, развертывание, сопровождение и выведение из эксплуатации. Все эти этапы показаны на рис. 1.7.



Хотя есть тенденция рассматривать этапы жизненного цикла API (или SDLC) как независимые, следующие друг за другом, в действительности они тесно взаимосвязаны. Например, на этапе реализации могут обнаружиться факторы, не учтенные при проектировании, что приведет к изменениям в исходном проекте. Кроме того, мы не указываем, какой этап проекта является начальным, поскольку это зависит от временны́х рамок вашего проекта.

---

[1] См. статью: *Abouelhassan M. A.* Introducing Facebook Graph API v19.0 and Marketing API v19.0 («Новый Graph API v19.0 и marketing API v19.0 от Facebook») // Meta, 23.01.2024. https://oreil.ly/YOxWt.



Рис. 1.7. Жизненный цикл API

Вы могли заметить, что в нашей схеме отсутствуют два этапа, которые часто упоминаются другими авторами: обеспечение безопасности и создание документации. На некоторых схемах жизненного цикла API обеспечение безопасности представлено как отдельный этап. В результате создается впечатление, что безопасность важна только на одном из этапов. Для нас же безопасность — это универсальный принцип, одинаково применимый ко всем этапам цикла.

Создание документации в некоторых проектах откладывается до этапа сопровождения. Считается, что нет смысла документировать программное обеспечение, разработка которого еще не завершена. Главный недостаток такого подхода заключается в том, что разработка ПО продолжается до момента вывода его из эксплуатации, и если ждать завершения, то документация не появится никогда. Более реалистичный подход — составление черновых вариантов документации на каждом этапе жизненного цикла API.

Планирование

Разработчики, применяющие подход code-first («сначала код»), часто упускают этап планирования, важный для успеха любого проекта. На этом этапе решаются следующие вопросы.

- Какие данные пользователя будет запрашивать сервер?
- В какие сроки должна быть выполнена работа?
- Какие результаты ожидаются на каждом из этапов?

Кроме того, владельцы, архитекторы и разработчики API закладывают основы будущего продукта: определяют цели проекта, изучают целевую аудиторию и формулируют функциональные и нефункциональные требования к API.

Функциональные требования (functional requirements, FR) описывают возможности программного обеспечения, а *нефункциональные* (nonfunctional requirements, NFR) — его качественные характеристики. Например, при разработке поисковой системы функциональным требованием может быть возможность отправки поисковых запросов через API, а нефункциональным — возврат результатов поиска в течение одной секунды.

На этапе планирования необходимо ответить на следующие вопросы.

Каков охват API?

API может быть частным (созданным внутри организации) или публичным (для внешних пользователей). Кроме того, API может быть партнерским — для совместной работы между организациями.

Какие устройства у пользователей API?

Это могут быть устройства интернета вещей (IoT-устройства), браузеры, мобильные телефоны или ИИ-агенты.

Есть ли требования к проектированию API?

Есть ли язык программирования или фреймворк, навязанный разработчиком API? Есть ли лимит времени, в течение которого API должен выполнить запрос? Зависит ли API от внешних сервисов?

Каков масштаб работы API?

Каково приблизительное количество пользователей?

Какую стратегию управления версиями использовать?

Можно выполнять версионирование по семантическому принципу или по дате выпуска версии.

Есть ли ограничения для API?

Влияют ли на функциональность API регуляторные ограничения, такие как GDPR или HIPAA?

Есть ли установленные каналы связи с пользователями API?

Как вы будете предоставлять пользователям документацию? Будет ли ограничен доступ к коммуникации?

Это лишь несколько вопросов, на которые нужно ответить перед реализацией API. Кроме того, на этапе планирования в требованиях необходимо отразить безопасность и комплаенс.

ФУНКЦИОНАЛЬНЫЕ И НЕФУНКЦИОНАЛЬНЫЕ ТРЕБОВАНИЯ

На разных этапах жизненного цикла разработки ПО формулируются несколько типов требований к системе (https://oreil.ly/qvpIP): пользовательские, бизнес-требования, регуляторные, системные, архитектурные, функциональные, нефункциональные и требования к реализации.

Однако, с точки зрения проектировщика, на первых трех этапах жизненного цикла ключевую роль играют функциональные и нефункциональные требования (ФТ и НФТ соответственно). Разделение на ФТ и НФТ помогает расставлять приоритеты в проектировании и разработке, а также заранее выявлять риски, которые могут возникнуть из требований.

Рассмотрим определения ФТ и НФТ.

Функциональные требования (ФТ)

Конкретные функции, задачи или поведение системы. ФТ можно описывать с точки зрения пользователя, выполняющего определенные действия в системе, или с точки зрения самой системы. Например: «У пользователя должна быть возможность сбросить пароль» или «У администраторов должно быть право заблокировать пользователю доступ».

Нефункциональные требования (НФТ)

Возможности и ограничения, которым должна соответствовать система. Иными словами, НФТ описывают качество системы. Их также называют архитектурно значимыми требованиями, программными архитектурными характеристиками или атрибутами качества. НФТ описывают такие свойства системы, как развертываемость, сопровождаемость, производительность, безопасность и удобство использования (https://oreil.ly/s85Wq). Например: «Система должна быть быстрой» или «Система должна быть доступна 99,9 % времени».

Различие между ФТ и НФТ часто бывает размытым. Например, требование «Сервис должен предоставлять своевременные оповещения о погоде» может быть одновременно и функциональным, и нефункциональным:

- сервис предоставляет оповещения о погоде — это функция;
- сервис предоставляет своевременные оповещения — это качество реализации функции.

Если нужно определить, к какой категории относится требование, то ответьте на следующие вопросы.

- Связано ли требование с реализацией функции?
- Влияет ли требование на свойства системы (развертываемость, производительность и др.)?

- Сказано ли в требовании, что система или функция должна работать определенным образом?
- К каким данным относится требование — входным или выходным?

Система продолжает функционировать после этапа реализации, поэтому влияние НФТ проявляется не сразу.

Проектирование

На этапе проектирования разработчики (архитекторы и программисты) определяют подход к проектированию API на основе информации об аудитории, требованиях и ограничениях, полученной на этапе планирования. При проектировании принимается решение не только о том, какая должна быть функциональность, но и о том, как ее реализовать. Кроме того, решается вопрос о необходимости интеграции сторонних сервисов и о том, какие технологии и инструменты использовать в разработке.

ПОДХОД API-FIRST

Стратегия проектирования API-first предполагает, что разработка начинается с создания спецификаций API, собранных из требований организации и пользователей. Далее следует реализация. Этот подход полезен, если разработчики могут связаться с будущими пользователями и согласовать структуру API до реализации.

Как правило, спецификация API составляется в общепринятом формате, например OpenAPI (https://swagger.io) или AsyncAPI (https://www.asyncapi.com). Инструменты для работы с этими форматами помогают реализовывать и тестировать API, а также создавать документацию. Более подробно проектирование API рассматривается в главе 2.

Реализация

Этап реализации также называют *фазой прикладной архитектуры* (applied architecture phase).

На данном этапе требования, полученные в ходе планирования, и спецификации, составленные при проектировании, преобразуются в код. Кроме того, в процессе реализации может потребоваться пересмотреть ранее принятые решения и внести изменения в требования и спецификации.

Обратите внимание: реализация определяет фактическое поведение системы. Хотя детали реализации должны быть скрыты за интерфейсами, иногда они все же становятся видимыми.

API: ИНТЕРФЕЙС ИЛИ РЕАЛИЗАЦИЯ

Грегор Кицалес (Gregor Kiczales) в статье *Towards a New Model of Abstraction in Software Engineering* (https://oreil.ly/33Qqi) высказал идею о том, что абстракциям не всегда удается скрыть детали реализации, и это часто усложняет перенос системы:

> «Абстракции», которыми мы пользуемся, на самом деле не являются абстрактными. За ними стоят реальные фрагменты кода, работающие на реальных компьютерах, потребляющих энергию и занимающих место в пространстве. Попытка игнорировать детали реализации подобна попытке игнорировать законы физики: звучит заманчиво, но в конечном счете ведет в никуда.
>
> *Грегор Кицалес (Gregor Kiczales), 1992*

Эта идея известна как «Закон дырявых абстракций» (Law of Leaky Abstractions) (https://oreil.ly/mBLLI):

> Любая нетривиальная абстракция в некоторой степени дырявая.
>
> *Джоэл Спольски (Joel Spolsky), 2002*

Широко известен также *закон Хайрама* (Hyrum's law) (https://www.hyrumslaw.com) о том, что детали реализации проявляются через интерфейс API:

> При достаточном количестве пользователей API не имеет значения, что вы обещаете в контракте: все наблюдаемые изменения вашей системы будут зависеть от чьих-то действий.
>
> *Хайрам Райт (Hyrum Wright), 2012*

«Дырявые» абстракции приводят к тому, что пользователи API начинают полагаться на поведение системы, которое разработчики не планировали и не документировали (такое поведение известно как «неявный интерфейс»)[1]. Это усложняет внесение изменений в API.

От практических решений, принятых на этапе реализации, зависит и безопасность API. Защищенные API создаются на основе механизмов шифрования,

[1] Это поведение системы стало объектом для шуток в 1172-й части веб-комикса xkcd (https://xkcd.com/1172).

аутентификации, авторизации и других практик безопасного программирования, в том числе валидации входных и выходных данных (подробнее см. в разделе «Безопасность API» главы 2).

Тестирование

На данном этапе проверяется функциональность, безопасность и производительность API. За создание и проведение тестов отвечают разработчики и QA-команда (quality assurance)[1]. Тестирование может проводиться параллельно с реализацией, а его результаты могут влиять на другие этапы, например на проектирование, реализацию и развертывание.

Степень сложности тестирования варьируется в зависимости от того, что проверяется и как. Этот этап важен, поскольку тестирование позволяет создавать удобное в сопровождении программное обеспечение, проверять функциональность системы и выявлять наличие ошибок (но не их отсутствие). Кроме того, оно повышает эффективность рефакторинга.

Тесты можно выполнять вручную или автоматически, если они являются частью конвейеров непрерывной интеграции, поставки и развертывания (CI/CD) (см. подраздел «Развертывание» далее в этой главе). Запуск тестов в конвейере позволяет разработчикам оперативно выявлять проблемы в программном обеспечении и поддерживать его стабильное качество.

Интеграционное тестирование

Интеграционные тесты проверяют взаимодействие компонентов между собой, например взаимодействие программного обеспечения с внешними сервисами (базами данных, файловыми системами и сторонними API). В отличие от модульных тестов (unit tests), проверяющих отдельные фрагменты кода, интеграционные тесты контролируют взаимодействие компонентов в рамках большой системы.

Инструменты наподобие Docker (https://www.docker.com) упрощают настройку интеграционных тестов. Docker запускает изолированные контейнеры с необходимыми сервисами (например, базу данных, кэш или очередь сообщений). Это позволяет тестировать программное обеспечение на работающих копиях сервисов вместо использования заглушек (mocks), что делает результаты более реалистичными. Если сторонний сервис нельзя запустить локально, то можно настроить отдельный тестовый экземпляр и интеграционные

[1] Иногда более эффективным является бета-тестирование, при котором доступ к API сначала предоставляется небольшой группе пользователей. Практика показывает, что такой способ позволяет выявлять баги и недостающие функции быстрее, чем при стандартном тестировании разработчиками и QA-командой.

тесты для взаимодействия с ним. Тесты будут проверять работу реального сервиса в контролируемой среде. Однако такая настройка часто приводит к непредсказуемым результатам и ошибкам тестирования, которые нельзя воспроизвести.

Если данные из продакшена разрешено использовать для тестирования, то сохраните ответ сервиса в файл (например, `response.json`) и используйте его в интеграционных тестах для эмуляции сервиса. Это позволяет проводить регулярные тесты (иногда называемые тестами *записи и воспроизведения* (record and replay)), избегая незапланированного взаимодействия с продакшен-средой.

Контрактное тестирование и фаззинг

Метод *контрактного тестирования* берет начало в работах Роберта У. Флойда (Robert W. Floyd). В конце 1960-х годов Флойд предложил проверять корректность работы программ с помощью логических утверждений. Этот и аналогичные, более поздние методы, такие как логика Хоара (https://oreil.ly/DAhnS; https://ru.wikipedia.org/wiki/Логика_Хоара) (1969) и контрактное программирование (https://oreil.ly/KV5BZ; https://ru.wikipedia.org/wiki/Контрактное_программирование) (1986), заключались в том, чтобы доказать, что программа будет работать правильно при допустимых входных данных:

> Разработчики писали код, запускали тесты, находили ошибки и исправляли их, затем обнаруживали новые ошибки и снова вносили правки — так продолжалось до тех пор, пока тесты не переставали выявлять ошибки. Однако всегда сохранялся риск того, что на следующий день обнаружится новая, ранее неизвестная ошибка. Мы не осознавали, что можно построить строгое логическое доказательство корректности программы.
>
> *Дональд Э. Кнут (Donald E. Knuth), 2003*

Сегодня контрактное тестирование не предоставляет строгих доказательств корректности работы системы, поскольку клиенты и серверы постоянно подвергаются изменениям. Вместо этого контрактное тестирование проверяет, не нарушилась ли совместимость компонентов системы и соблюдают ли они условия контракта (спецификацию API). Спецификация часто представляется в виде статической схемы, описывающей структуру данных, ее содержимое, структуру запросов пользователей и ответов провайдеров. Однако API может быть свойственно и такое поведение, которое выходит за рамки статических схем. Например, не все комбинации входных данных могут считаться допустимыми, или API может ограничивать частоту запросов — ничего из этого не может быть описано в спецификации.

Однако по аналогии с методом записи и воспроизведения можно сохранять набор прошлых запросов и ответов и проверять их соответствие спецификации API (параллельно с проверкой текущего взаимодействия провайдера и пользователя). Такой подход к контрактному тестированию схож с регрессионным тестированием (https://oreil.ly/wwfCo; https://ru.wikipedia.org/wiki/Регрессионное_тестирование): он позволяет убедиться в том, что контракт API остается стабильным и пользователи и провайдеры продолжают его соблюдать. Например, при интеграции в процесс разработки контрактное тестирование позволяет обнаружить, что изменения со стороны провайдера несовместимы с поддерживаемыми версиями пользователей. Кроме того, тестирование позволяет определить, возможно ли удалить поле, которое используется только устаревшими версиями пользовательских приложений.

Контракт может составлять как потребитель API, так и провайдер. Соответственно есть два типа контракта: *управляемый потребителем* (consumer-driven contract, CDC) и *управляемый провайдером* (provider-driven contract, PDC). В CDC поведение сервиса определяет потребитель, а задача провайдера — реализовать контракт в API. Этот подход требует больше усилий от провайдера. В PDC усилия распределяются иначе: контракт определяет провайдер, а потребитель следует ему.

Основное ограничение контрактного тестирования заключается в том, что не всегда возможно собрать достаточно большой набор запросов и ответов, чтобы оценить соблюдение контракта API. Сбор входных данных осложняется при использовании популярного метода *защитного программирования* (defensive programming) (см. врезку «Защитное программирование» в подразделе «CRUD» в главе 5). В этом методе приоритет отдается обработке ошибок и оперативному восстановлению функциональности, вследствие чего спектр «допустимых» входных данных и шаблонов поведения, поддерживаемых API, расширяется. Этот подход противопоставляется так называемому *«атакующему» программированию* (offensive programming или fail-fast), в котором программа прекращает работу при получении некорректных входных данных.

Чтобы компенсировать нехватку тестовых данных, вместе с контрактным тестированием применяют *фаззинг* (fuzzing), где в набор входных данных входят случайные значения. Предполагается, что API корректно обработает такие входные данные и фаззер проверит бóльшую часть спецификации API.

Сквозное тестирование

Такое тестирование (end-to-end) предполагает имитацию взаимодействия пользователя с системой. Цель — проверить работоспособность системы в целом. Традиционно сквозные тесты проводились вручную. Такие инструменты,

как Cypress (https://www.cypress.io), Playwright (https://playwright.dev) и Selenium (https://www.selenium.dev), позволили автоматизировать этот процесс, однако результаты их работы не всегда стабильны[1].

Квадранты Agile-тестирования

Разработчики создают множество разных типов тестов. Чтобы классифицировать имеющиеся подходы к тестированию, используется диаграмма «Квадранты Agile-тестирования» (рис. 1.8). На горизонтальной оси расположены цели тестирования — от поддержки команды в разработке до критики продукта. Вертикальная ось показывает направленность тестирования: от технической составляющей продукта до удобства использования.



Рис. 1.8. Квадранты Agile-тестирования

Поясним назначение каждого квадранта.

Q1

Тестирование ориентировано на совершенствование технологий и поддержку команды. Для этого применяются модульные и компонентные тесты, которые создаются и используются разработчиками. Основные преимущества этих тестов — они автоматизируются, выполняются быстро и предоставляют моментальную обратную связь.

---

[1] См. статью Джастина Сирлса (Justin Searls) Why Playwright Is Less Flaky than Selenium («Почему Playwright надежнее Selenium») (https://oreil.ly/UF0sj).

Q2

Тестирование ориентировано на улучшение пользовательского опыта и поддержку команды разработки. Тесты проверяют функциональность, гарантируя, что система удовлетворяет потребности пользователей. Допускается как автоматизированное, так и ручное тестирование.

Q3

Тестирование ориентировано на потребности пользователя и критику продукта. Здесь обычно проводят приемочные тесты (user acceptance tests, UAT), которые выявляют проблемы, пропущенные при разработке, и улучшают пользовательский опыт. Основной недостаток UAT — их часто приходится выполнять вручную.

Q4

Тестирование ориентировано на совершенствование технологий и критику продукта. Тесты здесь имеют технический характер и проверяют соблюдение НФТ, таких как безопасность и производительность. Для их проведения часто требуются специализированные инструменты.

Таким образом, тесты в квадрантах Q1 и Q2 поддерживают процесс разработки, контролируя соблюдение функциональных требований. Тесты в квадрантах Q3 и Q4 оценивают продукт с точки зрения соответствия потребностям пользователей и соблюдения нефункциональных требований. Эта модель Agile-тестирования служит своего рода картой, которая охватывает основные виды тестирования, позволяющие создавать качественные и удобные в сопровождении приложения.

Развертывание

Развертывание — это перемещение программного артефакта в среду, где он будет работать. Обычно в этот процесс входит упаковка ПО, настройка и подготовка среды, выполняемые вручную или автоматически. Перемещение артефакта API (автономной программы вместе со спецификацией и документацией) на целевой хост кажется простой задачей, но любой, кто работал в команде эксплуатации, подтвердит, что на практике все сложнее. В этом подразделе мы рассмотрим несколько вопросов, на которые нужно ответить при планировании развертывания API.

Первый вопрос: как будет организована команда, управляющая развертыванием? Будет ли разделение на команды разработки и эксплуатации или вы выберете подход DevOps, где эти две команды работают вместе? DevOps позволяет создавать конвейеры, где интеграция (сборка), доставка (выпуск) и развертывание ПО являются этапами единого, непрерывного процесса.

На рис. 1.9 показаны этапы развертывания API, а также связи между конвейерами CI/CD[1].

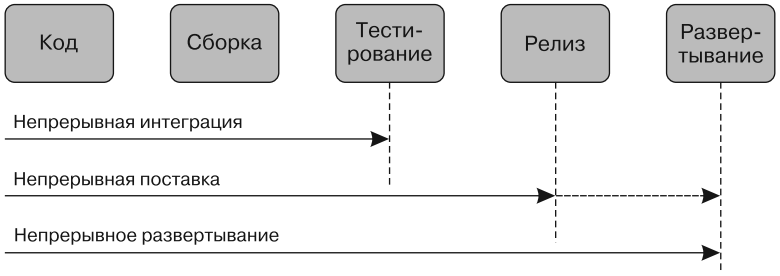


Рис. 1.9. Конвейеры CI/CD

Рассмотрим роли каждого типа конвейеров.

- *Непрерывная интеграция* — это конвейер сборки артефакта, состоящий из таких этапов, как:
  - линтинг — проверка соответствия исходного кода стандартам программирования;
  - статический анализ кода — предотвращение добавления в код нежелательных, сложных или небезопасных паттернов программирования (например, жестко запрограммированных секретов или инъекций кода);
  - анализ зависимостей — проверка кода на наличие потенциально устаревших или нелицензированных библиотек;
  - сборка — создание артефакта;
  - тестирование — проверка функциональности артефакта;
  - проверка безопасности — проверка собранного артефакта на уязвимости и выполнение автоматизированного пентеста.

  Результат работы конвейера CI — артефакт, например образ контейнера, нативный двоичный файл или платформенно зависимый пакет. Готовый артефакт доставляют в целевой репозиторий: в Docker Hub (https://hub.docker.com), GitHub Packages (https://oreil.ly/80QID), JFrog Artifactory (https://oreil.ly/iejv0) или на FTP-сервер классического типа.
- *Непрерывная поставка* — это конвейер, который запускается по завершении CI-конвейера. После ручного подтверждения он развертывает собранный

[1] При непрерывной поставке релиз созданного артефакта нужно подтверждать вручную. При непрерывном развертывании релиз артефакта осуществляется автоматически после завершения тестирования.

артефакт в нужную среду. *Стратегия развертывания* (https://oreil.ly/4hw1Y) зависит от таких факторов, как степень сложности API, организационные требования и возможности инфраструктуры. К популярным стратегиям относятся сине-зеленое (blue-green deployment) и канареечное развертывание (canary deployment).

- *Непрерывное развертывание* — этот конвейер автоматически развертывает артефакты в нужную среду, обходясь без участия человека. На данном этапе применяются автоматические тесты, среди которых сквозные (end-to-end), дымовые для проверки наиболее важной функциональности, нагрузочные и стресс-тесты.

Помимо описанных технических характеристик, в методологии DevOps культивируется сотрудничество между командами разработки и эксплуатации, а также совместная ответственность за жизненный цикл продукта. Проблемы безопасности команды DevOps решают по принципу *shift-left*: требования безопасности находятся в фокусе внимания с первых этапов жизненного цикла продукта, а не только на завершающем этапе *(shift-right)*. Такой подход называют *DevSecOps*.



В анонимной цитате из интернета справедливо отмечается: «shift-left повышает трение в команде, а shift-right повышает корректность разработки». Увеличение нагрузки на разработчиков снижает их продуктивность в плане создания новых функций ПО. При частых сбоях проверок безопасности команды могут отключать эти проверки, что снижает эффективность подхода shift-left.

Методология DevOps основана на автоматизации: считается, что перенос операционных задач на более низкий уровень (например, включение проверок безопасности в базовую платформу) снижает нагрузку на разработчиков (https://oreil.ly/153p3). Однако существование отдельных команд DevOps или DevSecOps говорит о том, что некоторые идеи движения DevOps трудноосуществимы, поскольку один разработчик не справится со всеми возложенными на него задачами.

После сборки артефакта API возникает вопрос о месте его развертывания. Это классическая дилемма «купить или создать» (buy-versus-build). В зависимости от потребностей артефакт API можно развернуть на проприетарной облачной платформе, на собственном облачном сервере или на локальных компьютерах. Если вы выбрали путь «создать», то готовы ли вы нести высокие затраты на настройку и поддержку платформы типа Kubernetes (https://kubernetes.io), которая поддерживает высокодоступную инфраструктуру для развертывания и масштабирования контейнеризованных приложений? Или вы предпочтете более низкие первоначальные затраты на простой инструмент наподобие Kamal

(https://kamal-deploy.org), понимая, что в будущем это может привести к более высоким затратам из-за необходимости управлять мощностями вручную?

Дилемма «купить или создать» затрагивает не только исходную цель развертывания, но и другие сервисы, поддерживающие операции через API. Необходимо расставить приоритеты. Будет ли выгоднее использовать интегрированную проприетарную систему облачного провайдера, у которого реализованы сервисы прокси-сервера, логирования, управления ключами и информационной безопасности? Или лучше развернуть собственное, более контролируемое решение на основе компонентов с открытым исходным кодом или разработанных внутри компании?[1]

Следующий вопрос касается количества сред для развертывания API. Продакшен-среда — это место, где API работает, но тестирование в ней может быть запрещено по требованиям комплаенса. Развертывание изменений напрямую в продакшен-среду без тестирования рискованно, поэтому перед развертыванием артефакта API тесты запускаются в тестовой или стейджинг-среде. Кроме того, может потребоваться одна или несколько отдельных сред разработки или функциональных сред для проведения потенциально разрушительных экспериментов, чтобы избежать блокировки доступа к общей промежуточной среде.

При работе с несколькими средами нужно также определиться, какую стратегию ветвления использовать в системе контроля версий[2].



Не забывайте, что использование дополнительных сред требует затрат на поддержку, и каждая может демонстрировать поведение, не воспроизводимое в продакшене[3].

> Стейджинг-среда — это не продакшен и никогда им не будет.
>
> *Чарити Мэйджорс (Charity Majors), 2019*

Как вы могли заметить, развертывание API требует учета многих факторов, среди которых инфраструктура хостинга и среды, ветвление исходного кода,

---

[1] Необходимо также выбрать систему CI/CD. Для этого нужно решить: готовы ли вы отлаживать сбои развертывания в удаленной системе, основанной на YAML, или инвестировать в разработку локальной среды? В связи с YAML см. также выступление I'm Sorry About the YAML («Простите меня за YAML») Джо Беды (Joe Beda) (https://oreil.ly/q0J3t).

[2] «Идея создания отдельной ветки кода для каждого окружения многим кажется привлекательной. Однако в действительности это классический антипаттерн — решение, привлекательное на первый взгляд, но создающее еще больше проблем в долгосрочной перспективе». Из статьи Мартина Фаулера (Martin Fowler) Patterns for Managing Source Code Branches («Паттерны управления ветками исходного кода») (https://oreil.ly/SpuOp).

[3] См. выступление Чарити Мэйджорс I Test in Production («Тестирование в продакшене») (https://oreil.ly/3kuby).

упаковка, тестирование, требования безопасности, масштабируемость, логирование и мониторинг. Стратегия развертывания определяется масштабом, сложностью и ожидаемой нагрузкой на API.

Сопровождение

Сопровождение — это продолжительная фаза жизненного цикла API, требующая значительных затрат. На данном этапе программное обеспечение обновляют, документируют, улучшают, а также отслеживают его производительность, активность пользователей и угрозы безопасности. Цель сопровождения — поддерживать актуальность API на протяжении всего времени его работы.

На рис. 1.10 показана традиционная классификация типов сопровождения ISO/IEC/IEEE 14764 (https://oreil.ly/8V2mi), основанная на характере изменений, вносимых в ПО. Запрос на изменение подразумевает либо исправление ошибок, либо улучшение системы. Корректирующий запрос связан с устранением дефектов программного обеспечения, а запрос на улучшение — с изменением имеющихся функций или добавлением новых.

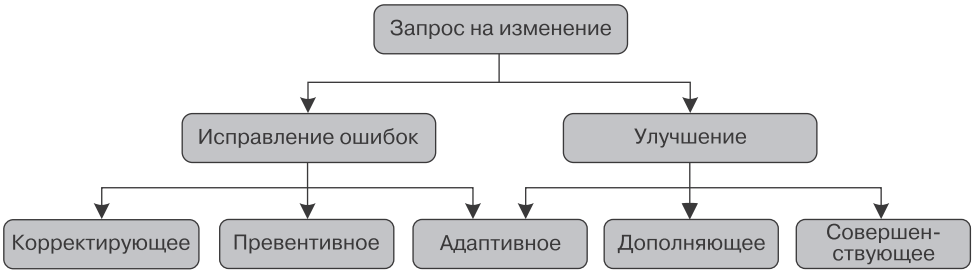


Рис. 1.10. Классификация типов сопровождения по стандарту ISO/IEC/IEEE 14764

Рассмотрим пять типов сопровождения, возникающих в результате запроса на изменение.

Корректирующее

Устраняет сбои в работе ПО, вызванные ошибками. Требует значительных затрат и может занимать много времени. Разновидность корректирующего — *аварийное сопровождение*, которое требует внесения внеплановых изменений для поддержания работоспособности системы.

Превентивное

Предполагает предотвращение сбоев ПО за счет их прогнозирования. Сопутствующий прием — *омоложение программного обеспечения* (software rejuvenation), связанное с устранением накопленных ошибок и оптимизацией системных ресурсов.

Адаптивное

Связано с внесением исправлений или улучшений в среду работы ПО. Необходимость такого сопровождения обусловлена тем, что среда, в которой функционирует программное обеспечение, подвержена изменениям.

Дополняющее

Наиболее распространенный тип сопровождения, связанный с необходимостью добавить новую функциональность или улучшить имеющуюся.

Совершенствующее

Предполагает усовершенствование ПО для повышения его производительности, удобства сопровождения и улучшения пользовательского опыта. Может выполняться тонкая настройка ПО (software tuning) для устранения узких мест производительности и рефакторинг для повышения удобства сопровождения кода.

Этап сопровождения предполагает наблюдение и сбор аналитики. К *наблюдению* относится мониторинг запросов и ответов API (например, соотношение успешных и неуспешных ответов), объема входящего трафика и его источников, а также времени последнего использования API. Собранные данные затем визуализируют в виде дашбордов с метриками. Кроме того, на этапе сопровождения настраивают инструменты оповещения для реагирования на чрезвычайные ситуации.

Итак, мы рассмотрели этапы жизненного цикла, ведущие к созданию API: планирование, проектирование, тестирование, реализацию, развертывание и сопровождение. В следующем подразделе поговорим о выводе API из эксплуатации, в том числе из-за его устаревания.

Вывод из эксплуатации

Вывод из эксплуатации — заключительный этап жизненного цикла API. Организация может прекратить поддержку API, если он не способствует достижению ее целей, представляет угрозу безопасности или затраты на его обслуживание превышают доходы от него. Кроме того, API выводится из эксплуатации, если к нему невозможно добавить новый функционал или если организация меняет стратегию монетизации и переходит с бесплатного API на платный.

При отсутствии плана вывода API из эксплуатации в инфраструктуре организации появится так называемый *зомби-API* (zombie API) — тот, который больше не поддерживается, но остается активным, и к нему можно получить доступ. Зомби-API представляют серьезную угрозу для безопасности организации.

С другой стороны, команда разработчиков и репутация организации могут пострадать и в случае неправильного вывода API из эксплуатации[1].

Для этого этапа разрабатывается план миграции, позволяющий поддерживать параллельную работу старой и новой версий API. Кроме того, создается план завершения жизненного цикла, определяющий, как объявить API устаревшим, не повлияв на текущую производительность, и как сообщить пользователям о выводе API из эксплуатации.

Объявление API устаревшим означает, что он будет доступен еще какое-то время, но его удаление (или замена) уже запланировано. Устаревание API тесно связано с частотой новых релизов. Например, организация может придерживаться политики поддержки максимум двух версий — тогда при выпуске третьей первую удалят. Допустим, она выпустила две версии API — *v1* и *v2*, а теперь готовит третью — *v3*. Это значит, что при выпуске *v3* версию *v1* удалят.

Устаревание и вывод из эксплуатации привязаны к конкретным датам. Дата устаревания указывает момент, начиная с которого поддержка обновлений API и исправление ошибок ограничиваются. Дата вывода из эксплуатации означает момент, с которого API становится недоступен.

Если организация планирует объявить API устаревшим, то необходимо уведомить об этом пользователей. Например, в Kubernetes (https://kubernetes.io) при создании устаревшего артефакта через CLI выводится предупреждающее сообщение (рис. 1.11).

```
Warning: networking.k8s.io/v1beta1 Ingress is deprecated in v1.19+,
unavailable in v1.22+; use networking.k8s.io/v1 Ingress
ingress.networking.k8s.io/echomap
```

Рис. 1.11. Уведомление об устаревании API Kubernetes

Уведомить пользователей можно, разместив баннер на сайте с документацией. На рис. 1.12 показано предупреждение об устаревании API PaLM от Google.

Warning: The PaLM API is **decomissioned**. The Vertex AI PaLM API is scheduled to be decomissioned in October 2024. Please upgrade to the Gemini API. Learn more about this in the PaLM API deprecation guide.

Рис. 1.12. Баннер с уведомлением об устаревании API

[1] См. статью: Twitter Only Gave Meerkat 2 Hours Notice before Cutting Access to the Social Graph («Twitter дал Meerkat всего два часа уведомления перед отключением доступа к социальному графу») Сары Кесслер (Sarah Kessler) (https://oreil.ly/ATBvS).

Еще один вариант — добавить заголовки об устаревании API в ответ (пример 1.4)[1].

Пример 1.4. HTTP-заголовки устаревания и завершения поддержки

```
Sunset: Wed, 11 Nov 2026 11:11:11 GMT
Deprecation: @1688169599
Link: <https://developer.example.com/deprecation>; rel="deprecation"
```

Метка времени в заголовке `Deprecation` может указывать как на прошедшую, так и на будущую дату. Прошедшая означает, что API уже устарел. Однако дата в заголовке `Sunset` всегда указывает на будущее, обозначая момент прекращения поддержки API. Период завершения поддержки публичного API обычно занимает несколько месяцев. Кроме того, в заголовке `Link` может содержаться дополнительная информация об устаревании API, например ссылка на документацию.

Помимо даты прекращения поддержки, в сообщение об устаревании можно добавить контактные данные, руководство по миграции или даты выпуска следующей версии, если они известны.

Информацию об устаревании и прекращении поддержки API может потребоваться сохранить для комплаенс-аудита или исторических справок.

Управление, администрирование и платформа API

В небольших компаниях API, как правило, управляет одна команда, которая принимает решения на всех этапах его жизненного цикла. Однако более крупным организациям с несколькими командами, разрабатывающими API, необходим план управления и контроля. При отсутствии контроля над API возникают риски получения несанкционированного доступа, утечки данных или незаконного использования. Управление API позволяет устранить эти риски.

Управление — это набор формальных и неформальных правил и процессов, которые направляют и контролируют деятельность организации. Соблюдение этих правил обеспечивают управляющие структуры организации.

Управление API, в свою очередь, предполагает создание регулирующих стандартов и политик, их распространение среди команд API и внедрение в сами

---

[1] Подробнее о заголовках Sunset и Deprecation см. в RFC 8594 (https://oreil.ly/09ryX) и в статье The Deprecation HTTP Header Field («Поле заголовка Deprecation в HTTP») (https://oreil.ly/Z35cR).

API. Контроль соблюдения политик выполняется за счет мониторинга API и деятельности команды.

Отсутствие управления приводит не только к несогласованности в работе API, но и к появлению *теневых API* (shadow APIs), которые активно используются, но не управляются, не документируются и развертываются без официального контроля организации. Управление же способствует безопасности, стабильности, позволяет повторно использовать API и обеспечивает их соответствие регулирующим политикам.

Управление также предполагает *администрирование API*, цель которого — стандартизировать процессы разработки на каждом этапе жизненного цикла API. Так, для этапов планирования и проектирования определяются стандарты и лучшие практики проектирования API, для этапов реализации и развертывания — инструменты для сборки, тестирования и развертывания API. На этапе сопровождения ведется мониторинг работы API и собирается обратная связь от пользователей.

Платформа API — это программное обеспечение для систематизации, управления и администрирования API, а также для взаимодействия с пользователями. Набор инструментов платформы позволяет командам разработчиков создавать и публиковать API, управлять ими и использовать их. Платформа API интегрируется с внешними инструментами, такими как конвейеры CI/CD, облачная инфраструктура, инструменты мониторинга, а также средства безопасности и комплаенса. Среди распространенных платформ API — Amazon API Gateway (https://oreil.ly/t87B6), Azure API Management (https://oreil.ly/LBZhB), Google Cloud Apigee (https://oreil.ly/FXFVz), Kong (https://konghq.com) и MuleSoft Anypoint Platform (https://oreil.ly/4JbwO).



API Landscape (https://apilandscape.apiscene.io) — это сайт с инструментами для проектирования, разработки и управления API. Пусть вас не смущает их огромное количество — многие из инструментов встречаются в списке несколько раз.

Будущее API

Закончить обсуждение истории и основ API логично взглядом в будущее.

По данным Statista, в 2024 году количество подключенных IoT-устройств достигло 18 млрд. Из них 7,2 млрд приходятся на подписки на смартфоны, которые приобрело около 67,1 % населения мира. Ожидается, что в ближайшие

годы количество IoT-устройств будет расти, а для реализации многих функций этих устройств необходимы сетевые API.

Провайдер CDN (Content Delivery Network) Cloudflare в отчете по безопасности API за 2024 год (https://oreil.ly/Q0QLD) сообщает, что 60 % динамического (некэшируемого) трафика приходится на API. Кроме того, API стали самой быстрорастущей категорией трафика в Cloudflare[1].

Еще одна причина возрастающего спроса на API — развитие ИИ и LLM. Обучение больших языковых моделей предполагает использование большого объема данных — текстов, изображений, аудио и видео, которые в основном поступают через API. По мере того как компании создают собственные сервисы на основе LLM, новые и имеющиеся стили API будут способствовать их развитию, выступая в роли «воронок» для входящих и исходящих данных.

Кроме того, искусственный интеллект будет способствовать увеличению количества доступных API, участвуя в проектировании, создании документации, реализации и обеспечении безопасности. Уже сегодня, если предоставить ИИ несколько примеров запросов и ответов, он может создать черновик программы для API[2].

По мере развития ИИ и увеличения количества IoT-устройств ключевым элементом API станет обеспечение безопасности. По прогнозам компании Gartner, сделанным в 2021 году, самым распространенным видом атак на корпоративные веб-приложения станут атаки на API (https://oreil.ly/52qic). В отчете Kong за 2023 год (https://oreil.ly/Xoayu) прогнозируется, что к 2027 году количество атак на API по всему миру возрастет более чем в два раза.

Для обеспечения безопасности API создаются новые протоколы и совершенствуются имеющиеся. В июне 2022 года вышла третья версия протокола HTTP (HTTP/3) (https://oreil.ly/3CK30). Для устранения недостатков WebSocket был создан протокол WebTransport (https://oreil.ly/GSqqu). Протокол MCP (Model Context Protocol) (https://oreil.ly/5uKl4) призван упростить поиск и использование данных LLM и агентным ИИ.

Одним словом, если верить прогнозам, в ближайшие годы популярность API будет только возрастать.

---

[1] См. публикацию Даниэля Молтени (Daniele Molteni) Landscape of API Traffic («Динамика трафика API») на Cloudflare (https://oreil.ly/PTWww).

[2] Высказываются предположения, что в будущем API будет состоять из «программ, которые сами разбираются, как им взаимодействовать между собой» (*Victor B*. The Future of Programming («Будущее программирования»), https://oreil.ly/qOIjv).

Резюме

В этой главе мы поговорили об основных понятиях API и погрузились в тему сетевых API, которым посвящена эта книга. Мы обсудили ключевые термины сетевых API: *сообщения*, *синхронная и асинхронная коммуникация*, а также *сетевые протоколы*. Далее мы рассмотрели историю развития API и прогнозы на будущее, указывающие на то, что в ближайшие годы ожидается бурное развитие технологий API.

Кроме того, мы классифицировали по стилям разные API, о которых пойдет речь в следующих главах. Значительная часть главы была посвящена жизненному циклу — процессу создания и управления API.

В следующей главе описаны распространенные паттерны проектирования API.

ГЛАВА 2

Паттерны проектирования API

Эта глава — практическое руководство к этапу проектирования в жизненном цикле API. Речь в ней пойдет о лучших практиках работы с API, таких как паттерны проектирования и паттерны безопасности API.

Паттерн проектирования программного обеспечения — это готовый план решения частной, но типовой проблемы, возникающей при создании ПО. Хотя нюансы процесса проектирования API разнятся в зависимости от стиля, есть ряд «универсальных» сложностей, которые не связаны с используемым стилем. Для их разрешения используются паттерны проектирования.

В этой главе паттерны представлены по отдельности, в виде списка готовых «рецептов», чтобы вы могли выбрать нужные и интегрировать их в процесс проектирования. Некоторые из них мы применим на практике в главе 5 при создании примера REST API.

Примеры паттернов проектирования API

Независимо от стиля API разработчику приходится решать следующие вопросы:

- на каком языке писать документацию;
- какую стратегию версионирования использовать;
- как называть интерфейсы;
- какое кодирование использовать;
- как предоставлять данные клиенту.

В этом разделе мы рассмотрим паттерны для решения всех перечисленных задач.

Для погружения в тему рекомендуем книгу Дж. Дж. Гивакса (JJ Geewax) *API Design Patterns* (Manning, 2021)[1], а также дополнительные источники:

- руководство по проектированию API от Google (https://google.aip.dev/1);
- рекомендации Microsoft по проектированию REST API (https://oreil.ly/q7A9o);
- обзоры REST API: The REST API Tutorial (https://restfulapi.net) и Learn REST API Design (https://www.restapitutorial.com).

Как правило, разработка API ведется без участия будущих пользователей, которые обычно выступают в качестве источников обратной связи, поэтому для промежуточной оценки проекта API и используемых в нем паттернов можно использовать LLM. Однако не стоит забывать о том, что основная задача больших языковых моделей — генерация текста, а не проектирование API, поэтому полученную от моделей информацию необходимо перепроверять.

Язык API

Стандартно API разрабатывается на английском языке, получившем статус языка ИТ благодаря ведущей роли США и Великобритании в развитии этих технологий. Хотя разговорный английский имеет множество вариантов, на письме есть два основных варианта орфографии: американский и британский. Рекомендуется использовать более распространенный — американский, если иного не требует целевая аудитория.

Наименование элементов API

Одна из сложных задач при создании API — выбор стратегии именования интерфейсов, схем, полей и параметров. Разработчику следует заранее собрать обратную связь от пользователей, поскольку после выпуска API в продакшен изменить наименования будет сложно — это потребует обновления всех имеющихся клиентов[2].

Независимо от того, что вы именуете: эндпоинт, маршрут, интерфейс, функцию, метод, процедуру, подпрограмму, свойство или параметр запроса, наименование должно быть интуитивно понятным и читабельным, а также отражать предназначение компонента и соответствовать контексту его использования. Хорошее наименование характеризует компонент однозначно и недвусмысленно. Необходимо поддерживать согласованность имен и соблюдать правила именования: использовать существительные во множественном числе, выбирать соответствующую часть речи (глагол или существительное),

---

[1] *Гивакс Д.* Паттерны проектирования API. — СПб.: Питер, 2026.

[2] См. определения понятий «клиент» и «сервер» в разделе «Socket API» главы 3.

указывать единицу измерения в полях со скалярными величинами. Рекомендуется также соблюдать соглашения об именовании, например *hungarianNotation*, *CamelCaseNotation* или *snake_notation*.

Наименования в API всегда существуют в контексте. Например, слово «канал» (channel) имеет разные значения в зависимости от того, о чем идет речь: о коммуникации, теле- или радиовещании или гражданском строительстве.



Даже самое понятное имя может быть истолковано неверно, поэтому предоставляйте вспомогательную документацию, которую разработчики API смогут использовать в спорных ситуациях.

Выбор между единственным и множественным числом при именовании ресурса зависит от стиля API. В REST API принято использовать ресурсоориентированные имена во множественном числе, например `/orders/123`. В GraphQL, gRPC и брокерских API применяется более гибкий подход, ориентированный не на адрес коллекции ресурсов, а на совершаемую операцию (intent-oriented approach). В рамках этого подхода для обозначения операций над данными допускается использование как единственного, так и множественного числа.

РЕСУРСООРИЕНТИРОВАННЫЕ И ОПЕРАЦИОННО-ОРИЕНТИРОВАННЫЕ API

API делятся на две категории: ресурсоориентированные (resource-oriented) и операционно-ориентированные (intent-oriented). Понятие ресурса обсуждается в разделе «HTTP, ресурс и URI» главы 5, но в рамках этого обсуждения вам достаточно понимать, что ресурс представляет собой данные.

Ресурсоориентированный подход к проектированию API берет начало в архитектурном стиле REST (см. подробнее в главе 5). Здесь в центре внимания находятся данные: ресурсы именуют существительными, например *bookings*, *orders*, *users*, а для манипуляции ресурсами используются HTTP-методы `POST`, `GET`, `PUT`, `PATCH`, `DELETE`. В том, что набор стандартных HTTP-методов ограничен, есть и плюсы, и минусы. С одной стороны, HTTP-методы не покрывают специальные операции, например *archive user* или *backup server*. Возможные обходные пути — реализовать пользовательские действия через поля действия внутри ресурса или поступиться принципами ресурсоориентированного подхода и вместо существительных использовать глаголы для именования эндпоинтов. Достоинство стандартных HTTP-методов заключается в том, что они применяются во всех ресурсоориентированных API. Соответственно, использование API становится более предсказуемым.

В *операционно-ориентированных API* в центре внимания находятся операции над данными. Имена интерфейсов обычно содержат глаголы, например *createUser*, *archiveUser*, *backupServer*. Известный пример операционно-ориентированного REST API — Stripe Payment Intents (https://oreil.ly/HRU7q). Этот подход более гибкий, но приводит к тому, что вариативность наименований повышается, а количество эндпоинтов увеличивается, поскольку под каждую операцию с данными создается отдельный эндпоинт.

Чтобы подчеркнуть различие, ресурсоориентированные API часто называют декларативными (declarative API), а операционно-ориентированные — императивными (imperative API).

В декларативных API главное — конечное состояние ресурса, а не конкретные шаги для достижения этого состояния. Названия стандартных HTTP-методов, такие как `GET` или `POST`, скрывают детали операций над данными. Соответственно, REST API относится к декларативным.

Императивные API, напротив, ориентированы на действия: их названия отражают, как достичь желаемого результата. API gRPC относятся к императивным, поскольку приоритет в них отдается операциям и процедурам, совершаемым над данными (см. подробнее в главе 8). API GraphQL сочетают признаки декларативного и императивного подходов: запросы в них декларативны, а мутации — императивны (подробнее об API GraphQL см. в главе 6).

Примеры названий интерфейсов для REST API, gRPC, GraphQL и RabbitMQ приведены в табл. 2.1, 2.2, 2.3 и 2.4 соответственно. Обратите внимание, что порядок операций в таблицах соответствует шаблону CRUD: Create — «создание», Read — «чтение», Update — «обновление», Delete — «удаление» (подробнее о CRUD поговорим далее в этой главе).

Таблица 2.1. Ресурсоориентированный REST API

Операция	HTTP-метод	Эндпоинт
Создать новый заказ	`POST`	`/api/orders`
Получить все заказы	`GET`	`/api/orders`
Получить один заказ	`GET`	`/api/orders/{orderID}`
Обновить заказ	`PUT`	`/api/orders/{orderID}`
Удалить заказ	`DELETE`	`/api/orders/{orderID}`

Таблица 2.2. Операционно-ориентированный gRPC API

Операция	RPC-метод
Создать новый заказ	`CreateOrder()`
Получить все заказы	`ListOrders()`
Получить один заказ	`GetOrder()`
Обновить заказ	`UpdateOrder()`
Удалить заказ	`DeleteOrder()`

Таблица 2.3. GraphQL API, сочетающий признаки операционно-ориентированного и ресурсоориентированного подходов

Действие	Операция GraphQL	Поле/операция
Создать новый заказ	Мутация	`createOrder`
Получить все заказы	Запрос	`orders`
Получить один заказ	Запрос	`order`
Обновить заказ	Мутация	`updateOrder`
Удалить заказ	Мутация	`deleteOrder`

Таблица 2.4. Операционно-ориентированный брокерский API RabbitMQ

Операция	Тип сообщения	Ключ маршрутизации
Создать новый заказ	Команда	`orders.create`
Получить все заказы	Запрос	`orders.list`
Получить один заказ	Запрос	`orders.get`
Обновить заказ	Команда	`orders.update`
Удалить заказ	Команда	`orders.delete`

Еще одна особенность — образование множественного числа существительных. В английском языке множественное число образуется с помощью окончаний *-s* или *-es*. Однако есть исключения, например слова *die («игральная кость»)*, *person («человек»)* и *mouse («мышь»)*, для которых формы множественного числа — *dice*, *people* и *mice* соответственно. В подобных случаях при наименовании элементов интерфейса используется форма-исключение или подбирается другое название.

Поля ответа API часто содержат скалярные величины, например температуру, скорость, высоту, давление, размер, длительность, длину и вес. При этом чаще всего не указывается, какая система измерения используется — метрическая (Международная система единиц, СИ (https://oreil.ly/sxYjG)) или британская (https://oreil.ly/ocDC6). В таких случаях разработчики могут предположить, что по умолчанию используется удобная для них единица измерения, что часто приводит к необратимым последствиям (https://oreil.ly/IrAJR). Например, в 2003 году в токийском «Диснейленде» американские горки сошли с рельсов из-за того, что ось задних колес у одной из вагонеток оказалась слишком короткой: при переводе из одной системы измерения в другую произошла ошибка (https://oreil.ly/U1MIk). По этой же причине в 1999 году провалилась миссия NASA Mars Climate Orbiter: спутник, двигавшийся в сторону Марса, отклонился от траектории и сгорел в атмосфере из-за того, что в его двигателе были запрограммированы единицы метрической системы, а команды с Земли поступали в британской (https://oreil.ly/JtOLy). Чтобы избежать подобных ошибок, в название скаляра включается единица измерения, например `durationSeconds`, `lengthMeters`, `temperatureCelsius`, `weightGrams` или `widthPixels`.

Эволюция API

API постоянно развиваются, поэтому необходимо предусмотреть стратегию версионирования, чтобы API продолжал работать в процессе внесения изменений.

Изменения в API могут затрагивать два уровня: уровень интерфейса (эндпоинт, процедура, запрос или мутация) и уровень ресурса (сообщение). Кроме того, изменения могут быть аддитивными (добавление новой функциональности) и субтрактивными (удаление имеющейся). Модификация существующей функциональности — это субтрактивное изменение, за которым следует аддитивное. Влияние изменения на систему API может быть критическим или некритическим[1].

В табл. 2.5 показаны критические и некритические последствия изменений для клиентов REST API. Изменения оказывают аналогичное влияние на систему и в других стилях. Все примеры в табл. 2.5 предполагают выполнение HTTP-запроса `GET` к фиктивному эндпоинту `https://api.example.com/v1/die/roll` и получение ответа в формате JSON (пример 2.1).

---

[1] См. обсуждение критических изменений и версионирования ПО в докладе Speculation Keynote («Ключевые прогнозы») Рича Хики (Rich Hickey) (https://oreil.ly/azu75).

Пример 2.1. Полезная нагрузка ответа API в формате JSON (JavaScript Object Notation)

```
{
  "data": {
    "message": "Rolling 5",
    "outcome": 5
  }
}
```

Рассмотрим табл. 2.5, в которой представлены возможные изменения в API и их последствия.

Таблица 2.5. Изменения в API и их последствия

Изменение	Последствие	Пример
Переименование эндпоинта	Критическое	Переименование `/v1/roll/die` в `/v1/roll-die`
Удаление версии API	Критическое	Удаление `v1`
Изменение поддерживаемого типа контента или прекращение поддержки	Критическое	Запрос `application/xml` вместо `application/json` или изменение `application/xml` на `application/json`
Удаление поля	Критическое	Удаление поля `outcome`
Переименование поля	Критическое	Переименование поля `outcome` в `roll`
Изменение типа данных поля	Критическое	Изменение `int (5)` на `string ("5")`
Добавление нового поля или эндпоинта	Некритическое	Добавление поля `sides` в `data`
Добавление параметра запроса со значением по умолчанию	Некритическое	Добавление параметра `sides` (например, `/v1/roll-die?sides=6`)
Добавление нового типа поддерживаемого контента	Некритическое	Добавление `application/xml`
Добавление необязательных пользовательских заголовков	Некритическое	Добавление заголовка `Example-Service-ID`

Обратите внимание: в REST API переименование, удаление или изменение порядка представления данных приводит к критическим изменениям для клиентов, использующих функции, связанные с этими данными[1].

Критические изменения не обязательно происходят со стороны провайдера API — они также могут возникать из-за нетипичного поведения клиента,

[1] Если клиент ожидает, что данные будут появляться в определенном порядке, то изменение этого порядка скажется на эффективности взаимодействия клиента с API.

который ожидает от API выполнения функций, не заложенных разработчиком. Например, если клиент ожидает, что поле `status` вернет значение `enabled`, а в самом API запрограммировано значение `active`, то для клиента логика взаимодействия нарушится, хотя со стороны API ошибки не произойдет.

API: ПРЯМАЯ И ОБРАТНАЯ СОВМЕСТИМОСТЬ

Обратная совместимость (backward compatibility) означает, что новая (то есть измененная) версия API совместима со старыми клиентами. *Прямая совместимость* (forward compatibility) предполагает, что старые версии API совместимы с новыми клиентами.

Обратная совместимость особенно важна для клиент-серверных API. Здесь предпочтительны аддитивные изменения — они обычно сохраняют совместимость со старыми клиентами, которые могут игнорировать новую функциональность. Однако после развертывания API могут возникнуть ситуации, когда внести обратно совместимое изменение невозможно, — например, нужно изменить тип данных в полезной нагрузке, исправить опечатку в поле или удалить эндпоинт. Такие изменения могут потребовать развертывания новой версии API и прекращения поддержки старых клиентов.

Прямая совместимость более критична в API на основе обмена сообщениями — событийно-ориентированных систем, систем с очередями и архитектур «издатель — подписчик» (pub/sub). В них отправители и получатели сообщений независимы друг от друга и могут обновляться в разное время. Соответственно, в системе могут одновременно передаваться и обрабатываться сообщения разных версий. Сильная сторона таких систем — относительная автономность компонентов, однако при проектировании системы необходимо учитывать не только обратную, но и прямую совместимость.

Версионирование API

API часто называют *контрактом* — соглашением между клиентом и провайдером. Контракт описывает структуру данных, формат и возможное поведение API. Он служит спецификацией, которая обеспечивает «взаимопонимание» между клиентом и сервером. Спецификация определяет, какие функции предлагает API и как клиент может ими пользоваться. На практике контракты, как и сами API, часто меняются. Версионирование отражает изменения в контракте и позволяет старым клиентам продолжать использовать старую версию API, а новым — работать с последней версией.

В табл. 2.6 представлено несколько подходов к версионированию API.

Таблица 2.6. Подходы к версионированию API

Подход к версионированию	Пример
Версионирование через URL-путь	• `https://example.com/v1/orders` • `https://api.v1.example.com/orders`
Версионирование через параметр запроса	• `https://example.com/orders?api-version=v1` • `https://example.com/v1/orders?resource-version=2.1.3`
Версионирование через заголовки	• `Api-Version: v1` • `Accept: application/vnd.example+json;api-version=1` • `Accept: application/vnd.example+json,resource-version=2` • `Accept: application/vnd.example+json; api-version=1;resource-version=2.1.3`
Версионирование на уровне сообщения	`{ "version" : "v1", "data": {...} }`

Вот более подробное описание этих подходов.

Версионирование через URL-путь

Номер версии API становится самостоятельной частью URL-пути (например, `/api/v1/resource` или `/api/v2/resource`). Такой подход четко сигнализирует клиентам о смене версий и хорошо подходит для API с критическими изменениями или независимо развивающимися ресурсами. Версия применяется ко всему API, поэтому метод работает на высоком уровне абстракции: в отличие от версионирования через заголовки, версионирование в URL не позволяет точечно управлять отдельными эндпоинтами. Без своевременного рефакторинга (например, унификации библиотек для сохранения единой логики) сопровождение нескольких версий может привести к дублированию кода и увеличению трудозатрат на обслуживание.

Версионирование через параметр запроса

Номер версии API отражается в URL как параметр запроса. Преимущество такого подхода в том, что меняется только значение ресурса. Базовый URL остается постоянным для всех клиентов и не требует изменений, пока эндпоинты не переименуют или не удалят. Такой подход к версионированию более точечный, поскольку применяется на уровне запроса, а не привязывается к идентификатору ресурса.

Недостаток такого версионирования заключается в том, что оно усложняет маршрутизацию запросов, поскольку маршрут не определяется путем в URL

(как при версионировании через URL-путь), а обрабатывается внутри функции, принимающей запрос. Это увеличивает объем кода и требует дополнительной логики направления запросов к нужному обработчику (классу или функции) для конкретной версии API. Кроме того, каждый ресурс нужно документировать на уровне параметров запроса.

Версионирование через заголовки

В этом подходе версия API хранится в заголовке запроса, например `Accept-Version: v1`, что позволяет клиентам использовать один и тот же URL независимо от эволюции API. Такой подход менее прозрачен, чем версионирование через URL-путь или параметр запроса, поскольку информация о версии не видна в адресной строке. Вдобавок низкая прозрачность усложняет отладку и логирование, поскольку разработчикам и тестировщикам приходится проверять заголовки запросов, чтобы определить версию API. Хотя версионирование через заголовки соответствует соглашениям о типах контента RESTful, оно может вносить элемент неоднозначности и усложнять управление версиями. Кроме того, некоторые промежуточные (прокси) системы могут удалять или блокировать пользовательские заголовки.

Версионирование на уровне сообщения

Версионирование можно выполнять на уровне содержимого сообщений — для номера версии в схеме сообщения создается специальное поле. Такую стратегию часто применяют в событийно-ориентированной архитектуре и в брокерских API, где сообщения могут храниться долго, ожидая обработки. Преимущество данного подхода заключается в том, что можно одновременно отправлять сообщения в разных версиях. Недостаток — получателю сообщений придется обрабатывать каждое версионированное сообщение отдельно.



Стратегии версионирования можно комбинировать. Например, можно одновременно использовать версионирование через URL-путь и через параметр запроса — в результате получится эндпоинт вида /api/v1/orders/123?version=1.2.3. Этот подход менее предпочтителен, поскольку усложняет работу приложения, затрудняет идентификацию ресурсов и приоритизацию, а также нарушает механизмы кэширования.

В операционно-ориентированных API, таких как GraphQL, версию обозначают через изменение имени поля (например, updateOrderV2). В gRPC, напротив, предпочитают версионировать имя пакета и сервиса. Добавление суффикса версии к именам методов gRPC (например, UpdateOrderV2) возможно, но встречается реже. Наконец, можно полностью отказаться от версионирования API[1].

[1] Рой Филдинг на конференции Evolve 2013 рекомендовал избегать внесения критических изменений: «НЕ ДЕЛАЙТЕ ЭТОГО — версионирование интерфейса тайно уничтожает приложения, находящиеся в эксплуатации».

Необходимо также выбрать формат версионирования API. Есть несколько подходов, которые можно комбинировать.

Семантическое версионирование (https://semver.org)

Используются метки версий в формате `MAJOR.MINOR.PATCH` (например, 1.3.5), где `MAJOR` указывает на критическое изменение, `MINOR` — на обратно совместимое изменение, а `PATCH` — на обратно совместимое исправление ошибки.

Календарное версионирование (https://calver.org)

В качестве меток версий используются даты выпуска версий (например, `2024-11-01` или `2024.47`).

Хеш-версионирование (https://oreil.ly/4j6UK)

В качестве метки версии используется хеш, например хеш коммита системы контроля версий (`237a2b4f`).

Основная сложность версионирования — необходимо отразить в одной метке версии множество изменений API. Поэтому важно не только правильно подобрать формат версии — важно проинформировать пользователей о новых функциях и критических изменениях API, а также дать время на адаптацию. Для информирования о предстоящих изменениях можно использовать такие каналы коммуникации, как журнал изменений на сайте с документацией API, Atom-ленты или добавлять метаданные о предстоящих изменениях в ответы API.



Для описания и проверки структуры JSON-данных используется JSON Schema (https://json-schema.org). Этот инструмент помогает определить, приведет ли внесенное изменение к нарушению работы API.

Работайте только с проверенными онлайн-инструментами и не передавайте конфиденциальную информацию третьим лицам.

Кодирование

При увеличении объема данных растут и расходы на их передачу между системами. Пока данные измеряются в мегабайтах, это может быть незаметно, но при переходе к гигабайтам временны́е и финансовые затраты на передачу данных становятся значительными. Например, большой поток данных через API может перегрузить принимающую систему и снизить производительность. Аналогично чрезмерный объем выходных данных создает нагрузку на системные ресурсы и увеличивает затраты на облачные услуги. В обоих случаях качество взаимодействия пользователя с системой ухудшается. Для решения части этих проблем используется сжатие — разновидность кодирования.



Кодирование — это процесс преобразования данных из одного формата в другой. Декодирование — обратный процесс, преобразующий закодированные данные в исходный формат.

В контексте API цикл кодирования/декодирования состоит из получения данных из отправляющей системы, их кодирования в определенный формат и передачи в принимающую систему. Принимающая система декодирует полученные данные, использует их, формирует ответ, который затем кодируется и отправляется обратно отправителю.



«Кодирование» и «сжатие» не синонимы. Сжатие (уменьшение размера данных) всегда предполагает кодирование, однако кодирование не всегда подразумевает сжатие.

Сжатие — это частный случай кодирования, который сокращает объем данных ценой увеличения нагрузки на CPU. При выборе алгоритма и степени сжатия необходимо соблюдать баланс между желаемым размером данных и вычислительными ресурсами системы.

С кодированием также связано шифрование, но его цель — защита данных, которая часто сопровождается увеличением их размера.

Передаваемые или хранимые данные могут кодироваться в текстовом или двоичном формате[1]. Текстовый формат удобен разработчикам для отладки и ведения логов, поскольку может читаться и компьютером, и человеком. Однако универсальная читаемость имеет недостаток: текстовые данные занимают больше места и медленнее обрабатываются компьютерами. К популярным текстовым форматам кодирования относятся ASCII (https://oreil.ly/DsP8w), UTF-8 (https://oreil.ly/WP71s), XML (https://oreil.ly/OIkpM), JSON (https://www.json.org) и YAML (https://yaml.org).

В примере 2.2 показано кодирование и представление текста на латинице.

Пример 2.2. Кодирование ASCII, шестнадцатеричное и двоичное

```
# Запись текста в файл  ❶
echo -n "Hello" > message

# Определение типа файла  ❷
file --brief message
ASCII text, with no line terminators

# Вывод двоичного содержимого файла  ❸
xxd -groupsize 1 -bits message
00000000: 01001000 01100101 01101100 01101100 01101111   Hello
```

[1] Некоторые прикладные протоколы поддерживают компрессию текстовых данных на уровне сообщений. Например, в HTTP есть алгоритмы сжатия gzip или Brotli.

```
# Вывод содержимого файла в шестнадцатеричном и ASCII-формате  ❹
xxd -groupsize 1 message
00000000  48 65 6c 6c 6f  Hello

# Преобразование шестнадцатеричного кода в ASCII-текст  ❺
echo -n "48 65 6c 6c 6f" | xxd -plain -revert Hello
```

❶ Команда `echo` записывает текст Hello в файл `message`. Параметр `-n` изменяет поведение `echo` по умолчанию, чтобы не добавлялся автоматический перенос на новую строку.

❷ Команда `file` проверяет содержимое файла и кратко описывает его тип — текстовый или двоичный.

❸ Команда `xxd` выводит содержимое файла. В первом столбце в шестнадцатеричном формате отображается позиция первого байта каждой строки относительно начала файла. В последующих столбцах содержатся значения каждого байта. В данном случае мы получаем двоичное представление пяти 8-битных слов. В последнем столбце показано ASCII-представление содержимого файла.

❹ Команда `xxd` выводит содержимое файла в шестнадцатеричном и ASCII-формате.

❺ Команда `echo` записывает пять шестнадцатеричных чисел и передает их программе `xxd` для преобразования в ASCII-текст.

Двоичное кодирование, как видно из названия, представляет данные в виде последовательности нулей и единиц. Его преимущества — меньший размер закодированных данных и более быстрый парсинг (поскольку компьютеры анализируют числа быстрее, чем текст). Например, число 12 345 в виде ASCII-текста занимает 5 байт, а в двоичном формате — всего 14 бит (чуть меньше 2 байт): 11000000111001. Кроме того, двоичные форматы часто опускают избыточные метаданные. К популярным протоколам и форматам двоичного кодирования относятся Amazon Ion (https://oreil.ly/MG5Bt), Apache Avro (https://avro.apache.org), MessagePack (https://msgpack.org), Protocol Buffers (https://protobuf.dev) и Apache Thrift (https://thrift.apache.org).

В примере 2.3 показано кодирование и декодирование данных с использованием формата сериализации Avro, а в примере 2.4 — анализ двоично-кодированного файла Avro с помощью утилит `file` и `xxd`.



Код в книге оптимизирован для удобства чтения, поэтому не всегда полностью соответствует исходному коду из репозитория. Сокращенные фрагменты обозначены многоточием (...).

КОДИРОВАНИЕ И СЕРИАЛИЗАЦИЯ

Кодирование — общий процесс изменения формата данных, а *сериализация* — преобразование структуры данных языка программирования в последовательность байтов для хранения или передачи. *Десериализация* — обратный процесс, восстанавливающий исходную структуру данных из потока байтов. Процессы сериализации и десериализации показаны на рис. 2.1.

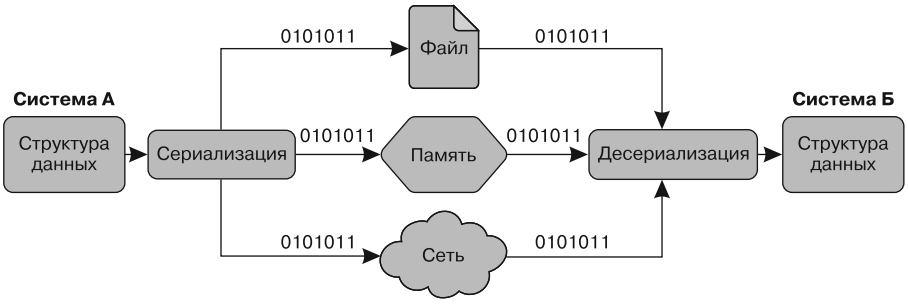


Рис. 2.1. Сериализация и десериализация данных

Пример 2.3. Сериализация Apache Avro на Python

```
import avro.schema
from avro.datafile import DataFileReader, DataFileWriter
from avro.io import DatumReader, DatumWriter

user_avsc = ' ' '{   ❶
 "type": "record",
 "name": "User",
 "fields": [
    {"name": "email", "type": "string"},
    {"name": "active", "type": "boolean", "default": false},
    {"name": "access_level", "type": "int", "default": 0}
 ]
}' ' '
schema = avro.schema.parse(user_avsc)  ❷

with DataFileWriter(open("users.avro", "wb"), DatumWriter(), schema) as writer:  ❸
    writer.append({"email": "ld@example.com", "active": False, "access_level": 0})
    writer.append({"email": "md@example.com", "active": True, "access_level": 3})

with DataFileReader(open("users.avro", "rb"), DatumReader()) as reader:  ❹
    for user in reader:
        print(user)
```

```
# Вывод программы:
# {'email': 'ld@example.com', 'active': False, 'access_level': 0}
# {'email': 'md@example.com', 'active': True, 'access_level': 3}
```

❶ Многострочная строка содержит JSON-определение схемы Avro. Схема определяет запись Avro, представляющую пользователя. Тип записи должен содержать ключи `type`, `name` и `fields`. Ключ `fields` определяет массив JSON-объектов со свойствами для сериализации/десериализации записей: `name`, `type` и `default`.

❷ JSON-строка преобразуется в схему Avro.

❸ Объект `writer` добавляет две записи в двоичный файл `users.avro`. В файл добавляются и схема Avro, и данные.

❹ Объект `reader` открывает файл `users.avro`, десериализует его и выводит каждую запись из файла. На выходе получаем десериализованную запись.

Пример 2.4. Анализ двоично-кодированного файла Avro

```
file --brief users.avro  ❶
Apache Avro version 1

xxd -groupsize 1 users.avro  ❷
00000000: 4f 62 6a 01 04 14 61 76 72 6f 2e 63 6f 64 65 63  Obj...avro.codec
00000010: 08 6e 75 6c 6c 16 61 76 72 6f 2e 73 63 68 65 6d  .null.avro.schem
00000020: 61 86 03 7b 22 74 79 70 65 22 3a 20 22 72 65 63  a..{"type": "rec
00000030: 6f 72 64 22 2c 20 22 6e 61 6d 65 22 3a 20 22 55  ord", "name": "U
00000040: 73 65 72 22 2c 20 22 66 69 65 6c 64 73 22 3a 20  ser", "fields":
00000050: 5b 7b 22 74 79 70 65 22 3a 20 22 73 74 72 69 6e  [{"type": "strin
00000060: 67 22 2c 20 22 6e 61 6d 65 22 3a 20 22 65 6d 61  g", "name": "ema
00000070: 69 6c 22 7d 2c 20 7b 22 74 79 70 65 22 3a 20 22  il"}, {"type": "
00000080: 62 6f 6f 6c 65 61 6e 22 2c 20 22 6e 61 6d 65 22  boolean", "name"
00000090: 3a 20 22 61 63 74 69 76 65 22 2c 20 22 64 65 66  : "active", "def
000000a0: 61 75 6c 74 22 3a 20 66 61 6c 73 65 7d 2c 20 7b  ault": false}, {
000000b0: 22 74 79 70 65 22 3a 20 22 69 6e 74 22 2c 20 22  "type": "int", "
000000c0: 6e 61 6d 65 22 3a 20 22 61 63 63 65 73 73 5f 6c  name": "access_l
000000d0: 65 76 65 6c 22 2c 20 22 64 65 66 61 75 6c 74 22  evel", "default"
000000e0: 3a 20 30 7d 5d 7d 00 4b e1 68 72 66 79 5c 50 ac  : 0}]}.K.hrfy\P.
000000f0: a2 1e 91 95 bb 11 f9 04 44 1c 6c 64 40 65 78 61  ........D.ld@exa
00000100: 6d 70 6c 65 2e 63 6f 6d 00 00 1c 6d 64 40 65 78  mple.com...md@ex
00000110: 61 6d 70 6c 65 2e 63 6f 6d 01 06 4b e1 68 72 66  ample.com..K.hrf
00000120: 79 5c 50 ac a2 1e 91 95 bb 11 f9 y\P........
```

❶ Команда `file` показывает, что `users.avro` — это файл Apache Avro версии 1.

❷ Команда `xxd` отображает содержимое файла в шестнадцатеричном и ASCII-представлении.

При передаче данных между системами нужно учитывать, что корректность сериализации и десериализации зависит от того, как один и тот же тип данных

обрабатывается в разных языках программирования. Например, JSON — это независимый от языка формат, который поддерживает ограниченный набор типов JavaScript: `object`, `array`, `string`, `number`, `boolean` и `null`.

В примере 2.5 показаны неожиданные результаты сериализации и десериализации JSON в Python и JavaScript.

Пример 2.5. Сериализация и десериализация JSON

```
# Преобразование строки JSON в словарь Python  ❶
python -c "import json; print(json.loads('{\"a\": null, \"b\": true}'))"
{'a': None, 'b': True}

# Преобразование объекта JavaScript в строку JSON  ❷
node --eval "console.log(JSON.stringify({number: 9007199254740999}))"
{"number": 9007199254741000}

# Парсинг строки JSON и чтение значения  ❸
node --eval "o = JSON.parse('{\"a\": 0.3}'); console.log(o.b)"
undefined
```

❶ При десериализации строки с JSON в Python значения JavaScript `null` и `true` интерпретируются как `None` и `True`.

❷ Попытка представить число за пределами диапазона, допустимого в выбранном типе данных, приводит к неточности в расчетах. Число `9007199254740999` округляется до `9007199254741000`, и сериализация не решает эту проблему.

❸ Обращение к отсутствующему значению в десериализованной строке JSON возвращает неопределенное значение JavaScript `undefined` вместо ошибки.



Представление чисел в виде строк облегчает перенос данных при работе в системах с поддержкой разных числовых типов. Строковый тип данных особенно актуален, когда речь идет о больших целых числах или когда число необходимо сохранить в исходном виде и округление не допускается. Однако строки занимают больше памяти и медленнее обрабатываются. Кроме того, если принимающая система попытается конвертировать строку в нативный числовой тип, то точность все равно может утратиться.

Один тип данных в системе API всегда должен обрабатываться единообразно. Например, нужно определить, как будет обрабатываться `null`: как пустая строка `""`, неопределенное значение `undefined` или как специальный тип `null` (например, `None` в Python). Вдобавок нужно установить правило для обработки отсутствующих полей. В некоторых форматах сериализации, например Protocol Buffers, используется значение по умолчанию, если поле сообщения отсутствует. Так, для `int32` значение по умолчанию — `0`, для `string` — `""`, для `bool` — `false`.

При кодировании логических значений (`true` и `false`) необходимо использовать стандартные значения логического типа, принятые в выбранном формате данных (например, `true`/`false` в JSON). Не следует представлять логические значения в виде чисел (`0` — `false` и `1` — `true`) или в виде строк (`no` и `yes` или `on` и `off`) — это может нарушить совместимость между разными системами[1]. Логические переменные состояния лучше именовать положительно, например `allowed` вместо `disallowed`, `enabled` вместо `disabled`. Человек лучше понимает отрицание положительного имени (`!enabled` — «не включено»), чем двойное отрицание (`!disabled` — «не отключено», то есть «выключено»). По возможности используйте значение по умолчанию для каждого логического типа и избегайте префиксов (например, `activated` вместо `isActive`)[2].

Кроме того, следует учитывать ограничения типов данных, связанных с полями API[3]. Например, для данных в поле «Фамилия» максимальной длины 255 символов может быть недостаточно. Лучше остановиться на 1024 символах — это значение достаточно велико, чтобы вместить самое длинное имя в мире (747 символов)[4]. Выбрать минимальную длину поля тоже непросто. Насколько коротким может быть имя? Четыре символа, два или даже один?



Ограничения значений и значения по умолчанию обсуждаются в книге Humans vs Computers Гойко Аджича (Neuri Consulting, 2017) (https://oreil.ly/V7uDc). Если у вас нет возможности прочитать всю книгу, то послушайте доклад автора People That Make Computers Go Crazy («Люди, которые сводят компьютеры с ума») (https://oreil.ly/Bm3Dm).

Многие языки программирования поддерживают *перечисления* (enums) — структуры данных, которые объединяют группу констант в один тип. Двоичные форматы сериализации, такие как Avro или Protocol Buffers, поддерживают перечисления нативно, а текстовые сериализаторы, например JSON, — нет. При работе с форматами, не поддерживающими перечисления, для представления значений перечисления (enum) можно использовать строки. Однако представление перечисления в виде строки требует дополнительных затрат: в приложении должна быть реализована логика для проверки того, что полученное строковое значение является валидным.

---

[1] Вспомним, например, проблему Норвегии в YAML 1.1: двухбуквенный код Норвегии No в YAML 1.1 считывается как false (https://oreil.ly/Iesmx).

[2] См. документацию Google API по логическим типам (https://oreil.ly/DM6X9).

[3] Распространенный способ проверки ограничений входных полей — использование регулярных выражений (Regex).

[4] См. самое длинное имя человека в Книге рекордов Гиннесса (https://oreil.ly/C3nOu).

Фильтрация

Фильтрация позволяет извлекать конкретные данные из большего набора по заданным критериям. В REST API фильтрация чаще всего реализуется через добавление критериев в виде параметров запросов в URI (например, https://example.com/api/v1/books?order_by=published_date&language=en). Есть два подхода к передаче критериев фильтрации, и оба предполагают использование параметров запроса.

В первом подходе используется *строка фильтра*, которая передается в API через параметр запроса. Ее формат зависит от базовой технологии хранения данных. Например, при работе с SQL-базой данных в строку фильтра можно добавить часть SQL-запроса, которую обработает бэкенд: `country='Denmark' AND snowfall >= 0.5 AND rain >= 0.1`. Не следует передавать полные SQL-запросы (например, `SELECT * FROM weather WHERE country='Denmark' AND snowfall >= 0.5 AND rain >= 0.1`), поскольку это создает риск SQL-инъекции, когда в исполняемый запрос встраивается вредоносный SQL-оператор, такой как `DROP`.

Преимущество использования строки фильтра заключается в том, что система адаптируется к переменам в формате данных, не внося изменений в API. За парсинг, очистку и валидацию изменений в базовой модели данных отвечает API, поэтому клиенту достаточно обновить строку фильтра в параметре запроса. Другими словами, при обновлении базовой модели данных клиенту не придется менять запрос — все изменения обработаются в API. Например, если в базовой модели столбец базы данных переименуют с `snowfall` на `snowfall_amount`, то клиенту потребуется обновить строку фильтра, но для API логика обработки запроса не изменится. API будет использовать новую схему (`snowfall_amount >= 0.5`), не изменяя свою кодовую базу. Недостаток подхода заключается в том, что клиент должен уметь формировать строку фильтра в соответствии с актуальной схемой запроса к базе данных. Кроме того, для интерпретации этой строки требуется пользовательский парсер.

При втором подходе — *фильтрация по полям* — используются параметры запроса для передачи условий фильтрации (например, `country=Denmark&snowfall>=0.5&rain>=0.1`). Этот подход более контролируемый, но менее гибкий, поскольку логику и структуру фильтрации определяет API. Фильтрация по полям имеет несколько преимуществ: сниженный риск SQL-инъекций благодаря сериализации и валидации входных данных, упрощенная интеграция с бэкенд-фреймворками API и более точная локализация ошибок, поскольку известно, какое поле проходит валидацию. Этот подход является наиболее распространенным, но усложняет адаптацию к изменениям в базовой модели данных. Это, в свою очередь, увеличивает затраты на поддержку API при обновлении модели.

Выбор метода фильтрации всегда остается за разработчиком. Если фреймворк предоставляет встроенные механизмы фильтрации, то следуйте его рекомендациям.

Кроме того, с помощью фильтрации можно определять структуру данных в ответе API. Так, GraphQL позволяет клиенту запрашивать данные из конкретных полей базы данных, обозначенных в запросе. В REST API клиент может отфильтровать ненужные поля, например, через параметр `exclude_fields`, но реализация такой функции требует дополнительной разработки на стороне API.

Подсчет и сортировка

Подсчет и сортировка записей — распространенная функциональность, необходимая для фронтенд-приложений. При *подсчете* определяется общее количество записей, соответствующих запросу, а его результаты используются при пагинации (см. подраздел «Пагинация» ниже). *Сортировка* — это упорядочение данных по возрастанию или убыванию на основе таких параметров, как время, имя или число. Однако эта функциональность может негативно сказываться на производительности системы.

В небольших системах подсчет и сортировка выполняются быстро, и создается впечатление, что система оптимизирована. Однако по мере роста объема данных производительность может ухудшаться, например, из-за отсутствия индексов базы данных или проблемы $n + 1$ (https://oreil.ly/CaKMC). Эта проблема возникает, когда для получения связанных данных отправляется несколько запросов вместо одного. В реляционных базах данных эта проблема обычно связана с отсутствием SQL-операторов JOIN.

Еще сложнее дела обстоят с распределенными системами. При распределении данных по нескольким узлам операции подсчета и сортировки требуют координации, что увеличивает время отклика. Кроме того, снижается отзывчивость API, если база данных не оптимизирована для операций агрегирования данных.

Механизмы подсчета и сортировки можно (и нужно) реализовывать, если на бэкенде используется нераспределенная база данных. В распределенных системах эти механизмы потребуют поддержки распределенной обработки данных. Для больших наборов данных рекомендуется ограничить или отключить подсчет и сортировку или рассмотреть их более гибкие альтернативы: приблизительные подсчеты, кэширование метаданных или отложенную обработку.

Пагинация

Пагинация (pagination или paging) — это паттерн проектирования, который позволяет загружать большие наборы данных постранично (частями). Каждая страница содержит подмножество данных, определяемое размером страницы

(количеством элементов), и предоставляет ограниченный обзор набора данных — в любой момент времени пользователь видит только часть набора. Пользователи или системы могут перемещаться по всему набору данных, перелистывая страницы вперед или назад, переходить к следующей/предыдущей странице или на конкретную страницу.

Есть несколько подходов к реализации пагинации. Ниже речь пойдет о наиболее распространенных — *пагинации со смещением* (offset-based pagination) и *курсорной пагинации* (cursor-based pagination).

Пагинация со смещением

Этот популярный метод имеет несколько вариантов. В первом варианте в параметре запроса используется номер страницы (пример 2.6). Обратите внимание на то, что значение `page` в полях `next` и `previous` увеличивается и уменьшается на единицу соответственно, а поле `count` показывает количество объектов в наборе данных.

Пример 2.6. Пагинация со смещением: использование номера страницы в параметре запроса page

```
curl https://example.com/api/v1/orders?page=2&page_size=100
{
  "count": 1478,
  "next": "https://example.com/api/v1/orders?page=3&page_size=100",
  "previous": "https://example.com/api/v1/orders?page=1&page_size=100",
  "results": [...]
}
```

Во втором варианте пагинации со смещением используются параметры запроса `limit` и `offset` (пример 2.7). Параметр `limit` определяет общее количество возвращаемых элементов, а `offset` указывает, с какой позиции в полном наборе данных начинать возвращать результаты.

Пример 2.7. Пагинация со смещением: использование параметров запросов limit и offset

```
curl https://example.com/api/v1/orders?limit=100&offset=400
{
  "count": 1478,
  "next": "https://example.com/api/v1/orders?limit=100&offset=500",
  "previous": "https://example.com/api/v1/orders?limit=100&offset=300",
  "results": [...]
}
```

Преимущество пагинации со смещением заключается в том, что клиент может увидеть общее количество страниц (на основе количества записей и размера страницы) и перейти на конкретную страницу. Недостаток — при работе

с большими наборами данных время отклика увеличивается. Это происходит потому, что перед тем, как вернуть запрошенный результат, база данных сканирует или пролистывает множество записей, чтобы найти требуемую позицию в наборе данных. Например, при запросе страницы 1000 базе данных сначала придется пролистать записи предыдущих 999 страниц.

Еще один недостаток пагинации со смещением проявляется при удалении или добавлении элементов. Например, при удалении элемента с предыдущей страницы последующие элементы сдвигаются на одну позицию назад. В результате элементы, которые должны были отобразиться на следующей странице, могут быть пропущены. Аналогично при добавлении элемента на предыдущую страницу последующие элементы сдвигаются на одну позицию вперед, что приводит к дублированию (например, один и тот же элемент может занять последнюю позицию на предыдущей странице и первую на текущей).

В табл. 2.7 показано, как удаление или добавление элементов влияет на пагинацию. Обратите внимание на то, что после удаления элемента на последней странице становится на один элемент меньше, а после добавления страницы сдвигаются вперед и один элемент перемещается на следующую страницу.

Таблица 2.7. Влияние удаления и добавления элементов на пагинацию

№ страницы	Изначальная пагинация	После удаления B	После добавления Z
1	A, B, C	A, C, D	A, Z, B
2	D, E, F	E, F, G	C, D, E
3	G, H, I	H, I	F, G, H

Пагинация со смещением основана на пропуске фиксированного количества элементов. Если во время пагинации перед нужной позицией удаляются или добавляются элементы, то набор данных сдвигается. В результате при запросе последующих страниц клиент либо пропустит элементы (при удалении), либо увидит дублирующиеся записи (при добавлении).

Курсорная пагинация

В этом подходе для обозначения позиции в наборе данных используется курсор (указатель). Он должен быть уникальным, последовательным и сортируемым значением. Пример такого значения — первичный ключ базы данных. Курсор кодируется в виде токена, который может расшифровать API, но не клиент. API знает, как «декодировать» курсор и использовать его для навигации по базе данных.

Распространенный метод создания курсора — объединение базового значения курсора со случайным числом, используемым *один раз* (nonce). Далее результат кодируется с помощью Base64. В примере 2.8 показаны ответы API, использующего курсорную пагинацию.

Пример 2.8. Курсорная пагинация

```
# Первый ответ REST API с курсорной пагинацией  ❶
curl https://example.com/api/v1/orders?limit=100
{
 "next": "123abc",
 "results": [...]
}

# Второй ответ REST API с курсорной пагинацией  ❷
curl https://example.com/api/v1/orders?limit=100&next=123abc
{
 "next": "456cde",
 "results": [...]
}
```

❶ В первом запросе клиент использует параметр `limit` для управления размером страницы. Ответ API содержит курсор `next`, который указывает, как получить следующую страницу результатов. Клиент использует значение `next`, чтобы продолжить извлечение данных. Пустое значение `next` означает, что результатов больше нет.

❷ Во втором запросе клиент `curl` использует `next` для перехода к следующей странице результатов. API предоставляет новый курсор, указывающий на следующую страницу.



В документации Slack API есть пример реализации курсорной пагинации, где используются другие имена полей (https://oreil.ly/QZvu0). Ответ API содержит next_cursor, а запросы клиента — cursor.

Курсорная пагинация эффективна для больших наборов данных, поскольку курсор идентифицирует конкретный элемент в наборе. Она особенно подходит для динамичных сред с часто меняющимися данными, например для новостных лент, обновляемых в режиме реального времени. В отличие от пагинации со смещением, которая может привести к пропуску или дублированию записей при удалении или вставке элементов, курсорная пагинация дает более стабильные результаты. Это происходит потому, что курсор представляет позицию относительно известного элемента и не зависит от изменений в окружающих данных.

Недостаток курсорной пагинации — сложная реализация логики для перехода на конкретную страницу или для произвольной навигации между страницами.

Длительные задачи

Через запросы к API можно запускать выполнение сложных, многоэтапных задач, таких как обработка видео, формирование финансового отчета или обучение модели. Клиент, взаимодействующий с API по модели «запрос — ответ», ожидает ответ на каждый отправленный запрос. Если запрос отправляется через REST API, то ответ должен приходить своевременно. Обычно клиенты и серверы API используют предустановленные тайм-ауты, чтобы ограничить время ожидания. Механизм тайм-аутов эффективен для задач с коротким сроком выполнения. Однако есть задачи, время выполнения которых превышает тайм-аут, например обработка видео. В таких случаях используется паттерн обработки *длительных задач* (long-running tasks или long-running operations).

Этот паттерн взаимодействия предполагает, что запрос клиента к API запускает процесс обработки задачи (обычно она выполняется не на том компьютере, который обслуживает API). Вместо результата API возвращает клиенту уникальный идентификатор, означающий, что задача принята в обработку. Клиент использует этот идентификатор для проверки статуса задачи (прогресса выполнения). Когда REST API принимает задачу, код состояния в ответе должен быть `202 (Accepted)`. Один из способов запуска задачи — определение ресурса задачи *(tasks)* на верхнем уровне API, например: https://example.com/api/v1/tasks.

При обработке длительных задач API сохраняет и отслеживает их метаданные: статус, прогресс, время начала и завершения, а также ошибки. Для одних задач результаты возвращаются в виде метаданных, для других — в виде местоположения, например пути к файлу.

Обработка задачи описывается с помощью модели конечного автомата, которая фиксирует все возможные переходы между состояниями задачи. Некоторые длительные задачи поддерживают управление состоянием: их можно приостанавливать, возобновлять или отменять. Однако есть задачи из категории «отправил и забыл» (fire-and-forget tasks) — клиент не может управлять их состоянием и не получает уведомлений об изменении статуса задачи после ее запуска. Если задача относится к недвоичным, то есть, помимо двух стандартных состояний «успех» и «неудача», имеет другие (например, «возобновленная», «приостановленная», «отмененная»), то в API потребуется реализовать

специальные действия для управления этими состояниями. Интерфейс, созданный для таких задач, может выглядеть так: https://example.com/api/v1/tasks/1?action=resume.



Действия для управления недвоичными состояниями создаются с помощью параметра запроса /resource/ID?action, а не /resource/ID/action. Проблема второго подхода заключается в том, что action может интерпретироваться не как действие, а как ресурс (действия).

Возникает закономерный вопрос: как клиент, запустивший длительную задачу, узнает о ее завершении? Клиент может использовать *регулярный опрос сервера* (polling), периодически запрашивая у API статус задачи (подробнее см. в главе 10). Недостаток метода — избыточное количество запросов, которые создает клиент.

Альтернативный способ — установить *долгоживущее соединение* с API. В этом случае соединение с сервером не прерывается до завершения задачи и отправки сообщения о завершении клиенту. Недостаток непрерывного соединения заключается в том, что API должен управлять всеми открытыми соединениями, что увеличивает потребление ресурсов на сервере.

Дедупликация запросов

Дублирование запросов происходит, когда один и тот же запрос отправляется несколько раз. Причиной могут стать быстрые щелчки на элементе пользовательского интерфейса или ошибки в логике приложения. Дублирование неидемпотентных запросов нежелательно, особенно когда речь идет о платных API: с карты клиента может дважды списаться плата за один и тот же запрос. Однако есть случаи, когда исходный запрос необходимо отправить повторно — например, при возникновении ошибки сервера, сети или валидации. Эти случаи описаны в подразделе «Повторная отправка запроса» (см. ниже).

Чтобы избежать нежелательных последствий дублирования запросов, клиент может сгенерировать уникальный ID запроса и добавить его в запрос. Повторно получив запрос с тем же идентификатором, API корректно обработает дубль — вернет предыдущий успешный ответ или ошибку.



Идентификатор запроса, генерируемый одним клиентом, должен быть уникальным. Чтобы минимизировать риск совпадения, идентификатор следует создавать в соответствии со стандартом UUID (Universally Unique Identifiers).

Повторная отправка запроса

Повторение неудачных запросов учитывается и при проектировании API. Запрос может быть неудачным из-за содержащихся в нем неверных данных, сбоя сети, ошибки на сервере или ошибки в логике приложения. Если проблема в самом запросе (например, неверные данные или форматирование), то он исправляется и отправляется заново. Если же сбой произошел на стороне API, то исходный запрос следует повторить позже, когда (если) проблема будет устранена.



Когда множество клиентов одновременно пытаются получить доступ к ресурсу, вызывая скачки нагрузки и конкуренцию за ресурсы, возникает проблема, известная как thundering herd (буквально — «проблема громоподобного стада»).

Retry storm — связанная проблема, которая возникает, когда множество клиентов многократно повторяют неудачные запросы, перегружая API и потенциально продлевая время простоя.

Пользователи отправляют запросы повторно в надежде рано или поздно получить ответ. Но если получение ответа маловероятно, то повторные попытки нужно приостановить, чтобы не продлевать сбой. Можно ограничить максимальное количество попыток и установить таймер, после срабатывания которого попытки можно возобновить. Есть также более продвинутый паттерн Circuit Breaker (https://oreil.ly/e1mZS), который приостанавливает отправку, исходя из соотношения количества успешных и неудачных запросов.

Частота повторных запросов контролируется по интервалу повтора — времени, по истечении которого клиент отправляет следующий запрос. Этот интервал можно изменить, добавив *случайную задержку* (timing jitter), чтобы распределить повторные попытки во времени. Кроме того, клиентские запросы ограничены *тайм-аутом* (request timeout) — временем ожидания ответа. Таким образом, клиент может отправить повторный запрос после получения неудачного ответа или по истечении тайм-аута.

Другой способ — динамическая настройка интервала повтора с помощью *метода экспоненциальной задержки* (exponential back-off method), при котором интервал увеличивается с каждой попыткой. Например, если начальный интервал составляет 2 секунды, то следующие попытки будут выполняться через 4, 8, 16 секунд и т. д. На рис. 2.2 показаны повторные попытки с экспоненциальной задержкой и последующим ограничением. Время между попытками удваивается. После четвертой неудачной попытки ограничение приостанавливает повторные запросы до истечения тайм-аута (на схеме не показан).

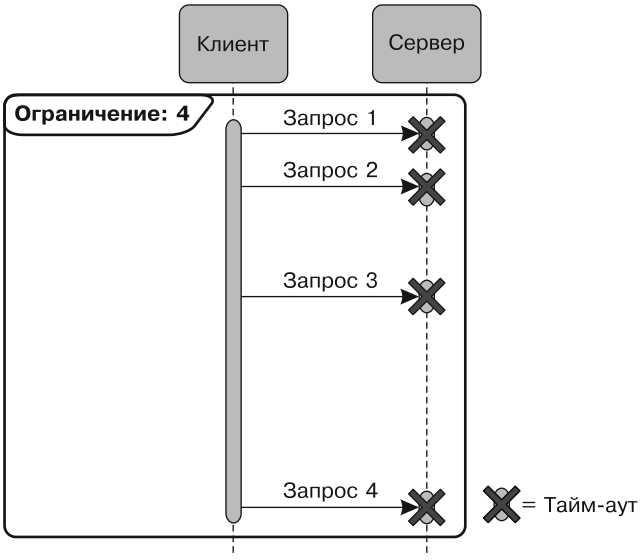


Рис. 2.2. Метод экспоненциальной задержки с последующим ограничением повторных запросов

Кроме того, сервер может участвовать в управлении повторными попытками, добавляя в ответ HTTP-заголовок `Retry-After`, указывающий, когда клиентам следует повторить запрос.

Ограничение частоты запросов

Ограничение частоты запросов, также известное как *троттлинг* (throttling), защищает API от наплыва легитимных или вредоносных запросов. При троттлинге ограничивается количество запросов, которые один клиент может выполнить к одному эндпоинту за заданный интервал времени, поэтому данный метод предполагает идентификацию клиента. Например, система может идентифицировать клиента по IP-адресу и ограничить количество запросов для этого клиента, тем самым сэкономив ресурсы на их обработке. Процедура троттлинга на сетевом пути запроса должна запускаться как можно раньше.

Троттлинг снижает риски традиционных брутфорс- и DoS-атак. Представим, что мы создали API с функцией сброса пароля. Пользователь, забывший пароль от учетной записи, указывает связанный с ней имейл и отправляет запрос в API. Через несколько секунд на почту (во входящие сообщения или в папку Спам) приходит письмо с инструкциями по сбросу пароля.

Предполагается, что человек, запускающий процедуру сброса, — владелец почты, который выполняет один или несколько запросов на сброс. Однако злоумышленник, желающий навредить API, может использовать эту функцию для DoS-атаки. В примере 2.9 показан бесконечный цикл, отправляющий HTTP-запрос `POST` к эндпоинту `forgot_password`. Один такой клиент вряд ли перегрузит сервис, но множество клиентов, действующих одновременно, приведут к DDoS-атаке. При этом каждый клиент может выполнять всего несколько запросов, поэтому одного троттлинга для прекращения атаки будет недостаточно.

Пример 2.9. Попытка DoS-атаки с использованием бесконечного цикла while

```
while true; do
  curl 'https://example.com/api/v1/forgot_password' \
    --header 'Content-Type: application/json' \
    --data '{"email": "user@example.com"}' \
    --request POST
done
```

Кэширование

Кэширование — это прием оптимизации цикла «запрос — ответ», который снижает количество повторных запросов к базе данных. Благодаря этому эффективность взаимодействия с клиентом повышается, а нагрузка на сервер снижается. Кэширование можно реализовать как на стороне клиента, так и на стороне сервера.

При *кэшировании на стороне клиента* клиент запрашивает ресурсы у сервера и сохраняет их локально. Например, браузер может кэшировать HTML, CSS, JavaScript и другие статические ресурсы, чтобы ускорить загрузку страницы при повторных посещениях. Такое кэширование снижает сетевой трафик и повышает отзывчивость клиента, поскольку загрузка ресурсов с диска происходит быстрее, чем по сети. Однако у кэширования на стороне клиента есть недостаток — оно может использовать устаревший контент и требует ручной инвалидации кэша. Чтобы поддерживать актуальность ответов и автоматически инвалидировать кэш, сервер может установить срок действия кэша (например, time-to-live, TTL) или использовать заголовки валидации `ETag` и `Last-Modified`, которые позволяют клиентам автоматически проверять актуальность кэшированного контента.

При *кэшировании на стороне сервера* сам сервер проверяет, сохранены ли запрошенные данные в кэше. Если нет, то сервер генерирует ответ, сохраняет его в хранилище кэша, а затем отправляет клиенту. Обычно (в целях повышения производительности) ответ сохраняется в локальной памяти сервера, но можно

использовать и распределенный кэш. Такой подход дает больше контроля над кэшированными ресурсами, поскольку не зависит от поведения отдельных клиентов. Хотя серверное кэширование менее эффективно для снижения сетевого трафика, оно уменьшает нагрузку на бэкенд-системы и уменьшает общее время отклика.

Удаление

Удаление ресурсов мы обсудим в контексте архитектуры REST.

В REST API есть два способа удаления ресурсов: *мягкое* (soft delete) и *жесткое* (hard delete). При мягком ресурс сохраняется, но помечается как удаленный, а при жестком полностью удаляется из системы.



Мягкое удаление не полностью соответствует требованиям по стиранию данных, которые содержатся в регламентах по защите данных, таких как GDPR и CCPA.

Для мягкого удаления требуется пометить ресурс — например, использовать флаг `soft_delete` вместе с меткой времени изменения статуса, чтобы указать, удален ли ресурс и когда. Можно также использовать несколько состояний для описания статуса удаления ресурса: `active`, `pending_deletion` или `soft_deleted`. Сложность использования нескольких состояний в том, что для отслеживания статуса удаления ресурса нужно хранить историю изменений.

При мягком удалении в REST API представление ресурса возвращается с обновленным состоянием и кодом `200 (OK)`. Последующий запрос на мягкое удаление уже удаленного ресурса должен возвращать `412 (Precondition Failed)`[1].

Одновременная реализация жесткого и мягкого удаления требует добавления действия `erase` (или `wipe`, `prune`) к ресурсу API — например, `HTTP DELETE /api/orders/123?action=erase`. При жестком удалении система должна возвращать либо код состояния `200 (OK)` вместе с удаленным ресурсом, либо код `204 (No Content)` без содержимого. Последующий запрос на жесткое удаление уже удаленного ресурса должен возвращать код `404 (Not Found)`.



Обязательно документируйте любое нестандартное поведение HTTP-методов.

---

[1] Подробнее о мягком удалении читайте в главе 25 книги «Паттерны проектирования API» Дж. Дж. Гивакса.

В API с версионированием мягкое удаление может стать критическим изменением. Перед внедрением необходимо оценить, как оно повлияет на данные, возвращаемые API, и не нарушит ли оно пользовательские ожидания. Если мягкое удаление приводит к неожиданному поведению имеющихся клиентов (например, к пропаже данных, которые раньше отображались), то добавляйте функциональность мягкого удаления при выпуске новой версии API.

От типа удаления зависит и способ возврата ресурсов. Например, при запросе ресурса после мягкого удаления метод должен возвращать сам ресурс, а не код состояния HTTP `404 (Not Found)`. Кроме того, при запросе нескольких ресурсов в ответе должны быть только те записи, у которых флаг `soft_delete` установлен в `false`. Интерфейс API должен содержать флаг `include_deleted`, позволяющий клиенту видеть (получать) ресурсы, удаленные мягко. Без этого флага API отфильтрует такие ресурсы при формировании ответа.

Кроме того, API должен поддерживать действие восстановления мягко удаленных ресурсов (`undelete`, `restore`). Для этого можно реализовать пользовательское действие, например: https://example.com/api/v1/order/1?restore. При попытке восстановить уже восстановленный объект API должен возвращать код состояния HTTP `412 (Precondition Failed)`. Кроме того, к ресурсу можно добавить поле `expiration_date`, чтобы периодически очищать базу данных от мягко удаленных ресурсов.

Безопасность API

Сегодня организации все чаще переходят на модель SaaS (software-as-a-service, программное обеспечение как услуга), что позволяет им мигрировать на готовые облачные решения. По этой причине злоумышленники сосредотачиваются не только на технических уязвимостях (например, на неправильной конфигурации решения), но и на стратегиях манипуляции человеческим поведением.

В результате возникло понятие *«социальная инженерия»* (social engineering), также известное как *«атака на человека»* (human hacking). Оно обозначает совокупность методов психологического манипулирования для получения несанкционированного доступа к ИТ-системам. Вот наиболее распространенные методы.

Фишинг

Злоумышленник обманом вынуждает жертву предоставить конфиденциальную информацию.

Претекстинг

Злоумышленник создает ложный сценарий и представляется уполномоченным лицом для получения конфиденциальных данных.

Байтинг

Злоумышленник оставляет вредоносный объект (например, зараженную флешку) с расчетом на то, что жертва им воспользуется.

Проект OWASP API Security Top 10

Open Web Application Security Project (OWASP) (https://owasp.org) — это онлайн-сообщество, цель которого — повысить безопасность программного обеспечения. Материалы, созданные сообществом: статьи, документация, инструменты, методы и технологии — находятся в свободном доступе.

OWASP реализует несколько проектов (https://oreil.ly/qape2), в том числе проект Top 10 (https://oreil.ly/XOHc6), ставший эталонным стандартом для оценки рисков безопасности веб-приложений.

Кроме того, OWASP реализует проект API Security Top 10 (https://oreil.ly/rU9sQ), в котором подробно рассматриваются риски безопасности API и методы их устранения. Риски, упомянутые в проекте, делятся на три категории.

Аутентификация и авторизация

Некорректная реализация аутентификации или авторизации (определение этих терминов см. в подразделе «Шифрование, аутентификация и авторизация» далее) позволяет злоумышленникам выдавать себя за других пользователей или получать несанкционированный доступ к защищенным данным. Атаки этой категории возникают из-за скомпрометированных токенов аутентификации, ошибок в реализации потоков аутентификации и ненадежных паролей. К этой же категории относятся брутфорс-атаки.

При атаках на авторизацию злоумышленники используют некорректные политики контроля доступа или отсутствие разделения обязанностей между ролями администраторов и обычных пользователей. Пример такой атаки — *повышение привилегий* (privilege escalation), когда злоумышленник использует уязвимости в аутентификации или авторизации для получения привилегированного доступа.

Управление системой

Причиной атак могут стать некорректная конфигурация или небезопасные практики программирования API и вспомогательных сервисов. К некорректной конфигурации относятся неправильная настройка межсетевого

экрана, отсутствие обновлений безопасности[1], утечка секретов в журналах API, отсутствие протокола TLS или использование слабых алгоритмов шифрования. Примеры атак в этой категории:

- *атака посредника* (man-in-the-middle, MITM) — злоумышленник перехватывает данные, которыми обмениваются клиент и сервер;
- *инъекция кода* — злоумышленник компрометирует систему, внедряя вредоносный фрагмент кода, например, через SQL-инъекцию (https://oreil.ly/a0hwv);
- *инъекция команд операционной системы* (https://oreil.ly/pUxMy), или межсайтовый скриптинг (https://oreil.ly/TAc44).

Управление ресурсами

Если использование API не контролируется, то возможность совершать неограниченное количество вызовов часто приводит к незапланированным расходам. Если API интегрируется с внешними системами, предоставляющими такие сервисы, как CMC, электронная почта или телефония, и каждый запрос к сервису тарифицируется, то злоумышленники могут повысить операционные расходы на использование сервиса. Пример атаки из этой категории — DoS-атака, при которой злоумышленник перегружает API потоком запросов.

Примеры паттернов безопасности API

При проектировании API необходимо учитывать риски безопасности. В проекте OWASP они классифицируются и изучаются в исторической перспективе. В этом разделе описаны некоторые паттерны для снижения таких рисков.

Шифрование, аутентификация и авторизация

Шифрование — наиболее распространенный способ защиты трафика API, обеспечивающий конфиденциальность, целостность и подлинность данных при передаче. Шифрование преобразует данные из исходного формата (открытого текста) в зашифрованный текст с помощью алгоритма шифрования

[1] В настоящее время облачные провайдеры и SaaS-компании не обязаны раскрывать уязвимости своих сервисов, даже если для их устранения требуются действия пользователей. См. выступление Шира Тамари (Shir Tamari) и Алона Шинделя (Alon Schindel) Security Industry Call-to-Action: We Need a Cloud Vulnerability Database («Обращение к индустрии безопасности: нам нужна база данных уязвимостей облачных сервисов») (https://oreil.ly/L3jVB), а также базу данных Open Cloud Vulnerability and Security Issue Database (https://www.cloudvulndb.org).

и ключа шифрования. Без ключа дешифровки прочитать зашифрованный текст невозможно.



Подробнее о шифровании см. файл src/network/README.md в репозитории кода книги.

Для шифрования трафика API обычно используют протокол TLS, который также позволяет клиенту аутентифицировать сервер через цифровой сертификат, выданный доверенным центром сертификации. Возможна двусторонняя аутентификация через TLS (mTLS, mutual TLS), когда клиент и сервер аутентифицируют друг друга. Наконец, TLS обеспечивает целостность данных, подтверждая, что они не были изменены при передаче. Для этого используют коды аутентификации сообщений (message authentication codes, MACs), обнаруживающие подмену данных.

Аутентификация — это проверка идентичности сущности. *Авторизация* гарантирует, что сущность получит доступ к запрашиваемым ресурсам. При возврате данных клиентам API рекомендуется применять *принцип наименьших привилегий*, согласно которому пользователям предоставляются минимальные права, необходимые для решения их задач.



Подробнее шифрование, аутентификацию и авторизацию мы рассмотрим в следующих главах, когда будем реализовывать разные стили API. См., например, раздел «Безопасность» в главе 5.

Очистка и валидация данных

Очистка и валидация защищают API от атак, которые связаны с эксплуатацией входных данных или с манипуляцией выходными данными, например от таких, как инъекционные атаки или утечки данных.

Очистка — это первый этап, позволяющий привести данные в формат, безопасный для обработки. На данном этапе удаляются нежелательные входные данные, например подозрительные символы, или обрезаются пробелы в начале и в конце входной строки. Далее, на этапе *валидации*, входные данные проверяются на предмет соответствия ограничениям и политикам валидации. Валидированные данные должны находиться в правильном диапазоне, а строки — иметь корректный формат (например, правильный числовой формат для почтового индекса).

Очистка и валидация — обязательные процедуры. Разработчик, пишущий безопасный код, исходит из презумпции недоверия входным данным. В противном

случае злоумышленник сможет внедрить вредоносный код в поле запроса, что приведет к утечке данных или получению несанкционированного доступа к ним. Кроме того, при валидации входных данных нужно учитывать не только их значения, но и типы.

При работе с входными данными рекомендуется использовать сериализаторы — стандартный компонент многих фреймворков и языков программирования. Сериализаторы часто содержат функции валидации данных. Вдобавок большинство фреймворков содержит встроенные валидаторы, которые поддерживают валидацию на уровне модели (object-relational mappers, ORMs).

Выходные данные API тоже требуют проверки, поскольку небезопасные API часто возвращают избыточные данные, среди которых могут быть и конфиденциальные. Например, при разработке API для сервиса аренды автомобилей может потребоваться отслеживать местоположение транспортных средств. Для реализации этой функции нужно добавить поле `location` в виде массива с историей местоположений. Это поле не должно быть доступно клиентам API, у которых нет соответствующих прав. При возврате данных клиентам следует руководствоваться принципом минимальных привилегий (https://oreil.ly/SuSn3), который позволяет контролировать доступ к данным и предотвращает раскрытие конфиденциальной информации.

Для валидации входных и выходных данных можно осуществить *пробный запуск* (dry run). Его цель — проверить валидность запроса, а не выполнить реальную операцию, изменяющую состояние системы. Один из способов реализовать пробный запуск — добавить флаг `validate_only` в интерфейс запроса. При тестировании запросов с `validate_only` процесс будет идемпотентным и безопасным[1].

Важно защитить и сами данные. Если злоумышленник стремится извлечь данные из системы, а не изменить ее поведение, то может использовать технику *скрейпинга данных* (data scraping или excessive data gathering). При скрейпинге из API или с веб-страниц извлекается большой объем данных, которые могут использоваться для различных целей, в том числе вредоносных. В следующем подразделе мы рассмотрим способы предотвращения скрейпинга.

---

[1] См. врезку «Идемпотентность, безопасность и кэширование HTTP-запросов» на с. 199. Обычно POST-запросы неидемпотентны и небезопасны — исключение составляют те, которые помечены флагом validate_only. При наличии этого флага эндпоинт обрабатывает запрос как идемпотентный и безопасный, позволяя клиентам проверять корректность операции без ее фактического выполнения.

Защита от скрейпинга данных

Скрейпинг позволяет программе извлекать данные из другой программы. Его наиболее распространенная форма — веб-скрейпинг, когда данные извлекаются с сайтов. Возможно также извлечение данных из API, десктопных приложений или баз данных. Компании используют скрейперы для сбора данных других компаний с целью дальнейшей обработки и анализа. Скрейпинг сайтов конкурентов может стать источником несправедливого конкурентного преимущества, что приводит к юридическим и этическим противоречиям.

Для защиты сайта от популярных «безобидных» веб-краулеров, например поисковых роботов, разместите файл `robots.txt` в корне сервера API (https://example.com/robots.txt). Однако важно учитывать, что `robots.txt` — добровольный стандарт и не все боты его соблюдают. Этот метод не защитит данные организации от вредоносных веб-краулеров[1].

Еще один паттерн для защиты публичных эндпоинтов API — избегать предсказуемых последовательностей в идентификаторах пользовательских ресурсов. Если в качестве идентификаторов используются целые числа, например `1` в `https://example.com/api/v1/books/1`, то парсеры смогут легко извлекать ресурсы, последовательно увеличивая идентификатор. Если вместо целых чисел использовать идентификаторы UUIDv4 вида `9f287bb6-7a74-44dc-b95f-2046116c5a78`, то их подбор займет у парсеров много времени из-за огромного диапазона возможных значений. UUIDv4 — это 128-битный идентификатор, в котором 6 бит предопределены, а 122 бита генерируются случайно. В итоге возможный диапазон включает примерно $5{,}32 \times 10^{36}$ (5,32 ундециллиона) уникальных идентификаторов.

Лучшие практики проектирования API

Статья Джошуа Дж. Блоха (Joshua J. Bloch) *How to Design a Good API and Why It Matters* («Как спроектировать хороший API и почему это может быть важно») содержит набор максим о проектировании API[2]. Эти максимы сформулированы с точки зрения программной библиотеки, но применимы к широкому

---

[1] Вредоносных ботов, игнорирующих robots.txt, можно заманить в ловушки (honeypots). См. статью Рейда Таториса (Reid Tatoris) Trapping Misbehaving Bots in an AI Labyrinth («Заманивание недобросовестных ботов в AI-лабиринт») в блоге Cloudflare (https://oreil.ly/eXGvV).

[2] *Bloch J.* How to Design a Good API and Why It Matters («Как спроектировать хороший API и почему это может быть важно») // Companion to the 21st ACM SIGPLAN Symposium on Object-Oriented Programming Systems, Languages and Applications, 2006. — P. 506–507. https://oreil.ly/RZxP3.

спектру API, в том числе к сетевым. Прежде чем дать описание набора лучших практик для проектирования сетевых API, мы приведем избранные максимы. Изучить все максимы вы можете самостоятельно в полной версии статьи.

Начнем с ответа на вопрос, почему хорошее проектирование API важно:

> API могут стать вашим главным активом или обязательством. Хорошие API создают постоянных клиентов; плохие — постоянные проблемы для поддержки.
>
> Публичные API, подобно бриллиантам, вечны. У вас есть один шанс сделать все правильно, так что приложите максимум усилий.
>
> Все программисты — проектировщики API. Хорошие программы — модульные, а API определяются межмодульными границами. Хорошие модули используются снова и снова.
>
> *Джошуа Дж. Блох, 2006*

Набор максим Блоха завершается такой максимой:

> Проектирование API — это искусство, а не наука. Стремитесь к красоте и доверяйте своей интуиции. Не следуйте слепо вышеуказанным эвристикам, но нарушайте их лишь изредка и только при достаточном основании.
>
> *Джошуа Дж. Блох, 2006*

Вот лучшие практики проектирования сетевых API.

Учет потребностей аудитории

API проектируется под потребности клиента. Клиентом может быть браузер, мобильное устройство, смарт-часы, IoT-устройство, ИИ-агент и т. д. Некоторые стили API обладают большей гибкостью для адаптации к нуждам клиента. Например, в GraphQL клиент может выбирать формат выходных данных.

Интуитивно понятный интерфейс

Интуитивно понятным API можно пользоваться даже без документации. Пример — REST API с HATEOAS (Hypermedia as the Engine of Application State), где ответ содержит ссылки на связанные ресурсы и допустимые действия.

Удобство сопровождения

API развивается без нарушения работы клиентов. Разработчики вносят обратно совместимые изменения, а в случае критических изменений выпускается новая версия.

Своевременное документирование

API поставляется с документацией. Наличие документации влияет на удобство использования и на внедрение API, а ее отсутствие может затруднить применение.

Предсказуемость

API следует принципу наименьшего удивления (principle of least surprise): простые задачи должны решаться интуитивно, сложные — относительно просто, а вероятность ошибок должна быть минимальной. Этот принцип помогает создать API, использование которого будет предсказуемым для разработчиков благодаря стандартным практикам проектирования, общепринятым сообщениям об ошибках и методам валидации данных.

Тестирование

Разработчики или пользователи должны иметь возможность тестировать API, который они создают или используют. Если тестирование API невозможно из-за его конфигурации, то проводится mock-тестирование.

Безопасность

API служит шлюзом к системе. Входные данные необходимо валидировать, чтобы злоумышленники не могли использовать уязвимости API. В выходных данных не должна раскрываться конфиденциальная информация. В библиотеках и внешних сервисах, которые использует API, должны быть установлены последние обновления безопасности. Безопасный API обрабатывает дублирующиеся запросы и эффективно управляет количеством входящих запросов.

Эффективность

API стремится минимизировать использование системных ресурсов. Предоставляя клиентам возможность настраивать ответы, API сокращает объем передаваемых данных. Для оптимизации используемых ресурсов применяются специальные методы, например фильтрация полей (sparse fieldsets в REST API), или создаются запросно-ориентированные API, такие как GraphQL.

Сообщения об ошибках

API сообщает о текущем состоянии через стандартные коды ошибок, соответствующие стилю API. Например, в REST API используются HTTP-коды состояний, а в gRPC API — gRPC-коды. Помимо стандартных кодов, API предоставляет сообщения на естественном языке об ошибках, содержащие пояснение возникшей проблемы. Сочетание стандартных кодов и сообщений помогает разработчикам понять и устранить операционные проблемы.

РЕКОМЕНДАЦИИ ПО ПОВЫШЕНИЮ ПРОИЗВОДИТЕЛЬНОСТИ API[1]

1. *Сжатие*. Подходящий алгоритм сжатия данных, например `gzip` или `Brotli`, уменьшает размер ответов API и ускоряет передачу данных по сети. Уменьшение объема полезной нагрузки повышает производительность, особенно при ограниченной пропускной способности канала. Наконец, проектировщикам API необходимо балансировать между вычислительными затратами и затратами на пропускную способность, поскольку сжатие увеличивает нагрузку на CPU.
2. *Кодирование*. Формат кодирования влияет на производительность API. Двоичные форматы, например Protocol Buffers (https://protobuf.dev) и Avro (https://avro.apache.org), обычно превосходят текстовые, такие как JSON (https://www.json.org) и XML (https://www.w3.org/XML), за счет меньшего размера данных при большей скорости обработки. Отдавать предпочтение текстовым форматам следует на этапе отладки или для улучшения удобочитаемости.
3. *Кэширование*. Механизмы кэширования оптимизируют использование ресурсов API, например базы данных или хранилища. Кэширование можно выполнять на стороне клиента или сервера. Побочный эффект кэширования — увеличение потребления памяти. Кроме того, при отсутствии синхронизации с источником данных возникает риск устаревания или утери кэшированных данных. Наконец, кэширование повышает затраты на разработку и эксплуатацию API.
4. *Аппаратное обеспечение*. Производительность системы можно повысить путем вертикального масштабирования — замены или обновления аппаратного экземпляра на более мощный с увеличенными ресурсами CPU, памяти и хранилища. Однако такой подход может скрыть исходную причину снижения производительности. Кроме того, вертикальное масштабирование имеет свои ограничения. Когда пределы достигнуты, возникает необходимость в горизонтальном масштабировании — добавлении дополнительного оборудования.
5. *Управление частотой запросов*. Ограничение количества запросов к API от одного клиента в заданный интервал времени снижает риск злоупотреблений и защищает ресурсы бэкенда.

[1] Дополнительные рекомендации по повышению производительности приведены в разделе Evergreen Performance Best Practices в книге High Performance Browser Networking (https://oreil.ly/gUTTa) Ильи Григорика (Ilya Grigorik) (O'Reilly, 2013).

Резюме

В этой главе мы обсудили паттерны проектирования сетевых API, которые чаще всего встречаются в REST API, но применимы и к другим стилям. Мы рассмотрели такие паттерны, как именование компонентов API, версионирование, фильтрация и пагинация, обработка длительных задач, дедупликация и повторная отправка запросов, троттлинг, кэширование и удаление ресурсов API. Затем мы поговорили о паттернах безопасности API, общих для всех стилей: о шифровании, аутентификации, авторизации и валидации данных.

В заключение мы рассмотрели общие рекомендации по проектированию сетевых API и советы по повышению производительности. Однако главная идея главы заключается в том, что все многообразие паттернов — это не набор жестких правил, обязательных к соблюдению, а лишь возможные решения, и последнее слово всегда остается за разработчиком.

В главах 3 и 4 мы обсудим низкоуровневые функции протоколов TCP и HTTP, лежащих в основе многих API. Далее, в главе 5, мы применим некоторые паттерны проектирования на практике при создании примера REST API.

ГЛАВА 3

Сеть

В главе 1 мы рассмотрели понятия, связанные с обменом данными по компьютерным сетям, в частности режимы передачи и синхронизацию. В этой главе поговорим о сетевых протоколах. Мы обсудим цели и способы использования сетевых протоколов и их программных API. Далее применим теорию на практике и реализуем сервер и клиент TCP ECHO. Кроме того, разберем такие инструменты, как `netcat`, `scapy`, `openssl`, `tcpdump` и `tshark`, и используем их в процессе изучения протоколов TCP и TLS. Эта глава поможет вам понять, как разные API, описанные в книге, взаимодействуют по сети.

Сетевые протоколы

Задумывались ли вы, что происходит на сетевом уровне, когда вы открываете сайт в браузере? Цель этой и следующей главы — раскрыть некоторые детали организации сетевого взаимодействия.

Сеть обеспечивает связь, организуя обмен данными. Современные компьютеры работают с двоичными данными, поэтому задача сети сводится к тому, чтобы передать последовательность нулей и единиц от одного участника коммуникации к другому.



Канал связи для обмена двоичными данными не обязательно должен быть двоичным. Вспомним, например, первоапрельское предложение организации Internet Engineering Task Force об использовании почтовых голубей для передачи IP-пакетов (https://oreil.ly/pw0G3). Если следовать этой логике, то блок двоичных данных можно передать и с помощью голубей.

В качестве примера рассмотрим медный кабель, используемый в компьютерных сетях. Как определить, какие данные передаются между отправителем

и получателем? Во-первых, как закодировать двоичные данные на физическом уровне? Один из вариантов — использовать низкое и высокое напряжение для представления 0 и 1 соответственно. Во-вторых, как определить, где начинается сообщение? Чтобы решить эту проблему, определяется последовательность нулей и единиц, указывающая на начало сообщения, чтобы отправитель и получатель могли синхронизироваться.

Правила обмена данными в определенной степени произвольны и тесно связаны с физическими ограничениями среды передачи. Практическая задача сетевого взаимодействия — обеспечить поток данных с соблюдением этих правил. Формализация и кодификация правил гарантирует, что обе стороны общения понимают друг друга, — в этом и заключается назначение сетевого протокола.



Сетевой протокол определяет набор правил для обмена данными между сетевыми устройствами.

Базовые сетевые протоколы уже реализованы в операционной системе, что снижает нагрузку на пользователей сети, которым иначе пришлось бы с нуля программировать эти протоколы в своих приложениях. Кроме того, операционная система предоставляет стандартизированный набор функций (системных вызовов)[1].

Как правило, сетевые приложения взаимодействуют с операционной системой, а операционная система — с оборудованием[2].

Socket API

Пример стека сетевых протоколов, реализованных в операционной системе, — TCP/IP. TCP расшифровывается как Transmission Control Protocol (протокол управления передачей), а IP — как Internet Protocol (межсетевой протокол). TCP/IP называют стеком протоколов, поскольку, помимо двух основных

---

[1] Стандартизация есть не только в API (высокоуровневом исходном коде), но и в ABI (application binary interface), где программы взаимодействуют на низшем уровне двоичного машинного кода.

[2] Есть исключения, например обход сетевого стека ядра, когда приложения напрямую взаимодействуют с сетевой картой. Пример — технология удаленного прямого доступа к памяти (Remote Direct Memory Access, RDMA), которую применяют в высокопроизводительных приложениях, где два удаленных компьютера получают доступ к памяти друг друга, сокращая задержки и нагрузку на CPU.

протоколов, отраженных в названии, он содержит и другие. TCP/IP были впервые внедрены в начале 1983 года в системах Управления перспективных исследовательских проектов США (Advanced Research Projects Agency Network, ARPANET). ARPANET стала первой глобальной компьютерной сетью с коммутацией пакетов. Она соединяла США, Великобританию и Норвегию.

Обязательным требованием интернет-стандарта было наличие нескольких совместимых реализаций сетевого протокола[1]. В 1980-е годы существовало несколько версий и реализаций TCP/IP. Работы по внедрению TCP/IP в операционную систему Berkeley Unix велись при финансировании Управления перспективных исследовательских проектов Министерства обороны США (Defense Advanced Research Projects Agency, DARPA). Эта реализация была официально выпущена в середине 1983 года, и ее интерфейс сегодня известен как Socket API (https://oreil.ly/E31mf).

Набор системных вызовов *socket API*[2] — основной механизм взаимодействия приложений со стеком сетевых протоколов операционной системы. Приложение вызывает socket API, который позволяет открыть сетевое соединение, отправить и получить сообщение и закрыть соединение.



Клиент — это сторона, инициирующая соединение, а сервер — сторона, ожидающая запроса на соединение от клиента.

Рассмотрим стандартный пример использования TCP socket API[3]. На рис. 3.1 показан клиент, отправляющий единичный запрос серверу, получающий ответ и затем закрывающий соединение. Сплошные стрелки обозначают порядок системных вызовов socket API. Обратите внимание на то, что вызовы `send` и `recv` образуют цикл. Это значит, что клиент и сервер могут обмениваться несколькими сообщениями. Кроме того, после вызова `close` сервер может

---

[1] «Интернет-стандарт... требует нескольких совместимых реализаций» (Скотт Брандер (Scott Brander), IETF Newcomers Presentation: IETF Documents, https://oreil.ly/6tBJT).

[2] То, что принято называть системными вызовами, на самом деле является функциями-обертками в пользовательском пространстве, активирующими соответствующие системные вызовы ядра. Для удобства мы будем придерживаться термина «системные вызовы».

[3] См.: *Comer D. E., Stevens D. L.* Internetworking with TCP/IP, Vol. III: Client-Server Programming and Applications, Linux/Posix Sockets Version. — Prentice Hall, 2000. Эта книга — введение в сетевую разработку под Linux на языке C.

выполнять вызов `accept` и ожидать новых подключений клиентов. Вертикальные пунктирные линии на схеме связывают системные вызовы с сообщениями, которыми обмениваются клиент и сервер.

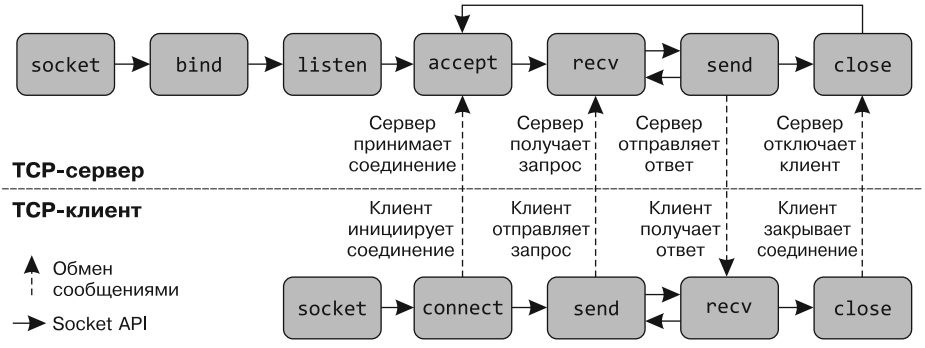


Рис. 3.1. Последовательность системных вызовов TCP socket API для клиента и сервера

TCP/IP и модель OSI

Сегодня TCP/IP обрабатывает большую часть интернет-трафика. Часто о TCP/IP говорят в контексте модели OSI (Open Systems Interconnection) (рис. 3.2). Эта модель была опубликована в 1984 году в виде стандарта ISO/IEC 7498 и используется для объяснения основ сетевого взаимодействия. В основе модели лежит уровневая категоризация сетевых устройств. Так, сетевой коммутатор работает на уровне 2 *(канальном)*, а маршрутизатор — на уровне 3 *(сетевом)*. В настоящее время модель OSI имеет преимущественно теоретическую ценность[1].



Для единиц данных, соответствующих каждому уровню, используются специальные названия: «фрейм Ethernet», «IP-пакет», «TCP-сегмент», «UDP-датаграмма», а также сообщение или данные для вышестоящих уровней или общих протоколов. Эти названия показывают, о каком уровне идет речь.

[1] См. статью Эндрю Л. Расселла (Andrew L. Russell) OSI: The Internet That Wasn't («OSI: интернет, которого нет») (https://oreil.ly/S4h32) и выступление Винта Серфа (Vint Cerf) Keynote: The Evolution of the Interplanetary Internet («Программная речь: Эволюция межпланетного интернета») (https://oreil.ly/GnJTh), в котором говорится: «OSI не существует, OSI — это фантазия».

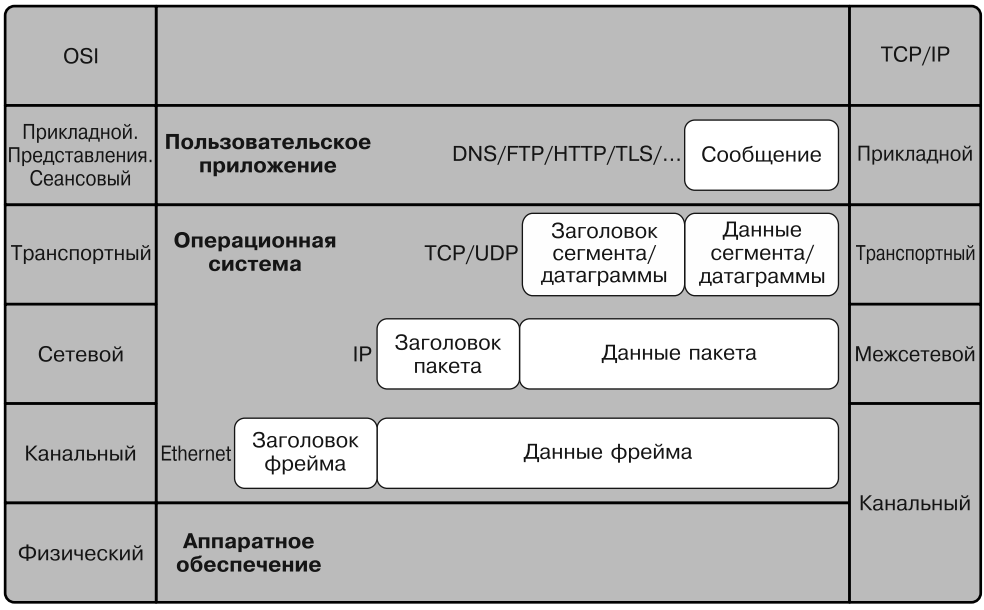


Рис. 3.2. Обзор моделей OSI и TCP/IP (RFC 1122)

Ниже мы описали уровни модели OSI, представленные на рис. 3.2. Список добавлен для общего ознакомления, поскольку многие нюансы становятся понятны только из практической работы с протоколами.

1. *Физический уровень* (physical) — передает сигналы по физическим средам и определяет связанные со средой спецификации, например типы разъемов, рабочий диапазон напряжения или радиочастоты.
2. *Канальный уровень* (data link) — обеспечивает адресацию устройств в локальной сети и обнаруживает ошибки.
3. *Сетевой уровень* (network) — назначает устройствам настраиваемые адреса и выполняет маршрутизацию данных между разными сетями.
4. *Транспортный уровень* (transport) — предоставляет дополнительные механизмы передачи, позволяющие сохранять упорядоченность данных, повторно отправлять данные при ошибках и реагировать на перегрузку сети.
5. *Сеансовый* (session), *представительский* (presentation) и *прикладной* (application) *уровни* — это три отдельных уровня в модели OSI, объединенные в один прикладной уровень в модели TCP/IP. Они поддерживают функциональность приложений, например шифрование (на *представительском* уровне), или протоколы обмена веб-данными, такие как HTTP (на *прикладном* уровне).

Многоуровневый подход к сетевым протоколам работает по принципу матрешки (https://oreil.ly/pLQlA) — каждый предшествующий уровень является частью последующего, вышестоящего.

Для примера возьмем данные IP-пакета. Они содержат заголовок протокола более высокого уровня (например, заголовок TCP-сегмента) и данные этого протокола. Иными словами, заголовок и данные протокола вышестоящего уровня (в данном случае TCP) встроены в данные протокола нижнего уровня (IP). Кроме того, заголовок нижнего уровня (IP) содержит поле, идентифицирующее протокол верхнего уровня. Это позволяет операционной системе (или приложению, если речь идет о сетевом стеке в пользовательском пространстве) вызывать соответствующие процедуры для обработки следующего, более высокого уровня.



В книге мы не рассматриваем детали спецификаций протоколов, например структуру заголовков. Современные инструменты, такие как tshark, позволяют определять и отображать уровни, не требуя от пользователя чтения двоичных данных. Если вас интересуют указанные детали, обратитесь к тематической литературе[1].

Вернемся к аналогии с матрешкой. Физический сигнал, поступающий на сетевую карту, — это самая большая кукла. Подобно тому как разбирают матрешку, операционная система или ПО декапсулирует данные, удаляя заголовок, и передает оставшиеся данные для обработки на следующий (вышестоящий) уровень протокола. Сообщение (данные) верхнего прикладного уровня — это самая маленькая кукла внутри матрешки. Если приложение отправляет сообщение, то процесс идет в обратном направлении. Данные инкапсулируются (добавляется заголовок) и передаются на следующий, более низкий, уровень протокола. Размер данных при этом увеличивается. Процесс продолжается, пока не будет достигнут физический уровень (самая большая кукла), где сигнал отправляется через сетевую карту.

Несмотря на привлекательность логической структуры, строгое разделение на уровни в модели OSI имеет слабые стороны[2]. Согласно статье *Layering Considered Harmful* (https://oreil.ly/17VY3), такой подход исключает возможность

---

[1] См.: *Comer D. E.* Internetworking with TCP/IP, Vol. I. — Pearson, 2013. Эта книга — введение в сетевые протоколы с точки зрения системного/сетевого администратора. Кроме того, в ней обсуждаются спецификации заголовков.

[2] См. замечание из доклада Дэвида Кларка (David Clark) The Network and the OS («Сеть и ОС») (https://oreil.ly/QcMNu): «Многоуровневая структура модели не предполагала разделение процесса реализации на модули. Скорее, она предполагала модуляризацию проектных комитетов».

межуровневых оптимизаций. Поэтому уровневое разделение не всегда соблюдается строго, и некоторые протоколы объединяют функциональность, ассоциируемую с разными уровнями OSI. Например, в IPsec (Internet Protocol Security) (https://oreil.ly/ey730) шифрование, относящееся к уровню представления, добавляется к протоколу IP, связанному с сетевым уровнем. Аналогично QUIC (https://oreil.ly/9FLk3) объединяет шифрование с транспортным уровнем UDP (User Datagram Protocol).

Прежде чем мы перейдем к применению socket API на практике, будет полезно освежить основы TCP/IP. Если вы предпочитаете обучающие видео, то изучите рекомендованные ресурсы[1]. Мы же воздержимся от низкоуровневого анализа битов, составляющих сетевые пакеты (такой анализ можно встретить в книгах по сетевому администрированию), и проанализируем высокоуровневую работу TCP/IP, проведя аналогию с офисом[2].

TCP/IP: ОФИСНАЯ АНАЛОГИЯ

Представим, что клиент работает в компании, где на каждом этаже находятся офисы с открытой планировкой. Клиент — часть «локальной сети» коллег на своем этаже, и на *сетевом* уровне его идентифицируют по IP-адресу `1.1.1.1`. Рабочий стол клиента идентифицируют по адресу clientMAC, который на *канальном* уровне соответствует MAC-адресу (Media Access Control address) Ethernet.

Теперь представим, что клиент хочет связаться с коллегой (сервером), IP-адрес которого — `2.2.2.2`. Коллега работает на другом этаже, то есть находится в удаленной сети. Этаж коллеги можно найти через корпоративный онлайн-справочник. Поиск пути к удаленной сети называют *маршрутизацией* (routing). В нашем офисе есть лифт — маршрутизатор с адресом routerMAC на *канальном* уровне. Он и доставит клиента на нужный этаж. Если коллега работает на том же этаже, то клиент, знающий, кто где сидит, пойдет прямо к столу коллеги.

Добравшись до нужного этажа, клиент громко спрашивает: «У кого адрес `2:2:2:2`?» — прерывая работу коллег и пытаясь найти нужный стол. Этот

[1] См. серию видео Бена Итера (Ben Eater) Networking Tutorial («Руководство по сетям») (https://oreil.ly/Z1NUV). В ней за два часа объясняются ключевые понятия сетевого взаимодействия.

[2] См. альтернативный пример о письме, отправленном авиапочтой, на с. 114–115 в книге TCP/IP Guide Чарльза Козьерока (Charles Kozierok) (No Starch Press, 2005).

процесс называют *широковещанием* (broadcasting). В контексте сетевой коммуникации широковещание используется протоколом ARP (Address Resolution Protocol) Ethernet на *канальном* уровне. Чтобы аналогия была приближена к реальной работе сетевых протоколов, вопрос должен транслировать лифт-маршрутизатор с адресом routerMAC, а отвечать — стол коллеги-сервера с адресом serverMAC (увы, офисы пока не настолько автоматизированы).

Широковещание может показаться навязчивым, но именно для такого общения и проектируют открытые пространства. Как только коллега ответит, клиент пожмет ему руку, представится и обозначит цель встречи — это и есть *трехэтапное рукопожатие* (three-way handshake) протокола TCP на *транспортном* уровне.

Клиент говорит: «Привет» — и сразу же возвращается в свой офис. Теперь коллега выполняет те же шаги, только в обратном порядке, чтобы добраться до офиса клиента и ответить ему: «Привет». После этого клиент завершает разговор. Вы, наверное, подумали: «Какая странная встреча!» Это был пример работы *службы TCP ECHO* с маршрутизацией между двумя сетями. Далее в этой главе мы проанализируем и реализуем аналогичный паттерн.

В связи с приведенным примером можно также спросить, в каком протоколе вместо IP-адресов используются имена клиента и сервера коллеги. Это протокол DNS (Domain Name System), где имя сопоставляется с IP-адресом. Необходимость такого протокола связана с тем, что людям проще запоминать имена, чем последовательности цифр.

Реализация службы TCP ECHO

В этом разделе мы реализуем службу TCP ECHO. Для сервера мы будем использовать модуль Python Socket API (https://oreil.ly/YbOzp), а для клиента — три подхода: готовую командную строку (не требующую написания кода), низкоуровневую реализацию на Python и высокоуровневую реализацию с помощью Python Socket API. Три разных варианта клиента позволят последовательно сосредоточиться на использовании службы TCP ECHO, на самом протоколе TCP и на возможностях socket API.



Служба TCP ECHO включает сервер, который ожидает входящего соединения от клиента, читает данные из соединения и записывает их обратно в соединение, пока клиент не завершит его.

Настройка сетевой лаборатории

Для настройки лаборатории запустите shell-скрипты в терминале. Из корневого каталога репозитория книги (см. рис. B.2 во вступлении) выполните следующие скрипты (пример 3.1).

Пример 3.1. Настройка сетевой лаборатории

```
cd src/network  ❶
bash scripts/setup_containers.sh  ❷
bash scripts/setup_network.sh  ❸
```

❶ Перейдите в каталог `src/network`. Все команды в этой главе выполняются из этого каталога.

❷ Создайте три контейнера: `client`, `router` и `server`.

❸ Настройте необходимые сети и маршрутизацию между контейнерами `client` и `server` через `router` (рис. 3.3)[1].

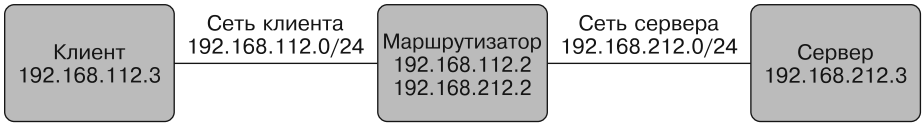


Рис. 3.3. Топология сети лаборатории

Теперь подробнее разберем понятие маршрутизации, которое упоминалось в аналогии с офисом открытого типа. Маршрутизация нужна для передачи сетевых пакетов между сетями. Исходный сетевой протокол Ethernet проектировали для соединения одноранговых устройств на одном проводе; он не предусматривал подключение нескольких сетей или предотвращение петель, возникающих из-за неправильно соединенных кабелей[2]. Маршрутизация требует, чтобы устройства в сети хранили таблицу направлений для соединения с устройствами из других сетей. Отправляя сетевой трафик, устройство либо передает его по своей локальной сети (например, Ethernet), либо, если трафик

---

[1] Выбранные диапазоны сетей в этой лаборатории предотвращают конфликты с вашей домашней сетью 192.168.X.0/24.

[2] История Ethernet обсуждается в известном докладе Радьи Перлман (Radia Perlman) Network Protocol: Myths, Missteps and Mysteries («Сетевой протокол: мифы, ошибки и загадки») (https://oreil.ly/vSBLW) с конференции SharkFest Wireshark Developer and User Conference 2023 года. См. также лекцию Боба Меткалфа (Bob Metcalfe), прочитанную в честь вручения премии Тьюринга на веб-конференции Ассоциации вычислительной техники (ACM) в 2023 году (https://oreil.ly/hesYZ).

предназначен для удаленной сети, отправляет его по наиболее точному маршруту, который соответствует целевому IP-адресу трафика.

В примере 3.2 показано, как будет выглядеть таблица маршрутизации нашего клиента. Выполните эту команду в терминале, который использовался для настройки лаборатории.

Пример 3.2. Таблица маршрутизации клиента (требует настройки из примера 3.1)

```
docker compose exec client bash -c "ip route"  ❶

default via 192.168.112.1 dev eth0             ❷
192.168.212.0/24 via 192.168.112.2 dev eth0    ❸
192.168.112.0/24 dev eth0 proto kernel scope link src 192.168.112.3  ❹
```

❶ Команда `ip route` выводит маршруты, доступные в контейнере client.

❷ Строка вывода указывает, что маршрут по умолчанию проходит через шлюз с IP-адресом `192.168.112.1` в сетевом интерфейсе `eth0`. Маршрут по умолчанию используется, если ни один из маршрутов в таблице маршрутизации не соответствует IP-адресу назначения пакета. Сетевой интерфейс — это программное представление сетевой карты, управляемое операционной системой. Термины «шлюз» (gateway) и «маршрутизатор» (router) часто используются как взаимозаменяемые, однако *шлюз* — более общий термин, в отличие от *маршрутизатора*, который функционирует на сетевом уровне модели OSI. Обратите внимание на то, что в Docker маршрут и шлюз настраиваются автоматически.

❸ Строка показывает, что сетевые пакеты, предназначенные для удаленной сети `192.168.212.0/24` (сеть сервера), отправляются через маршрутизатор с IP-адресом `192.168.112.2` в сетевом интерфейсе `eth0` клиента. Этот маршрут и маршрутизатор создаются явно в процессе настройки лаборатории.

❹ Строка представляет маршрут для локальной сети `192.168.112.0/24`. На то, что сеть локальная, указывает атрибут `scope link`, ссылающийся на канальный уровень модели OSI, доступный напрямую в интерфейсе `eth0`. Шлюз (`via`) не указан, а `src` — это IP-адрес клиента. Строка указывает, что пакеты, предназначенные для локальной сети, можно доставлять напрямую, без маршрутизации через посредника. Эту строку также настраивает Docker.

Маршрутизатор должен иметь доступ (быть подключенным) к каждой сети, с которой ему предстоит работать. Поэтому маршрутизатор имеет IP-адрес как в сети клиента (`192.168.112.2`), так и в сети сервера (`192.168.212.2`). Кроме того, серверу нужна таблица маршрутизации, аналогичная той, что есть у клиента. Чтобы получить ее, используйте команду `ip route` (пример 3.3).

Пример 3.3. Таблица маршрутизации сервера (требует настройки из примера 3.1)

```
docker compose exec server bash -c "ip route"

default via 192.168.212.1 dev eth0
192.168.112.0/24 via 192.168.212.2 dev eth0
192.168.212.0/24 dev eth0 proto kernel scope link src 192.168.212.3
```

Служба TCP ECHO

В этом подразделе мы обсудим протокольный уровень обмена сообщениями между клиентом и сервером TCP ECHO. В примере 3.4 захват пакетов в соединении с TCP ECHO выполняется с помощью `tshark` (https://www.wireshark.org) — анализатора сетевых протоколов с интерфейсом командной строки. Из вывода удалена информация, нерелевантная на данный момент.

Пример 3.4. Захват пакетов клиента TCP ECHO с помощью tshark (требует настройки из примера 3.1)

```
CLIENT=netcat
docker compose exec client bash -c \
       "tshark --read-file tests/tcp_echo_client_${CLIENT}_reference.pcap | \
       bash scripts/trim_client_pcap.sh"
```

Вывод содержит последовательность фреймов Ethernet (пример 3.5). Ниже приведены пояснения к элементам последовательности, отражающим разные этапы обмена сообщениями на уровне протокола TCP: трехэтапное рукопожатие TCP (строки 3–5), обмен сообщениями Hello (строки 6–9) и завершение TCP-соединения (строки 10–12).



Мы даем достаточно подробные пояснения к примеру, чтобы вы могли продолжить изучение TCP самостоятельно. Прежде чем перейти к пояснениям, рекомендуем посмотреть одно из следующих видео: 10-минутное видео TCP Connection Walkthrough («Пошаговый разбор TCP-соединения») Бена Итера (Ben Eater) (https://oreil.ly/g9ukF); How TCP Works — The Handshake («Принцип работы TCP: рукопожатие») Криса Грира (Chris Greer) (https://oreil.ly/vEXrV) или его двухчасовую трансляцию TCP Fundamentals (Основы TCP) в двух частях (https://oreil.ly/Cv5gn).

Пример 3.5. Последовательность пакетов клиента TCP ECHO

```
clientMAC → Broadcast ARP Who has router? Tell client                          ❶
routerMAC → clientMAC ARP router is at routerMAC                               ❷
client → server       TCP [SYN]      Seq=0=c0                       Len=0      ❸
server → client       TCP [SYN, ACK] Seq=0=s0       Ack=1=c0+1      Len=0      ❹
client → server       TCP [ACK]      Seq=1=c0+1     Ack=1=s0+1      Len=0      ❺
client → server       TCP [PSH, ACK] Seq=1=c0+1     Ack=1=s0+1      Len=5      ❻
```

```
server → client       TCP [ACK]      Seq=1=s0+1      Ack=6=c0+1+5   Len=0  ❼
server → client       TCP [PSH, ACK] Seq=1=s0+1      Ack=6=c0+1+5   Len=5  ❽
client → server       TCP [ACK]      Seq=6=c0+1+5    Ack=6=s0+1+5   Len=0  ❾
client → server       TCP [FIN, ACK] Seq=6=c0+1+5    Ack=6=s0+1+5   Len=0  ❿
server → client       TCP [FIN, ACK] Seq=6=s0+1+5    Ack=7=c0+1+5+1 Len=0  ⓫
client → server       TCP [ACK]      Seq=7=c0+1+5+1  Ack=7=s0+1+5+1 Len=0  ⓬
```

❶ Клиент обращается к таблице маршрутизации, чтобы получить IP-адрес маршрутизатора. Затем он отправляет широковещательный ARP-запрос по Ethernet, чтобы определить MAC-адрес маршрутизатора (`routerMAC`). Протокол ARP позволяет сопоставить IP-адрес с MAC-адресом в локальной сети.

❷ Маршрутизатор (`routerMAC`) отвечает клиенту по Ethernet, подтверждая, что IP-адрес маршрутизатора соответствует его MAC-адресу.

❸ Запускается трехэтапное рукопожатие TCP. Клиент инициирует TCP-соединение, отправляя сегмент с нулевой полезной нагрузкой (`Len=0`) и установленным флагом `SYN`, и присваивает сегменту номер в последовательности (`Seq=0`). Первый номер последовательности клиента (`c0`) и сервера (`s0`) генерируется по алгоритму и обычно отличен от ноля. Однако для удобства чтения `tshark` присваивает последовательностям клиента и сервера относительные номера, начиная с нуля (`Seq=0`).

❹ Сервер отвечает клиенту сегментом с нулевой полезной нагрузкой, устанавливает свой номер последовательности (`Seq=s0`) и подтверждает получение «фантомного» байта клиента (phantom byte, также называемый *ghost byte*), добавляя 1 байт к номеру последовательности клиента (`Ack=c0+1`). Как упоминалось ранее, `tshark` отображает это действие как `Ack=1` (вместо `Ack=c0+1`, которое может оказаться большим числом). В чем же смысл подтверждения, если длина полезной нагрузки равна нулю? Считается, что сегменты с установленными флагами `SYN` или `FIN` имеют дополнительную длину 1 байт. Этот «фантомный» байт и нужно подтвердить. Отправленный сегмент имеет оба флага: `SYN` и `ACK`.

❺ Номер последовательности клиента увеличивается на 1 байт (`Seq=c0+1`) за счет отправленного «фантомного» байта. Клиент также подтверждает получение «фантомного» байта сервера, добавляя 1 байт к начальному номеру последовательности сервера (`Ack=s0+1`), и отправляет этот сегмент с установленным флагом `ACK`. Трехстороннее рукопожатие TCP завершается, и клиент готов отправлять сообщения.

❻ Клиент отправляет 5-байтовое сообщение `Hello` (`Len=5`) серверу. Установленный флаг `PSH` указывает, что стек TCP на сервере должен доставить сегмент приложению немедленно без буферизации.

❼ Сервер подтверждает получение 5-байтового сегмента от клиента (`Ack=c0+1+5`).

❽ Сервер возвращает клиенту то же 5-байтовое сообщение `Hello` (`Len=5`).

❾ Клиент увеличивает номер своей последовательности на длину (5) отправленного сообщения `Hello` (`Seq=c0+1+5`) и отправляет сегмент `ACK`.

❿ Клиент отправляет сегмент `FIN`, чтобы завершить соединение. Это происходит, когда клиенту больше нечего отправлять.

⓫ Сервер увеличивает номер последовательности на длину сообщения `Hello`, которое он вернул клиенту, подтверждает получение 7 байт от клиента (включая два фантомных: `Ack=c0+1+5+1`) и устанавливает флаг `FIN`, завершая соединение.

⓬ Клиент увеличивает номер подтверждения на 1 фантомный байт, полученный от сервера, и подтверждает завершение соединения флагом `ACK`.

TCP-СОЕДИНЕНИЕ И TCP-ПОТОК

TCP-поток — это последовательность *TCP-сегментов*, передаваемых по *TCP-соединению*. Сегменты состоят из *байтов* — 8-битных единиц данных.

Иногда вместо термина «TCP-соединение» используют «TCP-сессия» (TCP session), но в технической документации TCP (стандарт RFC) этот термин не встречается. «Сессия» чаще встречается в протоколах более высоких уровней.

TCP-поток из примера 3.5 охватывает только часть конечного автомата для TCP-соединения (https://oreil.ly/ZIDus). В примере показано трехэтапное рукопожатие TCP, после которого клиент отправляет серверу сообщение `Hello`, сервер отвечает тем же сообщением, и клиент инициирует завершение TCP-соединения. Чтобы упростить объяснение, мы намеренно представили идеальный сценарий TCP-соединения без сетевых ошибок.

В реальных же сценариях протокол TCP обрабатывает разные сетевые ошибки, например потерю, дублирование или доставку пакетов в неправильном порядке. Кроме того, TCP управляет повторными передачами данных, регулирует объем отправляемых данных с помощью управления потоком, обнаруживает и минимизирует сетевые перегрузки. Ключевая характеристика TCP — протокол требует установки и поддержания соединения между взаимодействующими сторонами: клиентом и сервером.



Протокол пользовательских датаграмм (User Datagram Protocol, UDP) — это протокол передачи данных, не требующий установления соединения (в отличие от TCP). Оба протокола входят в стек TCP/IP, однако UDP проще, чем TCP: в UDP отсутствует начальное рукопожатие для установки соединения и нет функций, обеспечивающих надежность соединения, например повторные передачи, гарантированный порядок доставки пакетов или контроль сетевых перегрузок. За счет этого UDP передает данные быстрее и широко применяется в играх или потоковой передаче медиа, где периодическая потеря данных некритична.

Сервер TCP ECHO

В примере 3.6 показан полный код на Python для сервера TCP ECHO, который позволяет установить TCP-соединение из примера 3.5 (где клиент и сервер обмениваются сообщением `Hello`). В коде используются системные вызовы — API-сокеты TCP-сервера, показанные на рис. 3.1 (см. выше): `socket`, `bind`, `listen`, `accept`, `recv`, `send` и `close`.

Обратите внимание: это не полная реализация TCP-сервера. В ней не поддерживаются одновременные подключения клиентов и не обрабатывается завершение соединения при сбое или остановке клиента или сервера[1].

Пример 3.6. Сервер TCP ECHO

```
# src/network/src/tcp_echo/server.py

import socket

BUFSIZE_BYTES = 8  ❶
CLOSE_WAIT = 8  ❶

with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as server_socket:  ❷
    server_socket.bind(("", 8080))  ❸
    server_socket.listen()  ❹
    while True:  ❺
        print("Waiting for a new connection from client ...")
        conn_socket, client_address = server_socket.accept()  ❻
        while True:  ❼
            data = conn_socket.recv(BUFSIZE_BYTES)  ❽
            if data:
                data_decoded = data.decode()  ❾
                print(f"Received {data_decoded} from client {client_address}")
```

[1] См.: *Stevens W. R.* Unix Network Programming. The Sockets Networking API, Vol. I. — Pearson Education, 2004.

```
            conn_socket.send(data)  ❿
            print(f"Sent {data_decoded} to client {client_address}")
        conn_state = conn_socket.getsockopt(socket.SOL_TCP, socket.TCP_INFO)
        if conn_state == CLOSE_WAIT:  ⓫
            print(f"Client {client_address} closed connection")
            conn_socket.close()
            break
```

❶ Этот блок определяет параметры, используемые socket API.

❷ Программа создает TCP-сокет сервера для IP версии 4 (`AF_INET`). Сокет `server_socket` создается с использованием контекстного менеджера `with`. Контекстный менеджер гарантирует закрытие сокета после завершения кода внутри блока `with`, даже при возникновении ошибок.

❸ Метод `bind()` (системный вызов) связывает серверный сокет с любым сетевым интерфейсом на хосте сервера и с портом `8080`.

❹ Сокет работает на сервере, прослушивая порт и ожидая подключения нового клиента.

❺ Бесконечный внешний цикл поддерживает непрерывную работу сервера.

❻ Когда подключается новый клиент, системный вызов `accept()` принимает подключение. Этот вызов создает и возвращает сокет соединения для подключенного клиента вместе с кортежем (IP-адрес, порт).

❼ Запускается бесконечный внутренний цикл, который не прекращается, пока клиент не отправит данные или не закроет соединение.

❽ Сокет соединения на сервере принимает от клиента данные, закодированные в байтах. Размер буфера составляет `BUFSIZE_BYTES` и должен быть степенью двойки.

❾ Метод `decode()` преобразует данные из байтов в строку, чтобы их можно было вывести.

❿ Сервер отправляет закодированные данные клиенту по сокету соединения.

⓫ Это условие отслеживает состояние TCP-соединения, установленного с клиентом. Если соединение находится в состоянии `CLOSE_WAIT`, то сокет соединения закрывается и TCP-сокет сервера возвращается в режим ожидания нового подключения клиента.

Клиент TCP ECHO с использованием netcat

В этом подразделе для разбора работы службы TCP ECHO мы будем использовать высокоуровневый клиент командной строки `netcat` (https://nc110.sourceforge.io) — это утилита командной строки Unix для обмена данными по TCP или UDP. Инструмент устраняет необходимость писать клиентский сетевой код и позволяет сосредоточиться на потоке данных. В следующем

упражнении мы будем использовать `netcat` в качестве клиента, отправляющего серверу сообщение `Hello`.

На рис. 3.4 показан снимок экрана с последовательностью команд для этого упражнения.

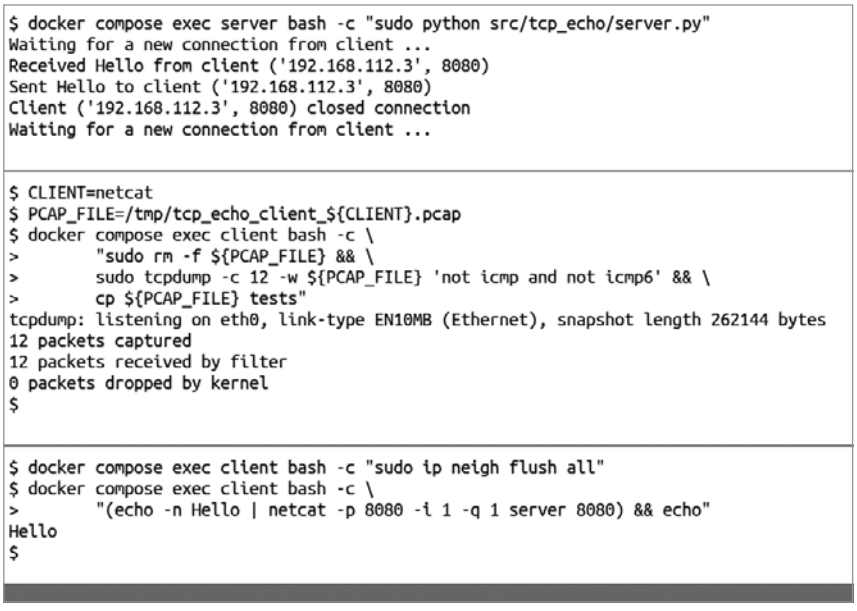


Рис. 3.4. Использование службы TCP ECHO с netcat (требует настройки из примера 3.1)

Если вы делаете упражнение параллельно с чтением, то выполните шаги из примера 3.7. Потребуется окно с тремя панелями — можно создать его с помощью `tmux` (см. пример B.2 во вступлении).

Пример 3.7. Использование службы TCP ECHO с netcat (требует настройки из примера 3.1)

```
# Терминал  ❶
tmux new-session \; split-window -v \; split-window -v \;

# Верхняя панель  ❷
docker compose exec server bash -c "sudo python src/tcp_echo/server.py"

# Центральная панель  ❸
CLIENT=netcat
PCAP_FILE=/tmp/tcp_echo_client_${CLIENT}.pcap
docker compose exec client bash -c \
       "sudo rm --force ${PCAP_FILE} && \
       sudo tcpdump -c 12 -w ${PCAP_FILE} 'not icmp and not icmp6' && \
       cp ${PCAP_FILE} tests"
```

```
# Нижняя панель  ❹
docker compose exec client bash -c "sudo ip neigh flush all"
docker compose exec client bash -c \
       "(echo -n Hello | netcat -p 8080 -i 1 -q 1 server 8080) && echo"
```

❶ Создайте три панели с помощью `tmux`.

❷ Запустите сервер TCP ECHO.

❸ Запустите `tcpdump` для захвата пакетов. Сначала программа будет в режиме ожидания. Далее, когда обмен сообщениями со службой TCP ECHO закончится, программа завершит работу и захватит ожидаемые 12 пакетов. В каталог `tests` сохранится файл `tcp_echo_client_netcat.pcap`. Там же находятся все эталонные файлы `.pcap` из этой главы, имеющие суффикс `*_reference.pcap`.

❹ Прежде чем запускать клиент на третьей панели, очистите ARP-кэш на стороне клиента. ARP-кэш хранит соответствия IP-адресов MAC-адресам; его очистка нужна, чтобы удостовериться, что клиент выполняет ARP-запрос, а не использует кэш. Затем запустите клиент TCP ECHO с помощью `netcat`. Клиент должен вывести сообщение `Hello`, полученное от сервера, и завершить работу.



Если при запуске сервера возникает ошибка OSError: [Errno 98] Address already in use, то подождите несколько минут, пока TCP автоматически удалит оставшиеся соединения от предыдущих попыток. Если количество захваченных пакетов не совпадает с ожидаемым, то убедитесь, что на сервере нет старых TCP-сокетов. Их можно вывести с помощью команды ss (socket statistics):

```
docker compose exec server ss --all --numeric –tcp
```

Если команда выводит сокет-соединения в состоянии TIME-WAIT, то подождите еще несколько минут — они очистятся самостоятельно. Если возникают другие проблемы (например, при захвате пакетов появляются дополнительные ARP-запросы или предупреждение TCP Previous segment not captured), то может потребоваться выполнить действия, описанные в примечании «Отключение сетевой лаборатории» в конце этой главы. После этого нужно повторить действия из примера 3.1, чтобы восстановить исходное состояние лаборатории. Если вы работаете в Linux, то может также потребоваться остановить службу avahi (например, sudo systemctl stop avahi-daemon в Debian/Ubuntu), чтобы ее трафик не попадал в захват пакетов.

После завершения работы клиента и `tcpdump` можно остановить сервер, нажав Ctrl+C. Захваченная последовательность пакетов должна соответствовать представленной в примере 3.5.

Клиент TCP ECHO с использованием scapy

В предыдущем подразделе мы показали, как отправлять сообщения с помощью `netcat` в роли TCP-клиента, и объяснили, как происходит обмен данными между клиентом и сервером по TCP. В этом подразделе мы рассмотрим низкоуровневую реализацию клиента TCP ECHO на Python, чтобы лучше разобраться в самом протоколе TCP.

Scapy (https://scapy.net) — это библиотека Python для взаимодействия с сетевыми пакетами. Она работает на уровнях IP, TCP и др., представленных в виде классов Python. Эта библиотека позволяет управлять TCP-соединением на более низком уровне, чем при использовании клиента наподобие `netcat` или socket API, и при этом не погружаться в байтовую структуру протоколов. С помощью этого инструмента можно явно вычислять TCP-номера последовательности и подтверждения, добавлять фантомные байты или устанавливать TCP-флаги. Все это помогает лучше понять конечный автомат TCP.

В примере 3.8 мы сосредоточимся только на трехэтапном рукопожатии, поскольку оно демонстрирует основные возможности `scapy` при работе с TCP.

Пример 3.8. TCP ECHO-клиент на scapy

```
# src/network/src/tcp_echo/client_scapy.py

from scapy.all import IP, RandInt, TCP, send, sr

F = {  ❶
    "ACK": "A",
    "FIN-ACK": "FA",
    "PSH-ACK": "PA",
    "SYN": "S",
    "SYN-ACK": "SA",
}

ip = IP(src="client", dst="server")  ❷

sport = dport = 8080  ❸
c0 = int(RandInt())  ❹

tcp = TCP(sport=sport, dport=dport, flags=F["SYN"], seq=c0)  ❺
packet = ip / tcp  ❻
ans, unans = sr(packet, verbose=0)  ❼

s0 = ans[0][1][TCP].fields["seq"]  ❽

tcp = TCP(sport=sport, dport=dport, flags=F["ACK"], seq=c0 + 1, ack=s0 + 1)  ❾
packet = ip / tcp
send(packet, verbose=0)  ❿
```

❶ В этом словаре содержатся полные названия TCP-флагов — ключи к коротким наименованиям, используемым в `scapy`.

❷ IP-уровень `scapy` определяет исходный и целевой адреса (клиента и сервера соответственно). Обратите внимание, что имена `client` и `server` преобразуются в IP-адреса.

❸ Определяется TCP-порт клиента (`sport`) и сервера (`dport`). Клиент и сервер используют один номер порта на своих компьютерах.

❹ В качестве начального номера последовательности TCP для клиента (`c0`) устанавливается случайное целое число. Значение `c0` задают для ручного расчета номеров последовательности клиента. Процедура безопасного выбора начального номера последовательности сложнее[1].

❺ TCP-уровень `scapy` определяет TCP-сегмент `SYN` для отправки от клиента к серверу.

❻ Эта строка определяет IP-пакет, состоящий из IP- и TCP-слоев. Оператор `/` в `scapy` обозначает композицию двух слоев.

❼ Начинается трехстороннее рукопожатие TCP. Для отправки и приема пакетов в `scapy` используется функция `sr()`. Функция возвращает два списка (кортеж из двух элементов). Первый элемент, `ans`, — список пар (отправленный пакет, ответ), а второй, `unans`, — список пакетов, на которые ответ не был получен. Чтобы отключить детальный вывод прогресса обмена пакетами, устанавливается `verbose=0`.

❽ Эта строка извлекает начальный номер последовательности сервера (`s0`) из ответа. Наш код управляет начальным номером последовательности клиента (`c0`), а сервер управляет своим.

❾ Эта строка определяет TCP-уровень для TCP-сегмента `ACK`, который клиент отправит серверу.

❿ Клиент отправляет серверу сегмент `ACK`. Трехстороннее рукопожатие TCP завершается.

Чтобы реализовать код на практике, будем действовать по аналогии с примером из предыдущего подраздела «Клиент TCP ECHO с использованием netcat». В примере 3.9 используются похожие команды, за исключением переменной окружения `CLIENT=scapy` (которая отвечает за сохранение файла `.pcap` с нужным именем) и использования скрипта `client_scapy.py` вместо `netcat` в качестве TCP-клиента.

---

[1] См. RFC 1948, Defending Against Sequence Number Attacks («Защита от атак с подменой порядковых номеров») (https://oreil.ly/Exp1y).

Пример 3.9. Использование службы TCP ECHO со scapy (требует настройки из примера 3.1)

```
# Терминал
tmux new-session \; split-window -v \; split-window -v \;

# Верхняя панель
docker compose exec server bash -c "sudo python src/tcp_echo/server.py"

# Центральная панель
CLIENT=scapy
PCAP_FILE=/tmp/tcp_echo_client_${CLIENT}.pcap
docker compose exec client bash -c \
       "sudo rm --force ${PCAP_FILE} && \
       sudo tcpdump -c 12 -w ${PCAP_FILE} 'not icmp and not icmp6' && \
       cp ${PCAP_FILE} tests"

# Нижняя панель
docker compose exec client bash -c "sudo ip neigh flush all"
docker compose exec client bash -c \
       "sudo python src/tcp_echo/client_scapy.py && echo"
```

Клиент `scapy` должен генерировать ту же последовательность пакетов, что и `netcat`, как было показано в примере 3.5.

Безопасность

Проблема безопасности сетевых протоколов существовала с момента их возникновения, несмотря на то что изначально протоколы развивались в доверенных сообществах исследовательских институтов. Например, в 1980-х годах обнаружилось, что TCP-соединения подвержены атакам перехвата, поскольку начальный номер последовательности можно подобрать. Методы защиты от таких атак были предложены в документе RFC 1948 *Defending Against Sequence Number Attacks* («Защита от атак с подменой порядковых номеров») (https://oreil.ly/ObzPA).

Сегодня безопасность сетевого обмена данными обеспечивается с помощью шифрования. Во второй половине XX века стандартизированные алгоритмы шифрования были недоступны из-за «криптографических войн» (Crypto Wars) (https://oreil.ly/ltIKL), в ходе которых США и их союзники ограничивали доступ общественности и других стран к криптографии (не путать «криптографические войны» с более поздними «криптовалютными войнами»). Ограничения на доступ иностранных государств к технологиям шифрования сняли только в 1990-х годах, что привело к разработке веб-протоколов шифрования, таких как SSL/TLS (Secure Sockets Layer/Transport Layer Security).

Как видно на рис. 3.5, в протоколе TLS задействованы три области криптографии: *асимметричные шифры*, *симметричные шифры* и *хеш-функции*. Криптография — это раздел *криптологии*, изучающей методы безопасной передачи данных. В криптологию входят *криптография*, в рамках которой изучают защиту передачи данных, и *криптоанализ*, изучающий методы дешифровки зашифрованной информации (взлома шифров)[1]. В криптографии безопасность, подразумевающая конфиденциальность, целостность, аутентичность и неоспоримость информации, достигается с помощью криптографических примитивов[2].

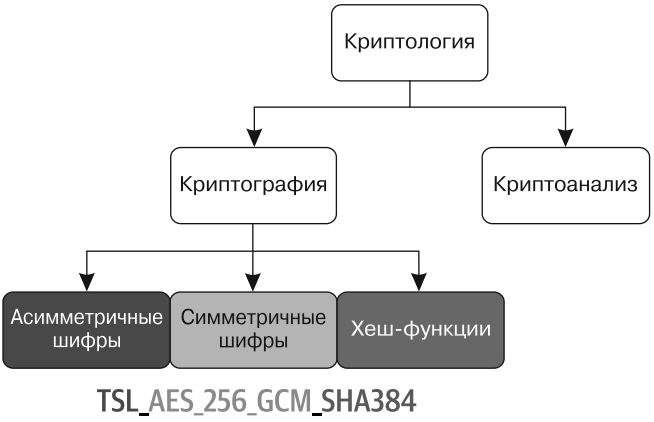


Рис. 3.5. Области криптографии, используемые в наборе шифров TLS версии 1.3

Перед обсуждением TLS рекомендуем вспомнить основы криптографии и сертификатов TLS. В репозитории кода книги, в файле `src/network/README.md`, можно найти небольшое введение прикладного характера по указанным темам. Если вы уже знакомы с криптографией и сертификатами открытых ключей X.509 (Public Key Infrastructure, PKI), то переходите к следующему подразделу.

---

[1] Введение в математические основы криптографии см. в книге Кристофа Пара (Christof Paar) и др. Understanding Cryptography (Springer, 2024). См. также бесплатные лекции Пара, являющиеся логическим продолжением книги (https://oreil.ly/sQM71). Криптографии для программистов, в частности реализации основных алгоритмов, посвящены бесплатные курсы Сирилла Гёсси (Cyrill Gössi) на YouTube (https://oreil.ly/8uC11) и книга Practical Cryptography in Python Сета Джеймса Нильсона (Seth James Nielson) и Кристофера К. Монсона (Christopher K. Monson) (Apress, 2019).

[2] См. также лекцию Сирилла Гёсси Cryptography with Python 2: Cryptography and Its Classic Security Goals («Криптография с Python 2: Криптография и ее классические цели безопасности») (https://oreil.ly/3SOYZ).

Клиент TCP ECHO с OpenSSL

Для защиты сервиса TCP ECHO с помощью TLS требуется цепь сертификатов X.509 для аутентификации сервера. В большинстве операционных систем TLS не реализован в пространстве ядра по умолчанию. Вместо этого его можно реализовать с помощью сторонних библиотек в пользовательском пространстве. Можно также реализовать клиент TCP ECHO поверх TLS, используя стандартный API библиотеки SSL в Python (https://oreil.ly/nbFNU). Однако в этом подразделе, чтобы сократить объем кода, мы будем использовать утилиту командной строки `openssl` в качестве клиента ECHO-сервиса, защищенного TLS. Итоговый захват пакетов в примере 3.12 объясняется с точки зрения ключевых характеристик протокола TLS.

> **СЕРТИФИКАТЫ ОТКРЫТЫХ КЛЮЧЕЙ X.509**
>
> Цель *стандарта X.509* — снизить вероятность атак посредника (MITM) за счет введения сертификатов. Сертификат X.509 связывает открытый ключ с дополнительной идентификационной информацией, например с именем сервера, и подписывает этот ключ с помощью алгоритма цифровой подписи.
>
> Сертификаты, которые используют алгоритмы цифровой подписи, основанные на шифровании с открытым ключом, сами подвержены MITM-атакам. Поэтому сертификаты X.509 (также называемые *SSL*- или *TLS*-сертификатами) полагаются не только на технологические средства для предотвращения MITM-атак, но и на внешние источники доверия, в качестве которых выступают центры сертификации (certificate authorities, CA) — реальные организации, управляемые людьми. Доверие формируется и поддерживается с помощью сертификатов. Однако изначальное доверие к центру сертификации устанавливается через отдельный, безопасный канал связи, например через распространение доверенных сертификатов поставщиками операционных систем.

Теперь выполним действия из примера 3.10, предварительно настроив окружение в соответствии с примером 3.1.

Пример 3.10. Служба TCP ECHO, защищенная с помощью TLS (требует настройки из примера 3.1)

```
# Терминал ❶
tmux new-session \; split-window -v \; split-window -v \;

# Верхняя панель ❷
docker compose exec server bash -c "sudo python src/tcp_echo/server.py"
```

```
# Центральная панель  ❸
CLIENT=openssl
PCAP_FILE=/tmp/tcp_echo_client_${CLIENT}.pcap
docker compose exec client bash -c \
       "sudo rm --force ${PCAP_FILE} && \
       sudo tcpdump -c 18 -w ${PCAP_FILE} 'port 443' && \
       cp ${PCAP_FILE} tests"

# Нижняя панель  ❹
docker compose exec client bash -c \
       'export SSLKEYLOGFILE=tests/SSLKEYLOGFILE.client && \
        (echo -n Hello && sleep 1 && echo -n) | \
        openssl s_client -brief -connect server:443 \
       -CAfile ca.crt -verify_return_error \
       -keylogfile ${SSLKEYLOGFILE} -servername server -tls1_3'
```

❶ Создайте три панели с помощью `tmux`.

❷ Запустите сервер. В отличие от предыдущего примера, здесь, помимо сервера, в фоновом режиме запущена программа `stunnel` (не отражено в коде). Программа прослушивает порт 443 и выступает в роли TLS-обертки для сервера TCP ECHO[1].

❸ Запустите `tcpdump`. В этот раз захват трафика выполняется на TLS-порте `443`. Для удобства трафик ARP исключен из захвата.

❹ Вызовите клиент `openssl`. По аналогии с другими примерами клиент выводит `Hello`, полученное от сервера, и завершает работу. Обратите внимание на наличие корневого сертификата CA (`ca.crt`) среди параметров командной строки. Это значит, что решение доверять этому CA было принято на стороне клиента.

   Операционные системы и крупные программы, например браузеры, имеют собственное хранилище сертификатов от CA, которым они решили доверять. Однако клиент может поместить в это хранилище любой сертификат или довериться CA разово, как в нашем примере.

   Еще один важный элемент — файл `SSLKEYLOGFILE` (в данном случае `SSLKEYLOGFILE.client`). Многие программы, реализующие TLS, например `openssl` или браузеры, поддерживают создание такого файла. Он хранит секреты, необходимые для расшифровки данных, передаваемых во время TLS-сессии. При работе с реальными системами не создавайте этот файл или храните его в закрытом доступе!

В примере 3.11 показано, как расшифровать захваченные пакеты с помощью секретов из `SSLKEYLOGFILE` и сохранить их в файле `.pcapng` (PCAP Next Generation Dump File Format).

---

[1] Детали конфигурации stunnel смотрите в файле src/network/scripts/setup_containers.sh.

Пример 3.11. Расшифровка захвата TLS-пакетов с помощью editcap (требует настройки из примера 3.10)

```
CLIENT=openssl
docker compose exec client bash -c \
       "cd tests && \
       editcap --inject-secrets tls,SSLKEYLOGFILE.client \
       tcp_echo_client_${CLIENT}.pcap tcp_echo_client_${CLIENT}.pcapng"
```

В примере 3.12 показана расшифрованная последовательность пакетов, прочитанная командой `tshark --read-file tests/tcp_echo_client_openssl.pcapng`.

Хотя пояснения к следующему примеру достаточно объемные, они охватывают только основные возможности TLS. Например, не упоминаются общий ключ, сессионные билеты и некоторые типы производных секретов[1]. Предлагаем сравнить нашу последовательность шагов с расширенным справочным примером TLS-сессии версии 1.3 (https://tls13.xargs.org).

Пример 3.12. Расшифрованная последовательность пакетов клиента OpenSSL ECHO

```
client → server TCP     [SYN]      Seq=0     Len=0                          ❶
server → client TCP     [SYN, ACK] Seq=0     Ack=1       Len=0              ❶
client → server TCP     [ACK]      Seq=1     Ack=1       Len=0              ❶
client → server TLSv1.3 Client Hello (SNI=server)                           ❷
server → client TCP     [ACK]      Seq=1     Ack=254     Len=0              ❸
server → client TLSv1.3 Server Hello, Change Cipher Spec,                   ❹
                        Encrypted Extensions, Certificate, Certificate Verify, ❺
                        Finished                                            ❻
client → server TCP     [ACK]      Seq=254   Ack=1569    Len=0
client → server TLSv1.3 Change Cipher Spec, Finished                        ❼
client → server TLSv1.3                                                     ❽
server → client TCP     [ACK]      Seq=1569  Ack=361     Len=0
server → client TLSv1.3                                                     ❾
client → server TCP     [ACK]      Seq=361   Ack=1596    Len=0
client → server TLSv1.3 Alert (Level: Warning, Description: Close Notify)   ❿
client → server TCP     [FIN, ACK] Seq=385   Ack=1596    Len=0              ⓫
server → client TLSv1.3 Alert (Level: Warning, Description: Close Notify)   ❿
client → server TCP     [ACK]      Seq=386   Ack=1620    Len=0
server → client TCP     [FIN, ACK] Seq=1620  Ack=386     Len=0              ⓫
client → server TCP     [ACK]      Seq=386   Ack=1621    Len=0              ⓫
```

❶ Клиент инициирует трехэтапное TCP-рукопожатие с сервером.

❷ Клиент начинает TLS-рукопожатие, отправляя сообщение `Client Hello`. `Client Hello` — это тип сообщения, определенный протоколом TLS,

[1] Обзор тем, не вошедших в наше обсуждение, представлен в лекции Сирилла Гёсси TLS Essentials 29: TLS 1.3 Master Secret, PSK, Encryption Keys and HKDF («Основы TLS 29: Главный секрет TLS 1.3, PSK, ключи шифрования и HKDF») (https://oreil.ly/GepH8).

отличный от сообщения `Hello`, содержащегося в TCP-сегменте. `Client Hello` содержит список поддерживаемых наборов шифров, причем первым часто указывается `TLS_AES_256_GCM_SHA384`, как наиболее предпочтительный для клиента (см. рис. 3.5 выше).

`Client Hello` также содержит дополнительную информацию в виде TLS-расширений, среди которых Server Name Indication (SNI), сообщающее имя сервера, поддерживаемые алгоритмы подписи сертификатов, версия TLS, предпочитаемая клиентом, и публичный ключ клиента для установления ключа сессии.

Например, для установления ключа клиент может выбрать эллиптическую кривую `X25519`. Эта информация передается по сети в незашифрованном виде.

В качестве упражнения откройте захват пакетов в графическом интерфейсе Wireshark и попробуйте найти эти данные в `Client Hello`.

❸ Сервер подтверждает получение TCP-сегмента с TLS-запросом `Client Hello` от клиента. Избыточное расходование ресурсов системы на TLS-рукопожатие заметно по размеру ACK в несколько сотен байт. Детали TCP-сессии, включающие номера последовательности и номера подтверждения, здесь не рассматриваются. Для сравнения можно обратиться к объяснению TCP, показанному в примере 3.5.

❹ Сервер продолжает TLS-рукопожатие, отправляя несколько TLS-записей в одном TCP-сегменте. Обратите внимание, что `Server Hello` — это тип сообщения, определенный протоколом TLS, а не ответ `Hello` сервера, содержащийся в TCP-сегменте. `Server Hello` подтверждает используемый набор шифров и используемую версию в TLS-расширениях. Затем сервер отправляет свою часть открытого ключа, вычисленную с помощью выбранного клиентом алгоритма. Далее следует устаревшая запись `Change Cipher Spec`, которая уже не используется в TLS версии 1.3.

С этого момента и до завершения TLS-сессии все TLS-записи шифруются. Если не использовать `SSLKEYLOGFILE` для расшифровки трафика, все TLS-записи отобразятся как зашифрованные `Application Data`. До конца TLS-рукопожатия для шифрования используется временный ключ (handshake secret). Он генерируется функцией формирования ключа HKDF (HMAC-based Extract-and-Expand Key Derivation Function)[1] на основе общего секрета и хеша `SHA384` сообщений `Client Hello` и `Server Hello`. Хотя обмен данными уже шифруется, клиент еще не видел сертификат сервера, поэтому сервер пока не прошел аутентификацию.

---

[1] См. RFC 5869, HMAC-Based Extract-and-Expand Key Derivation Function («HKDF (функция вывода ключей на основе HMAC)») (https://oreil.ly/L7uxz).

❺ Сервер отправляет свою цепь сертификатов, в зависимости от размера — в одной или нескольких TLS-записях. Затем сервер отправляет запись `Certificate Verify`, которая содержит подписанный цифровой подписью хеш сообщений TLS-рукопожатия. При подписании используется закрытый ключ сервера, связанный с его сертификатом. Цель этой процедуры — доказать клиенту, что сервер владеет сертификатом, представленным в рамках этой TLS-сессии. Подпись связывает временный открытый ключ сервера, сгенерированный в ходе TLS-сессии (и имеющийся в `Server Hello`), с закрытым ключом сервера, связанным с его сертификатом. Для проверки подписи клиент использует открытый ключ сервера из сертификата.

❻ Сервер отправляет запись `Finished`, содержащую хеш `SHA384` всех записей TLS-рукопожатия (от `Client Hello` до записи сервера `Certificate Verify`), зашифрованный с использованием временного ключа. Запись отправляется для того, чтобы согласовать хеш с клиентом и, таким образом, обеспечить целостность рукопожатия. Теперь у сервера есть вся информация, необходимая для генерации итоговых ключей шифрования.

❼ Клиент проверяет подпись сертификата, отправленного сервером, отправляет устаревшее сообщение `Change Cipher Spec` и свою запись `Finished`, основанную на всех записях рукопожатия (от `Client Hello` до `Finished`). TLS-рукопожатие завершается. Теперь и клиент, и сервер сгенерировали ключи шифрования. Ключ клиента, сгенерированный самим клиентом, совпадает с ключом клиента, сгенерированным сервером. Аналогично ключ сервера, сгенерированный клиентом, совпадает с ключом сервера, сгенерированным сервером.

❽ Клиент отправляет серверу сообщение `Hello`, зашифрованное с помощью `AES_256_GCM` и ключа клиента.

❾ Сервер отвечает сообщением `Hello`, зашифрованным с помощью `AES_256_GCM` и ключа сервера.

❿ Клиент, у которого больше нет данных для отправки, сообщает серверу о завершении TLS-соединения, отправляя запись `Alert`. Сервер подтверждает завершение соединения своей записью `Alert`.

⓫ Клиент инициирует разрыв TCP-соединения, и соединение завершается.

В завершение главы кратко охарактеризуем назначение каждого компонента набора шифров `TLS_AES_256_GCM_SHA384`[1] версии TLS 1.3, который использовался в примерах этого раздела.

---

[1] На эту же тему см. видео Сирилла Гёсси TLS Essentials 10: TLS Cipher Suites Explained («Основы TLS 10: Объяснение наборов шифров TLS») (https://oreil.ly/aR3vM).

Компонент `TLS` в `TLS_AES_256_GCM_SHA384` обозначает алгоритм установления ключа, который согласовывается во время TLS-рукопожатия. Это может быть эфемерный алгоритм Диффи — Хеллмана (Diffie-Hellman Ephemeral, DHE), эфемерный алгоритм Диффи — Хеллмана на эллиптических кривых (Elliptic Curve Diffie-Hellman Ephemeral, ECDHE) или общий ключ. Компонент `AES_256_GCM` с помощью симметричного шифрования обеспечивает конфиденциальность, целостность и подлинность данных приложения, которыми обмениваются по TLS. Финальный компонент `SHA384` — это хеш-функция, которую использует HKDF для генерации временных ключей на основе общего секрета, вычисленного с помощью алгоритма установления ключа. Для аутентификации сторон во время рукопожатия TLS версии 1.3 применяются сертификаты X.509. Криптографические алгоритмы с открытым ключом, используемые для проверки сертификатов, задаются самими сертификатами и не входят в состав набора шифров TLS.

Упражнения

1. По примеру подключения к службе TCP ECHO, приведенному ранее в этой главе, расширьте код трехэтапного рукопожатия TCP в `scapy`, чтобы отправлять и получать сообщение `Hello`. Затем завершите TCP-соединение. Сохраните код под названием `src/network/src/tcp_echo/client_scapy.py` в соответствии с корневым каталогом репозитория кода книги. Полный код решения уже доступен по этому пути. Используйте `F["PSH-ACK"]` для TCP-флагов клиентского сегмента и `ans, unans = sr(packet, multi=True, timeout=0.1, verbose=0)` для отправки данных и приема ответа (в ответе сервера должно быть два TCP-сегмента: `ACK`, а затем `PSH-ACK`). Не забудьте использовать фантомный байт при завершении TCP-соединения.
2. Напишите клиент для сервера ECHO, используя Python socket API. Сохраните код как `src/network/src/tcp_echo/client.py`. Пример реализации доступен в репозитории кода книги.

Чтобы протестировать обе реализации, настройте лабораторную среду (см. пример 3.1), запустите сервер и `tcpdump` с переменными окружения `CLIENT=scapy` и `CLIENT=python` соответственно. Далее очистите ARP-кэш клиента и запустите нужный клиент, например:

```
docker compose exec client bash -c \
       "sudo python src/tcp_echo/client.py && echo"
```

Реализации также можно протестировать с помощью скрипта из репозитория книги. Чтобы запустить автоматические тесты, не открывая окна вручную, установите GNU Screen (https://oreil.ly/50BkG):

```
CLIENT=python
bash tests/tcp_echo_client.sh $CLIENT
```

Если у вас нет Screen, проанализируйте имеющийся файл `.pcap` с помощью `tshark`, как было показано в примере 3.5. Захват пакетов снова должен содержать ту же последовательность.



Отключение сетевой лаборатории

Завершив работу с сетевой лабораторией из этой главы, остановите контейнеры, выполнив следующую команду:

```
bash scripts/teardown_containers.sh
```

Резюме

В этой главе мы обсудили сетевые протоколы, в частности TCP и TLS. Кроме того, мы реализовали сервер и клиент TCP ECHO, с которыми сервер обменивался текстовыми сообщениями по TCP. Это подводит нас к теме следующей главы, где речь пойдет о протоколе HTTP, лежащем в основе интернета. В двух словах, HTTP — это набор правил для обмена текстовыми сообщениями по TCP.

Помимо HTTP, в главе 4 описаны и другие характеристики сетевых протоколов, важные для веба, например производительность, надежность и безопасность.

ГЛАВА 4

Веб-протоколы

В предыдущей главе мы использовали TCP/IP для отправки и получения последовательности байтов, представляющей текст. Эта идея лежит в основе самого распространенного в современном вебе протокола HTTP (Hypertext Transfer Protocol).

Эта глава посвящена протоколу HTTP, его функциям и развитию. Мы рассмотрим вариации протокола программно, используя такие сетевые инструменты анализа, как `netcat`, `tcpdump`, `iperf`, `curl`, `nghttp2`, `tshark` или `Wireshark`. Для облегчения понимания версии HTTP обсуждаются в главе в хронологическом порядке. Примеры работы HTTP и связанного с ним протокола QUIC на уровне сетевых пакетов подготовят вас к более сложной теме — сетевым API, в основе которых лежит HTTP. В этой главе возможности HTTP описываются с точки зрения клиентов веб-API, в том числе браузеров.

Что такое гипертекст

Термин «гипертекст» был введен Тедом Холмом Нельсоном (Ted Holm Nelson) в 1965 году[1]:

> Под *гипертекстом* будем понимать текстовый или графический материал, характеризующийся настолько сложными внутренними связями, что его невозможно отразить на бумаге.
>
> *Тед Холм Нельсон, 1965*

[1] *Nelson T. H.* Complex Information Processing: A File Structure for the Complex, the Changing and the Indeterminate («Сложная обработка информации: структура файлов для сложного, изменяющегося и неопределенного») // ACM '65: Proceedings of the 1965 20th National Conference, 1965. — P. 84–100. https://oreil.ly/wAO7I.

Однако это не единственное значение термина[1]. В современной веб-среде больший акцент делается не на способах представления информации в гипертексте, а на способах ее связывания[2].



Есть смежный термин «гипермедиа», указывающий на то, что концепция гипертекста применима не только к текстам.

Чтобы понять, что такое гипертекст, нужно обратиться к его истокам. Гипертекст подразумевает установление связей между различными фрагментами информации. Эта идея существовала в разных формах на протяжении всей истории человеческой мысли, и в этом кратком обзоре мы приведем лишь некоторые примеры.

Еще в I веке нашей эры римский философ Сенека рекомендовал вести заметки и «смешивать» их, подобно тому как пчелы, смешивая пыльцу, получают мед[3]. Традицию иллюстрирует также «Атлантический кодекс» (https://oreil.ly/Ux8Oj) — собрание рисунков и записей Леонардо да Винчи, созданных в XV–XVI веках. В кодексе нет связей между заметками, но есть заметки на полях и комментарии. В других трудах, например в работе Карла Линнея «Виды растений» (https://oreil.ly/GNGnC), созданной в XVIII веке, связи между концепциями представлены в виде ссылок на другие работы и номера страниц.

В 1930-х годах Вэнивар Буш (Vannevar Bush), вдохновленный технологией микрофильмов (которую также исследовали другие изобретатели)[4], представил механическое устройство[5] для навигации по индексированным ссылкам.

---

[1] Как показывает миф о Вавилонской башне, достичь согласия в концепциях и их взаимосвязях непросто. Для стандартизации представления информации в вебе используется концепция семантической паутины (Semantic Web, https://oreil.ly/J6swr). Цитируя Бернерса-Ли (Berners-Lee), «семантическая паутина — это расширение текущей веб-среды, в которой информация представлена в стандартизированной форме и интерпретируется однозначно, что улучшает взаимодействие между человеком и компьютером» (2001).

[2] В 1989 году Бернерс-Ли в проекте Information Management: A Proposal («Управление информацией: предложение») (https://oreil.ly/2jYCQ) определил гипертекст как «информацию, предназначенную для чтения человеком и связанную произвольным образом».

[3] Сенека, «Нравственные письма к Луцилию», письмо 84 «О собирании идей» (https://oreil.ly/THUti).

[4] См. статью The Internet Before the Internet: Paul Otlet's Mundaneum («Интернет до интернета: Mundaneum Поля Отле») (https://oreil.ly/WV5yj) Сидни Перковица (Sidney Perkowitz).

[5] *Bush V.* As We May Think («Как мы можем мыслить») // Atlantic Monthly, 1945. https://oreil.ly/zLxqU.

Проект устройства так и не был воплощен в жизнь, однако именно он лег в основу концепции гипертекста Теда Нельсона и некоторых функций современных браузеров. Буш описал устройство так[1]:

> Если пользователь хочет обратиться к определенной книге, он набирает ее код с клавиатуры, и титульная страница книги сразу появляется перед ним в одной из рабочих областей... Специальная кнопка мгновенно переносит его на первую страницу указателя. Таким образом, можно вызвать и просмотреть любую книгу из библиотеки пользователя, и это гораздо удобнее, чем лезть за ней на полку. Поскольку у пользователя есть несколько проекционных областей, он может одновременно просматривать несколько элементов, а также добавлять заметки и комментарии.
>
> *Вэнивар Буш, 1945*

Технология гипертекста получила широкое распространение после реализации проектов Тима Бернерса-Ли в Европейской организации по ядерным исследованиям (ЦЕРН) в 1989 и 1990 годах. Сыграл роль и последующий выпуск исходного кода его программы (браузера) WorldWideWeb на безвозмездной основе в 1993 году[2]. Для поддержки этой реализации[3] был разработан специальный сетевой протокол прикладного уровня — HTTP. Это событие стало началом веба в его современном понимании.

HTTP: настройка лабораторной среды

Прежде чем переходить к обсуждению протокола HTTP, настроим лабораторную среду. Из корневого каталога репозитория кода книги (см. рис. B.2 во вступлении) выполните команды, приведенные в примере 4.1.

Пример 4.1. Настройка лабораторной среды для изучения HTTP

```
cd src/http  ❶
bash scripts/setup_containers.sh  ❷
```

❶ Команда позволяет перейти в каталог `src/http` (все команды в этой главе выполняются из него).

❷ Скрипт создает три контейнера Docker: `client`, `http-server` и `https-server`.

---

[1] Уже в 1968 году в «Матери всех демонстраций» Дага Энгельбарта (Doug Engelbart) были кликабельные ссылки.

[2] См. статью A Short History of the Web («Краткая история веба») на сайте ЦЕРН (https://oreil.ly/cVT0a).

[3] См. краткий 20-страничный обзор HTTP, включая HTTP/2, в книге HTTP Protocols Ильи Григорика. См. также его бесплатную книгу High Performance Browser Networking (https://hpbn.co) (O'Reilly, 2014).

Теперь, когда мы настроили среду, обсудим первую версию HTTP. В отличие от современной версии в ней отсутствовали заголовки и коды состояния и был доступен только метод `GET`. Историю и назначение этих и других функций HTTP мы подробно рассмотрим в этой главе.

HTTP/0.9

Первая версия HTTP, известная как HTTP/0.9, была создана в 1991 году. Протокол спроектировали для работы поверх любого транспортного протокола для установления соединения (в частности, TCP)[1]. Поскольку доставка данных в правильном порядке гарантировалась TCP[2], то в спецификациях HTTP в центре внимания были поиск и получение текстовой информации. Краткое описание протокола[3] выглядит так.

После установки TCP-соединения с сервером (который по умолчанию прослушивает порт `80`) клиент отправляет серверу запрос в виде строки ASCII-символов (https://oreil.ly/epG2A), завершаемой парой `<CR><LF>` (где `<CR>` — возврат каретки, `\r`, а `<LF>` — перевод строки, `\n`) или просто символом `<LF>`. Отправляемый запрос состоит из слова `GET`, пробела и адреса документа. Адрес документа представляет собой одно слово без пробелов; префикс `http:`, имя хоста и номер порта опускаются, если запрос не предназначен для шлюза. В ответе сервера, имеющем формат HTML, используются символы ASCII. Строки в ответе разделяются необязательным символом `<CR>`, за которым следует обязательный `<LF>`. Сервер закрывает TCP-соединение после полной передачи документа с ответом.



Данные, передаваемые в TCP-сегменте, — это последовательность байтов, даже если они кодируют символы ASCII. Отправитель кодирует текст в байты перед передачей по TCP, а получатель декодирует их обратно в текст. Например, слово Hello по схеме ASCII кодируется как последовательность байтов [72, 101, 108, 108, 111] в десятичной записи.

В кодировке ASCII используются 7-битные двоичные числа. Для выравнивания по 8-битным байтам старшему биту присваивается значение 0.

---

[1] Пример работы протокола прикладного уровня, не требующего транспорта и ориентированного на соединение, см. в подразделе «Влияние DNS на соединения в браузере» далее в этой главе.

[2] В RFC 9293 (https://oreil.ly/VhS6f) надежность TCP определяется как обнаружение потерь пакетов (через порядковые номера) и ошибок (через контрольные суммы для каждого сегмента), а также их исправление с помощью повторной передачи. Конечно, TCP предоставляет механизмы для надежной передачи данных, но не дает абсолютной гарантии успешной доставки.

[3] См. The Original HTTP as Defined in 1991 («Оригинальный HTTP, определенный в 1991 году») на сайте Консорциума Всемирной паутины (W3C) (https://oreil.ly/3BbSI).

Протокол HTTP создали поверх TCP для облегчения обмена данными по сети, а HTML разработали на основе стандартного обобщенного языка разметки (Standard Generalized Markup Language, SGML) для структурирования гипертекста с поддержкой ссылок между документами. Адресацию документов HTTP (https://oreil.ly/uimzH) ввели для того, чтобы можно было обращаться к документам.

HTTP и HTML в действии

Ранняя спецификация HTML (https://oreil.ly/CkZBG) не содержала обязательных HTML-тегов, поэтому ответ от HTTP-сервера мог быть любым текстом ASCII. Эта особенность отражена в примере HTTP/0.9, где клиент отправляет серверу запрос `GET /Hello.html`, чтобы получить содержимое файла `Hello.html` (HTML-документа со словом `Hello`). Если мы сравним обмен сетевыми пакетами по HTTP (пример 4.4, см. далее) и по TCP (см. пример 3.5 из главы 3), то увидим, что HTTP/0.9 — это набор правил для обмена текстовыми документами поверх TCP/IP.

В нашем примере `/Hello.html` в запросе клиента — это адрес документа. Кроме того, в адресе может содержаться / или https://www.w3.org/Addressing/HTTPAddressing.html. Например, по адресу документа https://www.oreilly.com/search/?q=learning+api+styles можно найти ключевые слова `learning`, `api` и `styles` в указателе, расположенном по адресу https://www.oreilly.com/search. Напомним, что в большинстве случаев схема `http`, `":"`, `host` и `port` опускаются в строке запроса HTTP `GET` и используются только для установки TCP-соединения с сервером.

На рис. 4.1 показаны этапы формирования запроса клиента HTTP/0.9.

```
$ CLIENT=http0.9
$ PCAP_FILE=/tmp/client_${CLIENT}.pcap
$ docker compose exec client bash -c \
>       "sudo rm -f ${PCAP_FILE} && \
>       sudo tcpdump -w ${PCAP_FILE} 'port 80' && \
>       cp ${PCAP_FILE} tests"
tcpdump: listening on eth0, link-type EN10MB (Ethernet), snapshot length 262144 bytes
^C10 packets captured
10 packets received by filter
0 packets dropped by kernel
$
```

```
$ docker compose exec client bash -c \
>       "echo -en 'GET /Hello.html\r\n\r\n' | netcat -p 8080 http-server 80"
Hello
$
```

Рис. 4.1. Запрос HTTP/0.9 с использованием netcat

СХЕМА ВЫПОЛНЕНИЯ ПРИМЕРОВ В ГЛАВЕ

Большинство примеров имеют схожий порядок действий. Потребуется открыть окно с двумя панелями и выполнить в них команды: на верхней панели захватить сетевой трафик с помощью `tcpdump`, а на нижней — вызвать клиентский запрос для генерации этого трафика, затем остановить захват `tcpdump` и прочитать его с помощью `tshark`.

В стандартных случаях вы увидите только выполненные команды и их вывод, без снимков экрана. Снимки будут показаны, если порядок выполнения команд различается.

Инструменты `tcpdump`, `tshark` и `netcat`, используемые в этой главе, мы уже обсуждали в главе 3. Эталонные файлы `.pcap` из этой главы представлены в каталоге `tests` под заголовком `*_reference.pcap`.

В примере 4.2 представлены выполненные команды с пояснениями. Для реализации потребуется окно с двумя панелями, открытое с помощью `tmux`. Основы работы с `tmux` см. в примере B.2 во вступлении.

Пример 4.2. Порядок выполнения демонстрации HTTP/0.9 (требует настройки из примера 4.1)

```
# Терминал ❶
tmux new-session \; split-window -v \;

# Верхняя панель ❷
CLIENT=http0.9
PCAP_FILE=/tmp/client_${CLIENT}.pcap
docker compose exec client bash -c \
       "sudo rm --force ${PCAP_FILE} && \
       sudo tcpdump -w ${PCAP_FILE} 'port 80' && \
       cp ${PCAP_FILE} tests"

# Нижняя панель ❸
docker compose exec client bash -c \
       "echo -en 'GET /Hello.html\r\n\r\n' | netcat -p 8080 http-server 80"
```

❶ Создайте две панели с помощью `tmux`.

❷ Запустите на клиенте новый захват пакетов с помощью `tcpdump`. Захваченный трафик сохранится в файл `client_http0.9.pcap` в каталоге `tests`.

❸ Вызовите HTTP-запрос `GET /Hello.html` с клиента, используя `netcat`. Клиент должен вывести HTML, полученный от `http-server` (в нашем случае текст `Hello`), и завершить работу.

Остановите `tcpdump`, нажав Ctrl+C, и прочитайте сохраненный захват пакетов с помощью `tshark` (пример 4.3).

Пример 4.3. Чтение файла захвата пакетов с помощью tshark (требует настройки из примера 4.2)

```
CLIENT=http0.9
docker compose exec client bash -c \
       "tshark --read-file tests/client_${CLIENT}.pcap"
```

Вы увидите последовательность, подобную той, которая представлена в примере 4.4. Обратите внимание: некоторые пакеты, имеющиеся в захватах в этой главе (например, сегменты `TCP ACK` или `FIN`), не объясняются, так как TCP мы уже рассматривали в главе 3.

Пример 4.4. Последовательность пакетов клиента HTTP/0.9

```
client → server TCP  8080 → 80   [SYN]      Seq=0 Len=0          ❶
server → client TCP    80 → 8080 [SYN, ACK] Seq=0  Ack=1  Len=0  ❶
client → server TCP  8080 → 80   [ACK]      Seq=1  Ack=1  Len=0  ❶
client → server HTTP             GET /Hello.html\r\n\r\n         ❷
server → client TCP    80 → 8080 [ACK]      Seq=1  Ack=20 Len=0  ❸
server → client HTTP             Continuation                    ❹
client → server TCP  8080 → 80   [ACK]      Seq=20 Ack=8  Len=0  ❺
server → client TCP    80 → 8080 [FIN, ACK] Seq=8  Ack=20 Len=0  ❻
client → server TCP  8080 → 80   [FIN, ACK] Seq=20 Ack=9  Len=0  ❻
server → client TCP    80 → 8080 [ACK]      Seq=9  Ack=21 Len=0  ❻
```

❶ Устанавливается трехстороннее рукопожатие TCP между клиентом, использующим порт `8080`, и сервером, использующим порт 80. Обычно клиентский порт выбирается случайно, но здесь для наглядности он зафиксирован на `8080`.

❷ Клиент отправляет HTTP-запрос для получения документа `/Hello.html` с сервера. Обратите внимание на окончание строки `\r\n\r\n` (`<CR><LF><CR><LF>`) вместо `\r\n` (`<CR><LF>`). Это артефакт реализации HTTP/0.9 на конкретном сервере, который требует наличия HTTP-заголовков после первой пары `<CR><LF>`. Однако HTTP 0.9 не использует заголовки. Длина данных этого TCP-сегмента составляет 19 байт.

❸ Сервер подтверждает получение HTTP-запроса от клиента (`Ack=20=1+19`).

❹ `Continuation` означает, что HTTP-ответ не содержит заголовков, но все равно распознается как HTTP, поскольку его TCP-данные заканчиваются на `<CR><LF>`. То есть TCP-сегмент данных, возвращаемый сервером, имеет вид `Hello\r\n` длиной 7 байт.

❺ Клиент подтверждает получение HTTP-ответа от сервера.

❻ После отправки всего HTML-документа клиенту сервер инициирует завершение TCP-соединения и оно разрывается.

В этой последовательности пакетов показаны ключевые особенности HTTP/0.9. Базовый транспортный протокол — TCP; поверх него через HTTP выполняется обмен сообщениями в ASCII-кодировке. Кроме того, из примера видно, что TCP-соединение закрывается сервером. Оно завершается после того, как сервер полностью отправил сообщение в ответ на единственный запрос клиента.

HTTP/1.0

В 1990-х годах интернет продолжал развиваться: количество сайтов достигло 10 тысяч в 1994 году, 75 тысяч в 1995-м, 600 тысяч в 1996-м и 9 миллионов в 1999-м[1]. Появились модификации исходного протокола HTTP, и в 1996 году был опубликован[2] информационный документ Request for Comments (RFC)[3]. В нем была описана общепринятая на тот момент практика использования HTTP, которую назвали HTTP/1.0.

В RFC для HTTP/1.0 HTTP определялся как «протокол прикладного уровня, обладающий легкостью и скоростью, необходимыми для распределенных, совместных, гипермедийных информационных систем».

В документе были зафиксированы некоторые из уже существовавших возможностей HTTP:

- версия протокола и правила совместимости версий, которые должны соблюдать клиенты и серверы;
- разграничение заголовков и тела сообщения для передачи метаданных в HTTP-запросах и ответах;
- кодировка содержимого для сжатия тела сообщения в форматах `x-gzip` и `x-compress`, которые указываются в заголовке `Content-Encoding`;
- модификаторы сообщений на основе типов MIME (Multipurpose Internet Mail Extensions), теперь чаще называемых MIME-типами (используются для указания желаемого формата представления тела HTTP-сообщения с помощью заголовка `Content-Type`);

---

[1] Согласно данным Netcraft Web Server Survey (https://oreil.ly/4U08P), в период с 2010 по 2024 год количество активных сайтов стабилизировалось на уровне около 200 миллионов. Общее их количество в настоящее время сокращается после достижения пика в 1,8 миллиарда в 2017 году.

[2] См. RFC 1945, Hypertext Transfer Protocol — HTTP/1.0 («Протокол передачи гипертекста — HTTP/1.0») (https://oreil.ly/HOlgK).

[3] Информационный RFC не является официальным интернет-стандартом.

- дополнительные методы запроса помимо `GET`: `HEAD`, `POST`, а также `PUT`, `DELETE`, `LINK` и `UNLINK`, изменяющие состояние сервера по запросу HTTP-клиента[1];
- требование, чтобы запрос `POST` содержал заголовок `Content-Length`;
- коды состояния: `1xx` (информационные), `2xx` (успех), `3xx` (перенаправление), `4xx` (ошибка клиента), `5xx` (ошибка сервера);
- схема базовой аутентификации, а также спецификация для дополнительных схем аутентификации и механизмов шифрования.

Заголовки и тело HTTP-сообщения

Если сравнить запрос `GET` в HTTP/0.9 и HTTP/1.0, мы увидим ряд особенностей версии 1.0: версию протокола, код статуса ответа и разделение сообщения на заголовки и тело. Последовательность выполнения команд для следующего упражнения приведена в примере 4.5.



Сообщение HTTP/1.0 или HTTP/1.1 начинается со стартовой строки. За ней следуют <CR><LF> и последовательность, состоящая из строк заголовков, пустой строки, обозначающей конец заголовков, и необязательного тела сообщения[2].

На этот раз используем корректный HTML-документ вместо текста `Hello`. Минимальный HTML-документ, содержащий `Hello`, — это `<!DOCTYPE html> <title>Hello</title>`, как указано в спецификации HTML5 (https://oreil.ly/ocJ6G).

Для этого примера потребуется окно с двумя панелями. Здесь соблюдается стандартная последовательность шагов, описанная во врезке «Схема выполнения примеров в главе» (см. выше).

Пример 4.5. Порядок выполнения демонстрации HTTP/1.0 (требует настройки из примера 4.1)

```
# Терминал  ❶
tmux new-session \; split-window -v \;

# Верхняя панель  ❷
CLIENT=http1.0
PCAP_FILE=/tmp/client_${CLIENT}.pcap
```

---

[1] HTTP развивается на основе практики его использования. Например, методы LINK и UNLINK позднее исключили из спецификации HTTP/1.1 из-за их низкого распространения (см. раздел «HTTP/1.1» далее в этой главе).

[2] См. интернет-стандарт RFC для HTTP/1.1 (https://oreil.ly/BASAg) от 2022 года.

```
docker compose exec client bash -c \
       "sudo rm --force ${PCAP_FILE} && \
       sudo tcpdump -w ${PCAP_FILE} 'port 80' && \
       cp ${PCAP_FILE} tests"

# Нижняя панель  ❸
docker compose exec client bash -c \
       "echo -en 'GET /HelloValid.html HTTP/1.0\r\n\r\n' | \
       netcat -p 8080 http-server 80"
```

❶ Создайте две панели с помощью `tmux`.

❷ Запустите новый захват пакетов с помощью `tcpdump` на клиенте. Файл `.pcap` сохранится в `client_http1.0.pcap` в каталоге `tests`.

❸ Выполните HTTP-запрос `GET /HelloValid.html` с клиента с помощью `netcat`. Обратите внимание, что HTTP/1.0 не требует заголовков в `GET`-запросе. Клиент выведет HTML, полученный от `http-server`, и завершит работу.

В примере 4.6 показано, что, помимо HTML-контента, в ответе сервера выводится дополнительная информация: код состояния ответа `200 (OK)` и HTTP-заголовки.

Пример 4.6. Ответ сервера HTTP/1.0

```
HTTP/1.0 200 OK\r\n                                  ❶
Date: Wed, 17 Jul 2024 17:48:40 GMT\r\n              ❷
Server: Apache/2.4.61 (Unix)\r\n                     ❸
Last-Modified: Wed, 17 Jul 2024 17:48:35 GMT\r\n     ❹
Content-Length: 37\r\n                               ❺
Connection: close\r\n                                ❻
Content-Type: text/html\r\n                          ❼
\r\n
<!DOCTYPE html><title>Hello</title>\r\n
```

❶ Код состояния HTTP `200 (OK)` указывает на успешную обработку запроса сервером. Последовательность `<CR><LF>` не видна в выводе, но мы добавили ее в этот фрагмент, чтобы вы не забывали о ее существовании.

❷ Заголовок `Date` содержит метку времени ответа на сервере (источнике).

❸ Заголовок `Server` указывает серверное ПО. Обычно этот заголовок удаляют в целях безопасности.

❹ Заголовок `Last-Modified` содержит метку времени последнего изменения HTML-документа, то есть метку времени изменения файла `/HelloValid.html` на сервере.

❺ Заголовок `Content-Length` содержит длину тела сообщения HTTP в байтах.

❻ Заголовок `Connection` впервые появился в HTTP/1.1. Его наличие в этом ответе — особенность реализации HTTP/1.0 на конкретном ПО сервера[1].

❼ Сервер отправляет MIME-тип `text/html`, указанный в заголовке `Content-Type`.

Остановите `tcpdump`, нажав Ctrl+C, и прочитайте сохраненный захват пакетов с помощью `tshark` (пример 4.7).

Пример 4.7. Чтение файла захвата пакетов с помощью tshark (требует настройки из примера 4.5)

```
CLIENT=http1.0
docker compose exec client bash -c \
       "tshark --read-file tests/client_${CLIENT}.pcap"
```

Вы увидите последовательность, как в примере 4.8. Она похожа на ту, которую мы видели в примере 4.4, за исключением того, что `tshark` теперь распознает и запрос, и ответ как HTTP.

Пример 4.8. Последовательность пакетов клиента HTTP/1.0

```
client → server TCP  [SYN]      Seq=0           Len=0       ❶
server → client TCP  [SYN, ACK] Seq=0   Ack=1   Len=0       ❶
client → server TCP  [ACK]      Seq=1   Ack=1   Len=0       ❶
client → server HTTP GET /HelloValid.html HTTP/1.0\r\n\r\n  ❷
server → client TCP  [ACK]      Seq=1   Ack=34  Len=0       ❸
server → client HTTP HTTP/1.0 200 OK  (text/html)           ❹
client → server TCP  [ACK]      Seq=34  Ack=234 Len=0       ❺
server → client TCP  [FIN, ACK] Seq=234 Ack=34  Len=0       ❻
client → server TCP  [FIN, ACK] Seq=34  Ack=235 Len=0       ❻
server → client TCP  [ACK]      Seq=235 Ack=35  Len=0       ❻
```

❶ Клиент устанавливает с сервером трехэтапное TCP-рукопожатие.

❷ Клиент отправляет HTTP-запрос общим размером 33 байта.

❸ Сервер подтверждает получение HTTP-запроса от клиента (`Ack=34=1+33`).

❹ Сервер успешно отвечает HTML-сообщением с MIME-типом `text/html`.

❺ Клиент подтверждает получение 233 (234 – 1) байт с момента последнего ответа сервера. В это число входят 196 байт кода состояния и заголовков HTTP, а также 37 байт тела HTTP-сообщения.

❻ Клиент инициирует завершение TCP-соединения, и соединение разрывается.

Введение заголовков и стандартизированных кодов состояния в HTTP/1.0 позволяет передавать и получать дополнительную (мета)информацию о со-

[1] Некоторые реализации HTTP/1.0 поддерживают постоянное TCP-соединение через использование заголовка Connection: Keep-Alive и значения из раздела Changes from HTTP/1.0 в RFC 9112 (https://oreil.ly/sBhxp).

общении. Однако поскольку заголовки добавляются в каждый запрос и ответ, то возникают дополнительные расходы. HTTP/1.0 позволяет сжимать тело сообщения, но не его заголовки. Сжатие заголовков стало частью спецификации HTTP/2 (будет описана ниже, в разделе «HTTP/2»).

Использование HTTP в браузере

До сих пор мы рассматривали HTTP с точки зрения клиента, выполняющего один запрос. Этот сценарий отражает изначальную концепцию веба, когда требовалось получить один целый документ. Однако современные сайты состоят из множества ресурсов, таких как каскадные таблицы стилей (CSS) (https://oreil.ly/b2CWw), изображения или JavaScript. Согласно отчету State of the Web (https://oreil.ly/pbjxd), на 2024 год стандартный сайт имел размер более 2 Мбайт, содержал 70 ресурсов и требовал 10 TCP-соединений на страницу. Размер и количество ресурсов влияют на время загрузки сайта: для небольших ресурсов время загрузки ограничивается задержкой их получения по сети, а для крупных — пропускной способностью сети. Кроме того, зависимости между объектами в объектной модели документа HTML (Document Object Model, DOM), объектной модели CSS (CSSOM) и JavaScript влияют на порядок загрузки этих ресурсов, что также сказывается на времени загрузки сайта[1].

При этом человеческое восприятие приемлемого времени отклика остается неизменным: не более 0,1 секунды для ощущения мгновенного ответа и несколько секунд до начала потери внимания[2].

Продолжая аналогию, приведем несколько способов сократить время загрузки отдельного ресурса сайта на сетевом уровне: можно укоротить шланг (уменьшить RTT), увеличить давление воды, использовать несколько шлангов (несколько TCP-соединений) или передавать разные ресурсы по одному шлангу одновременно (мультиплексировать их)[3]. Первые три способа реализованы в HTTP/1.1, а последний — в HTTP/2 и HTTP/3 (см. разделы «HTTP/2» и «HTTP/3» далее в этой главе).

---

[1] См. раздел Primer on Web Performance из книги Ильи Григорика High Performance Browser Networking (https://oreil.ly/utgiA) или более позднюю статью HTTP/3 Prioritization Demystified («HTTP/3: демистификация приоритизации») Робина Маркса (Robin Marx) (https://oreil.ly/SmtMt).

[2] См. статью Response Times: The 3 Important Limits («Время отклика: 3 важных ограничения») Якоба Нильсена (Jakob Nielsen) (https://oreil.ly/jJLwD).

[3] Термин «мультиплексирование» (multiplex) имеет латинское происхождение и буквально означает «много переплетенных частей». В контексте сетевого взаимодействия мультиплексирование означает передачу нескольких сигналов по одному соединению.

ЗАДЕРЖКА И ПРОПУСКНАЯ СПОСОБНОСТЬ

Задержка (latency) — это время от момента отправки сообщения источником до его получения. Связанный показатель, измеряющий время прохождения сообщения от источника к получателю и обратно, — время кругового обращения (round-trip time, RTT). Задержка редко равна половине RTT, поскольку задержки в двух направлениях могут различаться. Обратите внимание, что значения, которые выдают сетевые утилиты, такие как `ping` или `traceroute`, представляют собой RTT, хотя их часто называют задержкой.

Сервис Connection Health Check (https://oreil.ly/DTAa1) позволяет в режиме онлайн измерить RTT от пользовательского компьютера до разных регионов Amazon Web Services (AWS). В 2024 году максимальное RTT до некоторых локаций составляло чуть больше 1 секунды. Важно помнить, что на пути от источника к получателю обычно находится несколько промежуточных узлов и каждое устройство увеличивает время задержки.

Пропускная способность сети (bandwidth) — это максимальный объем данных, передаваемых за единицу времени. Стандартная единица измерения пропускной способности — Мбит/с (англ.: Mbit/s или Mbps; мегабит в секунду = 1 000 000 бит в секунду). Если провести аналогию с садовым шлангом, то задержку определяет длина шланга, а пропускную способность — его диаметр.

Влияние DNS на соединения в браузере

Протокол IP, лежащий в основе стека TCP/IP, работает с IP-адресами. Но когда браузер или другой клиент отправляет HTTP-запрос к сайту, например http://example.com, он обычно указывает имя HTTP-сервера (`example.com`). Клиентами управляют люди, и им проще запоминать имена, а не числовые адреса. Кроме того, ссылка с фиксированным именем более стабильна по сравнению с IP-адресом, который может измениться. Протокол DNS работает на прикладном уровне и обеспечивает преобразование понятного человеку доменного имени сервера в IP-адрес.

Стандартные DNS-запросы отправляются по UDP[1] — транспортному протоколу из стека TCP/IP. В DNS-запросах содержится кортеж из четырех

[1] В DNS-запросах по UDP имя сервера и IP-адрес клиента передаются без шифрования. Со временем появились другие протоколы DNS, позволяющие отправлять запросы через TLS (DoT) или через HTTPS (DoH), где имя сервера скрыто от сторонних наблюдателей. Если используется Oblivious DNS через HTTPS (ODoH), то скрывается также IP-адрес клиента. По сравнению с DNS на основе UDP указанные протоколы увеличивают RTT из-за использования TCP и TLS.

параметров: IP-адреса источника, порта источника (клиента), IP-адреса назначения, порта назначения. В качестве сервера назначения здесь выступает DNS-сервер, он же и предоставляет IP-адрес HTTP-сервера. Важно заметить, что клиент не может одновременно отправить два запроса с одинаковыми значениями параметров к одному DNS-серверу.

Перед установкой TCP-соединения (или соединения по QUIC) клиент должен знать IP-адрес HTTP-сервера. Соответствие доменного имени сервера IP-адресу меняется редко, и оно кэшируется браузером (обычно на 1 минуту) и операционной системой. Тем не менее при отправке HTTP-запроса к серверу, к которому ранее не обращались, при анализе производительности HTTP-клиента нужно учитывать задержку из-за запроса RTT DNS.

Отдельный цикл «DNS-запрос/ответ» можно выполнить с помощью команды **`docker compose exec client bash -c "dig +short example.com"`**. В примере 4.9 показана соответствующая последовательность пакетов, прочитанная с помощью `tshark`. `192.168.114.2` — это IP-адрес клиента, `1.1.1.1` — IP-адрес DNS-сервера (может различаться), а `93.184.215.14` — IP-адрес HTTP-сервера `example.com`.

Пример 4.9. DNS-запрос/ответ

```
192.168.114.2 → 1.1.1.1       DNS-запрос example.com
      1.1.1.1 → 192.168.114.2 ответ на DNS-запрос example.com 93.184.215.14
```

Последовательность «запрос — ответ» содержит только два DNS-сообщения, и каждое инкапсулировано в UDP-датаграмму. Из примера видно, что базовый транспортный протокол UDP предназначен для выполнения транзакций[1], в отличие от TCP, который предназначен для установления соединений. TCP, подобно подписке на онлайн-сервис, устанавливает соединение и отслеживает его состояние, что создает обязательства для обеих сторон до завершения соединения. UDP, напротив, предполагает, что программы обмениваются отдельными единицами данных формата «запрос — ответ», подобно разовым финансовым операциям без заключения долгосрочного контракта. По этой причине UDP часто называют протоколом без установления соединения, где каждая датаграмма рассматривается как независимая сущность, а отправитель и получатель не отслеживают ранее отправленные или полученные датаграммы.

UDP-клиент отправляет запрос серверу получателя. Такая транзакция может быть однонаправленной; пример — благотворительное пожертвование. Другие примеры подобных протоколов, не ожидающих ответа от сервера, — Simple Network Management Protocol (SNMP) и Network Time Protocol (NTP). Транзакция может предполагать обмен информацией. Именно так UDP

[1] См. RFC 768, User Datagram Protocol («Протокол пользовательских датаграмм») (https://oreil.ly/w0uSX). Это один из самых коротких интернет-стандартов.

используется в DNS: клиент отправляет DNS-запрос и ожидает DNS-ответ от сервера. Однако всегда есть риск того, что запрос придет поврежденным (если контрольная сумма UDP не совпадает, то датаграмма будет отброшена), или того, что ответ будет потерян.

Если протокол транспортного уровня не обеспечивает надежность передачи данных, то ответственность ложится на протокол прикладного уровня, например DNS. Если DNS-клиент не получает ответ в течение тайм-аута (1 секунда в Windows и 5 секунд в Unix-подобных системах), то он повторяет запрос. Если механизмы расширения DNS (EDNS) не используются и DNS-ответ приходит усеченным из-за превышения размера данных 512 байт, то DNS-клиент переключается с UDP на TCP и отправляет новый запрос, чтобы получить полное, упорядоченное DNS-сообщение в нескольких TCP-сегментах.

UDP не предоставляет характерных для TCP функций, обеспечивающих надежность передачи данных. Однако в большинстве случаев эти функции и не нужны, поскольку DNS-ответ помещается в одну UDP-датаграмму. Это справедливо и в том случае, если ответ содержит несколько IP-адресов, например, когда HTTP-сервер находится за балансировщиком нагрузки. Преимущество UDP в том, что в нем отсутствует этап рукопожатия TCP и DNS-ответ доставляется клиенту за один RTT[1].

Параллельные TCP-соединения в браузере

Для анализа производительности браузера при загрузке сайта есть специальные инструменты, например Web Vitals (https://oreil.ly/7QOt4). В этом подразделе мы рассмотрим сетевой трафик на уровне протокола, который генерируется браузером при загрузке сайта с использованием HTTP/1.1. На рис. 4.2 показан вид сайта в браузере, а в примере 4.10 приведен его исходный HTML-код. Это простой сайт, содержащий CSS и JavaScript.

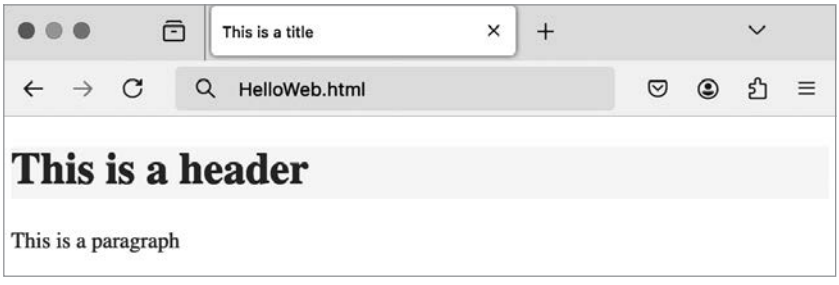


Рис. 4.2. Сайт в Firefox

[1] Из-за отсутствия механизма, аналогичного трехэтапному рукопожатию TCP, DNS-запросы через UDP уязвимы для атак с амплификацией (https://oreil.ly/BinS7).

Пример 4.10. Исходный код HelloWeb.html

```
<!DOCTYPE html>
<html>
<head>
    <title>This is a title</title>
    <link rel="stylesheet" href="background.css">
    <link rel="stylesheet" href="heading.css">
    <link rel="stylesheet" href="paragraph.css">
    <script src="script1.js"></script>
</head>
<body>
    <h1>This is a header</h1>
    <p>This is a paragraph</p>
    <script src="script2.js"></script>
</body>
</html>
```

Чтобы захватить трафик, который генерирует Firefox при загрузке этого сайта, следуйте инструкциям из примера 4.11.

Пример 4.11. Процесс захвата трафика, генерируемого Firefox (требует настройки из примера 4.1)

```
# Терминал ❶
tmux new-session \; split-window -v \;

# Верхняя панель ❷
CLIENT=firefox
PCAP_FILE=/tmp/client_${CLIENT}.pcap
docker compose exec client bash -c \
       "sudo rm --force ${PCAP_FILE} && \
       sudo tcpdump -w ${PCAP_FILE} 'port 80' && \
       cp ${PCAP_FILE} tests"

# Нижняя панель ❸
docker compose exec client bash -c \
       "rm -rf ~/.cache/mozilla/firefox/* && \
       rm -rf ~/.mozilla/firefox/*.profile && \
       firefox --headless --screenshot /tmp/website-in-firefox.png \
       http://http-server/HelloWeb.html && \
       cp /tmp/website-in-firefox.png tests"
```

❶ Создайте две панели с помощью `tmux`.

❷ Запустите новый захват пакетов с помощью `tcpdump` на клиенте. Перехваченный трафик сохранится в файл `client_firefox.pcap` в каталоге `tests`.

❸ Загрузите сайт с помощью `firefox` в режиме headless. Firefox сохранит снимок экрана сайта и завершит работу.

Остановите `tcpdump`, нажав Ctrl+C. В захвате много пакетов, поэтому для удобства чтения лучше открыть его в Wireshark (https://www.wireshark.org).

Содержание захвата может различаться — важно то, что в нем есть несколько TCP-сегментов SYN (в нашем случае три; на рис. 4.3 выделены серым). Это говорит о том, что браузер открывает несколько TCP-соединений с сервером для параллельной загрузки ресурсов.

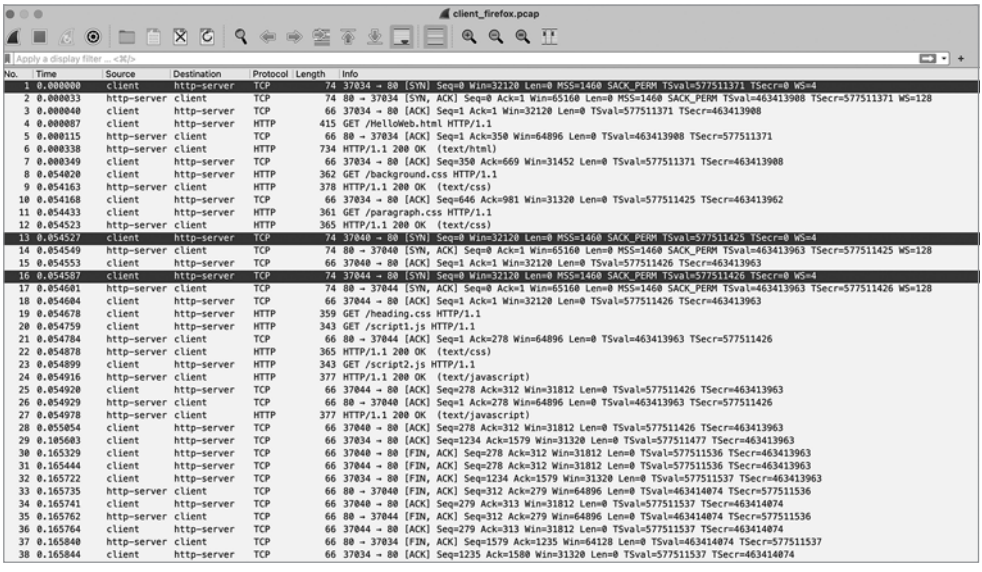


Рис. 4.3. Сетевой трафик, генерируемый Firefox при доступе к сайту

Содержание TCP-потока (см. определение потока во врезке «TCP-соединение и TCP-поток» в главе 3) можно проанализировать в Wireshark: выберите сегмент SYN, щелкните правой кнопкой мыши и выберите Follow ▸ TCP Stream. Вы увидите, что в первом TCP-потоке загрузились HTML и все CSS-файлы, а в оставшихся двух потоках загрузилось по одному файлу JavaScript.

Ограничения TCP

Для работы поверх TCP был выбран исходный протокол HTTP/0.9 (см. раздел «HTTP/0.9» ранее в этой главе). С одной стороны, HTTP стали доступны такие преимущества TCP, как надежность и доставка в порядке отправки, а с другой — на HTTP стали распространяться ограничения TCP.

Некоторые характеристики TCP негативно влияют на производительность HTTP. Среди них блокировка начала очереди, медленный старт и алгоритм предотвращения перегрузок. В современных версиях HTTP/2 и HTTP/3 (см. разделы «HTTP/2» и «HTTP/3» далее в этой главе) вводятся оптимизации для преодоления этих проблем.

Блокировка начала очереди TCP

Функции TCP, обеспечивающие надежность и доставку сообщений в порядке отправки, увеличивают время задержки, поскольку потерянные сегменты требуют повторной передачи[1].

Эту проблему называют *блокировкой начала очереди* (head-of-line, HOL). Термин «начало» заимствован из теории массового обслуживания и указывает на первый сегмент очереди. При блокировке начала очереди потеря сегмента приводит к буферизации последующих сегментов; собрать их в правильном порядке можно только после повторной передачи потерянного сегмента[2].

Зачастую обсуждение блокировки начала очереди ограничивается иллюстративными диаграммами. Чтобы понять, в какой момент возникает блокировка и как она влияет на время задержки, рассмотрим пример на рис. 4.4.

```
$ docker compose exec client bash -c "netcat -l -p 80 && echo"
SEGMENT1DELAYMESEGMENT11SEGMENT111
$
```

```
$ CLIENT=tcp
$ PCAP_FILE=/tmp/client_${CLIENT}.pcap
$ docker compose exec client bash -c \
>       "sudo rm -f ${PCAP_FILE} && \
>       sudo tcpdump --interface lo -w ${PCAP_FILE} 'port 80' && \
>       cp ${PCAP_FILE} tests"
tcpdump: listening on lo, link-type EN10MB (Ethernet), snapshot length 262144 bytes
^C13 packets captured
26 packets received by filter
0 packets dropped by kernel
$
```

```
$ docker compose exec client bash -c \
>       "exec 3<>/dev/tcp/127.0.0.1/80 && \
>       echo -n SEGMENT1 >&3 && \
>       echo -n DELAYME >&3 && \
>       echo -n SEGMENT11 >&3 && \
>       sleep 1 && \
>       echo -n SEGMENT111 >&3 && \
>       exec 3<&- && \
>       exec 3>&- && \
>       sleep 5"
$
```

Рис. 4.4. Блокировка начала очереди TCP

---

[1] «Если низкоуровневые компоненты системы связи пытаются обеспечить идеальную побитовую передачу, это приводит к неконтролируемым задержкам в доставке пакетов. Например, запрос повторной передачи поврежденных пакетов задерживает доставку последующих пакетов». Из статьи: *Saltzer J. H., Reed D. P., Clark D. D.* End-To-End Arguments in System Design («Сквозные аргументы в проектировании систем») // ACM Transactions on Computer Systems (TOCS), 1984. — № 4. — P. 277–288. https://oreil.ly/rfr4h.

[2] См. раздел Head-of-Line Blocking в книге Ильи Григорика High Performance Browser Networking (https://oreil.ly/ocWbl).

На этот раз для захвата пакетов потребуется окно с тремя панелями (пример 4.12).

Пример 4.12. Реализация блокировки начала очереди в TCP (требует настройки из примера 4.1)

```
# Терминал ❶
tmux new-session \; split-window -v \; split-window -v \;

# Верхняя панель ❷
docker compose exec client bash -c "netcat -l -p 80 && echo"

# Центральная панель ❸
CLIENT=tcp
PCAP_FILE=/tmp/client_${CLIENT}.pcap
docker compose exec client bash -c \
       "sudo rm --force ${PCAP_FILE} && \
       sudo tcpdump --interface lo -w ${PCAP_FILE} 'port 80' && \
       cp ${PCAP_FILE} tests"

# Нижняя панель ❹
docker compose exec client bash -c \
       "exec 3<>/dev/tcp/127.0.0.1/80 && \
       echo -n SEGMENT1 >&3 && \
       echo -n DELAYME >&3 && \
       echo -n SEGMENT11 >&3 && \
       sleep 1 && \
       echo -n SEGMENT111 >&3 && \
       exec 3<&- && \
       exec 3>&- && \
       sleep 5"
```

❶ Создайте три панели с помощью `tmux`.

❷ Запустите TCP-сервер с помощью `netcat`. Сервер запускается в контейнере клиента, а обмен данными происходит через интерфейс обратной связи для упрощения настройки. После получения полного TCP-потока от клиента сервер заблокируется и завершит работу.

❸ Запустите новый захват пакетов с помощью `tcpdump` на клиенте. Захваченный трафик сохранится в файл `client_tcp.pcap` в каталоге `tests`.

❹ Используйте функцию `bash`, чтобы открыть TCP-сокет для отправки данных. Отправляются четыре TCP-сегмента: `SEGMENT1`, `DELAYME`, `SEGMENT11` и `SEGMENT111` — с явной задержкой в 1 секунду перед последним сегментом. Сегменты названы так, чтобы их длина в байтах отражалась в захвате пакетов.

   Сегмент `DELAYME` искусственно задерживается на 5 секунд с помощью инструментов `iptables` и `tc`, чтобы вызвать блокировку начала очереди. Способы активации блокировки не обсуждались в этом разделе, но для

выполнения примеров эта настройка необходима. Подробности см. в файле `tests/client.sh` с соответствующими командами. Пятисекундная задержка добавляется после отправки клиентом всех TCP-сегментов, чтобы TCP мог восстановиться после блокировки, вызванной задержкой сегмента `DELAYME`.

Дождитесь завершения работы сервера после получения полного TCP-потока от клиента, остановите `tcpdump`, нажав Ctrl+C, и прочитайте сохраненный захват пакетов с помощью `tshark` (пример 4.13).

Пример 4.13. Чтение файла захвата пакетов с помощью tshark (требует настройки из примера 4.12)

```
CLIENT=tcp
docker compose exec client bash -c \
       "tshark --read-file tests/client_${CLIENT}.pcap"
```

Вы увидите последовательность как в примере 4.14. Ее обсуждение требует понимания деталей TCP, поэтому прежде, чем вы продолжите чтение, рекомендуем пересмотреть пример 3.5 из главы 3.



Строки SEGMENT1, DELAYME, SEGMENT11 и SEGMENT111 содержатся в данных TCP-сегментов. Их можно найти, анализируя захват пакетов в Wireshark, однако в базовой сводке последовательности данные сегментов не отображаются — там указывается только длина данных (Len).

Пример 4.14. Последовательность пакетов при блокировке начала очереди в TCP

```
0.0000 client → server                    [SYN]          Seq=0        Len=0  ❶
0.0000 server → client                    [SYN, ACK]     Seq=0  Ack=1  Len=0  ❶
0.0000 client → server                    [ACK]          Seq=1  Ack=1  Len=0  ❶
0.0002 client → server                    [PSH, ACK]     Seq=1  Ack=1  Len=8  ❷
0.0002 server → client                    [ACK]          Seq=1  Ack=9  Len=0  ❸
0.2173 client → server [TCP Previous ...] [PSH, ACK]     Seq=16 Ack=1  Len=9  ❹
0.2174 server → client [TCP Dup ACK 5#1]  [ACK]          Seq=1  Ack=9  Len=0  ❺
1.0054 client → server                    [FIN, PSH, ACK] Seq=25 Ack=10 Len=8  ❻
1.0054 server → client [TCP Dup ACK 5#2]  [ACK]          Seq=1  Ack=9  Len=0  ❼
5.0029 client → server [TCP Retransmission] [PSH, ACK]   Seq=9  Ack=1  Len=7  ❽
5.0032 server → client                    [ACK]          Seq=1  Ack=36 Len=0  ❾
5.0089 server → client                    [FIN, ACK]     Seq=1  Ack=36 Len=0
5.0090 client → server                    [ACK]          Seq=36 Ack=2  Len=0  ❿
```

❶ Устанавливается трехстороннее рукопожатие TCP между клиентом и сервером. В реальном захвате пакетов и клиент, и сервер используют loopback-адреса `127.0.0.1`, поэтому их можно идентифицировать только по портам. Поскольку обмен трафиком происходит через loopback, то задержка составляет доли миллисекунды.

❷ Клиент отправляет серверу SEGMENT1 (Len=8).

❸ Сервер подтверждает (ACK) получение SEGMENT1.

❹ Клиент отправляет серверу SEGMENT11 (Len=9) после отправки сегментов SEGMENT1 (Len=8) и DELAYME (Len=7), что отражено в Seq=1+8+7=16. Метка [TCP Previous segment not captured] не является частью TCP-сегмента — ее добавляет tshark.

❺ Увидев Seq=16 в сегменте от клиента, сервер обнаруживает потерю сегмента и повторно подтверждает (ACK) последний полностью полученный SEGMENT1, чтобы сообщить клиенту о проблеме. tshark помечает эту проблему как [TCP Dup ACK 5#1] — дублирующее подтверждение. Кроме того, ACK содержит диапазон избирательного подтверждения 16–25 (SACK, не показан), который указывает, что сервер также получил SEGMENT11. TCP SACK — это механизм обработки потери пакетов, который позволяет получателю информировать отправителя обо всех успешно доставленных сегментах, чтобы отправитель повторно передавал только отсутствующие[1].

❻ Клиент находится в состоянии Seq=1+8+7+9, что означает отправку сегментов SEGMENT1, DELAYME и SEGMENT11. Клиент отправляет SEGMENT111 (Len=10) и, поскольку это последний сегмент, устанавливает флаг FIN для завершения TCP-соединения. На стороне клиента есть ожидаемая задержка в 1 секунду.

❼ Сервер повторно сообщает, что SEGMENT1 — последний полученный сегмент перед потерей. tshark помечает эту проблему как [TCP Dup ACK 5#2]. Вдобавок это подтверждение (ACK) содержит диапазон избирательного подтверждения 16–36 (снова не показан), который указывает, что сервер также получил SEGMENT111 и сегмент с флагом FIN.

❽ После истечения искусственно введенной задержки 5 секунд клиент повторно передает сегмент DELAYME длиной 7 байт (Len=7) с Seq=1+8=9 (после отправки SEGMENT1). tshark помечает это событие как [TCP Retransmission].

❾ Сервер подтверждает прием всех сегментов, отправляя Ack=1+8+7+9+10+1=36. Последнее +1 — это добавленный фантомный байт (см. пример 3.5 в главе 3).

❿ TCP-соединение завершено.

В этом примере блокировка начала очереди возникает из-за искусственной задержки в TCP-потоке. Сервер (получатель) воспринимает задержку как потерю пакета, но продолжает принимать последующие сегменты от

---

[1] Демонстрацию работы TCP SACK можно найти в видео Криса Грира (Chris Greer) How TCP Works — Selective Acknowledgment (SACK) («Как работает TCP — выборочное подтверждение») (https://oreil.ly/dX768).

клиента (отправителя), буферизуя их и напоминая клиенту о потере. Только после получения повторно переданного сегмента сервер может собрать все сегменты и завершить TCP-соединение. В *петлевой* сети с задержкой менее миллисекунды прогнозируемая задержка всей последовательности составляет около 1 секунды из-за явной задержки в 1 секунду на стороне клиента (приложения). Однако из-за потери пакета фактическая задержка составляет около 5 секунд.

Медленный старт и алгоритм предотвращения перегрузок

В TCP реализовано управление перегрузкой для достижения максимальной пропускной способности в заданных сетевых условиях, а также для рационального распределения сетевых ресурсов между компьютерами, использующими общую сеть. Основные механизмы управления перегрузкой включают медленный старт и предотвращение перегрузок[1].



Более подробно изучить медленный старт и предотвращение перегрузок можно с помощью примера в файле src/http/README.md в репозитории кода книги.

При медленном старте TCP отправитель исследует сеть, постепенно увеличивая количество сегментов в пределах одного RTT после подтверждения их получателем. Увеличение обычно мультипликативное: отправитель может начать, например, с двух сегментов, после получения подтверждений отправить четыре и т. д.[2] При обнаружении потери пакета отправитель снижает интенсивность, уменьшая количество сегментов. Обе стороны TCP-соединения управляют перегрузкой независимо.

Таким образом, TCP позволяет быстро наращивать объем данных, отправляемых за единицу времени. После достижения *порога медленного старта* (slow start threshold, ssthresh) алгоритм управления перегрузкой переходит в менее агрессивную фазу *предотвращения перегрузок*. Фаза медленного старта, несмотря на название, увеличивает пропускную способность быстрее, чем фаза предотвращения перегрузок. Кроме того, управление перегрузкой требует времени, поэтому оно неэффективно для кратковременных TCP-соединений, например, когда браузер запрашивает небольшой ресурс.

---

[1] Подробнее см. в разделе Congestion Avoidance and Control книги Ильи Григорика High Performance Browser Networking (https://oreil.ly/6fooP).

[2] В проекте RFC 2013 года Increasing TCP's Initial Window («Увеличение начального окна TCP») (https://oreil.ly/l9BeZ) рекомендовалось увеличить начальное количество сегментов с 2–4 до 10.

HTTP/1.1

Протокол HTTP продолжал развиваться, и в 1997 году, менее чем через год после выпуска HTTP/1.0, появилась версия HTTP/1.1[1]. За ней последовал проект стандарта RFC в 1999 году[2].

Спецификация HTTP/1.1 1997 и 1999 годов не выходила за пределы возможностей транспорта TCP, однако предпринимались попытки повысить скорость загрузки гипертекстовых ресурсов[3] и решить другие проблемы, обнаруженные в 1990-х годах (например, истощение пула адресов IPv4)[4].

В HTTP/1.1 появились следующие возможности.

Пофрагментное кодирование передачи

Тело ответа делится на фрагменты, постепенно генерируемые сервером. Для их отправки используется одно TCP-соединение. Функция полезна, если клиент принимает промежуточные результаты, поступающие в потоковом режиме.

Постоянные TCP-соединения по умолчанию

Сервер больше не обязан закрывать TCP-соединение после отправки всего документа клиенту. Эта функция снижает задержку при получении последующих ресурсов: поскольку трехэтапное рукопожатие TCP выполняется только в первый раз, то время последующих взаимодействий сокращается на один RTT. Возвращаясь к аналогии с садом: если мы поливаем его регулярно, то нет необходимости убирать шланг.

Кроме того, постоянные соединения повышают пропускную способность, поскольку долгоживущие TCP-соединения эффективнее используют алгоритмы управления перегрузкой для оптимизации передачи данных в текущих сетевых условиях.

---

[1] См. RFC 2068, Hypertext Transfer Protocol — HTTP/1.1 («Протокол передачи гипертекста — HTTP/1.1») (https://oreil.ly/LQANn).

[2] Hypertext Transfer Protocol — HTTP/1.1 («Протокол передачи гипертекста — HTTP/1.1») (https://oreil.ly/KIZId) и некоторые другие RFC появились позже. Однако первый интернет-стандарт HTTP/1.1 (https://oreil.ly/UGtuU) был опубликован только в 2022 году.

[3] Подробный анализ производительности версий HTTP 1 и 2 в браузерах см. в книге: *Pollard B.* HTTP/2 in Action. — Manning, 2019. (*Поллард Б.* HTTP/2 в действии. — М., 2021.) В книге также представлен анализ производительности загрузки реальных сайтов с помощью WebPageTest (https://www.webpagetest.org).

[4] См. статью IPv4 Address Exhaustion («Исчерпание адресов IPv4») в Википедии (https://oreil.ly/vodvR).

Конвейеризация HTTP-запросов

Появляется возможность отправлять несколько запросов без ожидания ответа сервера. Сервер обязан отправлять ответы в том же порядке, в котором получил запросы. Эта функция не получила широкого распространения из-за блокировки начала очереди, а также из-за несовместимости ее реализаций в клиентах, серверах и промежуточных узлах.

Добавление и удаление HTTP-методов

Добавлены новые методы запросов: `OPTIONS`, `TRACE` и `CONNECT`. Методы, которые не получили поддержки в реализациях, такие как `PATCH`, `LINK` и `UNLINK`, были удалены[1].

Добавление дополнительных HTTP-кодов состояния

Добавлены новые HTTP-коды состояния, в том числе `101 (Switching Protocols)`, который до сих пор используется для протокола WebSocket (см. главу 10).

Согласование содержимого

Клиент сообщает серверу о своих предпочтениях с помощью заголовков запроса, таких как `Accept` или `User-Agent`.

Добавление спецификации кэширования HTTP

Кэширование позволило сократить количество отправляемых запросов и ответов[2].

Обязательное добавление заголовка Host в запросах клиента

Это требование ввели для поддержки виртуального хостинга[3] HTTP-серверов. Виртуальный хостинг позволяет обслуживать контент с нескольких доменов (например, `server1.example.com` и `server2.example.org`)

---

[1] Впоследствии был выпущен отдельный RFC, восстановивший метод PATCH для HTTP (https://oreil.ly/QZm9f).

[2] «Интересное наблюдение о сетевых приложениях: приложение демонстрирует лучшую производительность, если сеть не используется» (Рой Томас Филдинг) (Roy Thomas Fielding) (https://oreil.ly/sSySy).

[3] «Одной из главных ошибок в раннем дизайне HTTP стало решение отправлять не полный URI, являющийся целью запроса, а только те части, которые не использовались при установке соединения. Предполагалось, что сервер узнает собственный стандарт именования на основе IP-адреса и TCP-порта соединения. Однако не учитывалась возможность существования нескольких стандартов именования на одном сервере. Это стало критической проблемой по мере экспоненциального роста веба, когда количество новых доменных имен (основа для стандарта именования в пространстве имен http URL) намного превысило количество доступных IP-адресов» (из диссертации Роя Филдинга) (https://oreil.ly/CiIDl).

с одного IP-адреса. Клиент выбирает виртуальный сервер, отправляя нужный домен в заголовке запроса — `Host: server1.example.com`. Функция позволила сократить использование IP-адресов.

Рекомендация клиентам использовать несколько TCP-соединений с сервером

В RFC рекомендовалось использовать не более двух соединений, но оптимальным количеством для браузеров стало шесть соединений на домен. В контексте аналогии со шлангами это значит, что лучше использовать несколько более узких шлангов вместо одного широкого при равной общей площади поперечного сечения. Общая пропускная способность[1] остается той же, но несколько TCP-потоков позволяют ускорить загрузку некоторых ресурсов.

Постоянство HTTP-соединения

Наиболее важным нововведением в HTTP/1.1 стало постоянное TCP-соединение. Рассмотрим его реализацию в примере 4.15 и одновременно обсудим конвейеризацию HTTP, поскольку постоянное соединение — ее обязательное условие. В процессе вы увидите в заголовках и другие возможности HTTP/1.1. Для захвата пакетов потребуется окно с двумя панелями.

Пример 4.15. Реализация постоянного TCP-соединения в HTTP/1.1 (требует настройки из примера 4.1)

```
# Терминал  ❶
tmux new-session \; split-window -v \;

# Верхняя панель  ❷
CLIENT=http1.1
PCAP_FILE=/tmp/client_${CLIENT}.pcap
docker compose exec client bash -c \
       "sudo rm --force ${PCAP_FILE} && \
       sudo tcpdump -w ${PCAP_FILE} 'port 80' && \
       cp ${PCAP_FILE} tests"

# Нижняя панель  ❸
docker compose exec client bash -c \
       'GET="GET / HTTP/1.1\r\nHost:host\r\n" && echo -en \
       "${GET}Connection:keep-alive\r\n\r\n${GET}Connection:close\r\n\r\n" | \
       netcat -p 8080 http-server 80'
```

❶ Создайте две панели с помощью `tmux`.

❷ Запустите новый захват пакетов с помощью `tcpdump` на клиенте. Захваченный трафик сохранится в файл `client_http1.1.pcap` в каталоге `tests`.

[1] Обычно трафик со всех соединений проходит через одну сетевую карту устройства.

❸ Отправьте два последовательных конвейерных запроса `GET /` на сервер с помощью `netcat`. В первом запросе явно используется `Connection:keep-alive`, несмотря на то что для HTTP/1.1 это поведение по умолчанию (удалите заголовок, чтобы убедиться в этом). Во втором запросе используется `Connection:close`, поэтому сервер закроет соединение после отправки всего документа, что является поведением по умолчанию для версий, предшествующих HTTP 1.1. Значение `host` в `Host:host` в примере произвольное, так как на сервере настроен виртуальный хост по умолчанию, но этот заголовок должен быть в запросе HTTP/1.1. Удалите заголовок, чтобы проверить, соответствует ли сервер спецификации.

Ожидается, что клиент дважды выведет стандартный ответ `http-server`, содержащий HTML-документ `It works!`, и завершит работу. Вывод показан в примере 4.16.

Пример 4.16. Ответ сервера HTTP/1.1

```
HTTP/1.1 200 OK                                    ❶
Date: Tue, 23 Jul 2024 16:47:14 GMT
Server: Apache/2.4.61 (Unix)
Last-Modified: Mon, 11 Jun 2007 18:53:14 GMT
ETag: "2d-432a5e4a73a80"                           ❷
Accept-Ranges: bytes                               ❸
Content-Length: 45
Keep-Alive: timeout=5, max=100                     ❹
Connection: Keep-Alive                             ❺
Content-Type: text/html

<html><body><h1>It works!</h1></body></html>
HTTP/1.1 200 OK
Date: Tue, 23 Jul 2024 16:47:15 GMT
Server: Apache/2.4.61 (Unix)
Last-Modified: Mon, 11 Jun 2007 18:53:14 GMT
ETag: "2d-432a5e4a73a80"                           ❻
Accept-Ranges: bytes
Content-Length: 45
Connection: close                                  ❼
Content-Type: text/html

<html><body><h1>It works!</h1></body></html>
```

❶ Код состояния HTTP `200 (OK)` в ответе сервера указывает на успешное выполнение запроса клиента. Не забывайте, что каждая строка ответа заканчивается невидимой последовательностью `<CR><LF>`.

❷ Заголовок `ETag` назван по аналогии с тегом сущности (entity tag). Сущность отсылает к ресурсу документа, в данном случае — к файлу `index.html` 2007 года, который сервер предоставляет по умолчанию (здесь сервер Apache

возвращает устаревшее содержимое!). `ETag` используется для поддержки кэширования и зависит от сведений о ресурсе, например от размера HTML-файла или даты его изменения.

❸ Заголовок `Accept-Ranges` указывает на то, что сервер поддерживает запросы частичного содержимого. Это значит, что клиент может запросить часть файла или повторно загрузить файл.

❹ Заголовок `Keep-Alive` сообщает клиенту о лимитах продолжительности постоянного соединения на сервере. В данном случае сервер будет ждать следующий запрос от клиента на том же соединении в течение 5 секунд и примет максимум 100 запросов в одном соединении.

❺ Значение `Keep-Alive` в заголовке `Connection` указывает на то, что сервер должен сохранить соединение открытым после отправки всего документа клиенту.

❻ Значение заголовка `ETag` не изменилось. Это значит, что с момента предыдущего запроса клиента в файл `index.html` не вносились изменения.

❼ Значение `close` в заголовке `Connection` указывает, что сервер должен закрыть соединение после отправки всего документа клиенту, то есть оно соответствует поведению HTTP до версии 1.1.

Остановите `tcpdump`, нажав Ctrl+C. Далее прочитайте сохраненный файл захвата пакетов с помощью `tshark` (пример 4.17), чтобы проверить, сохраняется ли TCP-соединение.

Пример 4.17. Чтение файла захвата пакетов с помощью tshark (требуется настройка из примера 4.15)

```
CLIENT=http1.1
docker compose exec client bash -c \
       "tshark --read-file tests/client_${CLIENT}.pcap"
```

Вы увидите последовательность, показанную в примере 4.18.

Пример 4.18. Последовательность пакетов клиента HTTP/1.1

```
client → server TCP  [SYN]      Seq=0          Len=0   ❶
server → client TCP  [SYN, ACK] Seq=0   Ack=1  Len=0   ❶
client → server TCP  [ACK]      Seq=1   Ack=1  Len=0   ❶
client → server HTTP GET / HTTP/1.1 GET / HTTP/1.1     ❷
server → client TCP  [ACK]      Seq=1   Ack=100 Len=0  ❸
server → client HTTP HTTP/1.1 200 OK (text/html)       ❹
client → server TCP  [ACK]      Seq=100 Ack=327 Len=0
server → client HTTP HTTP/1.1 200 OK (text/html)       ❺
client → server TCP  [ACK]      Seq=100 Ack=616 Len=0
server → client TCP  [FIN, ACK] Seq=616 Ack=100 Len=0  ❻
client → server TCP  [FIN, ACK] Seq=100 Ack=617 Len=0  ❻
server → client TCP  [ACK]      Seq=617 Ack=101 Len=0  ❻
```

❶ Клиент устанавливает с сервером трехстороннее рукопожатие TCP.

❷ Клиент отправляет два конвейерных HTTP-запроса для получения документа по умолчанию / с сервера.

❸ Сервер подтверждает получение HTTP-запросов от клиента.

❹ Сервер отправляет клиенту первый HTTP-ответ, оставляя TCP-соединение открытым.

❺ Сервер отправляет клиенту второй HTTP-ответ.

❻ После отправки всего HTML-документа сервер инициирует завершение TCP-соединения, и соединение разрывается.

Представленная последовательность пакетов подтверждает, что постоянное TCP-соединение в HTTP/1.1 сокращает время задержки для последующих HTTP-запросов за счет отсутствия RTT трехстороннего рукопожатия.

HTTP/2

В новой версии HTTP/2 изменился формат сообщений[1]. В 2009 году Google представил протокол SPDY[2], который впоследствии лег в основу HTTP/2[3]. Главной целью HTTP/2 было увеличение скорости загрузки сайтов по сравнению с HTTP/1.1. Протокол проектировался на основе экспериментальных данных реального веб-трафика[4], поскольку Google контролировал и клиентов (браузер Chrome, выпущенный в 2008 году), и серверную инфраструктуру. Основные возможности протокола HTTP/2 представлены ниже[5].

- HTTP-сообщения разделяются на двоично закодированные фреймы, где каждый тип фрейма выполняет определенную функцию в протоколе — например, `DATA` (тело HTTP-сообщения), `HEADERS` (заголовки

---

[1] См. RFC 2145, Use and Interpretation of HTTP Version Numbers («Использование и интерпретация номеров версий HTTP») (https://oreil.ly/pr01D).

[2] См. статью A 2x Faster Web («Веб в два раза быстрее») в блоге Chromium за 2009 год (https://oreil.ly/CEF6z).

[3] См. RFC 7540, Hypertext Transfer Protocol Version 2 (HTTP/2) («Протокол передачи гипертекста версии 2 (HTTP/2)») от 2015 года (https://oreil.ly/36n4d).

[4] Развертывание HTTP/2 и HTTP/3 в интернете не гарантирует ускорение загрузки каждого сайта. Чтобы использовать возможности новых протоколов, сайт нужно адаптировать к ним (подобно тому как многие сайты оптимизировали под HTTP/1).

[5] Редко используемые возможности HTTP/2, такие как приоритизация ресурсов или server push (уже удаленный из Chrome), здесь не рассматриваются. Подробное описание HTTP/2 можно найти в книге Барри Полларда (Barry Pollard) HTTP/2 in Action (Manning, 2019).

HTTP-сообщения), `SETTINGS` (настройки, связанные с HTTP/2) или `GOAWAY` (намерение не начинать новые потоки или сигнал о серьезных ошибках). Благодаря включению длины сообщения в формат фрейма[1] компьютерам проще читать фреймы HTTP/2. Длина и тип фрейма считываются из его начала, после чего данные читаются и интерпретируются в соответствии с типом фрейма. В HTTP/1 границы сообщений отмечают с помощью `<CR><LF>`, а длину тела сообщения определяют по заголовку `Content-Length`[2].

- Заголовки HTTP-сообщений сжимаются. Поскольку в HTTP сервер не сохраняет предыдущие состояния (за исключением cookie-файлов), заголовки повторно отправляются с каждым HTTP-запросом/ответом. Если в HTTP/1.0 доступно только сжатие тела сообщения, то в HTTP/2 сжимаются и заголовки в специальном формате HPACK[3]. За счет этого в запросах размер заголовков уменьшается относительно тела сообщения. Аналогичным образом сокращаются HTTP-ответы[4].
- HTTP/2 мультиплексирует HTTP-сообщения, как показано на рис. 4.6. В качестве транспортного протокола по-прежнему используется TCP, и TCP-сегмент может содержать несколько фреймов HTTP/2.

  В HTTP/2 вводится понятие потока (отличающееся от TCP-потока, см. врезку «TCP-соединение и TCP-поток» в главе 3). Поток определяется как «независимая двунаправленная последовательность фреймов, которыми обмениваются клиент и сервер в рамках одного HTTP/2-соединения». Иными словами, один TCP-сегмент может содержать фреймы HTTP/2, принадлежащие к разным потокам ресурсов документа (например, к разным CSS-файлам).

  Кроме того, TCP-сегмент может нести один фрейм `DATA`, принадлежащий конкретному потоку. Такие сегменты могут быть в TCP-потоке циклически (по принципу round-robin). Клиент также может изменять приоритетность потоков, отправляя фрейм `PRIORITY`. Эти возможности HTTP/2 позволяют эффективно реализовать мультиплексирование.

---

[1] См. раздел Frame format («Формат фрейма») в спецификации HTTP/2 (https://oreil.ly/axiwI).

[2] См. главу The HTTP/2 Protocol в книге Learning HTTP/2 Стивена Людина (Stephen Ludin) и Хавьера Гарса (Javier Garza) (https://learning.oreilly.com/library/view/learning-http-2/9781491962435).

[3] Формат HPACK (https://oreil.ly/K9ija) создали, чтобы снизить уязвимость сжатых заголовков к атакам, подобным CRIME (Compression Ratio Info-leak Made Easy).

[4] См. пост HPACK: The Silent Killer (Feature) of HTTP/2 («HPACK: “Тихий убийца” (функция) HTTP/2») (https://oreil.ly/6qZaR) в блоге Влада Краснова (Vlad Krasnov) на Cloudflare.

> Тем не менее HTTP/2 не решает проблему блокировки начала очереди (см. подраздел «Блокировка начала очереди TCP» выше в этой главе), поскольку используется одно TCP-соединение[1]. Мультиплексирование на уровне приложений, реализованное в HTTP/2, позволяет снизить задержку при получении нескольких HTTP-ресурсов по сети без потерь пакетов. В случае потерь TCP-сегмент, несущий один поток HTTP/2, блокирует сборку оставшихся сегментов в TCP-потоке, а значит, и другие потоки HTTP/2[2].

Хотя сообщения, передаваемые по HTTP/2, схожи по формату с HTTP, кодирование в виде двоичных фреймов усложняет развертывание протокола в интернете. Эволюции сетевых протоколов препятствует наличие промежуточных устройств[3], ожидающих определенный формат пакетов (это явление получило название «окостенение протоколов»).

Некоторые промежуточные устройства блокируют нераспознанный трафик или изменяют метаданные (заголовки HTTP или TCP), нарушая работу новой версии протокола. На внедрение новой версии промежуточными устройствами и операционными системами могут уйти годы. Чтобы решить эту проблему, можно скрыть структуру пакетов от промежуточных устройств с помощью шифрования или реализовать протоколы вне ядра операционной системы, в пользовательском пространстве (см. раздел «HTTP/3» далее в этой главе). Спецификация HTTP/2 не требует шифрования, но браузеры поддерживают только зашифрованную версию, называемую `h2`. Другие типы клиентов также поддерживают незашифрованную версию — `h2c`.

Фреймы и потоки в HTTP/2

Основные особенности HTTP/2, отличающие его от HTTP/1.1, включают использование двоичных фреймов, сжатие заголовков и мультиплексирование с использованием потоков. При изучении HTTP/2 рекомендуем использовать клиент `nghttp2` (https://oreil.ly/vhHNM), который отображает сообщения

---

[1] Подробнее о блокировке начала очереди в HTTP/2 и примеры см. в разделе HOL blocking in HTTP/2 over TCP («Блокировка HOL в HTTP/2 поверх TCP») в статье Робина Маркса (Robin Marx) (https://oreil.ly/Jghqj).

[2] Поскольку HTTP/2 не решает проблему блокировки начала очереди в TCP, то возник вопрос, можно ли ускорить HTTP, внеся минимальные изменения в протокол, добавив дополнительные заголовки в HTTP/1.1 (https://oreil.ly/jx5J9).

[3] Промежуточное устройство — это любое устройство-посредник, выполняющее функции, отличающиеся от функций IP-маршрутизатора; см. RFC 3234, Middleboxes: Taxonomy and Issues («Промежуточные устройства (Middleboxes): классификация и проблемы») (https://oreil.ly/umnRJ). Примеры промежуточных устройств: NAT, межсетевые экраны, балансировщики нагрузки, системы обнаружения и предотвращения вторжений, а также кэши CDN.

в формате, удобном для чтения человеком. В примере 4.19 показано выполнение HTTP/2-запроса с использованием `nghttp2`.

Пример 4.19. Выполнение запроса в HTTP/2 (требует настройки из примера 4.1)

```
# Терминал  ❶
tmux new-session \; split-window -v \;

# Верхняя панель  ❷
CLIENT=h2c
PCAP_FILE=/tmp/client_${CLIENT}.pcap
docker compose exec client bash -c \
       "sudo rm --force ${PCAP_FILE} && \
       sudo tcpdump -w ${PCAP_FILE} 'port 80' && \
       cp ${PCAP_FILE} tests"

# Нижняя панель  ❸
docker compose exec client bash -c \
       "nghttp --verbose --timeout 3 --no-dep --hexdump \
       http://https-server:80"
```

❶ Создайте две панели с помощью `tmux`.

❷ Запустите новый захват пакетов с помощью `tcpdump` на клиенте. Захваченный трафик сохранится в файл `client_h2c.pcap` в каталоге `tests`.

❸ Отправьте HTTP/2-запрос на сервер с помощью клиента `nghttp2`. Клиент завершит работу после успешного получения ответа от сервера. Обратите внимание, что целевой сервер называется `https-server`, но, помимо обслуживания `h2` на порте `443` (стандартный HTTPS-порт), он настроен на поддержку незашифрованного `h2c` на порте `80`. Для упрощения примера в `nghttp2` используется аргумент `--no-dep`, чтобы клиент не отправлял серверу информацию о приоритетных ресурсах.

При выполнении примера вы получите вывод как в примере 4.20, но с цветовой разметкой.

Пример 4.20. Ответ сервера h2c на запрос nghttp2

```
[  0.001] Connected
[  0.001] send SETTINGS frame <length=12, flags=0x00, stream_id=0>          ❶
          (niv=2)
          [SETTINGS_MAX_CONCURRENT_STREAMS(0x03):100]
          [SETTINGS_INITIAL_WINDOW_SIZE(0x04):65535]
[  0.001] send HEADERS frame <length=32, flags=0x05, stream_id=1>           ❷
          ; END_STREAM | END_HEADERS
          (padlen=0)
          ; Open new stream
          :method: GET
          :path: /
```

```
          :scheme: http
          :authority: https-server
          accept: */*
          accept-encoding: gzip, deflate
          user-agent: nghttp2/1.62.1
[  0.005] recv SETTINGS frame <length=18, flags=0x00, stream_id=0>          ❸
          (niv=3)
          [SETTINGS_MAX_CONCURRENT_STREAMS(0x03):128]
          [SETTINGS_INITIAL_WINDOW_SIZE(0x04):65536]
          [SETTINGS_MAX_FRAME_SIZE(0x05):16777215]
[  0.005] recv WINDOW_UPDATE frame <length=4, flags=0x00, stream_id=0>
          (window_size_increment=2147418112)
[  0.005] recv SETTINGS frame <length=0, flags=0x01, stream_id=0>
          ; ACK
          (niv=0)
[  0.005] recv (stream_id=1) :status: 200                                   ❹
[  0.005] recv (stream_id=1) server: nginx/1.27.0
[  0.005] recv (stream_id=1) date: Sat, 27 Jul 2024 19:21:01 GMT
[  0.005] recv (stream_id=1) content-type: text/html
[  0.005] recv (stream_id=1) content-length: 55
[  0.005] recv (stream_id=1) last-modified: Sat, 27 Jul 2024 19:17:56 GMT
[  0.005] recv (stream_id=1) etag: "66a547e4-37"
[  0.005] recv (stream_id=1) accept-ranges: bytes
[  0.005] recv HEADERS frame <length=106, flags=0x04, stream_id=1>
          ; END_HEADERS
          (padlen=0)
          ; First response header
[  0.005] send SETTINGS frame <length=0, flags=0x01, stream_id=0>
          ; ACK
          (niv=0)
<!DOCTYPE html><title>Welcome to nginx!</title></html>                      ❺
[  0.008] recv DATA frame <length=55, flags=0x01, stream_id=1>
          ; END_STREAM
[  0.008] send GOAWAY frame <length=8, flags=0x00, stream_id=0>             ❻
          (last_stream_id=0, error_code=NO_ERROR(0x00), opaque_data(0)=[])
```

❶ После подключения к серверу клиент отправляет фрейм `SETTINGS` с настройками протокола HTTP/2: количеством параллельных потоков и начальным размером окна, который определяет объем данных, отправляемых в потоке. Передача настроек всегда выполняется через поток `0`.

❷ Клиент отправляет фрейм `HEADERS`, содержащий HTTP-заголовки, в потоке `1`.

❸ Сервер отправляет несколько фреймов `SETTINGS` и `WINDOW_UPDATE`.

❹ Сервер отправляет заголовки HTTP-сообщения. Обратите внимание, что из-за особенностей реализации `nghttp2` (https://oreil.ly/FCJM1) фактические значения заголовков появляются в выводе перед фреймом `HEADERS`, который отображается несколькими строками ниже.

❺ Сервер отправляет тело HTTP-сообщения. Фрейм `DATA` с данными также появляется в выводе после самих данных.

❻ Клиент отправляет серверу фрейм `GOAWAY`, обозначая намерение не запускать новые потоки.

Остановите `tcpdump`, нажав Ctrl+C, и прочитайте сохраненный захват пакетов с помощью `tshark` (пример 4.21).

Пример 4.21. Чтение файла захвата пакетов с помощью tshark (требуется настройка из примера 4.19)

```
CLIENT=h2c
docker compose exec client bash -c \
       "tshark --read-file tests/client_${CLIENT}.pcap"
```

Вы увидите такую же последовательность, как в примере 4.22. Она показывает связь HTTP/2 с TCP, включая передачу нескольких фреймов HTTP/2 в одном TCP-сегменте.

Пример 4.22. Последовательность пакетов клиента h2c

```
client → server TCP   [SYN]      Seq=0           Len=0       ❶
server → client TCP   [SYN, ACK] Seq=0   Ack=1   Len=0       ❶
client → server TCP   [ACK]      Seq=1   Ack=1   Len=0       ❶
client → server HTTP2 Magic, SETTINGS[0], HEADERS[1]: GET /  ❷
server → client TCP   [ACK]      Seq=1   Ack=87  Len=0
server → client HTTP2 SETTINGS[0], ..., HEADERS[1]: 200 OK   ❸
client → server TCP   [ACK]      Seq=87  Ack=165 Len=0
client → server HTTP2 SETTINGS[0]
server → client HTTP2 DATA[1] (text/html)                    ❹
client → server HTTP2 GOAWAY[0]                              ❺
client → server TCP   [FIN, ACK] Seq=113 Ack=229 Len=0       ❻
server → client TCP   [FIN, ACK] Seq=229 Ack=114 Len=0       ❻
client → server TCP   [ACK]      Seq=114 Ack=230 Len=0       ❻
```

❶ Устанавливается трехстороннее TCP-рукопожатие между клиентом и сервером. Напомним, что в этом примере сервер принимает незашифрованный трафик `h2c`.

❷ Клиент отправляет сообщение-триггер[1], указывающее на использование протокола HTTP/2. Далее отправляются фреймы `SETTINGS` и `HEADERS`.

❸ Сервер отвечает фреймами `SETTINGS`, `HEADERS` и некоторыми другими.

❹ Сервер отправляет тело HTTP-сообщения клиенту во фрейме `DATA` по потоку `1`.

---

[1] Сообщение-триггер выглядит так: PRI * HTTP/2.0\r\n\r\nSM\r\n\r\n. Возможно, оно связано с секретной программой PRISM (https://oreil.ly/cENO7), о которой в 2013 году рассказал Эдвард Сноуден (Edward Snowden).

❺ Клиент отправляет серверу фрейм GOAWAY, обозначая намерение не запускать новые потоки.

❻ Клиент инициирует завершение TCP-соединения, и соединение разрывается.

HTTP/3

С начала 2010-х годов получили распространение мобильные устройства: смартфоны (iPhone представили в 2007 году) и планшеты. Поскольку производительность их сетей была ниже, то главным приоритетом HTTP оставалось сокращение задержки при получении доступа к веб-ресурсам. Кроме того, внимание уделялось множественной адресации (использование устройством нескольких сетевых интерфейсов или IP-адресов) и миграции соединений. Когда мобильные устройства переключаются между домашней сетью Wi-Fi и сотовой сетью (например, 5G) или перемещаются на большие расстояния в пределах сотовой сети (например, во время поездки на скоростном поезде), их IP-адреса меняются. Вследствие этого TCP-соединение разрывается и устанавливается повторно[1].

В то же время уже существовали протоколы, снимавшие некоторые ограничения TCP: T/TCP (Transaction TCP, RFC 1994 года) (https://oreil.ly/6elYM), STCP (Stream TCP, RFC 2000 года) (https://oreil.ly/mCzYq), MPTCP (Multipath TCP, проект RFC 2012 года) (https://oreil.ly/Ueg8S) и TCP Fast Open (проект RFC 2012 года) (https://oreil.ly/55Gva). Однако эти решения были неполными, имели проблемы с безопасностью или не получили распространения из-за окостенения TCP в интернете.

В 2013 году Google объявил о разработке[2] транспортного протокола QUIC[3]. Разработчиков QUIC вдохновила идея протоколов Stream TCP и TCP Fast

---

[1] «Когда мы отделили IP от TCP, мне казалось правильным использовать IP-адрес как часть идентификатора конечной точки соединения... Оказалось, что это была серьезная ошибка, которая препятствовала множественной адресации и снижала мобильность при смене IP-адресов». Из доклада Винта Серфа (Vint Cerf) Reimagining the Internet («Переосмысливая интернет») (https://oreil.ly/xagef), сделанного в 2010 году.

[2] См. анонс QUIC от Google за 2013 год (https://oreil.ly/WDECL). Более современный обзор предпосылок создания QUIC представлен в видео на YouTube под названием AWS re:Invent 2022 — Deliver Great Experiences with QUIC on Amazon CloudFront («AWS re:Invent 2022 — Обеспечьте высокое качество взаимодействия с QUIC на Amazon CloudFront») от Джима Роскинда (Jim Roskind), архитектора QUIC в Google (https://oreil.ly/h8bmt).

[3] Сравнение TCP и QUIC можно найти в видео на YouTube под названием Explaining QUIC: The Protocol That Is Both Very Similar to and Very Different from TCP («Объясняем QUIC: протокол, который одновременно очень похож на TCP и очень сильно от него отличается») (https://oreil.ly/GnWF0) от Питера Дорнбоша (Peter Doornbosch).

Open, и впоследствии QUIC стал основой для HTTP/3 (https://oreil.ly/I7U34). Как и в HTTP/2, в QUIC детали протокола скрывались от промежуточных устройств с помощью шифрования. Внедрение обязательного шифрования в QUIC совпало по времени с распространением в интернете TLS (в 2014 году появился Let's Encrypt). Распространению QUIC способствовало и то, что протокол работал в пространстве пользователя, а не в ядре операционной системы. Кроме того, в его основе лежал протокол UDP, что упрощало развертывание QUIC в интернете.

На рис. 4.5 представлены особенности HTTP/3 и QUIC на уровне протокола в сравнении с версиями HTTP/1.1 и HTTP/2.

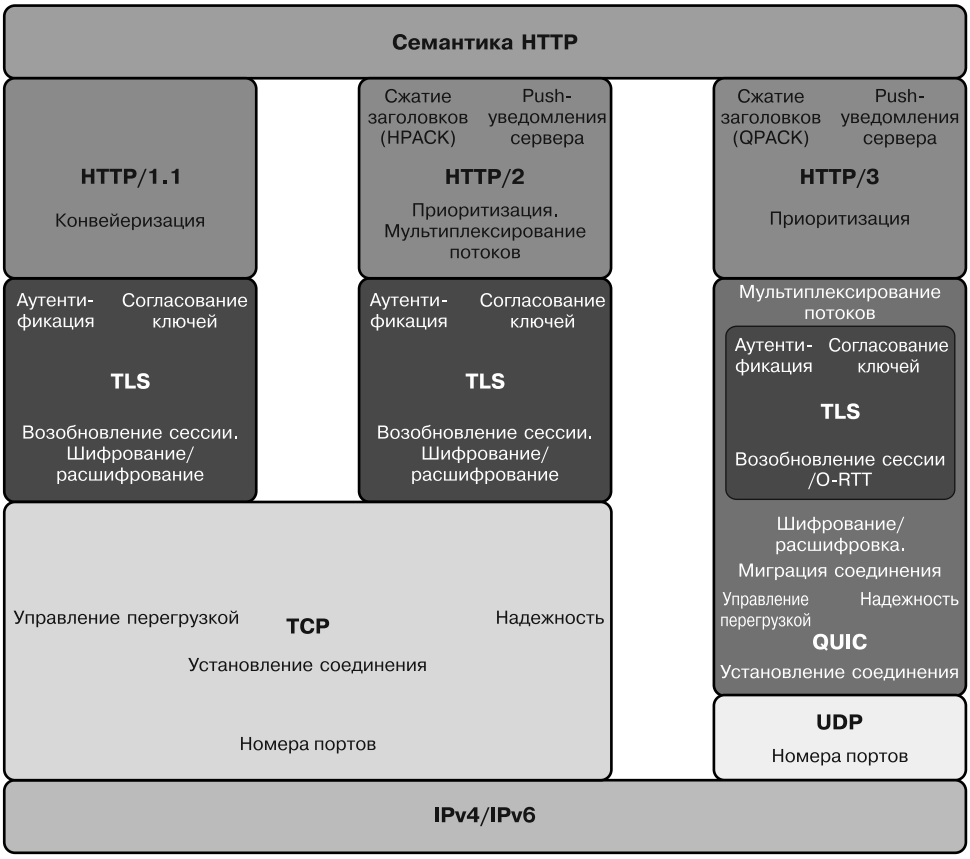


Рис. 4.5. Сравнение версий HTTP (по Робину Марксу)[1]

[1] См. раздел HTTP/3 Protocol Stack Diagrams («Схемы стека протоколов HTTP/3») (https://oreil.ly/128mT) в публикации Робина Маркса.

Кратко рассмотрим основные возможности QUIC[1].

Обязательное шифрование[2] и интеграция с TLS

Функции TLS в QUIC отличаются от стандартных функций протокола транспортного уровня, предусмотренных моделью OSI. В QUIC используются некоторые API TLS для получения криптографической информации. QUIC шифрует каждый пакет индивидуально, тогда как TLS работает с записями до 16 Кбайт (то есть одна запись TLS охватывает несколько сегментов TCP). Кроме того, QUIC устраняет риски блокировки начала очереди, возникающие из-за того, что TLS требует всю запись для расшифровки данных.

Сокращение числа RTT перед отправкой HTTP-сообщений клиентом

Достигается с помощью особой процедуры рукопожатия. В QUIC клиент отправляет предположение о параметрах TLS сервера, тем самым сокращая время начального соединения до одного RTT. Более того, по аналогии с TCP Fast Open QUIC поддерживает повторное соединение с 0-RTT (возможность TLS 1.3) для возвращающегося клиента при использовании безопасного HTTP-метода (https://oreil.ly/0agNj). При 0-RTT клиент может сразу отправить HTTP-сообщение, предполагая, что информация TLS не изменилась с момента последнего соединения. Предотвращать атаки отражения в случае обращения нелегитимного клиента позволяет дополнительный механизм, состоящий из токена проверки и отката к использованию большего числа RTT. Сокращение числа RTT в QUIC наиболее полезно для сетей с большой задержкой (например, мобильных).

Поддержка миграции соединения

Вместо кортежа из четырех параметров (IP-адрес отправителя, порт отправителя, IP-адрес получателя, порт получателя), которые используются в TCP для идентификации соединения, в QUIC клиент и сервер генерируют набор зашифрованных идентификаторов соединения и обмениваются ими при рукопожатии. При миграции соединения клиент (из соображений конфиденциальности[3]) использует один из этих идентификаторов, что позволяет серверу перенести соединение, не прерывая его. Эта возможность

---

[1] Подробный обзор QUIC и HTTP/3 представлен в серии публикаций Робина Маркса HTTP/3 From A To Z: Core Concepts («HTTP/3 от А до Я: основные концепции») (https://oreil.ly/0BmQb).

[2] Одним из последних незашифрованных элементов остается имя сервера (Server Name Indication, SNI), отправляемое в ClientHello и раскрывающее целевой домен. См. статью Encrypted Client Hello — The Last Puzzle Piece to Privacy («Зашифрованное приветствие клиента — последний элемент головоломки для обеспечения конфиденциальности») (https://oreil.ly/ziZNH) Ахила ван дер Манделе (Achiel van der Mandele) и др.

[3] Идентификатор соединения передается в открытом пакете, что позволяет отслеживать клиента.

положительно влияет на работу веб-приложений, где требуется непрерывное взаимодействие с пользователями.

Снижение рисков блокировки начала очереди по сравнению с TCP

В QUIC используются независимые потоки (рис. 4.6), поэтому проблема блокировки начала очереди ограничивается теми потоками, которые переносили потерянный пакет. Потоки мультиплексируются внутри одного пакета и между пакетами. Однако потеря пакетов в интернете часто носит характер кратковременного всплеска: за короткий промежуток времени может быть утеряно много пакетов, несущих несколько потоков. Выгода от снижения рисков блокировки в таких случаях неочевидна. При блокировке начала очереди в QUIC обнаруживается, что связи всегда упорядочены внутри одного потока, но не всегда упорядочены между разными потоками. Сжатие HPACK, используемое в HTTP/2, требовало упорядоченной доставки потоков, поэтому в QUIC был разработан специальный алгоритм сжатия заголовков QPACK.

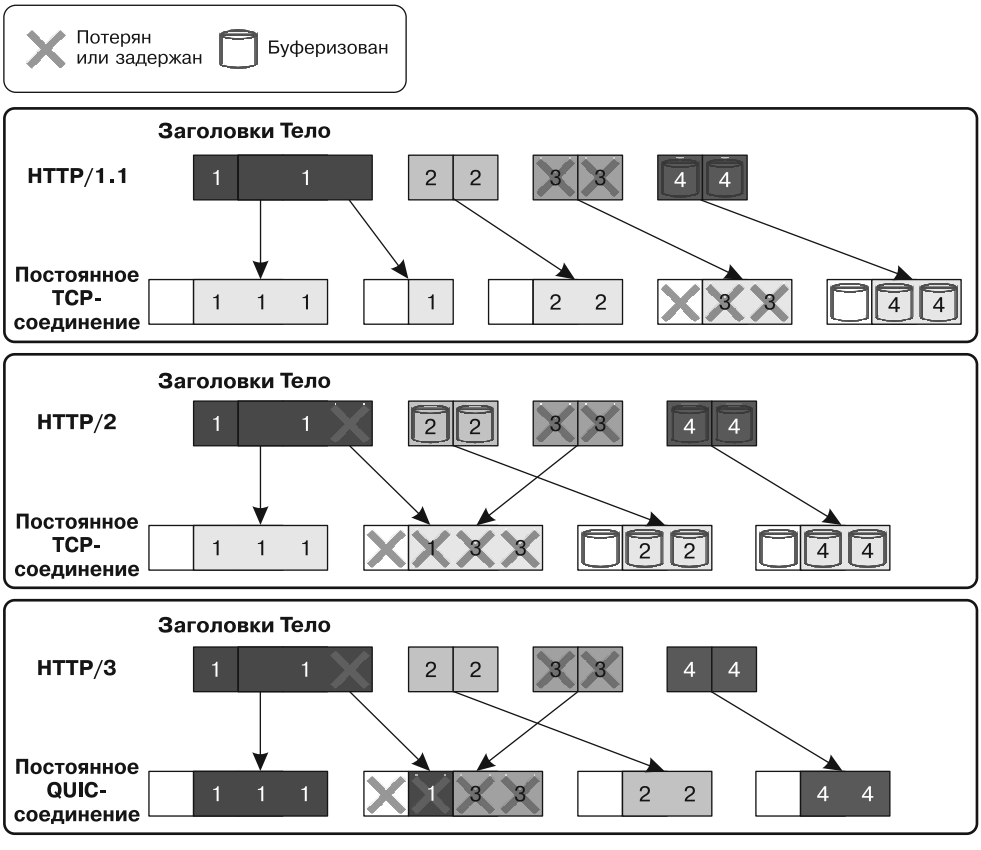


Рис. 4.6. Обзор мультиплексирования потоков и блокировки начала очереди в HTTP

В HTTP/1.1 при потере или задержке TCP-сегмента (цифра *3* на рисунке) отправитель передает последующие сегменты, но на стороне получателя они буферизуются. В результате HTTP-сообщения *3* и *4* не могут быть доставлены приложению-получателю до повторной передачи отсутствующих сегментов и восстановления TCP-потока. Постоянное TCP-соединение работает с целым потоком (на рисунке обозначен светло-серым цветом), и каждый TCP-сегмент должен доставляться по порядку.

HTTP/2 позволяет мультиплексировать фреймы, принадлежащие разным HTTP-сообщениям, в одном TCP-сегменте, а также мультиплексировать разные HTTP-сообщения между TCP-сегментами. В случае, показанном на рис. 4.6, это приводит к буферизации HTTP-сообщений *2* и *4*, поскольку потерянный или задержанный второй TCP-сегмент содержит части сообщений HTTP *1* и *3*.

В HTTP/3 риски блокировки начала очереди снижаются, поскольку каждый QUIC-поток независим (потоки обозначены разными оттенками). При потере или задержке QUIC-пакета, несущего фреймы сообщений HTTP *1* и *3*, блокировка затрагивает только потоки, содержащие эти фреймы.

Установление соединения QUIC за 1 RTT

На рис. 4.7 показаны различия в количестве RTT, необходимых для инициализации соединения с сервером до отправки клиентом первого HTTP-запроса. HTTP/2 и HTTP/1 требуют двух RTT, а HTTP/3 — одного. Кроме того, на рисунке показано переподключение (клиент с мобильного устройства на Wi-Fi отключается от сервера и подключается снова) и миграция соединения (клиент переключается с Wi-Fi на сеть 5G). Оба процесса в идеальных условиях проходят по алгоритму 0-RTT, где клиент может отправить HTTP-запрос в течение первого RTT.

Используя пример 4.23, рассмотрим последовательность пакетов при установке QUIC-соединения 1-RTT в HTTP/3, чтобы увидеть отличия от TLS. Нам потребуется окно с двумя панелями.

Пример 4.23. Установка QUIC-соединения 1-RTT в HTTP/3 (требует настройки из примера 4.1)

```
# Терминал  ❶
tmux new-session \; split-window -v \;

# Верхняя панель  ❷
CLIENT=http3
PCAP_FILE=/tmp/client_${CLIENT}.pcap
docker compose exec client bash -c \
       "sudo rm --force ${PCAP_FILE} && \
       sudo tcpdump -w ${PCAP_FILE} 'port 80 or port 443' && \
       cp ${PCAP_FILE} tests"
```

```
# Нижняя панель  ❸
docker compose exec client bash -c \
       "export SSLKEYLOGFILE=tests/SSLKEYLOGFILE.client && \
       curl --http3 --insecure --verbose https://https-server"
```

❶ Создайте две панели с помощью `tmux`.

❷ Запустите новый захват пакетов с помощью `tcpdump` на клиенте. Перехваченный трафик сохранится в файл `client_http3.pcap` в каталоге `tests`.

❸ Отправьте HTTP/3-запрос на сервер с помощью утилиты командной строки `curl` (https://curl.se). Буква `c` в `curl` означает «клиент» (client), и утилита поддерживает множество сетевых веб-протоколов. Клиент завершит работу после успешного получения ответа от сервера. Используется параметр `--insecure`, позволяющий клиенту принять самозаверенный TLS-сертификат от сервера.

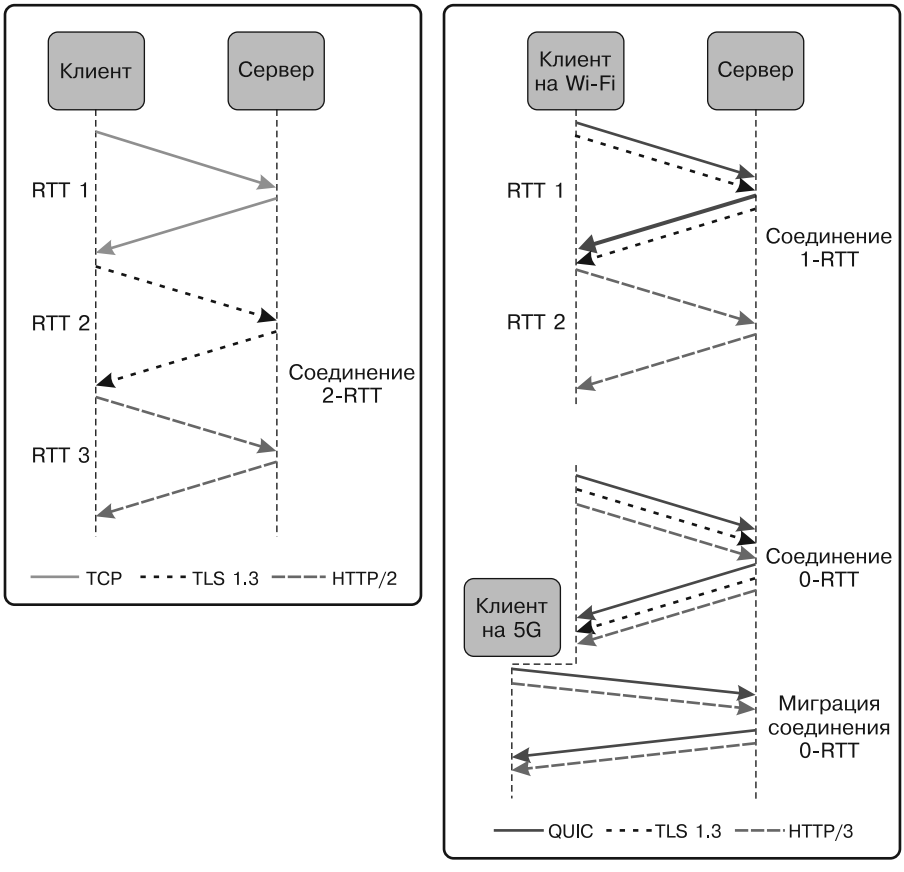


Рис. 4.7. RTT и миграция соединения в HTTP/3

В примере 4.24 показан ожидаемый ответ сервера на запрос `curl`.

Пример 4.24. Ответ HTTP/3-сервера на запрос curl

```
* Host https-server:443 was resolved.
* using HTTP/3
* [HTTP/3] [0] OPENED stream for https://https-server/  ❶
* [HTTP/3] [0] [:method: GET]
* [HTTP/3] [0] [:scheme: https]
* [HTTP/3] [0] [:authority: https-server]
* [HTTP/3] [0] [:path: /]
* [HTTP/3] [0] [user-agent: curl/8.8.0]
* [HTTP/3] [0] [accept: */*]
> GET / HTTP/3
> Host: https-server
> User-Agent: curl/8.8.0
> Accept: */*
>
* Request completely sent off
< HTTP/3 200                                          ❷
< server: nginx/1.27.0
< date: Wed, 31 Jul 2024 19:16:10 GMT
< content-type: text/html
< content-length: 55
< last-modified: Sat, 27 Jul 2024 19:17:56 GMT
< etag: "66a547e4-37"
< alt-svc: h3=":443"; ma=86400                        ❸
< accept-ranges: bytes
<
<!DOCTYPE html><title>Welcome to nginx!</title></html>  ❹
```

❶ После подключения к серверу клиент отправляет HTTP-запрос `GET /` по QUIC-потоку `0`.

❷ Клиент получает заголовки HTTP-сообщения от сервера.

❸ С помощью заголовка `alt-svc` сервер сообщает, что поддерживает HTTP/3 в течение следующих 24 часов.

❹ Тело HTTP-сообщения следует за заголовками в ответе сервера.

Остановите `tcpdump`, нажав Ctrl+C. В примере 4.25 показано, как расшифровать созданный файл захвата пакетов с помощью секретов из `SSLKEYLOGFILE` и сохранить его в файл `.pcapng`.

Пример 4.25. Расшифровка захвата пакетов HTTP/3 с помощью editcap (требует настройки из примера 4.23)

```
CLIENT=http3
docker compose exec client bash -c \
       "cd tests && \
       editcap --inject-secrets tls,SSLKEYLOGFILE.client \
       client_${CLIENT}.pcap client_${CLIENT}.pcapng"
```

Теперь файл `.pcapng` можно прочитать с помощью `tshark` (пример 4.26) или загрузить в Wireshark для более детального анализа.

Пример 4.26. Чтение файла захвата пакетов с помощью tshark (требует настройки из примера 4.25)

```
CLIENT=http3
docker compose exec client bash -c \
       "tshark --read-file tests/client_${CLIENT}.pcapng"
```

Вы увидите последовательность как в примере 4.27. Захват пакетов содержит больше информации о протоколе QUIC, чем представление со стороны клиента через `curl`. Обратите внимание: при реализации новых версий QUIC некоторые элементы этой последовательности могут измениться, поэтому отмечены только ключевые особенности HTTP/3.

Пример 4.27. Последовательность пакетов клиента HTTP/3

```
0.000 client → server QUIC   Initial, DCID=a, SCID=b, PKN: 0, CRYPTO, PADDING    ❶
0.003 server → client QUIC Handshake, DCID=b, SCID=c, PKN: 0, CRYPTO, CRYPTO     ❷
0.003 server → client QUIC Handshake, DCID=b, SCID=c, PKN: 1, CRYPTO, CRYPTO     ❸
0.006 client → server QUIC Handshake, DCID=c, SCID=b, PKN: 0, ACK, CRYPTO        ❹
0.006 client → server QUIC  PP (KP0), DCID=c, PKN: 1, PING, PADDING              ❺
0.007 client → server HTTP3 PP (KP0), DCID=c, PKN: 2, STREAM(0), HEADERS: GET /  ❻
0.007 server → client HTTP3 PP (KP0), DCID=b, PKN: 0, ACK, CRYPTO, DONE, NCI, ...
0.008 client → server QUIC  PP (KP0), DCID=c, PKN: 3, ACK
0.010 server → client HTTP3 PP (KP0), DCID=b, PKN: 1, STREAM(0),
                                                      HEADERS: 200 OK, DATA      ❼
0.011 client → server QUIC  PP (KP0), DCID=c, PKN: 4, PING, PADDING
0.011 client → server QUIC  PP (KP0), DCID=c, PKN: 5, CC                         ❽
```

❶ Клиент отправляет серверу начальный QUIC-пакет. Фрейм `CRYPTO` содержит `Client Hello` TLS 1.3 и расширения TLS, включая согласование протокола прикладного уровня (Application-Layer Protocol Negotiation, ALPN), где клиент сообщает серверу о поддержке `h3` (HTTP/3). Кроме того, имеется SNI с доменным именем сервера. Вдобавок во фрейме `CRYPTO` содержатся транспортные параметры QUIC. Фрейм `PADDING` добавляется для искусственного увеличения размера пакета, чтобы предотвратить фингерпринтинг сайта (угадывание посещаемого сайта по размерам пакетов). Клиент задает значение `DCID` (идентификатор соединения с сервером), которое будет перезаписано фактическим значением, указанным в `SCID` сервера (идентификаторе соединения с клиентом) в следующем QUIC-пакете. `PKN` (номер пакета) увеличивается для каждого типа пакетов (`Initial`, `Handshake` и `1-RTT`), причем для отправленных и полученных пакетов имеются отдельные последовательности. `PKN` играет роль, аналогичную порядковому номеру в TCP. Этот QUIC-пакет не зашифрован.

❷ Сервер отвечает пакетом `Handshake`, содержащим фреймы `CRYPTO`. В них находятся `Server Hello` TLS 1.3 и фрагмент сертификата сервера. Такая

информация, как сертификат или ALPN, шифруется. Временная метка этого пакета показывает, что RTT составляет примерно 0,003 секунды.

❸ Сервер завершает передачу зашифрованного сертификата.

❹ Рукопожатие QUIC (и TLS) завершается.

❺ Этот пакет начинает последовательность пакетов типа 1-RTT. PP — сокращение от Protected Payload (защищенная полезная нагрузка), указывающее на защиту QUIC-пакетов с использованием секретов трафика 1-RTT. Фрейм `PING` служит индикатором активности от клиента.

❻ Клиент отправляет HTTP-запрос в потоке `0`. Как и ожидалось, это происходит во время второго RTT.

❼ Сервер отправляет заголовки и тело HTTP-сообщения.

❽ Клиент отправляет фрейм `CC` (`CONNECTION_CLOSE`).

В 2024 году в вебе активно использовались все три версии HTTP (HTTP/1.1, HTTP/2 и HTTP/3). Объем трафика, приходящийся на каждую, зависит от типа клиента (десктопные браузеры, мобильные приложения или автоматизированные боты) и типа сервера, с которым они взаимодействуют[1]. Стабильно около 60 % сайтов, доступных через браузер, поддерживают HTTP/2 (этот показатель лишь незначительно снизился в последние годы)[2]. Около 30 % сайтов, напротив, поддерживают HTTP/3, и их количество продолжает расти. Однако в API по-прежнему доминирует версия HTTP/1.1 — примерно 50 % запросов отправляются по этому протоколу, но их количество также снижается. Тем не менее некоторые поисковые боты и боты в социальных сетях используют исключительно HTTP/1.1.

Как появление HTTP/3 сказалось на веб-API? Увеличение производительности за счет уменьшения числа RTT заметно в сетях с высокой задержкой, а также в регионах с менее производительной сетевой инфраструктурой. Клиент может сначала отправить запрос по HTTP/1.1 или HTTP/2 и, получив подтверждение поддержки HTTP/3 от сервера в виде заголовка `alt-svc: h3=":443"; ma=86400`, переключиться на HTTP/3. Конечно, в этом случае потребуются дополнительные RTT, и снижение задержки при установке первоначального соединения

---

[1] Подробнее см. в статье Дэвида Белсона (David Belson) и Лукаса Пардю (Lucas Pardue) Examining HTTP/3 Usage One Year On («Анализ использования HTTP/3 спустя год») на Cloudflare (https://oreil.ly/FQaYA), а также в отчете State of the Web («Состояние сайтов») (https://oreil.ly/UrAHU).

[2] По данным Can I Use (https://oreil.ly/AVr1J), с середины 2024 года HTTP/3 поддерживается во всех основных браузерах. В Debian Linux также поддерживается программа curl, которая использовалась в примерах в этой главе. По этой теме см. статью Сэмюэля Энрике (Samuel Henrique) Debian's curl Now Supports HTTP/3 («curl в Debian теперь поддерживает HTTP/3») (https://oreil.ly/GE9lY).

будет не так заметно. Тем не менее у HTTP/3 есть и более явное преимущество — сжатие заголовков, позволяющее уменьшить размер интернет-трафика.

С другой стороны, QUIC создает дополнительные вычислительные затраты: UDP не оптимизирован так, как TCP, а операции TLS на пакетном уровне выполняются чаще, чем операции на уровне записи. HTTP/3 не поддерживается нативно на некоторых облачных платформах для размещения приложений, таких как Google App Engine, Azure App Service или AWS Elastic Beanstalk. Для поддержки трафика HTTP/3 требуется прокси (CDN, шлюз или сервис-балансировщик нагрузки). Организации могут использовать устаревшие промежуточные устройства, которые не поддерживают TLS 1.3 или блокируют трафик QUIC (и UDP). Такие промежуточные устройства работают на старом протоколе TLS 1.2, где используются статические ключи в целях упрощения расшифровки и проверки трафика. По этой причине эти устройства могут быть использованы в MITM-атаках[1].

Основное преимущество протокола HTTP/3 заключается в том, что он позволяет развертывать QUIC в интернете[2]. Это, в свою очередь, дает возможность разрабатывать другие протоколы поверх QUIC и HTTP/3, например Apple iCloud Private Relay over QUIC, Microsoft Server Message Block (SMB) over QUIC, Secure Shell (SSH) over HTTP/3 или WebTransport over HTTP/3.

Упражнения

1. Сервер `http-server` реализует HTTP `POST` с помощью скрипта Common Gateway Interface (CGI), который находится в файле `src/http/httpd/post.sh`. CGI — это функция веб-сервера, которая запускает программу в ответ на HTTP-запрос. Следующий запрос `curl` возвращает код ошибки HTTP `400`. Используйте RFC для HTTP/1.0 и HTTP/1.1, чтобы проверить, соответствует ли реализация скрипта ожидаемому поведению `POST`, и выясните, как выполнить успешный `POST`-запрос к этому серверу. Подсказка: какие заголовки обязательны для `POST`-запроса HTTP/1.1?

```
docker compose exec client bash -c \
       "curl --data 'test' --http1.1 \
       --header 'Content-Length:' \
       --verbose http://http-server/cgi-bin/post.sh"
```

---

[1] Промежуточные устройства широко используются в корпоративной среде. Подробнее эта тема обсуждается в блоге Элли Бурштейн (Ellie Bursztein) на Cloudflare: см. запись Understanding the Prevalence of Web Traffic Interception («Анализ распространенности перехвата веб-трафика») (https://oreil.ly/ESMLv).

[2] См. видео Джаны Айенгар (Jana Iyengar) A Sleight of Hand! («Ловкость рук!»), являющееся частью YouTube-видео IETF 115 Technology Deep Dive: QUIC Part 1 («IETF 115: Техническое глубокое погружение в QUIC. Часть 1») (https://oreil.ly/PCzOI).

2. Сервер `http-server` также реализует пофрагментную передачу с помощью CGI. Протестируйте и изучите реализацию CGI, которая находится в файле `src/http/httpd/chunked.sh`:

```
docker compose exec client bash -c \
       "curl --http1.1 \
       --verbose http://http-server/cgi-bin/chunked.sh"
```

3. Измените пример 4.11, чтобы экспортировать переменную окружения `SSLKEYLOGFILE`. Затем запустите Firefox и откройте сайт с поддержкой HTTP/3: https://www.google.com или https://cloudflare-quic.com. Сохраните расшифрованный захват пакетов в формате `.pcapng`, как в примере 4.25, загрузите этот файл в Wireshark и ответьте на вопросы: использует ли браузер HTTP/3? Он использует HTTP/3 сразу или обновляет соединение с помощью заголовка `alt-svc`? Сколько соединений открывается на домен?

Резюме

В этой главе мы рассмотрели транспортные и прикладные протоколы, которые используются в вебе. Мы также обсудили современные возможности HTTP, сравнив их с возможностями первой версии протокола HTTP/0.9, которая позволяла получать только незашифрованный текст в кодировке ASCII по TCP. Наконец, мы поговорили о последней версии HTTP/3, позволяющей обмениваться текстом и другими форматами файлов через транспортный протокол QUIC (основанный на UDP) с двоичной кодировкой и шифрованием.

Основная цель оптимизации HTTP — ускорение загрузки сайтов в браузере. Однако при работе с API, основанным на HTTP, необходимо понимать и возможности протокола. В этой главе вы узнали, как разные характеристики сетевых протоколов влияют на безопасность HTTP (шифрование по умолчанию в QUIC), производительность (влияние DNS и трехэтапного рукопожатия TCP на время задержки) и надежность (способность TCP и QUIC восстанавливать соединение после потерь пакетов и сетевых ошибок). Кроме того, понимание основ HTTP важно и для работы с менее известными API, например gRPC или WebSocket, о которых пойдет речь в следующих главах.

На этом мы завершаем обсуждение протоколов, которые используются в сетевых API, рассматриваемых в книге. Следующие главы будут посвящены стилям API, и начнем мы со стиля REST, который описан в главе 5.

ГЛАВА 5

REST

Representational State Transfer (REST) — самый распространенный сетевой API[1]. Концепцию архитектурного стиля REST[2] в 2000 году сформулировал Рой Филдинг в диссертации *Architectural Styles and the Design of Network-Based Software Architectures* («Архитектурные стили и проектирование сетевых программных архитектур») (https://oreil.ly/YJo5o). В этой концепции предлагается подход к созданию масштабируемых сетевых приложений.

В этой главе мы рассмотрим основные аспекты REST API: его истоки, особенности реализации и тестирования, обеспечение безопасности и создание документации, а также плюсы и минусы работы со стилем REST. Кроме того, мы реализуем REST API для сервиса прогноза погоды (см. раздел «Сервис прогноза погоды» во вступлении) с помощью фреймворка Django REST Framework (DRF, https://www.django-rest-framework.org).

HTTP, ресурс и URI

В основе стиля REST лежат три кита: HTTP, ресурс и URI.

HTTP — «сердце» интернета. Этот протокол используется для обмена сообщениями в большинстве стилей API, описанных в книге. HTTP поддерживается большинством клиентов: браузерами, инструментами командной строки, например `curl` (https://curl.se), и библиотеками для разных языков

---

[1] По данным отчета State of the API Report («Отчет о состоянии API») Postman (https://oreil.ly/3LG67), REST признан самой популярной технологией API.

[2] В своей диссертации Architectural Styles and the Design of Network-Based Software Architectures («Архитектурные стили и проектирование сетевых программных архитектур») (https://oreil.ly/GuS8w) Рой Филдинг определяет архитектурный стиль как «согласованный набор архитектурных ограничений, который сокращает количество ролей/функций архитектурных элементов, а также перечень возможных отношений между ними».

программирования. HTTP — это протокол прикладного уровня, который поддерживает как текстовый, так и двоичный формат сообщений[1]. Подробнее об HTTP мы говорили в главе 4.

В REST ресурсом может быть любая информация, подлежащая наименованию: документ, изображение, числовое значение «ноль» или даже человек (невиртуальная именованная сущность, к которой нельзя получить доступ через интернет)[2]. Ресурс — это все, что может быть объектом гипертекстовой ссылки[3].

Ресурс может быть статическим или динамическим. *Статический ресурс* неизменяем, как изображение, которое веб-сервер отдает из файла. Пока статический ресурс существует, каждый клиент получает его в одном и том же виде. В случае внесения изменений ресурс считается новым. *Динамический ресурс*, напротив, формируется индивидуально для каждого клиента. Например, эндпоинт `/user/profile` возвращает содержимое, предназначенное для конкретного пользователя, вошедшего в систему.

Статические и динамические ресурсы идентифицируются по MIME-типам. *MIME-тип* описывает формат файла или тип содержимого ресурса, помогая клиентам понять, как его обрабатывать. Например, статический ресурс, такой как изображение, может иметь MIME-тип `image/png`, HTML-страница — `text/html`, а динамические данные JSON — `application/json`. Официально имеется более 1600 MIME-типов (https://oreil.ly/aijMf). Всего известно более 3000 MIME-типов для разных видов данных (https://oreil.ly/Jcg78).

В интернете примерно 1,1 миллиарда сайтов (https://oreil.ly/W4jFt), которые состоят из множества ресурсов: файлов, изображений, видео[4], блогов, руководств пользователя, новостей, журналов и многого другого. Цель унифицированного идентификатора ресурса (Uniform Resource Identifier, URI) (https://oreil.ly/fiIDU) — предоставить идентификатор, который отличает данный ресурс от

---

[1] Протоколы HTTP вплоть до версии HTTP/1.1 являются текстовыми, а начиная с версии HTTP/2 — двоичными.

[2] См. главу 5, Representational State Transfer (REST) («Передача состояния представления (REST)»), раздел Resources and Resource Identifiers («Ресурсы и идентификаторы ресурсов») в диссертации Роя Филдинга (https://oreil.ly/te5si).

[3] Концепция гипертекста была предложена Тедом Холмом Нельсоном в 1965 году. В 1989 году ее развил Тим Бернерс-Ли, определив гипертекст как информацию, предназначенную для чтения человеком и характеризующуюся неограниченным количеством внутренних связей. Подробнее о происхождении и значении терминов «гипертекст» и «гипермедиа» см. в разделе «Что такое гипертекст» главы 4.

[4] По данным на 2024 год, на YouTube опубликовано примерно 4,3 миллиарда видео (https://oreil.ly/geY8l).

других. При этом URI не гарантирует постоянства идентичности ресурса — исходный ресурс может быть изменен[1].

URI — это строка символов, идентифицирующая ресурс. Унифицированность предполагает, что разные типы идентификаторов следуют общей структуре, что обеспечивает их совместимость. URI — это общий термин, в который входят два частных понятия: *унифицированный указатель ресурса* (Uniform Resource Locator, URL) и *унифицированное имя ресурса* (Uniform Resource Name, URN) (https://oreil.ly/nhBF6).

URL указывает как способ получения доступа к ресурсу (протокол), так и его местоположение. Например, https://example.com/rfc3986.txt содержит протокол доступа к ресурсу (https), имя сервера, на котором он размещен (example.com), местоположение (/) и имя ресурса (rfc3986.txt).

URN, напротив, представляет собой постоянный идентификатор ресурса, независимый от его местоположения. По сути, URN — это URI, который идентифицирует ресурс, не описывая способ или место его получения. Например, urn:ietf:rfc:3986 идентифицирует RFC номер 3986, опубликованный Инженерным советом интернета (Internet Engineering Task Force, IETF). Префикс urn: указывает на то, что данная строка является URN.

Таким образом, URI может быть либо URL, который включает в себя и способ получения доступа, и местоположение ресурса, либо URN, который идентифицирует ресурс, не описывая способ или место его получения[2].

Чтобы вы могли лучше понять различия между типами URI, в табл. 5.1 приведены примеры URL и URN.

Таблица 5.1. Примеры типов URI

Тип URI	Пример
URL	https://www.example.com
URL	https://www.example.com/rfc3986.txt
URL	ftp://example.com/rfc3986.txt
URL	mailto:John.Doe@example.com
URL	tel:+1-816-555-1212
URN	urn:ietf:rfc:3986
URN	urn:isbn:978-1-098-15399-1

[1] Вопросы об идентичности вещи самой себе в целом неоднозначны, что убедительно демонстрирует миф о корабле Тесея (https://oreil.ly/GhC43).

[2] Подробнее о соотношении URI, URL и URN см. в RFC 3986 (https://oreil.ly/E6h4e).

На рис. 5.1 показана структура HTTP URL, а в табл. 5.2 объясняются ее элементы. Некоторые элементы, например порт, могут быть опущены. Опущенное значение заменяется значением по умолчанию, связанным со схемой.

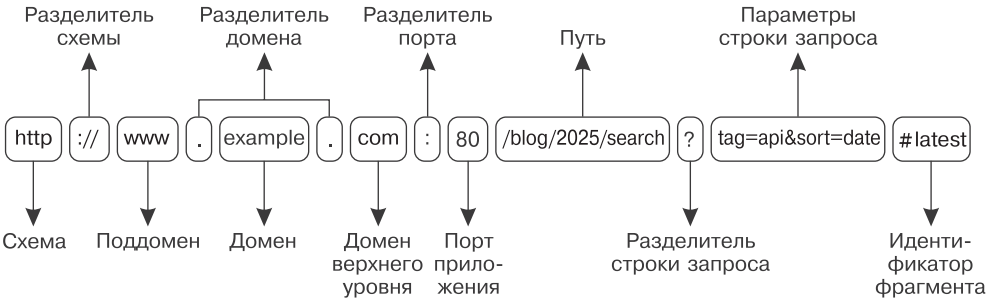


Рис. 5.1. Структура HTTP URL

Таблица 5.2. Объяснение компонентов HTTP URL

Значение в URL	Термин URL	Объяснение
http	Схема	Указывает на протокол прикладного уровня, который используют для доступа к ресурсу
://	Разделитель схемы	Отделяет схему от остальной части URL
www	Поддомен	Определяет подраздел домена
example	Домен	Основное доменное имя, которое идентифицирует сервер
.	Разделитель домена	Разделяет части доменного имени
com	Домен верхнего уровня (top-level domain, TLD)	Представляет высший уровень иерархии домена
:	Разделитель порта	Отделяет доменное имя от номера порта приложения
80	Порт приложения	Номер порта, на котором приложение обрабатывает запросы
/blog/2025/search	Путь	Указывает расположение ресурса на сервере
?	Разделитель строки запроса	Отделяет путь от строки запроса
tag=api&sort=date	Параметры строки запроса	Пары «ключ — значение», которые используются для передачи дополнительных данных на сервер
#	Идентификатор фрагмента	Обозначает начало идентификатора фрагмента внутри ресурса
latest	Фрагмент	Раздел или элемент внутри ресурса

Коммуникация по модели «клиент — сервер»

Компьютерные сети и HTTP мы подробно обсуждали в главах 3 и 4 соответственно. В текущем разделе эти темы рассматриваются в связи с HTTP-коммуникацией.

В основе HTTP лежит архитектура *«клиент — сервер»*. Мы уже дали определения понятиям «клиент» и «сервер» в разделе «Socket API» главы 3: клиент инициирует соединение, а сервер ожидает подключения со стороны клиента. На рис. 5.2 показан клиент (браузер), который запрашивает контент у веб-сервера, идентифицированного доменом `example.com`. Клиент запрашивает страницу `index.html`, отправляя HTTP-запрос `GET`, и сервер предоставляет запрошенный ресурс. С помощью кода состояния HTTP `200` (`OK`) сервер сообщает, что запрос клиента обработан успешно.

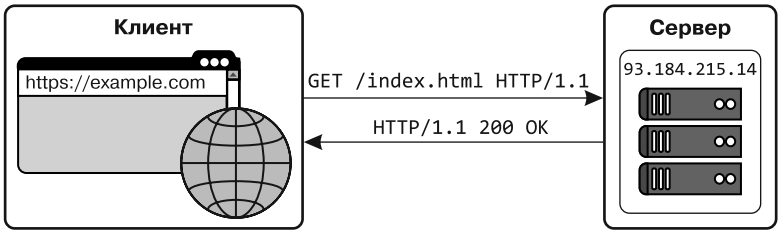


Рис. 5.2. Взаимодействие браузера и веб-сервера в HTTP

Термины «запрос» и «ответ» указывают направление потока данных между клиентом и сервером. Клиент отправляет запрос на сервер, который принимает его, обрабатывает и возвращает ответ. *Модель «запрос — ответ»* (в частности, ее HTTP-версию) можно реализовать синхронно или асинхронно (см. подраздел «Синхронная и асинхронная коммуникация» в главе 1).

HTTP находится на верхнем — прикладном — уровне модели OSI (уровень 7). Для начала определимся с понятием протокола прикладного уровня. Прикладные протоколы определяют правила обмена сообщениями, а также правила их структурирования и интерпретации (см. определение понятия «сообщение» в соответствующем подразделе главы 1). Эти правила обеспечивают коммуникацию между программными приложениями. Пока программные приложения соблюдают правила протокола, они могут взаимодействовать и обмениваться данными, несмотря на различия в языках программирования, фреймворках, средах запуска, архитектурах и операционных системах. Браузер и веб-сервер, или клиент `curl` и различные HTTP-библиотеки — это примеры

разных программных технологий, которые используют один и тот же протокол прикладного уровня (HTTP).

HTTP-сообщение — это текст, состоящий из строк (последовательностей символов), где каждая строка завершается символом возврата каретки (<CR>), за которым следует символ перевода строки (<LF>). HTTP-сообщение содержит следующие элементы:

1) *стартовую строку* для HTTP-запроса;
2) *строку состояния* для HTTP-ответа;
3) *заголовки*;
4) *пустую строку* (<CR><LF>), которая отделяет заголовки от тела сообщения;
5) *тело сообщения*, содержащее данные (в зависимости от используемого HTTP-метода тело сообщения может отсутствовать).

На рис. 5.3 показаны HTTP-сообщения запроса и ответа, полученные при обращении к http://example.com/index.html. Для удобства чтения ответ приведен в сокращенном виде и содержит только самые часто используемые заголовки.

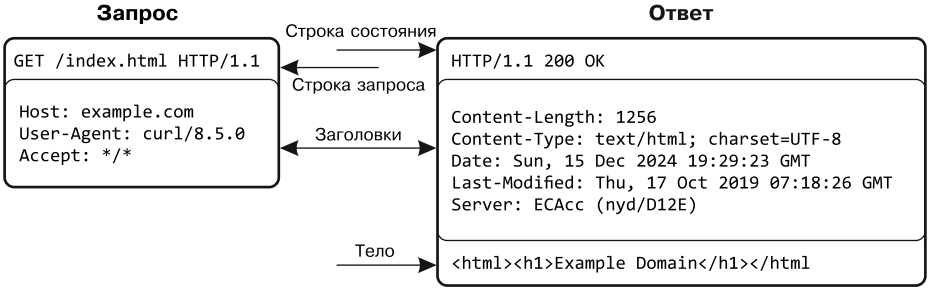


Рис. 5.3. HTTP-сообщения запроса и ответа

HTTP работает поверх TCP, который, в свою очередь, работает поверх IP. Инкапсулируя HTTP-сообщения в сегменты TCP, HTTP обеспечивает надежную доставку сообщений в упорядоченном виде и позволяет своевременно реагировать на ошибки. Поскольку HTTP работает поверх TCP, HTTP-сообщения отправляются в виде полезной нагрузки внутри TCP-сегмента (полезная нагрузка может быть в текстовом или двоичном формате).

На рис. 5.4 представлено взаимодействие по модели «клиент — сервер» и показан поток информации между клиентом и сервером в интернете. На схеме объединены все паттерны и протоколы, рассмотренные в этом разделе. Сейчас

HTTP-трафик шифруется с помощью TLS, однако для простоты чтения HTTP-сообщения представлены на рисунке в незашифрованном виде. Подробнее о TLS см. раздел «Безопасность» главы 3.

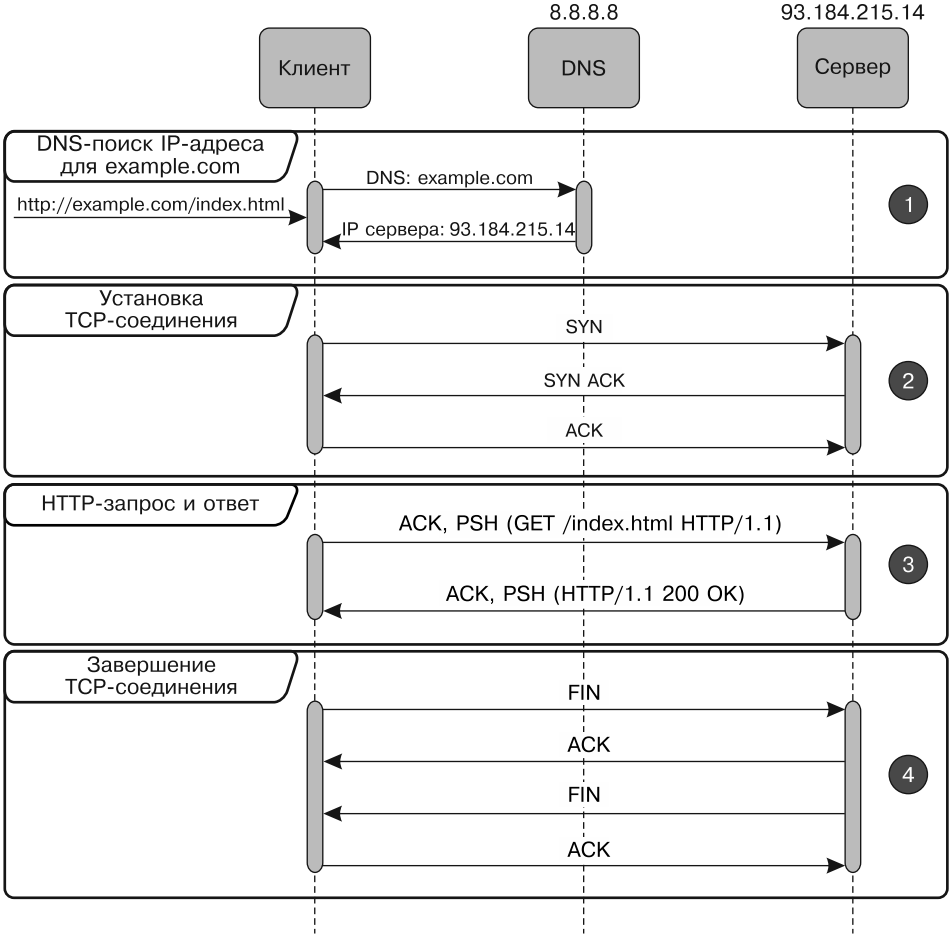


Рис. 5.4. Жизненный цикл HTTP-взаимодействия

Рассмотрим этапы HTTP-взаимодействия подробнее.

1. Клиент пытается получить доступ к адресу http://example.com/index.html. Схема URL задает протокол `http`. Доменное имя example.com идентифицирует веб-сервер. Поскольку номер порта не указан явно, то используется значение `80` (стандартный HTTP-порт для незашифрованного соединения). Ресурс index.html находится в корне (/) веб-сервера.

Сначала, используя протокол DNS[1], клиент отправляет запрос на запись типа «A»[2] на DNS-сервер с IP-адресом `8.8.8.8`. Сервер преобразует доменное имя example.com в IP-адрес веб-сервера `93.184.215.14`[3].

2. IP-адрес сервера и номер порта `80` используются для установки TCP-соединения через интернет между клиентом и сервером (здесь мы не будем подробно рассматривать маршрутизацию). Соединение устанавливается по алгоритму трехстороннего рукопожатия TCP (см. пример 3.5 в главе 3). IP-адрес клиента и случайный порт на схеме не показаны.
3. После установки TCP-соединения клиент отправляет HTTP-запрос `GET`, требуя от сервера предоставить содержимое ресурса index.html, расположенного в корне (/) сервера. Сервер выдает в ответ содержимое и код состояния `200` (`OK`), который указывает на успешную обработку запроса. Поскольку HTTP работает поверх TCP, то HTTP-сообщения представляют собой текст или двоичные данные, которые отправляются как полезная нагрузка внутри TCP-сегмента вместе с TCP-флагами (например, `ACK` или `PSH`).
4. Получив HTTP-ответ, клиент завершает TCP-соединение с сервером.

Происхождение стиля REST

Архитектурный стиль REST берет начало в работах Роя Филдинга (https://oreil.ly/D17wG), посвященных проектированию масштабируемых сетевых приложений.



Обзор стиля REST с точки зрения веб-разработки представлен в статье REST — Explained For Beginners («REST: разбор для начинающих») Карсона Гросса (Carson Gross) (https://oreil.ly/Y5Vh0).

Филдинг стремился решить следующие задачи:

- сохранение согласованности при соединении гипермедийных систем через интернет;
- независимое развертывание программных компонентов;

---

[1] Подробнее о DNS см. подраздел «Влияние DNS на соединения в браузере» главы 4.

[2] Запрос записи типа «A» применяется для четвертой версии IP (IPv4). Для IPv6 используется запрос записи типа «AAAA».

[3] Ответы DNS кэшируются на основе значения time-to-live (TTL), поэтому клиенту не нужно выполнять DNS-запрос при каждом обращении.

- *анархическая масштабируемость* (способность системы сохранять функциональность при неожиданно высокой нагрузке);
- поддержка промежуточных компонентов, таких как прокси или кэши, для повышения масштабируемости.

Под масштабируемостью мы понимаем способность системы справляться с растущим объемом нагрузки. На основе указанных задач были сформулированы ограничения стиля REST[1].

1. Архитектура «клиент — сервер» разделяет обязанности между клиентом и сервером: клиент формирует запрос, а сервер генерирует ответ. Это ограничение повышает переносимость ПО между платформами, так как клиент и сервер могут развиваться независимо.
2. Взаимодействие без сохранения состояний: каждый клиентский запрос содержит всю информацию, которая необходима серверу для его понимания и выполнения (то есть не зависит от предыдущих запросов). Сервер не хранит состояние сессии; если приложению требуется сохранение состояния, то оно реализуется на стороне клиента.
3. Кэшируемые ресурсы: в ответах должно быть явно указано, можно ли их кэшировать. Кэширование позволяет клиентам и промежуточным компонентам системы повторно использовать ответы сервера, когда это уместно. Благодаря этому повышается эффективность работы сети, снижаются задержки и улучшается видимая производительность.
4. Единообразие интерфейса: системные компоненты описываются единообразными интерфейсами, что делает взаимодействие между компонентами более прозрачным. Стандартизированный интерфейс упрощает их взаимодействие и снижает зависимость компонентов друг от друга. К интерфейсу предъявляются следующие требования.
   - Идентификация ресурсов: каждый ресурс имеет уникальный идентификатор, обычно в виде URI.
   - Манипуляция ресурсами через представления: в зависимости от потребностей клиента ресурсы могут быть представлены в разных форматах (например, HTML, JSON, XML).
   - Самодостаточность сообщений: каждое сообщение содержит всю информацию, необходимую для его понимания и обработки. Например, информация об HTTP-сообщении содержится в его теле и заголовках.

---

[1] См. главу 5 Representational State Transfer (REST) («Передача репрезентативного состояния (REST)») (https://oreil.ly/TCYQK) в диссертации Филдинга.

- HATEOAS (Hypermedia As The Engine of Application State): в ответ должны входить гиперссылки для навигации по REST API. В идеале клиент должен знать только начальную точку входа, а всю информацию о доступных взаимодействиях получать через гипертекстовые/гипермедийные ссылки.

5. Многоуровневая система: компоненты системы относятся к конкретному уровню в иерархии и могут непосредственно взаимодействовать только с компонентами следующего уровня. В качестве уровней, через которые проходит содержимое ресурсов, могут выступать шлюзы, прокси и обратные прокси[1].
6. Код по требованию (code-on-demand) (не обязательно): благодаря возможности загружать скрипты с сервера и выполнять их функциональность клиента можно расширить без повторного развертывания.

Модели зрелости API

Модели зрелости используются для измерения и оценки зрелости процессов или технологий. При оценке ПО модели зрелости позволяют измерить, насколько хорошо оно спроектировано и реализовано. Есть две широко используемые модели зрелости для API, основанных на HTTP: модель CohA (Classification of HTTP-based APIs) (https://oreil.ly/PoJtO) и модель зрелости Ричардсона (Richardson Maturity Model, RMM) (https://oreil.ly/WB0ys). Мы сосредоточимся на последней.

В 2008 году Леонард Ричардсон (Leonard Richardson) предложил фреймворк для оценки зрелости веб-сервисов (веб-API). Основным критерием стало соответствие сервисов принципам проектирования архитектурного стиля REST. В модели RMM учитываются три ограничения: использование методов HTTP, URI и элементов управления гипермедиа (HATEOAS).



Понятия «гипертекст» и «гипермедиа» обсуждаются в разделе «Что такое гипертекст» главы 4.

Модель RMM делится на четыре уровня, от низшей степени зрелости API (уровень 0) до высшей (уровень 3).

---

[1] Многоуровневые системы могут негативно влиять на пользовательский опыт, создавая дополнительную нагрузку и увеличивая время задержки.

Уровень 0

Используется один URI и один HTTP-метод. Пример — SOAP API, который обычно использует один URI (эндпоинт) и HTTP-метод `POST` для передачи полезной нагрузки по протоколу SOAP.

Уровень 1

Используются отдельные URI для идентификации системных ресурсов и один HTTP-метод (обычно `POST`).

Уровень 2

Используются отдельные URI для идентификации системных ресурсов и несколько HTTP-методов для каждого ресурса, что позволяет выполнять операции CRUD (Create, Read, Update, Delete; создание, чтение, обновление, удаление).

Уровень 3

Имеет все свойства API уровня 2 и содержит элементы управления гипермедиа — самодокументируемые ссылки, которые предоставляют информацию о возможных действиях в текущем состоянии ресурса API. HATEOAS позволяет клиентам ориентироваться в API и снижает зависимость разработчиков от документации.

Умение классифицировать API позволяет оценивать и улучшать их в соответствии с требованиями конкретного уровня модели RMM.

RESTful, RESTless или REST

Если API классифицируется как RESTless, то не достигает уровня 3 в модели RMM. Многие разработчики ошибочно относят свои RESTless API к категории *RESTful*. В этой книге мы сосредоточимся на REST API, которые, по нашему определению, относятся к категории RESTless. К категории RESTful будем относить только те REST API, в которых есть HATEOAS.



Поскольку RESTless API — самые распространенные, то в этой главе речь пойдет о них и мы будем называть их REST API.

Тем, кто обеспокоен точностью терминологии и хочет называть свой REST API RESTful, следует помнить, что в RESTful API должен быть реализован HATEOAS. Рассмотрим реализацию HATEOAS (пример 5.1). Обратите внимание на поле `_links`.

Пример 5.1. Запрос и ответ в API с HATEOAS

```
# HTTP-сообщение запроса
POST /order/123 HTTP/1.1
Host: example.com
Content-Type: application/json
Authorization: Bearer jwt_token

# HTTP-сообщение ответа
HTTP/1.1 200 OK
Content-Type: application/json
Date: Tue, 19 Mar 2025 10:00:00 GMT
Content-Length: 830

{
  "data": {  ❶
    "id": 123,
    "status": "processing",
    "item": "Laptop",
    "quantity": 1
  },
  "_links": {  ❷
    "self": {
      "href": "/order/123",
      "method": "GET",
      "description": "Получить детали заказа."
    },
    "cancel": {
      "href": "/order/123/cancel",
      "method": "POST",
      "description": "Отменить заказ.",
      "body": null
    },
    "notification": {  ❸
      "href": "/order/123/notification",
      "method": "POST",
      "description": "Получать уведомления о статусе заказа.",
      "content_type": "application/json",
      "body": {
        "parameters": {
          "channels":{
            "type": "array",
            "items": {
              "type": "string",
              "enum": ["sms", "email", "in-app"]
            },
            "description": "Предпочитаемые каналы уведомлений.",
            "required": true
          }
        }
      }
    }
  }
}
```

❶ Поле `data` содержит детали конкретного заказа.

❷ Поле `_links` указывает на то, что ответ соответствует спецификации Hypertext Application Language (HAL). В ней описывается структура API, которые позволяют клиентам перемещаться между ресурсами по ссылкам. Поле `_links` содержит три возможных действия с объектом: `self`, `cancel` и `notification`. Каждое действие содержит поле `href` — гипермедиа-ссылку на расположение ресурса. Поле `method` описывает HTTP-метод для взаимодействия с ресурсом, а `description` документирует действие. Для HTTP-метода `POST` есть поле `body`, которому присваивается значение `null`, если тело сообщения пустое.

❸ Поле `notification` описывает метод `POST`, а его тип определяется полем `content_type` (указывает на полезную нагрузку в формате JSON). В этом примере полезная нагрузка содержит обязательное поле `channels` — JSON-массив с предпочитаемыми каналами уведомлений.



Чтобы детально изучить структуру ответа в RESTful API (с HATEOAS), посетите ресурсы Adobe Experience Platform (https://oreil.ly/KvZqY) или Adobe Experience Manager (https://oreil.ly/FyzPd) и проанализируйте ответы их API. Обратите внимание на поля links или actions. Мы рекомендуем эти ресурсы, поскольку сам Рой Филдинг относил их к категории RESTful (HATEOAS)[1].

Достаточно ли создать RESTless API

Возникает вопрос: обязательно ли API должен относиться к категории RESTful? Однозначного ответа нет, но чаще всего RESTless API более чем достаточно. Создавать REST (RESTless) API можно и нужно. Недопустимо называть его RESTful, когда это не так[2].

Рассмотрим API нескольких известных компаний. Для этого воспользуемся инструментом Google APIs Explorer (https://oreil.ly/4Z2hU), который позволяет находить разные API Google и взаимодействовать с ними. Выберите любой API из списка, отправьте запрос и изучите ответы. Скорее всего, вы увидите, что в API не реализован HATEOAS, а значит, API относится к категории RESTless, а не RESTful. Такой же результат вы получите с Graph API от Meta (https://oreil.ly/Qehtq).

---

[1] См. слайд 25 в презентации Филдинга REST in AEM («REST в AEM») (https://oreil.ly/xThub).

[2] См. пост Роя Филдинга REST APIs Must Be Hypertext-Driven («REST API должны быть управляемы гипертекстом») (https://oreil.ly/gBsWR).

HATEOAS позволяет обнаружить API (ответы самодокументированы) и ослабляет взаимную зависимость клиента и сервера (ссылки на гипермедиа контролируются сервером, а не «вшиваются» в логику клиента). Возникает закономерный вопрос: почему, несмотря на эти преимущества, такие компании, как Google и Meta, не создают RESTful API? Есть ряд технических и нетехнических причин[1].

Сложности на стороне сервера

Поддержка API с HATEOAS усложняется по мере добавления новых ресурсов, между которыми нужно поддерживать связь. Сервер должен динамически генерировать ответы с гипермедийными ссылками, учитывая состояние приложения и права пользователя.

Кроме того, размер сообщения и время его обработки увеличиваются, поскольку API генерирует больше ссылок (что увеличивает объем полезной нагрузки), а для проверки корректности ответов API требуется больше процедур валидации.

Различия в способах фрагментации в разных форматах гипермедиа, таких как HAL (https://oreil.ly/w0YpX), Hydra (https://oreil.ly/iNqaC), JSON-LD (https://json-ld.org), JSON Hyper-Schema (https://oreil.ly/FD5Iq) и Siren (https://oreil.ly/ar50N), влияют на совместимость. С точки зрения обеспечения безопасности RESTful API должен быть спроектирован так, чтобы его гипермедиа не раскрывали «чувствительные» эндпоинты, доступ к которым должен быть закрыт или ограничен.

Наконец, из-за большего количества ссылок по сравнению с RESTless API в RESTful API трудно управлять версиями и поддерживать обратную совместимость. В случае RESTless API с проблемами версионирования сталкиваются в основном разработчики клиентов API.

Сложности на стороне клиента

Клиент API с HATEOAS должен уметь управлять текущим состоянием приложения и ориентироваться в нем, а также интерпретировать гипермедийные ссылки и принимать решения на основе ссылок, предоставленных сервером. Зависимость от гипермедийных ссылок может увеличить сетевой трафик, поскольку клиент динамически обнаруживает доступные действия во время взаимодействия с сервером. Например, если для навигации по ресурсам требуется несколько последовательных вызовов, то производительность приложения может снизиться, тем самым ухудшив пользовательский опыт.

Кроме того, усложняется отладка и тестирование клиентов HATEOAS, так как возникает необходимость имитировать разные состояния сервера

[1] См. распространенные проблемы и ограничения HATEOAS (https://oreil.ly/8UGC4).

и гипермедийные ответы. Клиент следует по ссылкам от сервера, поэтому любая ошибка на стороне сервера при генерации ссылок (или ошибка в их структуре) может нарушить работу клиента.

Выпуск новых версий API тоже влияет на клиентов HATEOAS, поскольку они должны адаптироваться к новым, измененным или устаревшим действиям. По мере увеличения количества ресурсов разработчики вынуждены усложнять архитектуру клиента, чтобы он эффективно обрабатывал динамические взаимодействия, не теряя производительности.

Прагматические соображения

Разработка и использование API с HATEOAS может увеличить трудозатраты, особенно для приложений, в которых не используются дополнительные фреймворки. В этом случае на ранних этапах требуется более тщательное проектирование для решения задач, связанных с внедрением и интерпретацией навигационных ссылок. API может достичь 3-го уровня RMM, если бэкенд- и фронтенд-разработчики обладают необходимой вовлеченностью, дисциплиной и экспертностью. В отличие от API 3-го уровня REST API на 2-м уровне не следуют принципам HATEOAS и используют множество URI и HTTP-методов, что снижает накладные расходы. Поэтому 2-й уровень RMM часто оказывается достаточным для решения большинства задач.

Документирование

HATEOAS позволяет серверу API информировать клиентов о доступных действиях, встраивая гипермедийные ссылки в ресурсы (см. выше пример 5.1). Такой подход упрощает обнаружение API, поскольку вместо жесткого кодирования конкретных URL клиенты могут переходить по ссылкам и взаимодействовать со связанными ресурсами. Используя гипермедийные ссылки от сервера, клиенты не привязываются к конкретной структуре URL, что снижает риск возникновения критических изменений при эволюции API. Отсутствие привязки к URL позволяет клиентам адаптироваться к разным типам взаимодействия и использовать разные пути ресурсов в зависимости от ссылок и метаданных, предлагаемых сервером.

Однако ссылки HATEOAS — лишь один из элементов использования API. Ссылки, безусловно, помогают в навигации, однако разработчику все равно необходимо изучать документацию API, чтобы увидеть набор доступных эндпоинтов, их функциональность, примеры использования, обработку ошибок, правила валидации и другие детали.

RESTful API целесообразно реализовывать в сценариях, где бизнес-логика приложения часто меняется, и клиент должен легко обнаруживать эти изменения. Вдобавок тип RESTful оптимален, если нужно создать самодокументируемый API.

Теперь, когда мы поговорили о происхождении REST API и принципах HATEOAS, а также выяснили, что в большинстве случаев достаточно API 2-го уровня RMM, мы можем перейти к реализации REST API.

Реализация

В этом разделе показано, как реализовать REST API с помощью Django REST Framework (DRF, https://www.django-rest-framework.org). Мы будем использовать паттерны проектирования API, описанные в главе 2. Чтобы выполнить примеры кода из этой главы, сначала настройте лабораторную среду (пример 5.2).

Пример 5.2. Настройка лабораторной среды для REST

```
cd src/Django  ❶
docker compose build --build-arg UID=$(id -u) --build-arg GID=$(id -g)  ❷
docker compose up --detach –wait  ❸
```

❶ Перейдите в каталог для настройки лабораторной среды.

❷ Соберите Docker-образы.

❸ Запустите Docker-контейнеры.

Эндпоинты API

Мы начали обсуждение паттернов проектирования в главе 2 с двух концепций: языка API и стандартов именования (см. подразделы «Язык API» и «Наименование элементов API» в главе 2). Язык, которым мы будем пользоваться при описании API, — американский английский (соответственно, мы будем соблюдать правила его орфографии). Чтобы определиться с принципами именования, рассмотрим эндпоинты REST API для сервиса прогноза погоды (WFS).

Эндпоинты, маршруты и интерфейсы REST API для WFS показаны в примере 5.3. В коде указаны эндпоинты приложения и соответствующие им обработчики (классы). В этой главе мы реализуем классы `CityView` и `CityListView`.

Пример 5.3. Эндпоинты REST API для WFS

```
# src/django/app/config/urls.py

from core.api.rest.v1.views import CityListView
from core.api.rest.v1.views import CityView
from core.api.rest.v1.views import GeocodingView
from core.api.rest.v1.views import WeatherForecastListView
from core.api.rest.v1.views import WeatherHistoryListView
from core.api.rest.v1.views import WeatherSeedView
...
```

```
urlpatterns = [
    ...
    path("api/cities", CityListView.as_view()),                ❶
    path("api/cities/<uuid:uuid>", CityView.as_view()),        ❷
    path("api/forecasts", WeatherForecastListView.as_view()),  ❸
    path("api/geocoding", GeocodingView.as_view()),            ❹
    path("api/history", WeatherHistoryListView.as_view()),     ❺
    path("api/seed", WeatherSeedView.as_view())],              ❻
    ...
]
```

❶ Эндпоинт `/api/cities` отвечает за получение списка всех городов, для которых доступен прогноз погоды.

❷ Эндпоинт `/api/cities/<uuid:uuid>` возвращает ресурс города, идентифицированный по `uuid`.

❸ Эндпоинт `/api/forecasts` предоставляет прогноз погоды для заданного города.

❹ Эндпоинт `/api/geocoding` преобразует названия городов в географические координаты, и наоборот.

❺ Эндпоинт `/api/history` предоставляет исторические данные о погоде для города.

❻ Эндпоинт `/api/seed` используется для первоначального заполнения базы данных WFS данными о погоде.

Наш API построен вокруг ресурса `cities` (города), в наименовании которого используется форма множественного числа. Ресурс города идентифицируется с помощью `uuid` в рамках родительской коллекции `cities/uuid` (этот эндпоинт управляет отдельным объектом города). Обратите внимание, что слово *history* («история») может быть как исчисляемым, так и неисчисляемым, а его форма множественного числа — *histories* (https://oreil.ly/CG5Ui). Мы выбираем форму единственного числа, чтобы обозначить исторические данные о погоде для отдельного города. Название `histories` противоречило бы здравому смыслу — один город не может иметь несколько разных историй погоды.

Еще один характерный случай именования — название ресурса, который отвечает за прогнозирование погоды. В американском английском чаще используют словосочетание *weather forecast* (например, Have you checked the weather forecast for tomorrow? («Ты уже смотрел прогноз погоды на завтра?»)). Однако использование формы единственного числа слова *forecast* указывало бы на то, что API предоставляет прогноз только на один день. Поэтому, несмотря на существование общеупотребительной фразы, мы выбираем форму множественного числа *forecasts*, что указывает на возможность получения прогноза на несколько дней вперед.

Кроме того, может возникнуть вопрос: почему не сделать эндпоинты `forecast` и `history` частью эндпоинта `cities` (например, `cities/uuid/forecast` и `cities/uuid/history`)? Проблема такого решения в том, что сервер будет возвращать прогноз только для конкретного города. Кроме того, не будет возможности получить прогнозы для нескольких городов сразу. Еще один аргумент против размещения эндпоинтов `forecast` и `history` под эндпоинтом `cities` заключается в том, что прогноз погоды и история служат разным целям. Наконец, плоская структура эндпоинтов (без вложенности) обеспечивает согласованность, простоту использования и эффективное кэширование, поскольку такие эндпоинты реже меняются.



В статье How to (and How Not to) Design REST APIs («Как проектировать REST API (и как не надо)») Джеффа Шнитцера (Jeff Schnitzer) (https://oreil.ly/6wqnt) даны дополнительные рекомендации по организации ресурсов в REST.

Версионирование API

В подразделе «Версионирование API» главы 2 мы рассмотрели несколько стратегий версионирования.

Глядя на REST-эндпоинты WFS, перечисленные в примере 5.3 (см. выше), можно подумать, что наш API не предоставляет никакой информации о версии. Это связано с тем, что мы используем версионирование через параметр запроса (например, `/api/cities?version=v1`). Если версия не указана, это значит, что клиент взаимодействует с версией v1 (версия по умолчанию, установленная разработчиком).

В примере 5.4 показана конфигурация версионирования в DRF и ее использование в классе представления API.

Пример 5.4. Реализация версионирования API

```
# src/django/app/config/base.py

REST_FRAMEWORK = {  ❶
    ...
    "DEFAULT_VERSIONING_CLASS": "rest_framework.versioning.
QueryParameterVersioning",
    "DEFAULT_VERSION": "v1",
    "ALLOWED_VERSIONS": ("v1", "v2"),
    "VERSION_PARAM": "version",
    ...
}

# src/django/app/core/api/rest/v1/views/city_views.py
```

```
class CityListView(GenericAPIView):
    ...
    def get_serializer_class(self):  ❷
        if self.request.version == REST_FRAMEWORK["DEFAULT_VERSION"]:
            return CitySerializer
        return super().get_serializer_class()
    ...
  ...
```

❶ В этом блоке мы настраиваем DRF, чтобы включить версионирование через параметр запроса с помощью `QueryParameterVersioning` (https://oreil.ly/8c3Cw). `DEFAULT_VERSION` устанавливает версию API по умолчанию как `v1`, а `ALLOWED_VERSIONS` определяет кортеж строк, представляющих версии API, которые поддерживает сервер. `VERSION_PARAM` задает имя параметра запроса, используемого для версионирования.

❷ Метод `get_serializer_class()` настраивает сериализатор, исходя из версии API, указанной во входящем запросе. Если версия запроса совпадает с версией по умолчанию (`v1`), то метод возвращает `CitySerializer` в качестве класса сериализатора. В противном случае он возвращает реализацию родительского класса.

CRUD

Термин *CRUD* (Create, Read, Update, Delete) был введен в 1983 году и обозначал четыре операции, которые можно выполнять над постоянным хранилищем[1]. С течением времени его начали применять не только в отношении баз данных, но и в отношении API и пользовательских интерфейсов (табл. 5.3).

Таблица 5.3. Операции CRUD в базах данных, приложениях и пользовательских интерфейсах

Операция CRUD	SQL	HTTP	UI
Create (создание)	`INSERT`	`POST`	Add (добавить)
Read (чтение)	`SELECT`	`GET`	View, Retrieve, Search (просмотреть, получить или найти)
Update (обновление)	`UPDATE`	`PATCH, PUT`	Edit (изменить)
Delete (удаление)	`DELETE`	`DELETE`	Remove (удалить)

[1] Термин был впервые использован в книге: *Martin J.* Managing the Data Base Environment. — Prentice Hall, 1983. — P. 381. https://oreil.ly/xS79e.

Следующий шаг после определения эндпоинтов — реализация обработчиков запросов (в Django они называются *views* — представления). В этом подразделе мы реализуем классы `CityListView` и `CityView`, которые содержат CRUD-операции для ресурса `city`.



Мы намеренно опустили детали реализации операции Update (обновление). Ее описание довольно длинное, хотя, по сути, она похожа на операцию Create (создание). Детали реализации Update (и других операций из CRUD) описаны в репозитории кода книги.

ИДЕМПОТЕНТНОСТЬ, БЕЗОПАСНОСТЬ И КЭШИРОВАНИЕ HTTP-ЗАПРОСОВ

Методы HTTP можно классифицировать по трем свойствам протокола: идемпотентность (https://oreil.ly/MIu2d), безопасность (https://oreil.ly/BVAwC) и кэшируемость (https://oreil.ly/VdFAj).

Идемпотентный метод HTTP одинаково воздействует на сервер как при однократном, так и при многократном вызове. Например, если несколько раз запросить ресурс, используя метод `GET`, то сервер будет каждый раз возвращать одни и те же данные. То же самое относится к методу `DELETE`. При выполнении первого запроса `DELETE` сервер ответит кодом состояния `200 (OK)`, а на любой последующий запрос `DELETE` к тому же ресурсу вернет код `404 (Not Found)`. Это происходит потому, что идемпотентная операция учитывает только состояние сервера.

Операция считается безопасной, если не изменяет состояние сервера. Например, метод `POST` не является безопасным, так как создает новый ресурс. Однако безопасный метод не обязательно предполагает только чтение. Более того, чтение ресурса влияет на состояние сервера, активизируя логирование или вызывая ограничение частоты запросов. Метод считается безопасным до тех пор, пока не изменяет состояние сервера.

В HTTP идемпотентность и безопасность определяются через ожидаемый эффект запроса с точки зрения клиента. Это означает, что даже безопасные (`GET`) и идемпотентные (`PUT`) методы, которые приносят клиенту ожидаемый ответ, могут вызывать побочные эффекты на сервере. Такая неоднозначность в определениях возникает потому, что в спецификации HTTP не разграничиваются первичный и побочные эффекты операции.

Кэширование предполагает, что имеющийся ответ можно использовать повторно для последующих запросов. Его применяют, чтобы сократить время ответа сервера и снизить потребление пропускной способности

сети. Для кэширования используется локальное или удаленное хранилище (кэш), которое управляет сохранением, извлечением и удалением ответов[1].

В табл. 5.4 представлена классификация стандартных методов HTTP по характеристикам идемпотентности, безопасности и кэшируемости[2].

Таблица 5.4. Классификация методов HTTP по критериям идемпотентности, безопасности и кэшируемости

HTTP-метод	Идемпотентность	Безопасность	Кэшируемость[a]
GET (https://oreil.ly/ob3a5)	Да	Да	Да
HEAD (https://oreil.ly/ebspK)	Да	Да	Да
POST (https://oreil.ly/EmINu)	Нет	Нет	Нет
PUT (https://oreil.ly/4oiLs)	Да	Нет	Нет
DELETE (https://oreil.ly/ZhqKK)	Да	Нет	Нет
CONNECT (https://oreil.ly/5vfnE)	Нет	Нет	Нет
OPTIONS (https://oreil.ly/26Fsx)	Да	Да	Нет
TRACE (https://oreil.ly/3vVWB)	Да	Да	Нет
PATCH (https://oreil.ly/Kn7az)	Нет	Нет	Нет

[a] Кэширование обычно применяется к операциям чтения и редко используется для операций записи, которые изменяют состояние сервера. Например, метод POST кэшируется только в том случае, если сервер явно предоставляет в ответе заголовки управления кэшем.

Create

В примере 5.5 показано, как реализовать CRUD-операцию Create с помощью HTTP-метода `POST`.

Пример 5.5. Реализация операции Create

```
# src/django/app/core/api/rest/v1/views/city_views.py

from core.helpers import error_response
from core.helpers import InputDataHandler
```

[1] См. RFC 9111, HTTP Caching («Кэширование в HTTP») (https://oreil.ly/TdtTO).

[2] Идемпотентность, безопасность и кэшируемость не гарантируются, если разработчик API не создает условия для этого.

```
from core.helpers import InputDataValidator
from core.helpers import IsSuperuser
from core.helpers import success_response
from core.helpers.serializers import CitySerializer
from core.helpers.utils import get_serializer_errors
from rest_framework.generics import GenericAPIView
from rest_framework.permissions import IsAuthenticated
...

class CityListView(GenericAPIView):  ❶
    allowed_methods = ["GET", "POST"]  ❷
    input_handler = InputDataHandler(InputDataValidator)  ❸
    permission_classes = [IsAuthenticated & IsSuperuser]  ❹
    serializer_class = CitySerializer  ❺

    def get_serializer_class(self):  ❻
        if self.request.version == "v1":
            return CitySerializer

        return super().get_serializer_class()

    def get_permissions(self):  ❼
        if self.request.method == "GET":
            return []

        return super().get_permissions()

    def post(self, request):  ❽
        serializer = self.get_serializer(data=request.data)  ❾

        if not serializer.is_valid():  ❿
            errors = get_serializer_errors(serializer)
            return error_response(errors, 400)

        try:
            serializer.save()  ⓫
        except Exception:
            return error_response(["Failed to create City object."], 500)

        return success_response(serializer.data, 201)  ⓬
    ...
```

❶ Класс `CityListView` связан с эндпоинтом `/api/cities` (см. выше пример 5.3). Класс `CityListView` наследует свойства класса из фреймворка DRF `GenericAPIView` (общее классовое представление, на основе которого создаются API-представления (https://oreil.ly/j1nux)).

❷ `allowed_methods` определяет список методов, разрешенных для ресурса. Для нашего представления метод `GET` извлекает список городов, а метод

`POST` создает ресурс города. Классовые представления DRF (https://oreil.ly/GNarL) требуют размещения всех HTTP-методов для конкретного эндпоинта в одном классе. С деталями этой функциональности можно ознакомиться на сайте DFR (https://oreil.ly/iQnUl).

❸ `input_handler` — это пользовательский класс, который мы реализуем для проверки данных в теле или параметре запроса.

❹ Поле `permission_classes` задает права доступа к представлению. Пользователь API должен пройти аутентификацию и иметь статус суперпользователя для работы с ресурсом города.

❺ `serializer_class` связывает представление с `CitySerializer`, который выполняет сериализацию и десериализацию объектов модели `City`. Сериализатор (https://oreil.ly/snCmV) преобразует модели Django (https://oreil.ly/P3ZzX) и QuerySets (https://oreil.ly/GuEfn) в типы данных Python, которые можно закодировать в JSON (https://www.json.org) и XML (https://www.w3.org/TR/xml).

❻ Метод `get_serializer_class()` меняет сериализатор по умолчанию на `CitySerializer` для версии API `v1`. В любых других версиях API этот метод по умолчанию вызывает сериализатор из родительского класса.

❼ Метод `get_permissions()` освобождает HTTP-запросы `GET` от проверки прав доступа и вводит проверку прав доступа для других методов (`permission_classes`).

❽ Метод `post()` отвечает на HTTP-запрос `POST`, давая разрешение на создание ресурса.

❾ Экземпляр `serializer` содержит данные из запроса клиента, которые используются для создания ресурса нового города.

❿ Этот блок кода проверяет данные, содержащиеся в запросе. Если содержимое невалидно, то сервер возвращает HTTP-ответ с кодом состояния `400 (Bad Request)`. Ошибки приложения, такие как `400 (Bad Request)`, по умолчанию не логируются. Это происходит потому, что DRF генерирует логи только для ответов, по которым были выброшены исключения[1].

⓫ `serializer` делает попытку сохранить объект города в базу данных. Если сохранение не было завершено, то возвращается HTTP-ответ с кодом состояния `500 (Internal Server Error)`.

⓬ Если сохранение завершено успешно, то возвращается HTTP-ответ с кодом состояния `201 (Created)`. Ответ содержит данные о новом созданном ресурсе.

---

[1] Подробнее о пользовательской обработке исключений в Django: https://oreil.ly/SPlNJ.

ЗАЩИТНОЕ ПРОГРАММИРОВАНИЕ

Обратите внимание на этапы, которые есть в каждом блоке кода в классах `CityView` и `CityListView`. Первый этап — валидация запроса, затем — действия с ресурсом и получение ресурса. На каждом этапе программа обрабатывает возникающие ошибки.

Такая последовательность этапов характерна для *защитного программирования (defensive programming)*, которое предполагает, что предоставленные входные данные могут быть невалидными или вредоносными и, следовательно, могут привести к некорректной работе ПО. Защитное программирование обеспечивает готовность к таким сценариям и обработку невалидного ввода без нарушения работы ПО.

Метод валидации ввода подразумевает проверки на нескольких уровнях: очистку, валидацию ввода, аутентификацию и авторизацию.

С помощью защитного программирования разработчики снижают риск неожиданного поведения системы из-за невалидного ввода, повышают ее безопасность и улучшают надежность. Недостаток подхода заключается в том, что он повышает сложность разработки.

В примере 5.6 клиент `curl` отправляет `POST`-запрос для создания нового ресурса города через эндпоинт `/api/cities`. В примере предполагается, что переменная окружения `$ACCESS_TOKEN` уже доступна, как будет показано в примере 5.18. Для отображения JSON-ответа в понятном человеку формате используется программа `jq` (https://jqlang.org).



Инструкции для получения ACCESS_TOKEN будут приведены в примере 5.18, а UUID, представленные в книге, будут отличаться от ваших.

Пример 5.6. Создание ресурса City с помощью HTTP-запроса POST (требует настройки из примера 5.2)

```
CREATE_CITY_PAYLOAD='{"name":"Copenhagen",
  "country":"Denmark",
  "region":"Europe",
  "timezone":"Europe/Copenhagen",
  "latitude":55.676100,
  "longitude":12.568300}'

docker compose exec app bash -c \
       "curl \
       --data '$CREATE_CITY_PAYLOAD' \
```

```
        --header 'Authorization: Bearer $ACCESS_TOKEN' \
        --header 'Content-Type: application/json' \
        --request 'POST' \
        --silent \
        'http://localhost:8000/api/cities' | \
        jq"

{
  "results": [
    {
      "uuid": "2a39b5ba-1dc5-4bd9-bba4-95c465067785",
      "name": "Copenhagen",
      "country": "Denmark",
      "region": "Europe",
      "timezone": "Europe/Copenhagen",
      "latitude": "55.676100",
      "longitude": "12.568300"
    }
  ]
}
```

Read

Операция Read применяется к эндпоинту `/api/cities` и предполагает, что сервер вернет названия нескольких городов. Детали реализации для получения нескольких ресурсов показаны в примере 5.7.

Помимо операции Read, мы реализуем эндпоинт с возможностью фильтрации, поиска и сортировки ресурсов. Конечная цель этой реализации — сделать так, чтобы API мог возвращать частичный ответ, который содержит не весь ресурс целиком, а только его подмножество[1].

Пример 5.7. Применение операции Read к списку ресурсов City

```
# src/django/app/core/api/rest/v1/views/city_views.py

from config import to_boolean
from core.helpers import InputDataHandler f
rom core.helpers import InputDataValidator
...

class CityListView(GenericAPIView):  ❶
    def get(self, request):
        fields = self.serializer_class.Meta.fields  ❷
        fields, fields_errors = self.input_handler.handle_fields(request, fields)
        search, search_errors = self.input_handler.handle_search(request, fields)
        sort, sort_errors = self.input_handler.handle_sort(request, fields)
        include_deleted = request.query_params.get("include_deleted", False)
```

[1] Частичный ответ — основная характеристика GraphQL.

```
        include_deleted = to_boolean(include_deleted)

        errors = get_errors(fields_errors + sort_errors + search_errors)  ❸

        if errors:
            return error_response(errors, 400)

        try:  ❹
            cities = CityRepository.get_all().order_by(sort).filter(**search)
            if not include_deleted:
                cities = cities.filter(deleted=False)
        except Exception:
            return error_response(["Failed to read from City database."], 500)

        return success_response_paginated(  ❺
            self.serializer_class, request, cities, fields
        )
    ...
```

❶ Класс `CityListView` определяет метод `get()`, который обрабатывает HTTP-запросы `GET` для получения списка городов.

❷ Этот блок кода валидирует параметры запроса `fields`, `search` и `sort` с помощью методов `handle_fields()`, `handle_search()` и `handle_sort()`. Каждый метод возвращает список Python, который содержит сообщение об ошибке валидации, если данные невалидны, и значение параметра запроса, извлеченное из имени поля, которое добавляется как суффикс к методу `handle_()`.

❸ Эта функция объединяет списки ошибок. Если обнаружены ошибки валидации, то возвращается HTTP-ответ с кодом состояния `400` (`Bad Request`).

❹ Этот блок кода описывает все города, выполняя сортировку и фильтрацию по полям в Django QuerySet (https://oreil.ly/uH38c). Переменная `include_deleted`, прочитанная из параметра запроса, определяет, должны ли мягко удаленные ресурсы добавляться в ответ, и по умолчанию имеет значение `False`. Метод `order_by()` выполняет сортировку, а `filter()` — фильтрацию по полям.

❺ Параметр запроса `fields`, передаваемый одноименной переменной, определяет, какие поля добавлять в ответ. Функция `success_response_paginated()` передает переменную `fields` как аргумент-ключ в `serializer(…, fields=fields)` (см. пример 5.12 далее). Таким образом, API возвращает частичный ответ, в котором есть только те данные, которые описаны в параметре запроса `fields`[1].

[1] В GraphQL возможность получения частичного ответа запрограммирована изначально (см. подробнее в главе 6).



DRF поставляется с классами для фильтрации (https://oreil.ly/bo7NQ), которые можно использовать при работе с GenericAPIView и ListAPIView (https://oreil.ly/2IuZJ).

Фильтрация, поиск, сортировка и получение частичного ответа — наиболее распространенные возможности, которые предоставляют серверы API (см. подраздел «Фильтрация» в главе 2). Фильтрация в API позволяет клиентам получать только те данные, которые соответствуют их потребностям. Поиск — это разновидность фильтрации, которая позволяет клиентам находить данные по точным совпадениям. Сортировка позволяет изменять порядок возвращаемых данных. Стандартный паттерн реализации REST API — добавление параметров запроса для настройки фильтрации. Например, параметр `fields` используется для ограничения возвращаемых данных, `search` — для поиска конкретных данных, а `sort` — для изменения порядка возвращаемых данных.

В примере 5.8 показано, как параметры запроса используются на эндпоинте `/api/cities`, а в примере 5.7 были представлены детали реализации. Используются следующие параметры запроса API: `fields`, `search_` с суффиксом одного из полей модели City (например, `search_region`; могут использоваться такие суффиксы, как `uuid`, `name`, `country`, `region`, `timezone`, `latitude`, `longitude`) и `sort`.

Пример 5.8. Применение операции Read с фильтрацией по полям к ресурсу City (требует настройки из примера 5.2)

```
QUERY_PARAMS="fields=name&search_region=Africa&sort=-name"
docker compose exec app bash -c \
      "curl --silent \
      'http://localhost:8000/api/cities?$QUERY_PARAMS' | \
      jq '.results'"

[
  {
    "name": "Cairo",
  }
]
```

Обратите внимание, что в ответе API возвращает только поле `name`. Из этого примера видно, что ответы REST API могут быть частичными.

Delete

Паттерн удаления ресурсов в API был описан в подразделе «Удаление» главы 2. Здесь же мы обсудим реализацию паттерна мягкого удаления.

В примере 5.9 показано, как удалить один ресурс City. Для этого примера нужно получить `ACCESS_TOKEN`, используя инструкции из примера 5.18.

Пример 5.9. Удаление ресурса City с помощью HTTP-метода DELETE (требует настройки из примера 5.2)

```
CITY_UUID=$(docker compose exec app bash -c \
  "curl --silent 'http://localhost:8000/api/cities?search_name=Mexico' | \
  jq --raw-output '.results[0].uuid'")  ❶

docker compose exec app bash -c \
       "curl \
       --header 'Authorization: Bearer $ACCESS_TOKEN' \
       --request 'DELETE' \
       --silent 'http://localhost:8000/api/cities/ | \
       $CITY_UUID?soft_delete=true' jq"  ❷

{
  "results": [
    {
      "uuid": "c1dc3ff5-5579-438b-bd8d-0aa0610cdb12",
      "name": "Mexico City",
      "country": "Mexico",
      "region": "North America",
      "timezone": "America/Mexico_City",
      "latitude": "19.432600",
      "longitude": "-99.133200",
      "deleted": true
    }
  ]
}
```

❶ Команда извлекает UUID для ресурса Mexico City и присваивает его переменной окружения `CITY_UUID`.

❷ Команда вызывает эндпоинт `/api/cities/uuid`, требующий аутентификации, с помощью HTTP-метода `DELETE`. Параметр запроса `soft_delete` определяет тип удаления: мягкий (ресурс помечается как удаленный, но физически не удаляется из базы данных) или жесткий (ресурс удаляется из базы данных).

В примере 5.10 приведены детали реализации удаления ресурса.

Пример 5.10. Реализация удаления ресурса City

```
# src/django/app/core/api/rest/v1/views/city_views.py

from config import to_boolean

class CityView(GenericAPIView):  ❶
    permission_classes = [IsAuthenticated & IsSuperuser]  ❷
```

```
    def delete(self, request, uuid):  ❸
        soft_delete = to_boolean(request.query_params.get("soft_delete", True))

        try:
            city = CityRepository.get_by_id(uuid)  ❹
        except Exception:
            return error_response(["Failed to read from City database."], 500)

        if not city: ❺
            return error_response(["Referenced City not found."], 404)

        double_soft_delete = (city and city.deleted and soft_delete)
        if double_soft_delete:  ❻
            return error_response(["Referenced City is already soft deleted."], 412)

        city = CityRepository.delete(uuid, soft_delete)  ❼
        if not city:
            return error_response(["Failed to delete City object."], 500)

        serializer = self.get_serializer(city)

        return success_response(serializer.data)  ❽
```

❶ Класс `CityView` наследует свойства класса `GenericAPIView` из DRF.

❷ Атрибут `permission_classes` гарантирует, что операцию `DELETE` могут выполнять только пользователи, которые прошли аутентификацию и имеют статус суперпользователя.

❸ Метод `delete()` обрабатывает HTTP-запросы `DELETE` для удаления ресурса, идентифицированного по `uuid`.

❹ Если при чтении ресурса (записи из базы данных) происходит сбой, то возвращается HTTP-ответ с кодом состояния `500` (`Internal Server Error`).

❺ Если ресурс не существует, то возвращается HTTP-ответ с кодом состояния `404` (`Not Found`).

❻ Если мягкое удаление ресурса уже было выполнено ранее, то программа возвращает HTTP-ответ с кодом состояния `412` (`Precondition Failed`).

❼ Переменная `soft_delete`, прочитанная из параметра запроса, определяет тип удаления ресурса. Если удаление не было завершено, то возвращается HTTP-ответ с кодом состояния `500` (`Internal Server Error`).

❽ Если удаление было завершено успешно, то возвращается HTTP-ответ, содержащий созданный ресурс, с кодом состояния `200` (`OK`).

Пагинация

Имеющиеся подходы к пагинации были описаны в подразделе «Пагинация» главы 2. Здесь же мы реализуем паттерн проектирования пагинации со смещением.

Пагинация позволяет серверу API ограничивать количество элементов, которое он возвращает клиенту за один раз. Она необходима при работе с большими наборами данных. Разделение набора на меньшие части повышает производительность API и снижает потребление памяти (как на клиенте, так и на сервере). Однако пагинация требует, чтобы клиенты API интерпретировали, анализировали и обрабатывали данные постранично, что может привести к несогласованности данных, полученных в ответ на разные запросы.

В REST API для WFS эндпоинт `/api/cities` возвращает ответ с данными, разбитыми на страницы. Чтобы протестировать нашу реализацию пагинации, вызовите WFS API, выполнив команду, показанную в примере 5.11, и сравните ответы для разных значений параметров запроса `page` и `page_size`. В примере 5.11 показана вторая страница данных, разбитых через пагинацию со смещением. Эта страница содержит только одну запись. Детали реализации пагинации со смещением см. далее в примере 5.12.

Пример 5.11. Использование пагинации в REST API (требует настройки из примера 5.2)

```
docker compose exec app bash -c \
       "curl --silent 'http://localhost:8000/api/cities?page=2&page_size=1' | \
       jq"
{
  "count": 6,  ❶
  "next": "http://localhost:8000/api/cities?page=3&page_size=1",  ❷
  "previous": "http://localhost:8000/api/cities?page_size=1",  ❷
  "results": [  ❸
    {
      "uuid": "89ee4c08-f483-4b73-a145-abb69892d39f",
      "name": "Istanbul",
      "country": "Turkey",
      "region": "Asia",
      "timezone": "Europe/Istanbul",
      "latitude": "41.008200",
      "longitude": "28.978400"
    }
  ]
}
```

❶ Поле `count` указывает общее количество доступных записей City, в нашем случае — шесть. Это позволяет клиентам узнать размер набора данных.

❷ Поля `next` и `previous` содержат адреса следующей и предыдущей страницы результатов. URL `next` ведет на третью страницу, а `previous` — на первую. Параметр `page_size` определяет количество элементов на одной странице.

❸ Массив `results` содержит фактические элементы данных, которые возвращает API. Каждый объект в массиве представляет город.

В примере 5.12 показана необходимая конфигурация для DRF и реализация функции `success_response_paginated()`, которая уже упоминалась в примере 5.7.

Пример 5.12. Реализация пагинации ответов

```
# src/django/app/config/base.py

REST_FRAMEWORK = {  ❶
    ...
    "DEFAULT_PAGINATION_CLASS": "rest_framework.pagination.PageNumberPagination",
    "PAGE_SIZE": 3,
    ...
}

# src/django/app/core/helpers/handlers/response_handler.py

from rest_framework.pagination import PageNumberPagination
...

def success_response_paginated(
    serializer, request, queryset, fields=None, status_code=200
  ):
    paginator = PageNumberPagination()  ❷
    paginator.page_size = int(request.query_params.get("page_size", 3))  ❸
    paginated_queryset = paginator.paginate_queryset(queryset, request)  ❹

    paginated_serializer = serializer(  ❺
      paginated_queryset, many=True, fields=fields
    )

    return paginator.get_paginated_response(paginated_serializer.data) ❻
```

❶ Словарь `REST_FRAMEWORK` содержит нужную конфигурацию DRF. `DEFAULT_PAGINATION_CLASS` указывает, что API будет использовать пагинацию со списком номеров страниц, что позволит клиентам запрашивать конкретные страницы данных, передавая номер страницы в запросе. Ключ `PAGE_SIZE` определяет количество элементов на одной странице.

❷ `PageNumberPagination` — это пагинатор DRF, который разделяет ответы на страницы.

❸ Параметр `page_size` определяет размер страницы. По умолчанию страница содержит три элемента, но ее размер можно контролировать через одноименный параметр запроса.

❹ Пагинатор получает исходные данные из параметра `queryset`, значение которого устанавливается в запросе.

❺ В результате мы получаем структуру данных Python `paginated_serializer`, на основе которых генерируются JSON-представления данных с пагинацией. Эти представления отправляются в качестве ответа API. Эта строка ограничивает исходный набор результатов запроса и оставляет лишь те данные, которые клиент запрашивает через параметр запроса `fields`.

❻ Из объекта `paginator` вызывается метод `get_paginated_response()`, который формирует итоговый ответ с постраничной разбивкой.

Ограничение частоты запросов

В одноименном подразделе главы 2 мы уже обсуждали ограничение частоты запросов — прием, который защищает API от перегрузки из-за чрезмерного количества запросов, предотвращает риски атак и обеспечивает справедливое использование API. Лимиты частоты запросов часто применяют, чтобы побудить бесплатных пользователей перейти на платные тарифы. Распространенная стратегия компаний — предоставлять полный доступ к функциональности публичных API, но ограничивать количество запросов в единицу времени, например в секунду, минуту, час или день. Это ограничение призвано стимулировать пользователей переходить на платную подписку, которая обычно предлагает более высокие лимиты.

DRF предоставляет инструменты для ограничения частоты запросов к API под альтернативным названием «троттлинг» (throttling). В примере 5.13 показана реализация троттлинга для эндпоинта `/api/cities`.

Пример 5.13. Реализация троттлинга

```
# src/django/app/config/base.py

REST_FRAMEWORK = {
    ...
    "DEFAULT_THROTTLE_CLASSES": ("rest_framework.throttling.AnonRateThrottle",), ❶
    "DEFAULT_THROTTLE_RATES": {"anon": "1000/minute"}, ❷
    ...
}
```

```
# src/django/app/core/api/rest/v1/views/city_views.py

from rest_framework.throttling import AnonRateThrottle
from rest_framework.decorators import throttle_classes
...

class CityListView(GenericAPIView):
    ...
    @throttle_classes([AnonRateThrottle])  ❸
    def get(self, request):
      ...
```

❶ Настройка `DEFAULT_THROTTLE_CLASSES` содержит класс `AnonRateThrottle`, который ограничивает количество запросов от анонимных клиентов.

❷ Настройка `DEFAULT_THROTTLE_RATES` определяет лимит частоты запросов для анонимных клиентов (`anon`), разрешая им выполнять до 1000 запросов в минуту. Идентификация клиента выполняется по IP-адресу[1].

❸ Декоратор `throttle_classes` применяет класс `AnonRateThrottle`, который защищает метод `get()`, предоставляющий доступ к эндпоинту `/api/cities`.

Кэширование API

Кэширование подробно рассматривалось в одноименном подразделе главы 2. Здесь же мы реализуем серверное кэширование для эндпоинта `/api/cities`, который предоставляет данные через идемпотентный и безопасный HTTP-метод `GET`. Механизм кэширования позволяет сохранять последовательные (одинаковые) запросы к API в хранилище кэша (обычно в базе данных, хранящейся в памяти), повышая отзывчивость API. Первый запрос клиента проходит через все уровни системы: кэш, бэкенд и базу данных, и на него формируется ответ. Последующие ответы на тот же запрос обслуживаются уже из кэша. Недостаток кэширования в том, что через некоторое время данные устаревают и их актуальность не гарантируется.

В примере 5.14 показано, как реализовать серверное кэширование с помощью DRF, используя базу данных Redis, хранящуюся в памяти (https://redis.io).

---

[1] Прокси-серверы или балансировщики нагрузки могут скрывать IP-адрес клиента. В таких случаях можно использовать постоянные или сессионные cookie-файлы, чтобы сохранить в браузере UUID клиента, сгенерированный сервером. Когда клиент выполняет последующие запросы, cookie-файлы отправляются обратно на сервер, который распознает клиента независимо от IP-адреса.

Пример 5.14. Реализация кэширования

```
# src/django/app/config/base.py

CACHES = {  ❶
    "default": {
        "BACKEND": "django.core.cache.backends.redis.RedisCache", \
        "LOCATION": f"{REDIS_PROTOCOL_SCHEMA}://{REDIS_HOST}:{REDIS_PORT}",
        "TIMEOUT": 300,
        "OPTIONS": {
            "db": 1
        }
    }
}

# src/django/app/core/api/rest/v1/views/city_views.py

from django.utils.decorators import method_decorator
from django.views.decorators.cache import cache_page
...

class CityListView(GenericAPIView):
    ...
    @method_decorator(cache_page(30))  ❷
    def get(self, request):
      ...
```

❶ Этот блок кода определяет настройки кэша для приложения Django. `BACKEND` указывает, что на бэкенде для реализации кэширования будет использоваться Redis. Остальные параметры задают расположение сервера Redis и номер базы данных (опция `db`).

❷ Чтобы кэшировать ответ на 30 секунд, универсальный декоратор метода `@method_decorator()` использует специализированный декоратор `@cache_page()` к методу `get()`, через который выполняется чтение данных с эндпоинта `/api/cities`.

Безопасность

В 2024 году компания Cloudflare опубликовала отчет *Application Security Report* («Отчет о безопасности приложений») (https://oreil.ly/fWdsO), в котором говорится, что 60 % динамического (некэшируемого) трафика генерируется в процессе взаимодействия с API. Увеличение этого показателя связано с тем, что API стал основным каналом связи между приложениями, базами данных, IoT-устройствами и другими системами.

Однако API могут выходить в продакшен без надлежащей каталогизации, аутентификации или аудита, что создает риски безопасности. Такое может происходить даже при подходе *API-first*, когда спецификация API создается до его реализации, и при использовании практик shift-left (см. подраздел «Развертывание» в главе 1). В рамках *подхода shift-left* тестирование, обеспечение безопасности и контроль качества реализуются на как можно более раннем этапе жизненного цикла разработки ПО или жизненного цикла API.

Помимо TLS, для обеспечения безопасности REST API используются стандартные методы аутентификации и авторизации. Один из методов, поддерживающих и аутентификацию, и авторизацию, — JSON Web Token (JWT, https://oreil.ly/UGaui). Этот метод позволяет аутентифицировать пользователя с помощью специального токена JWT, содержащего информацию о пользователе в виде «утверждений» (claims). Есть также методы OAuth 2.0 (https://oreil.ly/9IfVx) и OAuth 2.1 (https://oreil.ly/CgFlF), позволяющие выполнять только авторизацию. OAuth 2.0 содержит отдельные процедуры авторизации для веб-приложений, а также настольных и мобильных приложений. OpenID Connect (OIDC, https://oreil.ly/DaOnL) — еще один протокол аутентификации, основанный на фреймворке OAuth 2.0. В этом протоколе добавляется уровень идентификации, который предполагает проверку информации о пользователе и подтверждение личности. OIDC часто используют для приложений с единым входом (single sign-on, SSO).

Чтобы повысить уровень безопасности API, организации применяют такие инструменты, как OpenAPI, для стандартизации документации, а также практики аудита и платформы управления API для централизованного контроля (см. раздел «Управление, администрирование и платформа API» главы 1).

JSON Web Token (JWT)

Прежде чем приступить к реализации, разберем, что такое JWT. JSON Web Token (https://oreil.ly/vEOZD) — это структура из трех блоков, каждый из которых представляет собой строку в кодировке Base64url (https://oreil.ly/4MT8i). JWT содержит заголовок, полезную нагрузку и подпись, разделенные точками. В примере 5.15 показана закодированная версия JWT.



Отправка токенов в параметрах запроса может привести к их попаданию в историю браузера или логи сервера.

Пример 5.15. Пример JWT

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.
eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiwiaWF0IjoxNTE2MjM5MDIyfQ.
cThIIoDvwdueQB468K5xDc5633seEFoqwxjF_xSJyQQ
```



Декодировать JWT можно на сайте jwt.io (https://jwt.io). Используйте только проверенные интернет-инструменты и будьте осторожны, вводя в них личную информацию. В противном случае неавторизованные третьи лица могут получить доступ к конфиденциальной информации.

Заголовок и полезная нагрузка JWT содержат JSON-объекты в кодировке Base64url. JSON-объект в заголовке содержит информацию о типе токена и об используемом алгоритме шифрования, а в полезной нагрузке — информацию о пользователе и сроке действия токена (эту информацию называют утверждениями). Последний элемент JWT — подпись HMAC, которая генерируется на основе заголовка и полезной нагрузки, сконкатенированных через точку, и секретного ключа. Для генерации подписи, имеющей длину 256 бит, используется функция Secure Hash Algorithm (SHA). В табл. 5.5 показан декодированный JWT из примера 5.15.

Таблица 5.5. Структура JWT

Заголовок	Полезная нагрузка	Подпись

`{`
`  "alg": "HS256",`
`  "typ": "JWT"`
`}`
	`{`
	`  "sub": "1234567890",`
	`  "name": "John Doe",`
	`  "iat": 1516239022`
	`}`
		`HMAC_SHA256(secret,`
		`base64url(header) + "." +`
		`base64url(payload))`

JWT имеет ряд преимуществ.

Безопасность

Чтобы обеспечить целостность и подлинность JWT, для генерации подписи используются криптографические алгоритмы. Они могут быть симметричными, например HMAC (https://oreil.ly/sPSk5), или асимметричными, наподобие RSA (https://oreil.ly/es87d) или ECDSA (https://oreil.ly/soU1X). В последних используется два ключа: закрытый и открытый. Кроме того, JWT содержит утверждения, например `exp` (expiration, срок действия) — оно фиксирует время, после которого токен становится недействительным. Если система, выпустившая токен, явно не обновит время его действия, то его нельзя будет использовать для получения доступа.

Отсутствие сохранения состояний

JWT не хранит данные о сессиях на сервере. По этой причине JWT подходит для аутентификации в распределенных системах, которые используют API для междоменного взаимодействия. Однако — и это важно — состояния не сохраняются только в тех случаях, когда JWT используется для аутентификации. Реализация таких функций, как отзыв токенов или контроль

доступа к токенам через создание списков разрешений и запретов, неизбежно предполагает использование хранилища.

Самодостаточность

JWT содержит такие утверждения, как идентификатор пользователя, роли и разрешения. Благодаря этому серверу может рассматривать действующий токен как подтверждение аутентификации, что устраняет потребность в частых обращениях к базе данных.

Переносимость

Кодировка JWT содержит только символы, разрешенные синтаксисом URL[1]. Кроме того, она совместима с текстовыми протоколами, такими как HTTP. JWT можно передавать в параметре запроса, теле или заголовке HTTP-запроса.

Кастомизация

JWT содержит утверждения, которые можно адаптировать под конкретное приложение (роль пользователя, срок действия токена).

Теперь, когда вы знаете, что такое JWT, реализуем с его помощью защиту REST API WFS.

Для реализации JWT мы будем использовать Python-пакет djangorestframework-simplejwt (Simple JWT) (https://oreil.ly/OV8G3), который совместим с нашим основным фреймворком DRF (https://www.django-rest-framework.org). При отсутствии такого пакета функциональность JWT пришлось бы реализовывать с нуля. Чтобы интегрировать Simple JWT в DRF, в конфигурации DRF нужно установить класс аутентификации по умолчанию (пример 5.16).

Пример 5.16. Конфигурация JWT

```
# src/django/app/config/base.py

REST_FRAMEWORK = {  ❶
    ...
    "DEFAULT_AUTHENTICATION_CLASSES": (
        "rest_framework_simplejwt.authentication.JWTAuthentication",
    ),
    ...
}

SIMPLE_JWT = {  ❷
    "AUTH_TOKEN_CLASSES": ("rest_framework_simplejwt.tokens.AccessToken",),
    "USER_ID_CLAIM": "id",
}
```

[1] Существует набор безопасных символов URL (unreserved characters) (https://oreil.ly/IFUjl).

❶ Словарь `REST_FRAMEWORK` содержит настройки, которые настраивают поведение DRF. После установки класса `JWTAuthentication` в `DEFAULT_AUTHENTICATION_CLASSES` DRF будет использовать класс `JWT` из пакета Simple JWT для аутентификации запросов к API.

❷ Конфигурация гарантирует, что библиотека Simple JWT использует класс `AccessToken` для обработки токенов аутентификации. Он отвечает за генерацию, валидацию и декодирование токенов доступа JWT. `USER_ID_CLAIM` переопределяет имя ключа, содержащего ID пользователя в полезной нагрузке, на `id`.

После настройки классов аутентификации для DRF и Simple JWT нужно создать эндпоинты для управления токеном JWT (пример 5.17). Созданные эндпоинты позволяют получать, обновлять, отзывать и проверять JWT.

Пример 5.17. Эндпоинты JWT

```
# src/django/app/config/urls.py

...
urlpatterns = [
    ...
    path("api/jwt/obtain", TokenObtainPairView.as_view()),
    path("api/jwt/refresh", TokenRefreshView.as_view()),
    path("api/jwt/revoke", TokenRevokeView.as_view()),
    path("api/jwt/verify", TokenVerifyView.as_view()),
    ...
]
...
```

В примере 5.18 показано, как получить JWT. Обратите внимание, что ответ содержит как токен доступа, так и токен обновления. Первый предоставляет доступ к защищенным ресурсам, а второй используется для получения нового токена доступа, когда истекает срок действия текущего. После получения токена доступа можно взаимодействовать с эндпоинтами API, требующими его наличия. Для этого нужно сформировать запрос так, чтобы его заголовки содержали строку авторизации вида: `"Authorization": "Bearer ACCESS_TOKEN"` (`ACCESS_TOKEN` нужно заменить реальным токеном доступа).

Пример 5.18. Получение JWT-токена доступа для REST API с помощью curl (требует настройки из примера 5.2)

```
CREDENTIALS_PAYLOAD='{"username":"admin","password":"admin"}'  ❶
ACCESS_TOKEN=$(docker compose exec app bash -c \
  "curl \
  --data '$CREDENTIALS_PAYLOAD' \
  --header 'Content-Type: application/json' \
  --request 'POST' \
```

```
  --silent 'http://localhost:8000/api/jwt/obtain' | \
  jq --raw-output '.access'")  ❷
echo $ACCESS_TOKEN  ❸
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.
eyJ0b2tlbl90eXBlIjoiYWNjZXNzIiwiZXhwIjoxNzMy...
```

❶ Переменная окружения `CREDENTIALS_PAYLOAD` содержит данные учетной записи пользователя в формате JSON.

❷ Переменная окружения `ACCESS_TOKEN` содержит JWT-токен доступа. Токен извлекается из эндпоинта `/api/jwt/obtain`. Команда завершается вызовом утилиты `jq`, которая извлекает значение токена доступа из поля `access`.

❸ Команда `echo` выводит JWT.

В данном случае JWT используется для подтверждения личности пользователя, который взаимодействует с API. При анализе самого токена невозможно отличить реального пользователя от злоумышленника, использующего перехваченный токен. Один из способов решения этой проблемы — сокращение времени действия токена (превращение его в *краткосрочный токен доступа*). Таким образом, уровень безопасности повышается, но возникает необходимость генерации новых токенов[1]. Кроме того, даже при использовании краткосрочного токена риск перехвата сохраняется.

Transport Layer Security (TLS)

В этом подразделе мы обсудим, как использовать протокол TLS для защиты HTTP-трафика в приложении Django для WFS.

САМОПОДПИСАННЫЕ СЕРТИФИКАТЫ И СЕРТИФИКАТЫ, ПОДПИСАННЫЕ ЦЕНТРОМ СЕРТИФИКАЦИИ

В этом подразделе для TLS используется самоподписанный сертификат (https://oreil.ly/5wJjQ). Главный его недостаток — по умолчанию браузеры ему не доверяют. Посетить сайт, зашифрованный с его помощью, можно только в том случае, если пользователь самостоятельно подтвердит переход (примет сертификат).

Самоподписанных сертификатов, как правило, избегают. Однако есть несколько сценариев, для которых они подойдут. Самоподписанные серти-

[1] См. пост What Are Refresh Tokens and How to Use Them Securely («Что такое refresh-токены и как использовать их безопасно») Дэна Ариаса (Dan Arias) и Сэма Беллена (Sam Bellen) в блоге Auth0 (https://oreil.ly/bFs0K).

фикаты могут использоваться при внутреннем тестировании или в среде разработки, где доверие между сторонами уже установлено. Не рекомендуется использовать такие сертификаты в продакшен-среде. В ней для шифрования трафика приложения с помощью TLS применяются *сертификаты, подписанные доверенным центром сертификации (CA)*.

Такие сертификаты выпускаются доверенными организациями, которые проверяют личность владельца (субъекта) сертификата. Есть четыре уровня проверки, и для каждого из них требуется разный объем информации о субъекте сертификата[1].

Чтобы получить сертификат, подписанный CA, можно использовать бесплатный сервис Let's Encrypt (https://letsencrypt.org), который предлагает автоматизированные инструменты для получения и обновления сертификатов. Кроме того, можно использовать сертификаты от облачных провайдеров или приобрести сертификат у проверенного провайдера.



Подробности о реализации TLS для WFS описаны в репозитории с кодом книги.

Чтобы включить или отключить TLS для сервиса WFS, нужно установить соответствующее значение переменной окружения `TLS_ENABLE=1` или `TLS_ENABLE=0` в файле `src/django/compose.yaml`. Далее нужно перезапустить WFS с помощью команды **`docker compose up --detach --wait`**. Чтобы убедиться, что шифрование трафика началось, перейдите по адресу https://localhost:8000[2].



При локальном переходе в https://localhost:8000 (не в GitHub Codespaces) браузер покажет предупреждение о том, что соединение не защищено. Чтобы разрешить соединение в локальной среде, добавьте SSL-сертификат приложения в список исключений браузера.

На рис. 5.5 показаны зашифрованный HTTP-трафик WFS и информация о самозаверенном TLS-сертификате. Обратите внимание, что используется HTTP-схема `https://`, а не `http://`.

---

[1] См. статью What's the Difference Between DV, OV, IV and EV Certificates («В чем разница между сертификатами DV, OV, IV и EV») (https://oreil.ly/nHsEg) на сайте SSL.com.

[2] При работе с GitHub Codespaces используется уникальный URL, содержащий имя Codespace вместо адреса localhost. Этот URL привязан к порту 8000, и его можно найти на вкладке PORTS в VS Code.

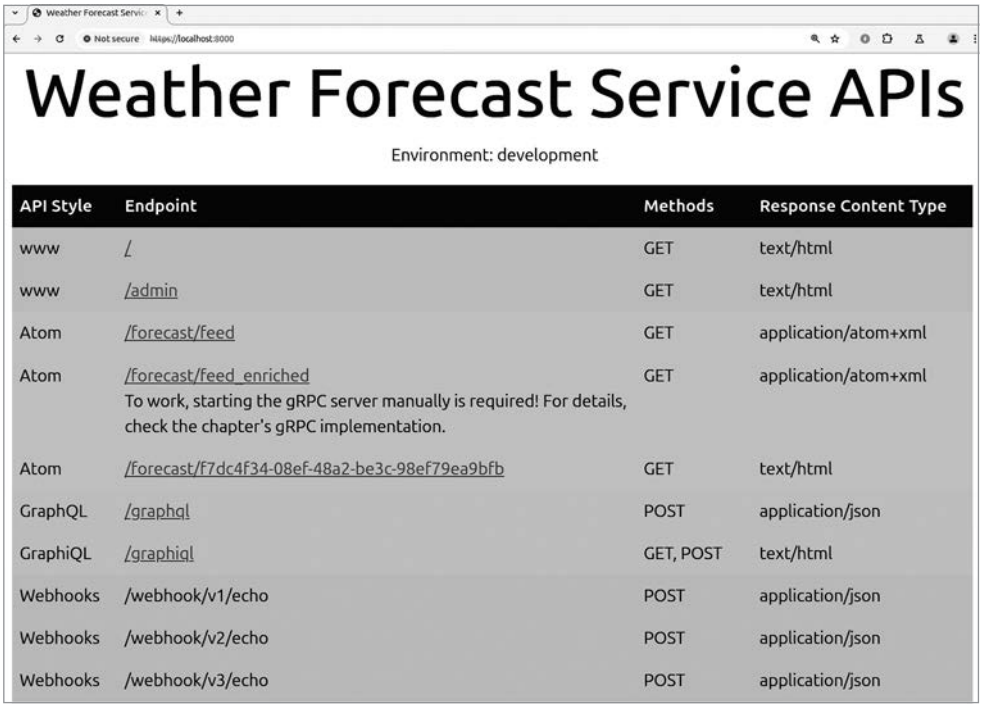


Рис. 5.5. Зашифрованное соединение WFS

Документация

В этом разделе мы обсудим создание документации для REST API. Как бы сильно разработчики ни старались сделать API интуитивно понятным, сохраняется риск того, что пользователи могут задействовать API некорректно. По этой причине документация API необходима. В противном случае пользователи предпочтут другой API.

Спецификация API — это контракт между провайдером и пользователем API. Провайдер использует спецификацию, чтобы предоставить описание API, а пользователь — чтобы понять, как работает API.

Спецификация OpenAPI

Спецификация OpenAPI (OpenAPI Specification, OAS, https://oreil.ly/s7dTX), ранее известная как Swagger, — самый популярный инструмент для документирования API, работающих на протоколе HTTP. Для создания текстовых документов со спецификацией API в OAS используются форматы, удобные

для чтения человеком, такие как YAML (https://yaml.org) или JSON (https://www.json.org).

Документ OAS можно сделать интерактивным с помощью инструментов ReDoc (https://redocly.com), Postman (https://www.postman.com) и Swagger UI (https://oreil.ly/4v1Yo). Благодаря поддержке разных инструментов и широкому распространению в сфере разработки ПО OAS получила большую популярность среди разработчиков. Во фреймворке FastAPI (https://oreil.ly/Ez21i) OAS является стандартом документации по умолчанию.



OAS широко применяется для документирования синхронных API, однако для асинхронных API возможности инструмента ограничены. Если ваши API работают асинхронно, то можно использовать AsyncAPI (https://www.asyncapi.com).

В примере 5.19 показан фрагмент документации для API WFS, созданный с помощью OAS.

Пример 5.19. Фрагмент спецификации OpenAPI для WFS

```
# src/django/docs/api/rest/api.resource-v1.schema.yaml

openapi: 3.1.0  ❶
info:  ❷
  title: Weather Forecast Service - REST API
  version: v1
  description: |
    Documentation of Weather Forecast Service (WFS) REST API.
tags:  ❸
  - name: Public
    description: Operations available to all users
paths:  ❹
  /api/cities:  ❺
    get:
      summary: Obtain all cities
      operationId: api_v1_cities_retrieve
      description: Obtain all cities.
      tags:  ❸
        - Public
      parameters:  ❻
        - name: version
          in: query
          description: API version
          required: false
          schema:
            type: string
            default: v1
          style: form
```

```
      explode: false
    - name: fields
      in: query
      description: Fields to return
      required: false
      schema:
        type: string
        default: uuid,name,country,region,timezone,latitude,longitude
      style: form
      explode: false
    - name: search_{key}
      in: query
      description: |
        Search parameter with a dynamic key.
        Replace `{key}` with the actual field name, e.g.,
        `uuid`, `name`, `country`, `region`, `timezone`, etc.
      required: false
      schema:
        type: string
  responses:  ❼
    200:
      description: OK
    400:
      description: Bad request. Invalid data provided
    500:
      description: Internal server error
```

❶ Поле `openapi` указывает версию OpenAPI, которая используется в спецификации OAS.

❷ Поле `info` содержит метаданные API, такие как заголовок (`title`), версия (`version`) и описание (`description`).

❸ Теги (`tags`) используются для группировки операций. Тег `Public` указывает на публичную доступность API. Этот тег используется далее в документе для HTTP-метода `GET`.

❹ Поле `paths` определяет доступные эндпоинты и операции над ними.

❺ Путь `/api/cities` определен HTTP-методом `GET`. Этот эндпоинт предоставляет клиенту данные по всем городам, информация о которых доступна в WFS.

❻ Поле `parameters` описывает доступные параметры запроса, включая `version` (описывающий версию API), `fields` и динамический параметр `search_{key}`.

❼ Поле `responses` определяет диапазон возможных ответов API.

OpenAPI также предоставляет интерактивный редактор Swagger Editor, в котором разработчики могут создавать и редактировать спецификации API. Swagger Editor — самый простой способ начать работу с OAS. Редактор доступен по

адресу https://editor-next.swagger.io. Чтобы изучить его возможности, попробуйте вставить в него спецификацию OAS из примера 5.19. Возможный результат показан на рис. 5.6.

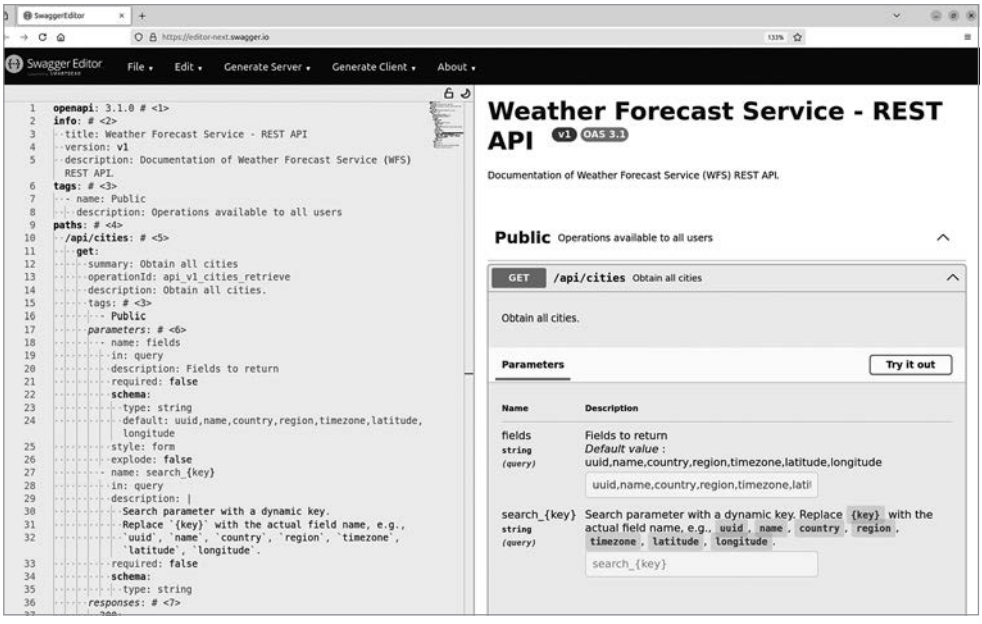


Рис. 5.6. Онлайн-редактор Swagger Editor

Если вы работаете с OAS локально, то вместо онлайн-редактора можно использовать Swagger UI (https://oreil.ly/qaZQF). Чтобы увидеть, как работает этот инструмент, выполните инструкции из примера 5.20.

Пример 5.20. Локальный запуск Swagger UI с помощью Docker

```
curl --silent --output schema.yaml http://localhost:8000/api/docs/v1/schema/manual
docker run --env SWAGGER_JSON_URL=schema.yaml \
       --publish 127.0.0.1:8888:8080 \
       --rm \
       --volume ${PWD}/schema.yaml:/usr/share/nginx/html/schema.yaml \
       swaggerapi/swagger-ui
```

При локальном запуске Swagger UI отобразит документ со спецификацией OAS (рис. 5.7)[1].

[1] При работе с GitHub Codespaces используется уникальный URL, содержащий имя Codespace вместо адреса localhost. Этот URL привязан к порту 8888, и его можно найти на вкладке PORTS в VS Code.

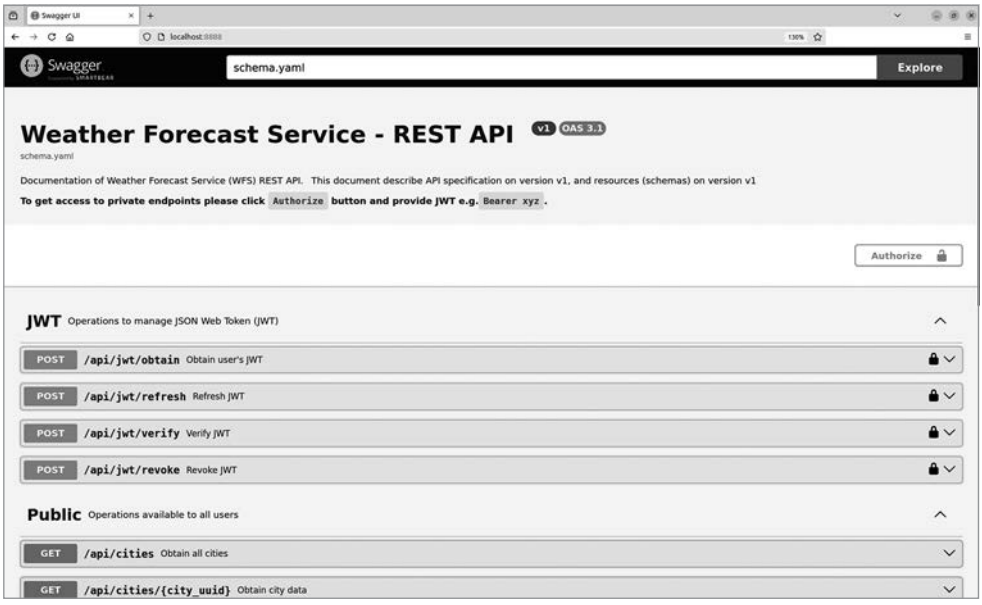


Рис. 5.7. Локальный запуск Swagger UI

Три подхода к созданию спецификации API

Есть три основных подхода к созданию спецификации.

Автоматическая генерация

Спецификация генерируется автоматически на основе исходного кода. Например, в WFS генерация выполняется с помощью библиотеки `drf_spectacular` (https://oreil.ly/XNVzo), созданной для работы с фреймворком DRF. Этот подход прост в реализации, но имеет недостатки. Автоматически сгенерированная спецификация не дает полного представления об API: инструменту, генерирующему спецификацию, часто не хватает контекста, и чтобы заполнить пробелы в документации, разработчику приходится дополнять код вручную.

Создание спецификации вручную

Спецификацию создают по принципу *bottom-up*: сначала пишется и реализуется код API, а затем на основе работающего кода с нуля создается спецификация. При таком подходе процесс создания спецификации полностью подконтролен разработчику и не ограничен возможностями автоматического генератора. Однако это трудоемкий процесс, который нередко приводит

к возникновению несоответствий между спецификацией и фактической реализацией API. В предыдущем подразделе, в примере 5.19, мы использовали спецификацию, созданную вручную.

Гибридный подход

При таком подходе первичная структура спецификации генерируется автоматически, а разработчики вручную заполняют недостающие детали. Как и при создании спецификации вручную, в этом подходе возможны несоответствия между реализацией и спецификацией.

В оставшейся части этого раздела рассматриваются автоматический и гибридный методы документирования REST API с помощью OAS.

Первый шаг — интеграция библиотеки `drf-spectacular` (https://oreil.ly/b0IRs) в проект Django (пример 5.21). После установки Python-пакета `drf-spectacular` его необходимо добавить в список `INSTALLED_APPS` Django.

Пример 5.21. Конфигурация Django drf-spectacular

```
# src/django/app/config/base.py

INSTALLED_APPS = [
    ...
    "drf_spectacular",
    ...
]
```

Следующий шаг — создание эндпоинтов, которые будут обслуживать OAS. Необходимая конфигурация показана в примере 5.22.

Пример 5.22. Создание эндпоинтов OAS

```
# src/django/app/config/urls.py

from drf_spectacular.views import SpectacularAPIView
from drf_spectacular.views import SpectacularSwaggerView
from core.www import swagger_rest_schema_view
...

urlpatterns = [
  ...
  path("api/docs/v1", SpectacularSwaggerView.as_view(url_name="oas_manual")),     ❶
  path("api/docs/v1/schema/auto", SpectacularAPIView.as_view(), name="oas_auto"), ❷
  path("api/docs/v1/schema/manual", swagger_rest_schema_view, name="oas_manual"), ❸
  ...
]
```

❶ Эндпоинт `/docs/api/v1` предоставляет доступ к Swagger UI (https://oreil.ly/Rr8OO) по адресу http://localhost:8000/api/docs/v1. Сгенерированная документация показана на рис. 5.7.

❷ Эндпоинт `/docs/api/v1/schema/auto` предоставляет автоматически сгенерированную спецификацию OAS. Чтобы увидеть ее, выполните команду **`curl --silent --output wfs.auto.schema.yaml http://localhost:8000/api/docs/v1/schema/auto`** и откройте файл `wfs.auto.schema.yaml`.

❸ Эндпоинт `/docs/api/v1/schema/manual` реализует гибридный подход к созданию спецификации: сначала спецификация генерируется автоматически, а затем ее обновляют вручную. Код функции `swagger_rest_schema_view()` читает статический файл OAS и возвращает его как HTTP-ответ (см. `src/django/app/core/www/views.py`). Чтобы увидеть спецификацию, выполните команду **`curl --silent --output wfs.manual.schema.yaml http://localhost:8000/api/docs/v1/schema/manual`** и откройте файл `wfs.manual.schema.yaml`.



Чтобы увидеть различия между автоматически сгенерированной и гибридной OAS, откройте Swagger Editor (https://editor-next.swagger.io) одновременно на двух вкладках в браузере. На одну скопируйте содержимое файла wfs.auto.schema.yaml из примера 5.22, а на другую — содержимое файла wfs.manual.schema.yaml. Кроме того, документацию можно сгенерировать локально с помощью Docker-образа swaggerapi/swagger-ui (https://oreil.ly/G5e0b).

REST API: преимущества и недостатки

Начнем обсуждение этой темы с сильных сторон REST API.

Масштабируемость

Архитектурный стиль REST предназначен для масштабируемых интернет-систем. В REST отсутствует сохранение состояний, поэтому система может обрабатывать каждый запрос независимо, что упрощает горизонтальное масштабирование.

Совместимость

REST обеспечивает совместимость систем за счет использования стандартизированного протокола HTTP. Поскольку HTTP поддерживается многими языками программирования, библиотеками и платформами, то REST API могут соединять системы, основанные на разных технологиях. Кроме того, отсутствие сохранения состояний упрощает интеграцию, позволяя компонентам разных приложений и систем взаимодействовать между собой.

Простота отладки и широкий инструментарий

Как правило, в REST API используются текстовые форматы кодирования, такие как JSON (https://www.json.org) или XML (https://oreil.ly/7WBP2). Эти форматы относятся к удобным для чтения человеком и поддерживаются разными языками программирования, фреймворками и библиотеками. В отличие от двоичных форматов, таких как Protocol Buffers (https://protobuf.dev), которые требуют специальных инструментов для чтения, текстовый формат кодирования упрощает отладку системы и отслеживание потоков данных. Кроме того, стиль REST API поддерживается многими инструментами (среди них серверы, инструменты кэширования, балансировщики нагрузки, прокси-серверы, брандмауэры, системы мониторинга, ПО для тестирования, утилиты отладки, инструменты документирования и др.).

Однако у REST API есть и недостатки.

Ограниченный набор методов

В REST API используется несколько основных HTTP-методов (см. выше, табл. 5.4). Соответственно, многие операции над ресурсами выходят за рамки словаря HTTP. Кроме того, ограничение количества доступных методов сказывается на структуре API (в REST API широко используются вложенные ресурсы, а один и тот же базовый HTTP-метод может применяться для реализации разных операций).

Проблемы безопасности

Функция «код по требованию» (code-on-demand, необязательная функция RESTful API) создает риски для безопасности API. Эта функция позволяет серверу передавать код клиенту в теле HTTP-ответа. Далее клиент выполняет этот код. Если злоумышленнику удастся внедрить в код вредоносный компонент (через межсайтовый скриптинг или инъекционную атаку), то возникает риск обхода механизмов авторизации и утечки данных.

Ограничения, связанные с отсутствием сохранения состояния

Как уже говорилось, REST API не сохраняет предыдущие состояния. Следовательно, каждый клиентский запрос содержит всю информацию, необходимую для его обработки. Недостаток такого подхода заключается в том, что объем данных, передаваемых в запросе, увеличивается. Помимо этого, REST API не подходит для приложений, которым требуется сохранение состояния (например, приложение интернет-магазина, где клиент выполняет цепочку запросов, при этом состав корзины пользователя не меняется). Сохранение состояний в REST API обеспечивается на стороне клиента и реализуется локально: с помощью cookie-файлов или локального хранилища.

Отсутствие стандарта управления версиями

Паттерны версионирования не закладывались при разработке стиля REST. Вместе с тем REST API часто требует критических изменений, к которым чувствительны клиенты. Например, Graph API от Meta, спроектированный в стиле REST (https://oreil.ly/4l1WZ), с момента выпуска версии 1.0 в 2010 году был изменен 37 раз и на момент написания книги имеет версию 22.0. Наличие HATEOAS может снизить вероятность критических изменений, но не устраняет их полностью. Клиенты могут видеть ссылку на новое доступное действие, но не понимать его смысл. Подробнее управление версиями API было описано в подразделе «Версионирование API» в главе 2.

Когда использовать REST

REST подходит для распределенных систем, особенно для систем, которые работают без сохранения состояния. Однако важно помнить, что для реализации распределенных систем одного стиля API недостаточно. Например, REST можно использовать для обработки *внешнего (вертикального) сетевого трафика*, когда запросы поступают в систему извне или наоборот (например, пользователь из интернета делает запрос в центр данных). Этот тип трафика получил название north-south. Для обработки *внутреннего (горизонтального) сетевого трафика* между серверами внутри сети (тип трафика east-west) может использоваться другой стиль API, например gRPC (см. главу 8) или Messaging (см. главу 11).

Ранее уже говорилось, что REST универсален и его поддерживают множество клиентов, платформ, языков программирования и инструментов. По этой причине REST используется для кросс-платформенной интеграции внутренних и внешних систем (в том числе устаревших), которые часто не контролируются разработчиком и требуют разных форматов данных.

Кроме того, в качестве транспорта в REST используется HTTP. Большинство бэкенд- и фронтенд-разработчиков, архитекторов ПО и администраторов защиты хорошо знают этот протокол, так как работали с ним на протяжении всей карьеры.

REST API поддерживает разные текстовые форматы кодирования, которые можно сжимать, а также двоичное кодирование данных. Отсутствие сохранения состояния делает каждый запрос к REST API самодостаточным, что позволяет системе распределять запросы между несколькими серверами. Поддержка кэширования дает возможность клиентам снижать время задержки для последующих ответов сервера. Наконец, REST API поддерживает технологию CDN (Content Delivery Network), которая позволяет кэшировать ответы на сервере в регионе пользователя.

Упражнения

1. В разделе «Документация» этой главы описано, как создать спецификацию OAS с помощью гибридного подхода и как конвертировать ее в формат, читаемый Swagger UI, с помощью специального модуля Python. В этом упражнении ваша задача — конвертировать OAS в формат Swagger UI (https://oreil.ly/6lLKK), не используя библиотеку `drf-spectacular`. Файл Swagger UI должен быть по-прежнему доступен по адресу localhost:8080.
2. Подсказка: используйте Docker-образ `swaggerapi/swagger-ui` (https://oreil.ly/wn4Li). Ответ к этому упражнению находится в файле `src/django/docs/solutions/rest-exercise.md`.
3. Для документирования API обычно используют OAS. В этом упражнении попробуйте альтернативный инструмент — ReDoc (https://redoc.ly). Задача — конвертировать OAS в HTML-страницу с помощью консольной программы `redoc`. Ответ к этому упражнению находится в файле `src/django/docs/solutions/rest-exercise.md`.

Резюме

В этой главе мы поговорили об архитектурном стиле REST API — самом популярном стиле веб-API. Мы начали с происхождения стиля REST, который создавался для реализации масштабируемых сетевых приложений в интернете. REST был основан на HTTP и предполагал использование технологии HATEOAS, которая отсутствует в большинстве современных реализаций.

Далее мы реализовали REST API (без HATEOAS) для сервиса прогноза погоды, используя Django REST Framework, защитили его с помощью TLS и JWT, а также задокументировали API, используя спецификацию OpenAPI (ранее известную как Swagger).

Помимо практических примеров реализации REST API, мы рассмотрели плюсы и минусы этого стиля, на основе которых вы можете решить, подойдет ли стиль REST для разработки вашего API. REST будет нашей точкой отсчета: в последующих главах мы будем часто ссылаться на его особенности и сравнивать с ним другие стили API. Следующая глава будет посвящена стилю GraphQL.

ГЛАВА 6

GraphQL

В REST API пользователь получает данные, отправляя запрос к конкретному эндпоинту, который описывает ресурс, и чаще всего возвращаемые данные имеют фиксированную структуру. Есть также стиль API на основе запросов: он позволяет создавать API, в которых возвращаемые данные точно соответствуют структуре запроса. В этой главе мы познакомимся с таким стилем — GraphQL.

Хотя самой популярной технологией API остается REST, GraphQL тоже используется широко[1]. Эту технологию выпустила компания Facebook (Meta) в 2015 году, и изначально это был проект с открытым исходным кодом. Позже для QraphQL была создана независимая площадка GraphQL Foundation. GraphQL — это *язык запросов* для API и *серверная среда* выполнения запросов, где используется *система типов*, которая определяет структуру данных[2]. Хотя GraphQL не привязан к конкретному протоколу передачи данных, чаще всего для обмена данными между клиентом и сервером используется HTTP. С помощью операции подписки (subscription) GraphQL также поддерживает одновременную двунаправленную передачу данных, которая часто выполняется по протоколу WebSocket (см. главу 10).

В этой главе мы обсудим происхождение стиля GraphQL, его спецификацию, практики обеспечения безопасности, а также плюсы и минусы этого стиля. Далее мы реализуем GraphQL API для сервиса WFS и сравним его со стилем REST API из главы 5.

---

[1] Согласно отчету State of the API Report («Отчет о состоянии API») Postman (https://oreil.ly/3LG67), GraphQL занимает третье место по популярности среди технологий API.

[2] Ознакомьтесь с разделом Introduction to GraphQL («Введение в GraphQL») на сайте GraphQL (https://oreil.ly/BclKH).

Ограничения REST

Несмотря на большую популярность, у REST API есть слабые стороны.

REST API часто жестко привязаны к структуре данных, которые они предоставляют: их ресурсы — это модели данных, доступные через эндпоинты. По этой причине внесение изменений в данные часто требует изменений в самом API. Чтобы изменить формат предоставления данных в API, может потребоваться выпустить новую версию API, что, в свою очередь, усложнит поддержку старых клиентов.

Кроме того, в REST API пользователи сталкиваются с проблемой *избыточной* или *недостаточной* выборки данных. При избыточной выборке *(over-fetching)* сервис API возвращает больше данных, чем нужно клиенту. При недостаточной выборке *(under-fetching)* API, напротив, предоставляет недостаточное количество данных, и клиенту приходится отправлять дополнительные запросы.

Наконец, в зависимости от реализации в REST API может отсутствовать HATEOAS или встраивание связанных данных в ответ. Из-за этого в API возникают трудности с управлением связями между ресурсами. Например, если клиенту нужны данные о пользователе вместе с его постами и связанными с ними комментариями, то потребуется несколько запросов к API, а их точный формат придется искать в документации. Можно реализовать специальный эндпоинт, который будет агрегировать эти данные, но такой подход усложнит реализацию и приведет к зависимости клиента от конкретного URL и формата ответа. Для упрощения навигации можно также реализовать HATEOAS, чтобы клиент получал ссылки на связанные ресурсы, или встраивание данных, которое позволяет сократить количество вызовов API, но увеличит объем передаваемых данных.

GraphQL решает сразу несколько проблем REST API, представляя связи между ресурсами в виде графовой модели и позволяя клиентам получать только конкретные, нужные им данные. Как и в REST, в GraphQL сохраняется риск избыточной или недостаточной выборки данных, однако этот риск значительно ниже, поскольку архитектура GraphQL позволяет клиентам точно указывать требования к данным.

Графовое мышление

Переход от REST к новому стилю GraphQL требует изменений на концептуальном уровне. Рассмотрим две основные характеристики GraphQL: *язык запросов* и *статическую типизацию*.



Если вы уже знакомы с основами JavaScript и только начинаете работать с GraphQL, рекомендуем к просмотру курс GraphQL for Beginners with JavaScript («GraphQL для начинающих на JavaScript») (https://oreil.ly/yDch9).

Согласно стандартной классификации языки программирования делятся на два типа: императивные и декларативные.

Императивные языки программирования требуют от разработчика пошагового описания того, *как* достичь желаемого результата. *Декларативные языки программирования*, напротив, не требуют явной последовательности шагов реализации. Вместо этого в них описывается желаемый *результат*, а среда выполнения определяет необходимые шаги. В декларативных языках часто используются запросы, которые абстрагируются от деталей процедурной логики. Примером декларативного языка является язык структурированных запросов (Structured Query Language, SQL), предназначенный для систем управления реляционными базами данных (СУБД).

SQL и GraphQL — это декларативные языки, в которых используются запросы. Разница между ними заключается в том, что SQL работает на уровне конкретной реляционной базы данных и позволяет делать запросы только к ней, а GraphQL работает на уровне API и позволяет выполнять запросы к разным источникам: базам данных, файловым системам, внешним API и сторонним сервисам.

Формируя запрос, клиент GraphQL описывает желаемую структуру данных, и система возвращает ему данные в соответствии с запросом. Каждому элементу запроса в GraphQL соответствует тип, определяющий структуру и характеристики данных. В GraphQL есть как встроенные, так и пользовательские типы. Как и в языках со статической типизацией, в GraphQL система типов принудительно применяется к каждому запросу, что позволяет выявлять ошибки типов еще до запуска программы. Прохождение проверки типов до запуска снижает вероятность ошибок во время выполнения программы.

В контексте GraphQL под *серверной средой выполнения* понимается система (служба или окружение), которая интерпретирует и выполняет запросы GraphQL, связывая их с источниками данных, с которыми взаимодействует сервер.

Основное различие между REST и GraphQL заключается в том, как выполняется запрос и структурирование данных. Стиль REST API строится вокруг понятия «ресурс» (см. раздел «Модели зрелости API» в главе 5), что находит отражение в именах интерфейсов API, которые соответствуют ресурсам. GraphQL основан на графах (https://oreil.ly/d9NT4), которые состоят из узлов (эквивалентов

ресурсов) и соединяющих их ребер (связей). Документ GraphQL определяет типы, где поля соответствуют узлам. Если одно поле вложено в другое, этому отношению соответствует ребро между узлами. Сервис GraphQL определяет интерфейс, способный взаимодействовать с любым бэкендом (базой данных, унаследованной системой или внешними API), а клиент GraphQL использует этот интерфейс для взаимодействия с самим сервисом.

Происхождение стиля GraphQL

Чтобы понять истоки стиля GraphQL, нужно учесть, в какое время он создавался и какие проблемы предполагалось решить с его помощью. GraphQL появился в период быстрого роста популярности мобильных устройств. Существовавший в 2012 году технологический стек мобильных браузеров и HTML5 не позволял добиться быстрого рендеринга страниц в веб-приложениях, что снижало их интерактивность. В это же время компания Facebook начала переписывать свои приложения под iOS и Android[1]. Эта работа заставила ее специалистов пересмотреть возможности REST API, когда требовалось получать по сети большие объемы графовых данных, то есть со сложными взаимосвязями. Так и родился GraphQL.

Стиль GraphQL создавался в период массового перехода пользователей с настольных приложений на мобильные. Две главные проблемы мобильных устройств — эффективность сети и энергопотребление, поэтому основной целью GraphQL было повышение эффективности запросов к источникам данных. Чтобы сократить объем информации, передаваемой по сети, и сэкономить заряд батареи, GraphQL позволяет пользователю запрашивать только те данные, которые ему нужны, и определять, как они будут структурированы.

Еще один фактор, повлиявший на разработку GraphQL, — увеличение количества клиентских платформ. REST API уже не справлялся с поддержкой большого разнообразия клиентов: обеспечение равной совместимости и эффективности для всех платформ и версий клиентов требовало все бóльших трудозатрат.

Наконец, в создании GraphQL сыграл роль медленный процесс разработки REST API, который не отвечал запросам времени на быстрое внедрение новых функций. Чтобы учесть специфические требования и изменения в дизайне

---

[1] Рекомендуем к просмотру документальный фильм GraphQL: The Documentary (https://oreil.ly/9vrMw) от Honeypot. Также советуем прочитать пост GraphQL: A Data Query Language («GraphQL: язык запросов к данным») Ли Байрона (Lee Byron) (https://oreil.ly/oNmNk).

со стороны клиента, в REST API необходимо менять формат предоставления данных на стороне сервера. Ограничение структуры данных, в свою очередь, замедляет циклы разработки. GraphQL, напротив, позволяет быстро разрабатывать широкий спектр приложений, где структуру данных, возвращаемых из API, определяет клиент.

GraphQL и REST

Преимущества GraphQL лучше всего понятны в сопоставлении с REST. REST API является ресурсоориентированным и использует несколько эндпоинтов для предоставления данных. GraphQL, напротив, относится к операционно-ориентированным (его интерфейсы описывают действие, выполняемое над ресурсом) и использует только один эндпоинт[1].

Чтобы вы могли лучше понять разницу между стилями REST и GraphQL, проведем аналогию с заказом еды в ресторане. Клиент читает меню и заказывает блюдо (данные); официант (API) передает заказ на кухню (сервер). Кухня обрабатывает заказ, готовит блюдо и передает заказ официанту, который доставляет его клиенту.

В REST API меню ресторана — это перечень фиксированных эндпоинтов API, где каждый эндпоинт — конкретное блюдо. Если клиенту нужно особое блюдо, то его нужно сначала добавить в меню, то есть обновить, распечатать и распространить новое меню. Из-за этого увеличивается расход ресурсов, особенно при частых обновлениях.

В GraphQL меню гибче, чем в REST. Оно позволяет клиенту формировать заказ на свой вкус. Вместо фиксированных пунктов меню клиент может указать точные ингредиенты, которые ему нужны. Затем кухня готовит блюдо и подает только то, что запросил клиент. Такой подход позволяет или сократить количество отходов (в случае чрезмерной выборки), или избежать дополнительных заказов (в случае недостаточной выборки). Иными словами, GraphQL может обрабатывать сложные требования к данным с помощью одного настраиваемого запроса.

Теперь, когда вы понимаете разницу между REST и GraphQL, мы можем обсудить основные элементы GraphQL. В следующем разделе описана структура документа GraphQL.

---

[1] Использование нескольких эндпоинтов для предоставления данных в GraphQL технически возможно, но такой подход считается антипаттерном.

Ключевые элементы GraphQL

В этом разделе вы познакомитесь с терминологией спецификации GraphQL, которая составляет значительную долю профессионального жаргона разработчиков GraphQL API. Для новичков эта терминология может показаться слишком абстрактной. Один из способов усвоить ее — поработать с GraphQL на практике, чем мы и займемся в этой главе в разделе «Реализация».

Обратите внимание, что в списке ниже даны только основные термины. Подробная информация об основных компонентах GraphQL доступна в последней версии спецификации GraphQL (https://spec.graphql.org).

Документ (https://oreil.ly/PgH6B)

Документ GraphQL — это строка запроса (или текстовый файл), которая описывает операции, разрешенные системой типов. В GraphQL используется статическая типизация — это значит, что перед выполнением программы выполняется парсинг документа и проверка на соответствие типам, представленным в схеме GraphQL. Парсинг гарантирует, что в документе соблюдается синтаксис GraphQL, а валидация подтверждает, что операции согласованы с типами, определенными в схеме, обязательными аргументами и используемыми переменными. Документ GraphQL создается клиентом.

Система типов (https://oreil.ly/6r4aK)

Система типов описывает возможности GraphQL API. Кроме того, ее используют для валидации операций GraphQL, поскольку системе типов должны соответствовать и входные данные (входные объекты, аргументы или переменные), и выходные (объектные типы или интерфейсы). Безопасность типов достигается за счет четкой границы между входными и выходными данными: перед выполнением запрос валидируют, чтобы подтвердить его соответствие схеме. Типы в системе типов делятся на входные и выходные (https://oreil.ly/IBpHN). *Входные типы* описывают значения, которые принимаются в качестве аргументов или переменных поля. *Выходные типы* описывают значения, которые могут возвращать поля.

Схема (https://oreil.ly/tqZe7)

Схема — это коллекция типов и поддерживаемых директив. Она служит корнем, определяющим точку входа для выполнения каждой операции в системе типов GraphQL. Схема — ключевая характеристика сервера GraphQL.

Операции (https://oreil.ly/LWzbc)

Операция получения или изменения данных применяется к *выборке* (ограниченному набору данных, запрошенных клиентом) (https://oreil.ly/OKJ4D).

В GraphQL можно выполнять три вида операций над данными: *Query* (запрос) — чтение данных; *Mutation* (мутация) — запись данных; *Subscription* (подписка) — долгоживущая операция, которая предполагает, что сервер отправляет клиенту данные в ответ на определенное событие в системе[1].

Поле (https://oreil.ly/cSizW)

В операции GraphQL поле — это основная единица выборки, которая возвращает значение. Опционально поле может иметь набор аргументов. В ответе поле представляет собой ключ, а в схеме — определяется по имени. Поле является отдельным фрагментом данных, который клиент может запросить в выборке, и возвращает выходные типы (https://oreil.ly/E7bQ1).

Резолвер (https://oreil.ly/3okpv)

Резолвер отвечает за заполнение поля данными. Он соединяет схему GraphQL и источники данных, например базы данных или внешние API. Технически резолвер — это функция, которая инкапсулирует логику для получения или вычисления значения поля в схеме при выполнении операции.

Скалярные типы (https://oreil.ly/BeWK1) *и перечисления* (https://oreil.ly/rsJ7K)

Скалярный тип — примитивное значение, например: *Int*, *Float*, *String*, *Boolean* или *ID*. Скалярные типы и перечисления не имеют вложенной структуры, поэтому в иерархии операций GraphQL они фигурируют как входные или конечные значения. Кроме того, GraphQL позволяет создавать пользовательские скалярные типы путем определения новых типов (https://oreil.ly/sQJQ9).

Принципы работы GraphQL

Основой GraphQL API служит схема, которая описывает возможности сервиса. Схема GraphQL расширяется путем добавления операций запроса, мутации и подписки, которые определяют, как клиенты могут взаимодействовать с API. Обычно клиенты получают доступ к этим операциям через один HTTP-эндпоинт.

В основе схемы лежат типы, описывающие структуру данных. Схема GraphQL содержит поля, каждое из которых представляет собой фрагмент информации определенного типа, и за каждым полем стоит резолвер.

Резолвер — это функция, которая инкапсулирует логику, необходимую для получения данных, их изменения или подписки на данные для конкретного поля. Каждому полю в схеме соответствует конкретный резолвер, который

[1] Для реализации подписок можно использовать протокол WebSocket (https://oreil.ly/5dmi9), а также механизмы HTTP Server-Sent Events (SSE) (https://oreil.ly/tFb3r) или long polling (https://oreil.ly/nEDqD).

содержит код для получения значения этого поля. При выполнении запроса клиента резолверы можно использовать для взаимодействия с базами данных, внутренними и внешними сервисами или другими источниками данных.

> **ПОДХОДЫ CODE-FIRST И SCHEMA-FIRST**
>
> Есть два подхода к созданию GraphQL API: code-first (сначала код) и schema-first (сначала схема).
>
> При подходе *code-first* сначала пишется код, который определяет поведение GraphQL-схемы. Этот подход используется в таких фреймворках GraphQL, как Graphene (https://graphene-python.org) и Strawberry (https://strawberry.rocks).
>
> При подходе *schema-first* сначала создается файл схемы, который описывает GraphQL-типы, определяющие структуру API. Далее разрабатывается код, в котором реализуются требования, заданные схемой. Этот подход используется в таких фреймворках, как Ariadne (https://ariadnegraphql.org) и Tartiflette (https://tartiflette.io).

Рассмотрим некоторые возможности GraphQL. В нем, как и в REST, в качестве транспорта используется HTTP, однако в REST взаимодействие с API выполняется через отправку запроса, содержащего HTTP-метод, на один из эндпоинтов. HTTP-методы выражают ограниченный набор операций, которые можно выполнить над ресурсом. Например, `GET` используется для чтения; `POST`, `PUT` и `PATCH` — для изменения, а `DELETE` — для удаления. Кроме того, параметры и тело запроса можно использовать для управления свойствами ресурса. В GraphQL для всех операций обычно используется HTTP-метод `POST`, а запросы отправляются на один эндпоинт. Другой возможный вариант настройки — HTTP-метод `GET` для операции запроса и `POST` для операции изменения.

Реализация

Самый простой способ взаимодействия с GraphQL API — использовать GraphiQL (https://oreil.ly/L5WD1). Это интерактивная графическая среда, доступная в браузере. Она позволяет создавать и выполнять документы (запросы) в GraphQL-сервисе. Чтобы получить доступ к GraphQL API для WFS, выполните действия, показанные в примере 6.1.

Пример 6.1. Настройка лабораторной среды GraphQL

```
cd src/Django  ❶
docker compose build --build-arg UID=$(id -u) --build-arg GID=$(id -g)  ❷
docker compose up --detach –wait  ❸
```

❶ Перейдите в каталог для настройки лабораторной среды.

❷ Соберите Docker-образы.

❸ Запустите Docker-контейнеры.

GraphiQL

В следующем примере показаны варианты взаимодействия с GraphQL API. Начнем с получения списка названий городов вместе с их UUID с помощью GraphiQL. Чтобы получить этот список, сделайте следующее.

1. Убедитесь, что WFS запущен, выполнив действия, указанные в примере 6.1.
2. Перейдите по адресу http://localhost:8000/graphiql[1].
3. Выполните `query GetCityUUID { cities { uuid name } }` (рис. 6.1). В GraphiQL можно отформатировать запрос с помощью кнопки Prettify query, расположенной под кнопкой Execute query. Раздел Docs (не виден на рисунке) содержит корневые типы схемы (`query` и `mutation`) и другие типы, доступные в этом GraphQL-сервисе.

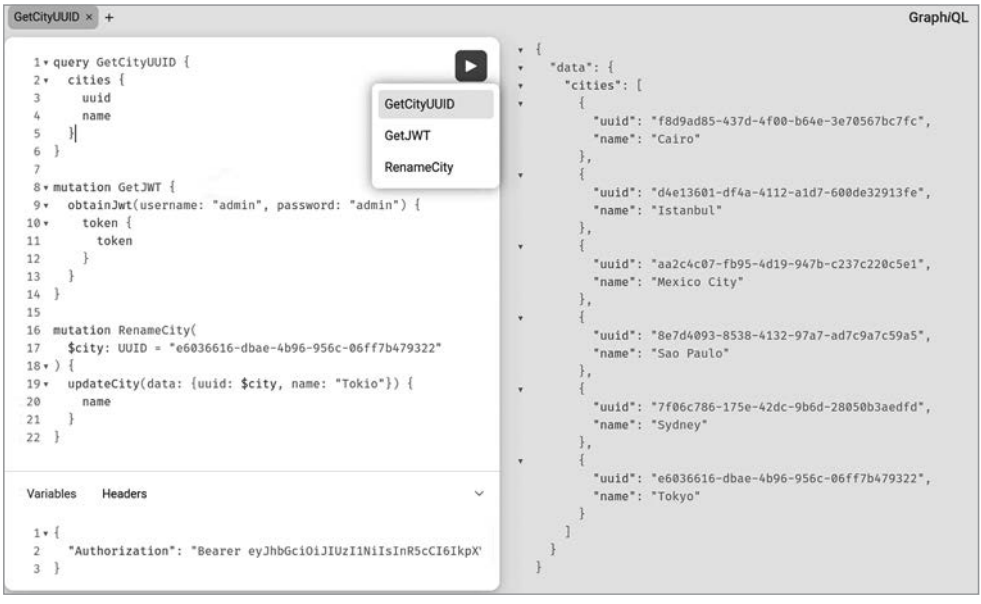


Рис. 6.1. GraphiQL (требует настройки из примера 6.1)

[1] При работе с GitHub Codespaces используется уникальный URL, содержащий имя Codespace вместо адреса localhost. Этот URL привязан к порту 8000, и его можно найти на вкладке PORTS в VS Code.



Доступ к эндпоинту GraphQL API не требует аутентификации, что создает угрозу безопасности, особенно в продакшен-среде. Однако в GraphiQL можно реализовать аутентификацию пользователей для контроля доступа. Например, для доступа к GraphiQL Explorer от GitHub (https://oreil.ly/3mTtC) пользователь GitHub должен сначала нажать кнопку Sign in with GitHub.

Запрос `GetCityUUID` продемонстрировал возможности GraphQL API по получению данных[1]. Обратите внимание, что для каждого города GraphQL API вернул только запрошенные данные — `uuid` и `name`. Ограничивая количество полей в ответе API, мы уменьшаем сетевую нагрузку.

Выполнение мутаций

В следующем примере показано изменение данных WFS путем обновления информации о городе через GraphQL API. Для этого примера также потребуются операции `GetJWT` и `RenameCity` (см. рис. 6.1).

Перед выполнением мутации для объекта города необходимо получить UUID города, информацию о котором нужно отредактировать. Для этого сделайте следующее.

1. Получите UUID для города Токио. Мы получили UUID (`e6036616-dbae-4b96-956c-06ff7b479322`) в предыдущем примере (см. рис. 6.1), но в вашем приложении он будет другим.
2. Получите JWT, который будет использоваться для аутентификации пользователя перед выполнением мутации `RenameCity` (табл. 6.1). Чтобы получить токен, выполните мутацию `GetJWT`, как показано на рис. 6.1.

GraphiQL — удобный инструмент для тестирования и изучения GraphQL API, но это не единственный способ взаимодействия с ним. Можно использовать и более привычные инструменты, такие как curl. В примере 6.2 показаны инструкции по выполнению запроса к GraphQL API без использования GraphiQL.

Пример 6.2. Получение JWT из GraphQL API с помощью curl (требуется настройка из примера 6.1)

```
# Установка переменных окружения  ❶
CSRF_TOKEN=$(docker compose exec app openssl rand -hex 16)
MUT='mutation {obtainJwt(username:\"admin\",password:\"admin\"){token{token}}}'

# Получение JWT-токена с помощью curl  ❷
docker compose exec app curl --silent \
       --request POST \
       --header "Content-type: application/json" \
```

[1] Сравните этот запрос с запросом из табл. 6.2 (см. ниже).

```
        --header "X-csrftoken: $CSRF_TOKEN" \
        --header "Cookie: csrftoken=$CSRF_TOKEN" \
        --data-raw "{\"query\": \"$MUT\"}" \
        "http://localhost:8000/graphql"

# Вывод:
{"data": {"obtainJwt": {"token": {"token": "eyJhbGciOiJIU..."}}}}
```

❶ Переменная `CSRF_TOKEN` реализует механизм защиты от межсайтовой подделки запросов (cross-site request forgery, CSRF). Переменная `MUT` содержит операцию мутации `GetJWT` (см. рис. 6.1), которая используется для получения JWT-токена доступа.

❷ Команда `curl` выполняет HTTP-запрос. Назначение запроса — эндпоинт GraphQL API WFS, доступный по адресу `/graphql` (не GraphiQL!). Запрос кодируется в формате JSON и содержит заголовки, которые требует защита CSRF, использующая `CSRF_TOKEN`. Отправляемый JSON-объект содержит только один ключ (`query`, не путать с операцией запроса Query), в котором находится GraphQL-документ, описывающий операцию мутации. Поле `obtainJwt` принимает данные учетной записи пользователя в качестве аргументов и возвращает токен доступа.

Последний шаг в этом примере — переименовать город. Для этого выполним мутацию `RenameCity`, как показано в табл. 6.1 и на рис. 6.1. На этот раз мутация выполняется в пользовательском интерфейсе GraphiQL.



Вне зависимости от того, используем ли мы REST API или GraphQL API, некоторые ресурсы WFS защищены механизмами авторизации, которые требуют JWT. Поэтому при взаимодействии с защищенными ресурсами API не забывайте добавлять заголовок Authorization. В GraphiQL заголовок добавляется в раздел Headers и заключается в фигурные скобки ({}), как показано ниже и на рис. 6.1:

```
{
"Authorization": "Bearer token"
}
```

Таблица 6.1. Мутация GraphQL для переименования города

Документ GraphQL	Ответ API

`mutation RenameCity(`
`  $city: UUID = "e6036616-dbae-4b96-956c-06ff7b479322"`
`) {`
`  updateCity(data: {uuid: $city, name: "Tokio"}) {`
`  name`
`  }`
`}`
	`{`
	`  "data": {`
	`  "updateCity": {`
	`  "name": "Tokio"`
	`  }`
	`  }`
	`}`

В этом разделе взаимодействие с GraphQL (через GraphiQL или без него) показано с точки зрения фронтенд-разработчика. В следующем разделе мы рассмотрим те же процессы с точки зрения бэкенд-разработчика и обсудим детали реализации GraphQL API для WFS.

CRUD

Для реализации GraphQL API мы выбрали библиотеку Strawberry (https://oreil.ly/5kup5), доступную для фреймворка Django (https://www.djangoproject.com).

Вы уже знакомы с применением команд CRUD к модели City в WFS через REST API (см. табл. 5.3). В этом примере мы реализуем CRUD для той же модели, но с использованием GraphQL.



Перед внедрением GraphQL API в продакшене ознакомьтесь с документом GraphQL Best Practices («Лучшие практики GraphQL») (https://oreil.ly/r83fM).

Реализация команд CRUD в GraphQL отличается от их реализации в REST API. Запросы GraphQL обычно отправляются HTTP-методом `POST`[1]. Соответственно, для манипуляции ресурсами нельзя использовать методы `DELETE`, `PUT` или `PATCH`. Вместо этого для чтения данных (операция Read в CRUD) используется запрос, содержащий операцию запроса или подписки. `POST`-запрос с операцией мутации обеспечивает запись данных (выполняя операции Create, Update или Delete из CRUD).

Read

В этом примере мы реализуем операцию Read для модели City.

Создание GraphQL API начинается с определения эндпоинта, через который он будет получать запросы. В отличие от REST API, который предоставляет множество HTTP-эндпоинтов, взаимодействие с GraphQL API обычно выполняется через один HTTP-эндпоинт, который предоставляет схему GraphQL. В примере 6.3 показано определение эндпоинта `/graphql`.

Пример 6.3. Конфигурация эндпоинта GraphQL API

```
# src/django/app/config/urls.py

from django.views.decorators.csrf import csrf_exempt
from strawberry.django.views import AsyncGraphQLView
from core.api.graphql.schema import schema
```

[1] При формировании запросов в GraphQL также может использоваться HTTP-метод GET.

```
...
urlpatterns = [
  ...
  path("graphql", AsyncGraphQLView.as_view(  ❶
        schema=schema, graphql_ide=False, allow_queries_via_get=False
  )),
  ...
]
...
if ENVIRONMENT == "development":
    ...
    urlpatterns += [path("graphiql", csrf_exempt(AsyncGraphQLView.as_view(  ❷
        schema=schema, graphql_ide=True, allow_queries_via_get=False
    ))]
...
```

❶ Функция `path()` предоставляет GraphQL API через эндпоинт `/graphql`. Аргумент `schema` указывает на переменную, содержащую корневую схему, которая хранит операции GraphQL.

❷ Функция `path()` делает GraphiQL доступным по эндпоинту `/graphiql`. Аргумент `graphql_ide=True` активирует доступ к GraphiQL в среде разработки. По умолчанию фреймворк Django ожидает, что CSRF-токен будет доступен для каждого `POST`-запроса. Функция `csrf_exempt()` отключает CSRF-защиту, чтобы разработчикам было проще работать с GraphiQL в среде разработки.



GraphiQL полезен во время тестирования и разработки, однако в продакшене технология создает риск раскрытия конфиденциальных данных и может привести к кибератакам.

Следующий шаг в проектировании GraphQL API — определение схемы, которая будет агрегировать операции GraphQL (пример 6.4).

Пример 6.4. Определение схемы GraphQL

```
# src/django/app/core/api/graphql/schema.py

from strawberry import Schema
from strawberry_django.optimizer import DjangoOptimizerExtension

from .queries import Query
from .mutations import Mutation


schema = Schema(  ❶
    query=Query,
    mutation=Mutation,
    extensions=[DjangoOptimizerExtension],
)
```

❶ Схема WFS содержит запросы и мутации, доступные через объекты `Query` и `Mutation`. Кроме того, библиотека Strawberry позволяет использовать расширения для этих операций. В данном случае используется `DjangoOptimizerExtension` (https://oreil.ly/8rvjq) для решения проблемы $n + 1$ запроса (https://oreil.ly/oZm4R). Проблема $n + 1$ — это проблема производительности, которая возникает, когда связанные данные извлекаются множеством запросов вместо одного. Она характерна для реляционных баз данных, где одна модель использует внешний ключ для связи с другой. Например, модель *WeatherForecast* имеет внешний ключ к модели *City*, который соответствует отношению «один ко многим» (прогнозы погоды для одного города меняются каждый день). Если проблему $n + 1$ не решить, то по умолчанию для одной искомой статьи реляционная база данных выполнит n дополнительных запросов, где n равно количеству комментариев к этой статье[1].

Следующий шаг после подготовки схемы — реализация запросов и мутаций. Начнем с запроса, показанного в примере 6.5: запрос `cities` возвращает из модели City города, доступные в WFS.

Пример 6.5. Реализация запроса GraphQL

```
# src/django/app/core/api/graphql/queries.py

import strawberry
from strawberry_django import field

from .filters import CityFilter
from .types import CityType

@strawberry.type
class CityQuery:
    cities: list[CityType] = field(filters=CityFilter, ...)  ❶

...

@strawberry.type
class Query(CityQuery, ...):  ❷
    pass
```

❶ `CityQuery` — это класс Python, который декоратор `@strawberry.type` превращает в тип GraphQL. Поле `cities` — это поле Strawberry Django (https://oreil.ly/1Aoew), к которому клиент выполняет запрос. Это поле возвращает список городов, описанных типом `CityType`. Кроме того, запрос позволяет выполнять фильтрацию, на что указывает класс `CityFilter`. Для

[1] В этом блоге (https://oreil.ly/nUxQl) описано решение проблемы $n + 1$ с использованием паттерна DataLoader.

оптимизации запросов используется ORM библиотеки Strawberry. Декорирование класса с помощью `@strawberry.type` создает резолвер для поля `cities`. Операция запроса (`Query`) (https://oreil.ly/xSnR9) позволяет только читать (извлекать) данные.

❷ Класс `Query` — это тип GraphQL, который агрегирует все запросы в WFS.

Рассмотрим типы из табл. 6.2, которые описывают объект города. Эти типы встретятся нам в примере 6.6.

Пример 6.6. Типы модели City в GraphQL

```
# src/django/app/core/api/graphql/types.py

import strawberry_django
from typing import Optional
from core.models import City

@strawberry_django.type(City, ...)
class CityType:  ❶
    uuid: UUID
    name: str
    country: str
    ...

# src/django/app/core/api/graphql/filters.py
@strawberry_django.filter(City, lookups=True)
class CityFilter:  ❷
    name: Optional[TextFilterLookup[str]]
    country: Optional[TextFilterLookup[str]]
    ...

# src/django/app/core/api/graphql/inputs.py
@strawberry_django.input(City)
class UUIDInput:  ❸
    uuid: UUID

@strawberry_django.input(City)
class CityInput:  ❸
    name: str
    country: str
    ...

@strawberry_django.partial(City)
class CityInputPartial(UUIDInput):  ❸
    name: Optional[str]
    country: Optional[str]
    ...
```

❶ Класс `CityType` — тип GraphQL, который создает расширение strawberry-graphql-django (https://oreil.ly/g4hP4). Декоратор `@strawberry_django.type` связывает модель Django City с системой типов Strawberry, позволяя экспортировать нужные поля модели `City`.

❷ Класс `CityFilter` позволяет применять операцию фильтрации к модели Django. Декоратор фильтра привязан к модели Django `City` и позволяет выполнять фильтрацию по конкретным полям. Аргумент `lookups=True` позволяет осуществлять поиск в модели `City` с помощью операторов `contains`, `endsWith`, `startsWith` и `regex`.

❸ Этот блок объявляет различные входные классы для операций Create, Update и Delete (CUD) (https://oreil.ly/53H3T). Типы входных данных (частичные и полные) наследуются от модели Django. *Частичный* тип позволяет передать только часть полей, тогда как *полный* требует указания всех полей.

Теперь протестируем реализацию поля `cities`. В табл. 6.2 показан документ с операцией запроса `GetAllCities`, которая извлекает названия всех доступных городов.

Таблица 6.2. Запрос cities (требует настройки из примера 6.1)

Документ GraphQL	Ответ API

`query GetAllCities {`
`  cities {`
`  name`
`  }`
`}`
	`{`
	`  "data": {`
	`  "cities": [`
	`  {"name": "Cairo"},`
	`  {"name": "Istanbul"},`
	`  {"name": "Mexico City"},`
	`  {"name": "Sao Paulo"},`
	`  {"name": "Sydney"},`
	`  {"name": "Tokyo }`
	`  ]`
	`  }`
	`}`

Create, Update, Delete

Реализовав операции чтения (Read) из CRUD с помощью GraphQL-запроса, далее необходимо использовать мутации для реализации оставшихся операций CUD. Эту задачу мы решим в примере 6.7.

Пример 6.7. Реализация мутаций для операций Create, Update и Delete

```
# src/django/app/core/api/graphql/mutations.py

import strawberry
from core.helpers import IsAdmin
from core.helpers import IsAuthenticated
from strawberry_django.mutations import create, update, delete
...

@strawberry.type
class CityMutation:  ❶
    create_city: CityType = create(
        CityInput,
        permission_classes=[IsAuthenticated, IsAdmin]
    )
    update_city: CityType = update(
        CityInputPartial,
        key_attr="uuid",
        permission_classes=[IsAuthenticated, IsAdmin]
    )
    delete_city: CityType = delete(
        UUIDInput,
        key_attr="uuid",
        permission_classes=[IsAuthenticated, IsAdmin]
    )

...

@strawberry.type
class Mutation(CityMutation, ...):  ❷
    pass
```

❶ `CityMutation` — это класс типа мутации с тремя полями: `create_city`, `update_city` и `delete_city`, которые используют мутации `create`, `update` и `delete` из strawberry-graphql-django (https://oreil.ly/JmSxf). Для каждого поля есть адаптированный тип входных данных, например `CityInput`, `CityInputPartial` и `UUIDInput`. Подробности реализации входных классов приведены в примере 6.6. Аргумент `key_attr` переопределяет стандартный поиск модели по ID (`pk`) в Strawberry, используя `uuid`.

Аргумент `permission_classes` разрешает выполнение мутации только авторизованным пользователям и позволяет реализовать авторизацию на уровне полей. Мы используем пользовательские классы `IsAuthenticated` и `IsAdmin`, которые дают разные права доступа к эндпоинту в зависимости от роли аутентифицированного пользователя (обычный пользователь или администратор).

❷ Класс `Mutation` агрегирует мутации WFS. Этот класс находится в корне схемы в примере 6.4.

После реализации всех трех операций CUD протестируем их. GraphQL позволяет последовательно выполнять операции, содержащие несколько полей, при условии, что все эти поля принадлежат к одному типу операции. В примере 6.8 показано выполнение трех мутаций.



Мутации защищены классом разрешений, который проверяет наличие JWT-токена в заголовке запроса. Инструкции по получению JWT-токена доступа представлены в примере 6.2.

Пример 6.8. Выполнение мутаций Create, Update и Delete

```
mutation CUDCity($city: UUID = "e6036616-dbae-4b96-956c-06ff7b479322") {
  createCity(
    data: {
      name: "Warsaw", country: "Poland", region: "Europe",
      timezone: "Europe/Warsaw", latitude: 52.249782, longitude: 21.012176
    }) {
      uuid
      name
      country
      region
      timezone
      latitude
      longitude
  }
  updateCity(data: {uuid: $city, name: "Tokio"}) {
    uuid
    name
  }
  deleteCity(data: {uuid: $city}) {
    uuid
  }
}
```

В примере 6.8 показана мутация с именем `CUDCity`, которая определяет переменную `$city`. Значение переменной получено из запроса `GetAllCities` до составления этого документа. Переменная `city` передается в поля `updateCity` и `deleteCity`.

В примере 6.9 показан ответ GraphQL API.

Пример 6.9. Ответы, возвращаемые мутациями из примера 6.8

```
{
  "data": {
    "createCity": {
      "uuid": "ed773f3f-84cd-41b3-b067-9961cdb9e912",
      "name": "Warsaw",
      "country": "Poland",
```

```
      "region": "Europe",
      "timezone": "Europe/Warsaw",
      "latitude": "52.249782",
      "longitude": "21.012176"
    },
    "updateCity": {
      "uuid": "e6036616-dbae-4b96-956c-06ff7b479322",
      "name": "Tokio",
    },
    "deleteCity": {
      "uuid": "e6036616-dbae-4b96-956c-06ff7b479322",
    }
  }
}
```

Структура ответа соответствует структуре мутации, показанной в примере 6.8. Обратите внимание, что один запрос выполнил три операции (Create, Update и Delete) над ресурсами городов. В REST API для достижения такого же эффекта потребовалось бы отправить три запроса. Поле `createCity` создает город `Warsaw`. Поле `updateCity` переименовывает `Warsaw` в `Tokio`, а поле `deleteCity` удаляет переименованный город `Tokio`. В последних двух полях в аргументе `data` используется `uuid` для идентификации и выбора города, к которому применяется мутация.

Безопасность

В подразделе «Проект OWASP API Security Top 10» главы 2 описаны проблемы безопасности API, характерные для различных стилей. Здесь же мы обсудим атаки, специфичные для GraphQL, а также реализацию механизма аутентификации пользователей.



Подробнее распространенные атаки и меры их предотвращения описаны в памятке OWASP по GraphQL (https://oreil.ly/kWQes).

Атаки на GraphQL

В этом подразделе мы рассмотрим распространенные атаки на GraphQL API.

Атака с использованием интроспекции

При этой атаке злоумышленник использует функцию интроспекции GraphQL для извлечения схемы. Атака может быть организована с помощью GraphiQL

или другого инструмента для отправки GraphQL-запросов. Чтобы снизить этот риск, функцию интроспекции можно ограничить или отключить[1].

Однако функция интроспекции необходима для автоматической генерации документации. Чтобы сбалансировать потребности безопасности и документирования, можно включить интроспекцию в среде разработки и отключить ее в продакшен-среде. Если интроспекция отключена, то документацию нужно сгенерировать предварительно и разместить на внешнем ресурсе.



Чтобы запустить функцию интроспекции в GraphQL API, отправьте полный запрос интроспекции (https://oreil.ly/KyK4G) на эндпоинт API[2]. Этот запрос выведет всю информацию о вашей схеме GraphQL.

Атака с использованием нежелательных запросов

Нежелательный запрос заставляет сервис GraphQL выполнять ресурсоемкие вычисления или раскрывать конфиденциальную информацию. К таким запросам относятся глубоко вложенные запросы, запросы на получение больших наборов данных, а также запросы с фильтрацией и сортировкой.

Один из способов предотвращения нежелательных запросов — введение списка разрешенных запросов. При таком подходе сервер GraphQL хранит список заранее известных, безопасных запросов. Каждый входящий запрос от клиента проверяется по этому списку. Если запрос есть в списке, сервер его выполняет. Если запроса нет, то он отклоняется с ответом HTTP `403 (Forbidden)`, что означает отказ сервера обработать запрос. Если сервер возвращает ошибки `4xx` и `5xx`, то рекомендуется использовать MIME-тип `application/graphql-response+json`[3].

Распространенная практика при работе со списками разрешенных запросов — *нормализация запросов* (удаление пробелов, структурирование в алфавитном порядке). Нормализация выполняется на стороне сервера, поэтому клиенты могут отправлять запросы в любом формате, не соблюдая ее правила.

Обычно списки разрешенных запросов отключают в средах разработки, чтобы разработчики могли свободно тестировать систему и создавать новые запросы.

---

[1] Учтите, что некоторые инструменты, например Clairvoyance (https://oreil.ly/2vWsF), могут извлекать схему GraphQL даже при отключенной интроспекции. Рекомендуем к прочтению руководство A Complete Guide to Hacking GraphQL («Полное руководство по взлому GraphQL») Стефано Ланаро (Stefano Lanaro) (https://oreil.ly/0Fwhg).

[2] Запрос можно найти в GitHub Gist (https://oreil.ly/nTDCL).

[3] См. MIME-тип application/graphql-response+json (https://oreil.ly/XZ1r3).

Однако в продакшен-среде использование таких списков гарантирует выполнение только доверенных запросов и снижает риск утечек данных и проблем с производительностью.

Атака с использованием рекурсивных запросов

В GraphQL связи определяются в схеме и описываются клиентом в документе через вложение полей. Если модель данных на сервере содержит циклическую связь, то можно сформировать запрос, аналогичный приведенному в примере 6.10. Такой запрос абсолютно корректен с точки зрения GraphQL. Проблема в том, что количество объектов, которые возвращает сервис, растет мультипликативно с каждым уровнем вложенности, что может потребовать значительных затрат ресурсов.

Пример 6.10. Рекурсивный запрос GraphQL

```
query RecursiveQuery {
  student {
    courses {
      student {
        courses {
          student {
            courses {
              student {
                uuid
                name
              }
            }
          }
        }
      }
    }
  }
}
```

Для защиты GraphQL API от рекурсивного вложения необходимо ограничить глубину вложенности запросов.

Атака с использованием пакетной обработки запросов

Эта атака эксплуатирует возможность формирования документа, содержащего несколько операций, в GraphQL. С таким документом вы встречались в примере 6.8.

При этой атаке злоумышленник отправляет множество запросов, объединенных в один документ GraphQL. Такой запрос может перегрузить сервис API, что приведет к исчерпанию ресурсов или простою сервиса.

Для защиты GraphQL API от этой атаки можно ограничить количество пакетных операций, разрешенных для одного запроса или мутации. Например,

можно установить лимит на пакетные запросы, разрешив максимум три операции на один запрос. Кроме того, можно установить лимиты для прерывания запросов, выполнение которых занимает слишком много времени. Наконец, можно блокировать клиентов, которые отправляют ресурсоемкие запросы.

Эта атака также эксплуатирует возможность *псевдонимов* (aliases). Если присвоить полю псевдоним, то можно выполнить несколько операций Query с помощью одного запроса (пример 6.11).

Пример 6.11. Использование псевдонима в GraphQL

```
query AliasQuery {
  Cairo: cities(filters: {name: {exact: "Cairo"}}) {
    uuid
    name
  }
  Sydney: cities(filters: {name: {exact: "Sydney"}}) {
    uuid
    name
  }
}
```

Аутентификация

В примере 6.7 мы кратко упомянули, что только аутентифицированные пользователи могут выполнять мутации над объектами городов. В этом подразделе мы подробно рассмотрим процесс аутентификации и его реализацию.

Для интеграции JWT в WFS используется библиотека strawberry-django-auth (https://oreil.ly/4WFjL). С ее помощью мутации, необходимые для процесса аутентификации, импортируются и интегрируются в схему WFS (пример 6.12).

Пример 6.12. Реализация мутаций JWT

```
# src/django/app/core/api/graphql/mutations.py

import strawberry

from gqlauth.user import arg_mutations
....

@strawberry.type
class JWTMutation:  ❶
    obtain_jwt = arg_mutations.ObtainJSONWebToken.field
    refresh_jwt = arg_mutations.RefreshToken.field
    revoke_jwt = arg_mutations.RevokeToken.field
    verify_jwt = arg_mutations.VerifyToken.field
    ...

@strawberry.type
class Mutation(CityMutation, JWTMutation):  ❷
    pass
```

❶ Класс `JWTMutation` агрегирует поля, относящиеся к процессу аутентификации, — поля для получения, обновления, отзы́ва и проверки JWT.

❷ Класс `JWTMutation` добавляется в класс `Mutation`, который регистрируется как корневой тип в схеме GraphQL (см. пример 6.4).

Реализация этого кода позволяет выполнить мутацию `obtainJwt` для получения JWT. Следующий шаг — выполнить валидацию токена. Здесь используется класс `IsAuthenticated` (пример 6.13).

Пример 6.13. Реализация аутентификации в GraphQL

```
# src/django/app/core/helpers/auth/permissions.py

import json
import jwt
...
from django.contrib.auth.models import User
from strawberry.permission import BasePermission as StrawberryBasePermission
from strawberry.types import Info

from config.constants import SECRET_KEY

class IsAuthenticated(StrawberryBasePermission):  ❶
    message = "User is not authenticated"

    async def has_permission(self, source: Any, info: Info, **kwargs) -> bool:  ❷
        authenticated = False

        request = info.context["request"]
        if "Authorization" in request.headers:
            jwt_token = request.headers["Authorization"].split(" ")[-1]  ❸
            if not jwt_token:
              raise Exception("Token is missing.")

            try:
              payload = jwt.decode(jwt_token, SECRET_KEY, algorithms=["HS256"]) ❹

              expiration_dt = datetime.fromtimestamp(int(payload["exp"]))
              token_expired = datetime.now() >= expiration_dt  ❺
              if token_expired:
                  rise Exception("Token has expired.")
              else:
                  authenticated = True

              qs = await sync_to_async(User.objects.filter)(pk=payload["id"])

              info.context.request.user = await sync_to_async(qs.first)()  ❻
```

```
        ...
        except Exception as e:
            raise Exception(f"Invalid token. {e}")

    return authenticated  ❼
```

❶ Класс `IsAuthenticated` расширяет функциональность базового класса разрешений `StrawberryBasePermission` (`BasePermission`).

❷ Класс переопределяет метод `has_permission()`, который возвращает логическое значение, указывающее, аутентифицирован ли запрос.

❸ Токен аутентификации JWT извлекается из HTTP-заголовка `Authorization`.

❹ Метод `jwt.decode()` выполняет проверку подписи и преобразует JWT в объект полезной нагрузки. Для декодирования требуется секретный ключ (`SECRET_KEY`), который использовался на этапе кодирования JWT. Этот блок кода извлекает дату истечения срока действия токена `expiration_dt`.

❺ Это условие проверяет, действителен ли токен.

❻ Если токен действителен, то на данном этапе выполняется аутентификация пользователя. Далее объект пользователя извлекается и помещается в контекст запроса.

❼ Если в процессе проверки не возникает ошибок, то класс возвращает `True`, что указывает на наличие у пользователя необходимых разрешений.

Документация

Написание документации — обязательный этап создания API. Даже если API хорошо спроектирован и интуитивно понятен, разработчик, который его использует, все равно должен иметь доступ к пониманию возможностей API. Роль документации заключается в том, чтобы подтверждать или опровергать эти предположения.

Одна из сложностей документирования API — поддержание документации в актуальном состоянии, в соответствии с текущей версией. Стремясь уложиться в назначенные сроки релиза, разработчики часто откладывают создание документации, и в итоге она пишется уже после выпуска API.

В GraphQL проблема создания документации решается с помощью функции интроспекции. Она позволяет просматривать схему GraphQL. В табл. 6.3 приведен пример запроса интроспекции и фрагмент полученных результатов.

Таблица 6.3. Запрос интроспекции типов схемы GraphQL

Документ GraphQL	Ответ API

`{`
`  __schema {`
`  types {`
`  name`
`  }`
`  }`
`}`
	`{`
	`  "data": {`
	`  "__schema": {`
	`  "types": [`
	`  {"name": "Query"},`
	`  {"name": "CityType"},`
	`  ...`
	`  ]`
	`  }`
	`  }`
	`}`

В GraphiQL есть встроенный инструмент *Documentation Explorer*, который генерирует документацию на основе запроса интроспекции. Инструмент получает схему, закодированную в формате JSON, от эндпоинта GraphQL API. Затем объекты схемы в формате JSON преобразуются в интерфейс документации, который можно просматривать.

Иногда из соображений безопасности лучше не предоставлять полную документацию вашего GraphQL API через GraphiQL. В качестве альтернативы можно использовать инструменты, которые преобразуют схему GraphQL в статическую документацию, например SpectaQL (https://oreil.ly/UmaoV) или Voyager (https://oreil.ly/3KDm8).

GraphQL: преимущества и недостатки

Начнем с рассмотрения сильных сторон GraphQL.

Настройка ответа API

Структура данных, возвращаемых GraphQL API, соответствует структуре запроса, который использовался для их получения. Настраиваемый формат данных решает проблему избыточной или недостаточной выборки данных, что снижает нагрузку на сеть. Кроме того, клиенты получают возможность работать с возвращенными данными, не выполняя дополнительные преобразования или нормализацию.

Уменьшение количества запросов по сравнению с REST

В GraphQL для получения и отправки данных используется один эндпоинт API. Благодаря наличию псевдонимов и возможности запрашивать сразу несколько полей у службы GraphQL количество запросов к системе

уменьшается. Иными словами, QraphQL менее «разговорчивый», чем REST, и для получения желаемого ответа пользователю требуется меньшее количество циклов коммуникации с системой.

Интроспекция схемы

GraphQL позволяет клиентам выполнять интроспекцию основных элементов API на основе его схемы. Анализируя схему, можно получить информацию о системе типов GraphQL, содержащей операции, поля и скалярные типы. Если название типа начинается с двойного подчеркивания (__), это говорит о том, что он является встроенным элементом системы интроспекции GraphQL.

Самодокументируемость

Документация в GraphQL доступна «из коробки» и формируется на основе его схемы. Схема позволяет выполнять интроспекцию типов GraphQL и получать представление о том, как взаимодействуют различные компоненты GraphQL. Такие инструменты, как Documentation Explorer в GraphiQL, позволяют просматривать схему через интерактивный интерфейс для отправки запросов и детального изучения структуры API. Кроме того, схему можно преобразовать в статическую документацию или даже в код с помощью таких инструментов, как GraphQL Codegen (https://oreil.ly/R6qFT), что упрощает интеграцию с другими этапами жизненного цикла разработки программного обеспечения.

Единый интерфейс для нескольких API

GraphQL может выступать в роли шлюза в *микросервисной архитектуре*, объединяя REST, GraphQL и другие API отдельных микросервисов в одну глобальную схему GraphQL. Каждый микросервис может предоставлять собственный эндпоинт, а глобальная схема GraphQL позволит клиентам запрашивать данные из разных сервисов.

Среди популярных архитектурных паттернов, применимых к GraphQL: Backends for Frontends (BFF) (https://oreil.ly/uhCXo) и федеративная архитектура (https://oreil.ly/xpaUO). В паттерне *BFF* отдельные клиентские приложения (веб-приложения, мобильные приложения и др.) взаимодействуют с GraphQL API, адаптированными под их нужды. *Федерация* GraphQL, напротив, объединяет данные из нескольких систем GraphQL в единый граф, к которому клиенты могут делать запросы. Этот подход позволяет сформировать запрос, который извлекает данные из разных источников, объединяет их в одном пакете и возвращает одним ответом.

Версионирование

В GraphQL версионирование не предполагается, поскольку акцент делается на непрерывной эволюции схемы (https://oreil.ly/vmCE5). Среди разработчиков

GraphQL API пропагандируется подход «без версий», поскольку структура запроса гибкая и сервер всегда возвращает настраиваемые данные. Расширение схемы, как правило, происходит за счет добавления новых полей или типов. Подобные изменения не являются критическими и не лишают пользователя возможности запросить только нужные ему данные. Таким образом обеспечивается обратная совместимость по мере развития API[1].

Однако у GraphQL есть и слабые стороны.

Проблемы безопасности

Интроспекция в GraphQL предоставляет клиентам документацию, позволяя им запрашивать схему API и видеть все доступные типы. Эта функция полезна на этапе разработки, однако раскрывает систему типов GraphQL API, что может сыграть на руку злоумышленникам. Они могут использовать эту информацию для обнаружения ресурсов, которые не предназначены для публичного использования, и получения доступа к ним.

Разница между GraphQL API и REST API заключается в поверхности атаки. В REST API злоумышленник атакует «вширь» и целится в несколько эндпоинтов сразу. В GraphQL же обычно имеется только один эндпоинт. Кроме того, в GraphQL API часто возникают сложности с реализацией ограничения частоты запросов, что упрощает организацию атаки с использованием пакетной обработки запросов.

Кроме того, GraphQL подвержен атакам чрезмерной выборки, при которых извлекается огромный объем данных, и атакам недостаточной выборки, при которых к API отправляется множество запросов. Эти атаки также могут быть реализованы одновременно.

Ресурсоемкие запросы

GraphQL позволяет решить проблему избыточной выборки, но сложные запросы все равно могут приводить к снижению производительности. Запросы, для которых требуются рекурсивные или глубоко вложенные данные, извлекается множество полей или используется функция псевдонимов, могут замедлять работу системы. Чтобы решить эти проблемы, разработчики API должны учитывать сложность запросов и устанавливать ограничения на их выполнение.

Сложности кэширования

Гибкая и динамическая природа GraphQL осложняет кэширование. Из-за того что все запросы проходят через один эндпоинт, различить их становится

---

[1] GraphQL позволяет избегать версионирования, но окончательное решение о его применении остается за проектировщиками API. Например, в GraphQL API GitHub (https://oreil.ly/14u6c) иногда осуществляется версионирование полей.

сложнее. В GraphQL нет встроенной поддержки HTTP-заголовков кэширования, поэтому этот механизм нужно реализовывать иначе. Стандартная рекомендация для решения этой проблемы[1] — строить запросы с глобально уникальными идентификаторами (globally unique identifiers, GUID).

Отсутствие встроенной поддержки multipart/form-data

Спецификация GraphQL не поддерживает MIME-тип `multipart/form-data`, который обычно используется для загрузки файлов. Чтобы реализовать такую функциональность, может потребоваться создать отдельный REST-эндпоинт или кодировать файл в строку Base64 (https://oreil.ly/jx2A9), что увеличит размера запроса. Можно также использовать клиентскую библиотеку Apollo (https://oreil.ly/50dLY) для реализации GraphQL Apollo Upload Client (https://oreil.ly/86xAu). Кроме того, добавление поддержки `multipart/form-data` может создать уязвимости для CSRF-атак[2].

Ограниченное использование HTTP-кодов состояния

В GraphQL ошибки обрабатываются самим сервером, и информация о них возвращается как часть полезной нагрузки ответа. Иными словами, GraphQL не использует HTTP-коды состояния: для всех ответов применяется код `200` (`OK`), независимо от того, что содержит ответ — данные, ошибки или и то и другое[3]. Вследствие этого код `200` (`OK`) фактически может означать ответ с ошибкой. Кроме того, могут возвращаться коды состояния `4xx` или `5xx`, но обычно они указывают на проблемы, внешние по отношению к GraphQL API.

Когда использовать GraphQL

GraphQL разработан для приложений, которым требуется низкое потребление сетевого трафика и энергии, например для смартфонов, умных часов или устройств интернета вещей (IoT). Кроме того, GraphQL применяют для объединения данных из различных внешних источников, таких как базы данных, сервисы логирования, унаследованные системы или сторонние поставщики данных.

Еще один вариант использования GraphQL — приложения с вложенными структурами данных, которые в случае применения другого подхода потребовали бы множественных вызовов API. GraphQL позволяет сократить

---

[1] См. статью GraphQL Caching («Кэширование в GraphQL») (https://oreil.ly/IE05r).

[2] См. раздел Security («Безопасность») в GraphQL multipart/form-data (https://oreil.ly/nDpqW).

[3] В проекте спецификации 2024 года сказано, что GraphQL постепенно уходит от постоянного использования кода состояния 200 (OK). См. материал GraphQL over HTTP («GraphQL поверх HTTP») (https://oreil.ly/CtlW9).

количество запросов к серверу, поскольку клиенты могут получать нужные данные за один запрос. Кроме того, разработка GraphQL требует меньше времени по сравнению с REST API, когда требуется быстрый прототип или проверка концепции продукта.

Упражнения

1. В примере 6.7 мы реализовали операции CUD, следуя руководству из библиотеки Strawberry (https://oreil.ly/3ZVRH). Однако у этого подхода есть проблема: если не изучать исходный код, то невозможно понять, как он реализован, и, хотя для простых случаев реализация работает, в более сложных сценариях она может оказаться недостаточной. В этом упражнении ваша задача — провести рефакторинг полей `create_city`, `update_city` и `delete_city` в файле `src/django/app/core/api/graphql/mutations.py`. Используйте собственные резолверы (https://oreil.ly/XPmQV), которые обеспечат соответствующую функциональность для этих полей. Решение для этого упражнения уже есть в исходном коде.
2. Реализуйте пагинацию для каждого типа из файла `src/django/app/core/api/graphql/types.py`, используя расширение Strawberry Django. Подход к решению описан в документации расширения, в разделе, посвященном пагинации со смещением (https://oreil.ly/rTfP4).

Резюме

В этой главе мы рассмотрели историю и основные понятия GraphQL — одного из самых популярных сетевых API. Мы также реализовали GraphQL API для WFS в рамках подхода code-first, инструменты для которого предоставляет библиотека Strawberry. Кроме того, обсудили проблемы безопасности GraphQL, реализацию механизма аутентификации, а также уязвимости GraphQL-запросов, влияющие на производительность. Наконец, мы увидели работу интроспекции, отправив полный интроспективный запрос через интерфейс GraphiQL.

В заключение мы обсудили сильные и слабые стороны GraphQL, знание которых поможет вам решить, стоит ли использовать стиль GraphQL в вашем следующем проекте.

ГЛАВА 7

Веб-фиды

В этой главе мы обсудим концепцию веб-фидов и уделим особое внимание формату синдикации Atom. *Веб-фид* (web feed), или *новостная лента* (news feed), — это цифровой документ, в котором обычно есть медиаконтент: содержимое веб-страниц (HTML) или ссылки на другие сайты. Веб-фиды используются для асинхронной рассылки информации и позволяют пользователям узнавать о новом контенте, не проверяя сайты постоянно. Обычно веб-фиды нужны в тех случаях, когда требуется регулярное обновление информации: на новостных порталах, в блогах, на платформах для размещения подкастов или видео.

Мы начнем с обсуждения эволюции веб-фидов и их значимости в контексте API. Далее мы рассмотрим примеры веб-фидов и инструкции по их реализации в соответствии со спецификацией Atom. Кроме того, мы поговорим о преимуществах и недостатках веб-фидов и типичных рисках, связанных с безопасностью. Для закрепления материала в конце главы предлагается набор упражнений, в которых нужно реализовать операцию чтения различных веб-фидов.

В 2000-х годах веб-фиды получили большую популярность, однако сейчас эта технология используется ограниченно. Ее основные потребители — представители новостных СМИ и технологических компаний, а также блогеры и ученые. Интерес к веб-фидам снизился после того, как контент стал перемещаться на платформы социальных сетей с авторизацией, такие как Meta или X. Тем не менее веб-фиды полезны как учебный материал: они сыграли важную роль в истории развития интернета и являются наглядным примером использования XML. По этой причине мы решили посвятить им отдельную главу.

Что такое веб-фид

Веб-фид (https://oreil.ly/KjuQN) — это формат данных, который доставляет пользователям регулярно обновляемый контент в виде документа со списками связанных данных. Есть несколько форматов фидов, таких как JSON feed

(https://www.jsonfeed.org) и Media RSS (https://oreil.ly/PFiyt), но в этой главе мы рассмотрим самые популярные: RSS и Atom.

RSS (https://oreil.ly/G47yw) — это тип веб-фида, предоставляющий обновления контента сайта в стандартизированном формате XML, который понятен одновременно и человеку, и компьютеру.

Atom (https://oreil.ly/scqYn) — это веб-стандарт, который включает в себя язык на основе XML, предназначенный для веб-фидов (Atom Syndication Format, https://oreil.ly/03MCd), и протокол прикладного уровня на основе HTTP для публикации и редактирования веб-ресурсов (Atom Publishing Protocol/AtomPub, https://oreil.ly/03MCd).

Зачем нужны веб-фиды

Большинство API, описанных в книге, основаны на *модели запроса* (pull-based API), и веб-фиды тоже относятся к этой категории. В таких API сервер возвращает данные по требованию, когда клиент их запрашивает. Бывает так, что клиенту нужно максимально быстро узнавать о появлении новых данных. В этих случаях и используется веб-фид, который позволяет асинхронно обнаруживать данные. Важно понимать: уведомление об обновлении содержимого фида инициирует не сервер, а клиент, периодически запрашивая фиды. Сервер возвращает документ с веб-фидом, а клиент сравнивает новое содержимое фида с уже имеющимся и добавляет все недавно опубликованные данные. Архитектура веб-фидов показана на рис. 7.1.

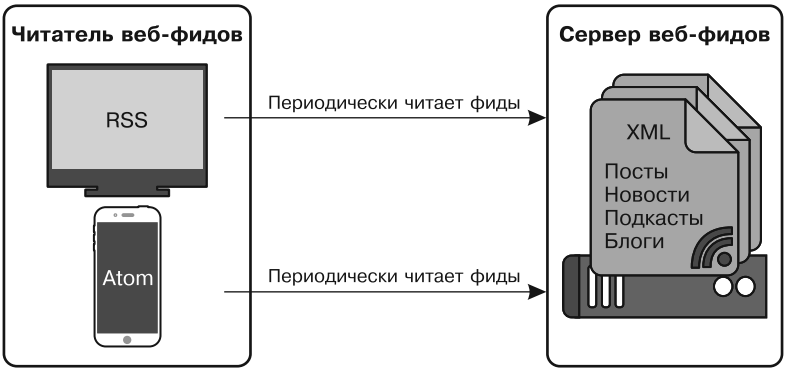


Рис. 7.1. Архитектура веб-фидов

Во введении к этой главе мы уже говорили, что веб-фид позволяет клиентам запрашивать у сервера самые свежие данные. Пример такого сценария

использования — новостной сайт, где автор публикации хочет уведомить читателей о появлении новой статьи.

Еще один сценарий использования веб-фидов — работа с облачными провайдерами, которые предоставляют свои сервисы через API и пользовательские интерфейсы. Команда эксплуатации, которая использует облачные сервисы, может подписаться на веб-фид поставщика, чтобы получать уведомления о любых изменениях в сервисе, например о новых функциях, устаревании или сбоях.

Кроме того, компании, специализирующиеся на кибербезопасности, используют веб-фиды в качестве канала для обновлений, чтобы быстро информировать клиентов о новых угрозах и обнаруженных уязвимостях. Можно подписаться на такой веб-фид, чтобы получать последние обновления и реагировать на эти угрозы.

В табл. 7.1 приведены ссылки на часто используемые веб-фиды и связанные с ними сайты, которые предоставляют актуальную информацию от облачных провайдеров и рекомендации по безопасности.

Таблица 7.1. Веб-фиды, предоставляемые облачными провайдерами и компаниями в сфере информационной безопасности

Служба	URL	Главная страница
AWS	RSS (https://oreil.ly/9iSH6)	https://health.aws.amazon.com
Azure Cloud	Atom (https://oreil.ly/FUdeT)	https://azure.status.microsoft
Google Cloud	Atom (https://oreil.ly/NPnX8)	https://status.cloud.google.com
Агентство по кибербезопасности и защите инфраструктуры США (CISA)	RSS (https://oreil.ly/61dNb)	https://www.cisa.gov/about/contact-us/subscribe-updates-cisa

Эволюция веб-фидов

RSS — это спецификация веб-фида в формате XML 1.0, опубликованная компанией Netscape в 1999 году. RSS-фид содержит сводку контента (новостных публикаций или записей в блоге), опубликованного на сайте. Способ организации такого веб-фида описан в спецификации RSS (https://oreil.ly/m19YA). Веб-фид создается и поддерживается владельцем контента и хранится на веб-сервере в виде текстового файла. Этот файл по запросу возвращается клиенту в виде HTTP-ответа. Чтобы просмотреть его, пользователь использует программу для чтения веб-фидов (*ридер* или *агрегатор*), которая отображает новый добавленный контент. Веб-фид идентифицируется по URL. Агрегатор может

объединять несколько фидов, что позволяет пользователю читать контент из разных источников в одном месте, а не посещать каждый сайт отдельно.

Рассмотрим веб-фид RSS в примере 7.1.

Пример 7.1. Спецификация RSS-фида версии 2.0

```
<?xml version="1.0" encoding="UTF-8" ?>  ❶
<rss version="2.0">  ❷
  <channel>  ❸
    <title>News from example.com</title>
    <description>Top stories from example.com</description>
    <link>https://example.com/main.html</link>
    <language>en-us</language>
    <item>  ❹
      <title>RSS feed explained.</title>
      <description>Shocking — RSS feed in 15 lines.</description>
      <link>https://example.com/blog/post/1</link>
      <guid>https://example.com/blog/post/1</guid>
      <category>APIs</category>
    </item>
  </channel>
</rss>
```

❶ Фид начинается с XML-декларации, в которой указаны версия спецификации XML и формат кодировки, использованный в документе.

❷ `rss` — корневой элемент фида, который содержит используемую версию спецификации RSS.

❸ Элемент `channel` содержит метаданные и элементы фида. Согласно спецификации RSS 2.0 (https://oreil.ly/VbcnO) для `channel` элементы `title`, `description` и `link` являются обязательными, в отличие от `language`.

❹ Элемент `item` содержит контент фида. Контентом могут быть новостная статья, запись в блоге, видео, ссылки на другие ресурсы и многое другое. Сейчас веб-фиды обычно содержат краткое описание и ссылку на сайт. Обязательные элементы `item` — `title`, `description` и `link`, в отличие от элементов `guid` и `category`[1].

Сейчас RSS-фиды считаются устаревшей технологией. Есть несколько факторов, которые привели к созданию новых форматов синдикации. Во-первых, большое количество версий RSS (0.90, 0.91, 0.92, 1.0, 1.1, 2.0) усложнило

[1] Чтобы понять, почему важно использовать GUID в веб-фидах, ознакомьтесь со статьей Джейка Говарда (Jake Howard) How I Messed up my RSS Feed («Как я испортил свою RSS-ленту») (https://oreil.ly/EWmYq).

процедуру валидации фидов. Во-вторых, веб-фид-ридеры поддерживали только ограниченное количество версий. Наконец, в ранних версиях не было возможности расширить фид, добавив новые элементы, поскольку нарушалась совместимость с ридером. В более поздних версиях RSS эта проблема была решена.

СИНДИКАТ И СИНДИКАЦИЯ

В издательском деле *синдикат* — это группа издателей, которые распространяют контент по лицензии, полученной от центральной организации, чтобы расширить охват потребителей. Соответственно, *синдикация* — это продажа контента другим организациям для его публикации или показа сразу в нескольких местах. Синдикация позволяет создателям контента охватить более широкую аудиторию (в печати, на радио или телевидении).

В контексте веб-фидов речь идет о веб-синдикации — процессе распространения контента (записей в блогах, подкастов или видео) на другие сайты или платформы и его последующего агрегирования.

Atom-фид

Стремление устранить недостатки RSS привело к созданию веб-фида Atom. Среди преимуществ Atom — расширение метаданных, поддержка Unicode, которая позволила корректно представлять текстовый контент на любом языке мира, и включение в элементы фида различных типов контента (текст, HTML и XML). Однако важнее всего то, что Atom стал официальной спецификацией веб-фида: в 2005 году он был стандартизирован IETF как формат синдикации в RFC 4287 (https://oreil.ly/5F8c6).

Являясь преемником RSS, Atom представляет собой формат синдикации на основе XML 1.0 (https://oreil.ly/EkB6f) для описания веб-фидов и протокол публикации (https://oreil.ly/KiLTV). Формат Atom-фидов доступен для чтения как человеком, так и компьютером и строится из элементов сущностей, у каждой из которых есть метаданные, подлежащие расширению.

Atom-фид создавался для синдикации веб-контента (блогов, подкастов, новостных заголовков), чтобы распространять его на другие сайты или платформы. Концепция Atom впервые была подробно описана в публикации *Anatomy of a Well-Formed Log Entry* («Структура правильно составленной записи журнала») в блоге Сэма Руби (Sam Ruby) (https://oreil.ly/390Jk).

В примере 7.2 показан Atom-фид с одной записью.

Пример 7.2. Atom-фид с одной записью

```
<?xml version="1.0" encoding="utf-8"?>  ❶
<feed xmlns="http://www.w3.org/2005/Atom">  ❷
  <id>https://example.com/</id>
  <title>Captain's log</title>
  <updated>2024-03-13T08:30:00Z</updated>
  <link rel="self" href="/captainslog"/>
  <author>
    <name>Jack Sparrow</name>
    <uri>https://example.com/</uri>
    <email>jack.sparrow@example.com</email>
  </author>
  <entry xml:lang="en">  ❸
    <id>https://example.com/captainslog/2024/03/13</id>
    <title>Copenhagen vs Warsaw Mermaid</title>
    <updated>2024-03-13T08:30:00Z</updated>
    <link rel="alternate" href="captainslog/2024/03/13"/>
    <summary>The are different</summary>
  </entry>
</feed>
```

❶ Фид начинается с XML-декларации версии 1.0 и указания формата кодировки документа.

❷ `feed` — корневой элемент Atom-фида, который определяет пространство имен XML (`xmlns`) для формата Atom (https://oreil.ly/T7OgS). `id`, `title` и `updated` являются обязательными элементами (https://oreil.ly/UdwCO) Atom-фида, в отличие от `link` и `author`.

❸ Элемент `entry` содержит контент фида. `id`, `title` и `updated` являются обязательными элементами записи (https://oreil.ly/IAlTd), а `link` и `summary` — нет.

Atom-фиды идентифицируются по MIME-типу `application/atom+xml`, и их обычно читают агрегаторы новостей. Фиды можно комбинировать, составляя новый фид из других фидов. Например, новости о баскетболе с разных спортивных сайтов можно объединить в один фид. Синтаксис Atom спроектирован с учетом модульности, и, согласно спецификации, некоторые его элементы являются обязательными. В отличие от RSS-фида Atom-фид позволяет устанавливать отдельный язык для каждой записи, формат даты соответствует спецификации RFC 3339 *(YYYY-MM-DDTHH:MM:SSZ)* (https://oreil.ly/W886O), а элементы Atom-фида можно использовать повторно, в том числе и в документах, не являющихся полным Atom-фидом.

В примере 7.3 элемент `<author>` из примера 7.2 повторно используется в XML-документе, который содержит Atom-элемент `<entry>`. Такой документ можно отправить через HTTP `POST`-запрос на сервер, поддерживающий протокол AtomPub (https://oreil.ly/nAwyP), чтобы создать или обновить веб-ресурсы. CDATA (character data) — это секция XML, которая позволяет клиентам отображать контент без экранирования специальных символов, таких как `<`, `/`, `&`.

Пример 7.3. XML-документ, содержащий запись Atom

```
<?xml version="1.0" encoding="utf-8"?>
<entry xmlns="http://www.w3.org/2005/Atom">
  <title>The Atom draft is finished</title>
  <id>urn:uuid:550e8400-e29b-41d4-a716-446655440000</id>
  <updated>2025-01-30T12:34:59Z</updated>
  <author>
    <name>John Sparrow</name>
    <uri>https://example.com/</uri>
    <email>jack.sparrow@example.com</email>
  </author>
  <content type="html">
    <![CDATA[
      <h1>The Atom draft is finished</h1>
      <p>This is an example of submitting a blog post using AtomPub.</p>
    ]]>
  </content>
</entry>
```

Почти 30 лет назад Atom-фид стал преемником RSS-фида и получил статус официального стандарта, поддерживаемого IETF. Тем не менее оба формата веб-фидов используются до сих пор. Отсутствие управляющей организации, которая могла бы унифицировать более ранние версии RSS и объявить их устаревшими, привело к тому, что отдельные разработчики стали создавать форки RSS-фида, и он разделился на две версии (v1.* и v2.*). Самая используемая на текущий момент — RSS 2.0. Тот факт, что RSS существует и используется до сих пор, вероятно, связан с тем, что он доминировал ранее, будучи первым и единственным веб-фидом. Вторая волна популярности RSS наступила после прекращения работы Google Reader[1], вызванного переходом к централизованным социальным сетям[2].

Теперь, когда мы кратко обсудили формат веб-фидов, можно перейти к его реализации. В следующем разделе мы создадим Atom-фид для нашего WFS.

---

[1] См. статью о Google Reader в Википедии (https://oreil.ly/Mcf7U).

[2] См. статью Дэвида Пирса (David Pierce) Who Killed Google Reader? («Кто уничтожил Google Reader?») (https://oreil.ly/zHKyL).

Реализация

WFS, который мы создаем, нужен для предоставления прогнозов погоды и исторических данных о погоде для заданного города. Мы хотим предоставлять данные широкому кругу клиентов, в том числе веб-фид-ридерам. Для этого добавим веб-фид Atom, который ссылается на веб-страницу с прогнозом погоды. Atom-фид позволит нам рассылать обновления прогноза погоды клиентам, которые на него подписаны.

В нашей реализации используются классы Feed из веб-фреймворка Django (https://oreil.ly/MM3EE). На рис. 7.2 показан ожидаемый результат выполнения упражнений из этой главы.

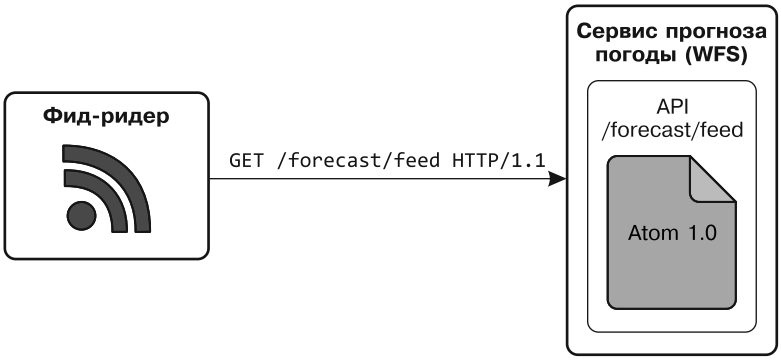


Рис. 7.2. Фиды прогнозов WFS



Каждый город в нашей базе данных имеет уникальный идентификатор (UUID) — во избежание конфликтов между городами с одинаковыми названиями.

Для выполнения упомянутых требований нужно добавить в WFS два эндпоинта API: первый — для отображения HTML-страницы с прогнозом погоды для заданного города, а второй — для отображения Atom-фида, который содержит список городов и перенаправляет читателя на первый эндпоинт. Эти два эндпоинта API для Atom-фидов WFS показаны в примере 7.4.

Пример 7.4. URL Atom-фида

```
# src/django/app/config/urls.py

from core.api.atom.feed import CityAtomFeedView
from core.api.atom.views import WeatherForecastFeedListView
...
```

```
urlpatterns = [
    ...
    path("forecast/feed", CityAtomFeedView(), name="city_feed"),  ❶
    path("forecast/<uuid:city_uuid>", WeatherForecastFeedListView.as_view(),  ❷
      name="city_forecast"
    ),
    ...
]
```

❶ Эндпоинт `/forecast/feed` отображает Atom-фид. За отображение отвечает класс `CityAtomFeedView` (см. пример 7.5 ниже). Фид содержит список городов и ссылку на веб-страницу HTML с прогнозом (второй эндпоинт).

❷ Эндпоинт `/forecast/<uuid:city_uuid>` отображает HTML-страницу с прогнозом погоды для города, идентифицированного по UUID (`city_uuid`).

Рассмотрим реализацию эндпоинта `/forecast/feed`, показанную в примере 7.5.

Пример 7.5. Представление Atom-фида

```
# src/django/app/core/api/atom/feed.py

from django.contrib.syndication.views import Feed as FeedView
from django.utils.feedgenerator import Atom1Feed
...

class CityAtomFeedView(FeedView):  ❶
    feed_guid = "WFS weather feed"  ❷
    feed_type = Atom1Feed  ❸
    language = "en-us"  ❹
    ...

    def items(self, city):  ❺
        return CityRepository.get_all()

    def item_title(self, city):  ❺
        return city.name

    def item_description(self, city):  ❺
        return f"{city.name}, {city.country}, {city.region}"

    def item_link(self, city):  ❺
        return reverse("city_forecast", args=[city.uuid])

    def item_lastupdated(self, city):  ❺
        return city.updated_at
```

❶ Класс `CityAtomFeedView` наследуется от встроенной функциональности Django syndication views. Представление Feed (`FeedView`) — это высокоуровневый фреймворк для генерации RSS- и Atom-фидов.

❷ `feed_guid` присваивает фиду уникальный идентификатор, чтобы предотвратить смешение данных из разных фидов.

❸ `feed_type`, которому присвоено значение `Atom1Feed`, — это генератор, который заставляет представление Feed генерировать Atom-фид.

❹ `language` указывает, что в фиде используется американский английский.

❺ Метод `items()` возвращает список городов, которые будут добавлены в фид в качестве записей. Отдельные методы `item_()` используются, чтобы присваивать значения конкретным записям в фиде. Эти значения содержат заголовок записи, ее описание, ссылку на запись и дату ее последнего обновления.

В примере 7.6 показан фрагмент Atom-фида, сгенерированного классом `CityAtomFeedView()` из примера 7.5. Документ можно скачать, перейдя по адресу http://localhost:8000/forecast/feed[1] после запуска WFS.

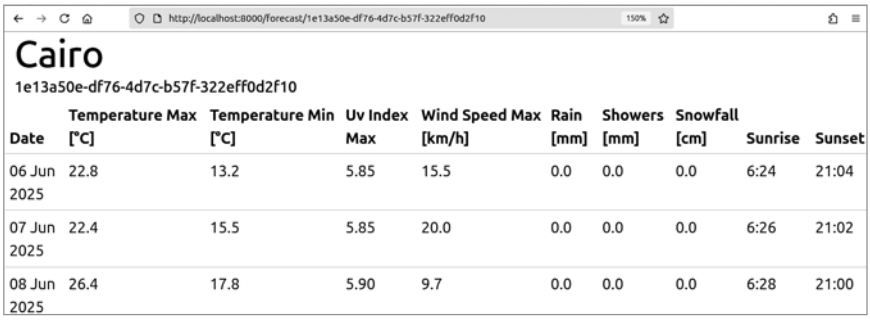
Пример 7.6. Atom-фид прогноза погоды WFS

```
<?xml version="1.0" encoding="utf-8"?>
<feed xmlns="http://www.w3.org/2005/Atom" xml:lang="en-us">
  <title>City Weather Forecast</title>
  <link href="http://localhost:8000/forecast/feed" rel="alternate" />
  <link href="http://localhost:8000/forecast/feed" rel="self" />
  <id>http://localhost:8000/forecast/feed</id>
  <updated>2024-03-15T09:43:02.062628+00:00</updated>
  <subtitle>One week forecast</subtitle>
  <entry>
    <title>Cairo</title>
    <link href="http://localhost:8000/forecast/05666603-d357-48bd-a554-5a5db5a67624"
      rel="alternate" />
    <id>http://localhost:8000/forecast/05666603-d357-48bd-a554-5a5db5a67624</id>
    <summary type="html">Cairo, Egypt, Africa</summary>
  </entry>
  ...
</feed>
```

Ссылка в элементе `entry` указывает на эндпоинт `/forecast/city_uuid`. Эндпоинт определяется методом `item_link()` в классе `WeatherForecastFeedListView`, который генерирует HTML-страницу с прогнозом погоды для конкретного города. Сгенерированная страница показана на рис. 7.3.

---

[1] При работе с GitHub Codespaces используется уникальный URL, содержащий имя Codespace вместо адреса localhost. Этот URL привязан к порту 8000, и его можно найти на вкладке PORTS в VS Code.



Date	Temperature Max [°C]	Temperature Min [°C]	Uv Index Max	Wind Speed Max [km/h]	Rain [mm]	Showers [mm]	Snowfall [cm]	Sunrise	Sunset
06 Jun 2025	22.8	13.2	5.85	15.5	0.0	0.0	0.0	6:24	21:04
07 Jun 2025	22.4	15.5	5.85	20.0	0.0	0.0	0.0	6:26	21:02
08 Jun 2025	26.4	17.8	5.90	9.7	0.0	0.0	0.0	6:28	21:00

Рис. 7.3. Страница прогноза погоды для города

Недостаточно просто реализовать Atom-фид — у пользователя должна быть возможность его прочитать. В следующем разделе мы поговорим о веб-фид-ридерах (агрегаторах) и о том, как их использовать.

Чтение фидов

Фид-ридеры позволяют пользователям подписываться на веб-фиды и агрегировать их в одном месте. Стандартный фид-ридер позволяет отмечать фиды как прочитанные, выполнять в них поиск по ключевым словам и организовывать фиды по категориям. Фид-ридер может быть настольным или мобильным приложением, плагином для браузера или веб-приложением. Среди популярных фид-ридеров — Feedly (https://feedly.com), Inoreader (https://www.inoreader.com), Newsboat (https://newsboat.org) и NewsBlur (https://newsblur.com).



При выборе ридера стоит учитывать, является ли он облачным (относится ли к веб-сервисам) и поддерживает ли он ваш смартфон. Кроме того, вы можете скачать и запустить ридер на своем сервере или устройстве, не используя сторонние онлайн-сервисы.

Чтобы получать обновления о новом контенте, публикуемом в веб-фиде, нужно вставить его заголовок или URL в фид-ридер. Далее ридер загрузит фид и отобразит контент. В примере 7.7 мы будем использовать текстовую консольную программу Newsboat (https://newsboat.org) для чтения нашего фида WFS.

Пример 7.7. Чтение Atom-фида WFS с помощью Newsboat

```
# Настройка лабораторной среды  ❶
cd src/django
docker compose build --build-arg UID=$(id -u) --build-arg GID=$(id -g)
docker compose up --detach --wait
```

```
# Получение Atom-фида  ❷
docker compose exec app bash -c \
       "mkdir --parents ~/.newsboat && \
       echo http://localhost:8000/forecast/feed > ~/.newsboat/urls"

# Чтение фида  ❸
docker compose exec app bash -c \
       "newsboat --refresh-on-start --execute=reload --execute=print-unread"
6 непрочитанных статей
```

❶ Выполните настройку лабораторной среды. Перейдите в соответствующий каталог, соберите Docker-образы и запустите контейнеры.

❷ Добавьте URL реализованного фида в список фидов, известных `newsboat`.

❸ Запустите `newsboat`, чтобы вывести количество непрочитанных статей.

Можно также запустить ридер в интерактивном режиме с помощью команды **`docker compose exec app bash -c "newsboat"`** и просматривать фид (рис. 7.4).

```
newsboat 2.21.0 - Articles in feed 'City Weather Forecast' (6 unread, 6 total)
   1 N  Mar 30    18   Cairo
   2 N  Mar 30    20   Istanbul
   3 N  Mar 30    32   Mexico City
   4 N  Mar 30    30   Sao Paulo
   5 N  Mar 30    24   Sydney
   6 N  Mar 30    16   Tokyo
q:Quit ENTER:Open s:Save r:Reload n:Next Unread A:Mark All Read /:Search ?:Help
```

Рис. 7.4. Интерактивное чтение Atom-фида WFS с помощью Newsboat

Может показаться, что такая простая технология, как Atom или RSS, не несет в себе рисков, однако это не так. В следующем разделе мы рассмотрим распространенные атаки, которым подвержены веб-фиды.

Безопасность

Обычно веб-фид создают для публичного доступа, чтобы любой, у кого есть ссылка, мог его читать. Но если фид содержит непубличный контент или конфиденциальные данные, то для контроля доступа применяют метод аутентификации.

Можно использовать *базовый метод аутентификации* (https://oreil.ly/uuL5y), который предполагает, что вместе с запросом передается имя пользователя и пароль. Этот метод не рекомендуется использовать, поскольку в нем постоянные учетные данные добавляются в каждый запрос. Альтернативный

подход — *аутентификация на основе токена*. Этот метод снижает риски раскрытия учетных данных, но требует частой повторной генерации токенов, поскольку у них есть срок действия.

Контент веб-фида передается через HTTP-эндпоинт `GET`, что делает фид уязвимым для ряда атак на уровне HTTP (даже если используется версия с шифрованием). Например, на эндпоинт фида можно направить DoS-атаку, при которой злоумышленник перегружает сервер фида, направляя на его эндпоинт множество HTTP-запросов. Чтобы снизить воздействие DoS-атаки, можно противодействовать ей на нескольких уровнях модели OSI (см. раздел «TCP/IP и модель OSI» главы 3).

На сетевом уровне (уровень 3) можно блокировать запросы, отправленные с одного IP-адреса или из определенного географического региона *(геоблокировка)*. Кроме того, для установки ограничений используются брандмауэры или маршрутизаторы (либо и то и другое): например, можно ограничить количество пакетов в секунду, байтов в секунду или количество передаваемых SYN-пакетов.

На прикладном уровне (уровень 7) можно подавлять интенсивность запросов с определенного IP-адреса, устанавливая квоты на количество запросов в секунду, минуту, час или день.

Кроме того, чтобы снизить прямое воздействие DoS-атаки на сервер, можно использовать балансировщик нагрузки, который распределит запросы между несколькими серверами и уменьшит вероятность возникновения единой точки отказа.

Еще один тип атаки на веб-фиды — *инъекция в RSS-фид* (RSS feed injection). Злоумышленник может внедрить вредоносный контент в документ фида с помощью межсайтового скриптинга (XSS) или изменить уже имеющийся внутри документа контент. Цель атаки — изменить содержимое ленты так, чтобы оно перенаправляло пользователя на другой сайт или включало в себя вредоносный код JavaScript. Эта атака возможна, поскольку содержимое ленты не проходит валидацию ни на стороне клиента, ни на стороне сервера. Последствия атаки — риски для безопасности пользователя, который перешел по вредоносной ссылке, и ущерб репутации владельца фида. Для защиты от инъекции в RSS-фид необходимо очищать содержимое фида и поддерживать ПО RSS в актуальном состоянии с установленными патчами безопасности (как на стороне клиента, так и на стороне сервера).

У большинства почтовых провайдеров есть инструменты для обнаружения вредоносного контента в письмах, что иногда приводит к попаданию важных писем в спам. У RSS-клиентов таких инструментов нет, поэтому пользователь

сам отвечает за проверку содержания фида и ссылок перед переходом. Это же относится к владельцу фида, который должен отслеживать наличие вредоносного контента и удалять его.

Веб-фиды: преимущества и недостатки

Начнем с преимуществ веб-фида как инструмента распространения и потребления контента.

Широковещание

Клиенты веб-фидов периодически запрашивают обновления через API. Когда содержимое фида меняется, клиенты узнают об этом, как будто информация транслируется им напрямую.

Своевременные обновления

Частоту отправки запросов к веб-фиду можно настроить на короткий интервал (каждые несколько секунд), что позволяет пользователям получать последние обновления сразу после публикации.

Агрегирование контента

Веб-фид-ридер агрегирует несколько фидов, позволяя пользователю читать контент из разных источников в одном месте вместо того, чтобы посещать каждый сайт отдельно.

Синдикация контента

Веб-фид позволяет владельцу публиковать контент (записи в блоге, подкасты или видео) на других сайтах или платформах и расширять охват пользователей, выходя за пределы исходной платформы.

Конфиденциальность

Ридеры, которые не требуют входа в систему, повышают анонимность пользователей, позволяя получать доступ к контенту без прямого взаимодействия с исходной платформой. Благодаря этому объем собираемых персональных данных уменьшается.

Есть у веб-фидов и несколько слабых сторон.

Отсутствие контракта

Веб-фид не предполагает контракта между поставщиком и потребителем. Если поставщик перестает создавать или обновлять фиды, потребитель не знает, будет ли еще обновляться фид. Иными словами, клиент рассчитывает, что поставщик будет поддерживать актуальность фида.

Низкая вовлеченность пользователей онлайн

Недостаточная вовлеченность пользователей веб-фидов на онлайн-платформах создает проблемы для авторов, издателей и платформ социальных сетей, особенно в части доходов от рекламы. Подписка на веб-фид не гарантирует, что пользователь будет проводить время на платформе. Для ее владельца это проблематично, поскольку ставка делается на длительное взаимодействие с пользователем для максимизации рекламы и сбора данных. Веб-фиды же препятствуют этому.

Проблемы безопасности

Содержимое веб-фида доступно через HTTP-эндпоинт `GET`. Из-за этого веб-фид становится уязвимым для атак на уровне HTTP, например DoS-атак или инъекций в RSS-фид. По этой причине важно поддерживать ПО для работы с фидами в актуальном состоянии (и на клиенте, и на сервере) и своевременно устанавливать патчи безопасности.

Информационная перегрузка

Агрегирование веб-фидов может быть одновременно преимуществом и недостатком. Пользователь, читающий слишком много фидов, может почувствовать, что он перегружен информацией, и вообще перестать читать контент.

Фрагментация

У формата веб-фидов есть множество версий. Соответственно, веб-фид-ридер должен поддерживать разные форматы синдикации, их версии и расширения.

Отсутствие мониторинга активности пользователей

Автор контента обычно хочет знать, как пользователи взаимодействуют с его веб-фидом. Однако самостоятельно отслеживать, кто обращается к фиду, как часто читает статьи или откуда приходит, как правило, неудобно. Отследить метрики веб-фида возможно, но для этого требуются специальные методы:

- анализ логов сервера — подсчет количества HTTP-вызовов к URL веб-фида и парсинг логов для получения временных меток, данных об агентах пользователя или IP-адресах;
- параметры запроса — добавление параметров запроса, например `?source=blog`, чтобы получить информацию о том, откуда приходят клиенты;
- редирект-ссылка — предоставление промежуточной ссылки, которая перенаправляет на контент веб-фида и одновременно собирает данные о пользователе.

Упражнения

1. Если вы интересуетесь архитектурой ПО, то посетите сайт Developer to Architect (https://oreil.ly/2kxDW) и подпишитесь на его RSS-фид. URL фида можно найти в нижней части страницы. Для этого упражнения можно использовать любой RSS-ридер или следовать инструкциям из примера 7.7.
2. Чтобы не пропустить новые видео от Modern Software Engineering (https://oreil.ly/f2EbJ), подпишитесь на Atom-фид его YouTube-канала. Для этого упражнения нам понадобятся идентификатор YouTube-канала и URL Atom-фида канала. Идентификатор можно найти в исходном коде страницы канала — перейдите по адресу https://www.youtube.com/@ModernSoftwareEngineeringYT, щелкните правой кнопкой мыши, выберите Просмотреть код страницы и найдите `youtube.com/channel/` (идентификатор канала должен выглядеть так: `UCCfqyGl3nq_V0bo64CjZh8g`). Далее сформируйте URL Atom-фида канала, добавив идентификатор канала к следующему URL: https://www.youtube.com/feeds/videos.xml?channel_id=UCCfqyGl3nq_V0bo64CjZh8g. Для этого упражнения также можно использовать любой RSS-ридер или следовать инструкциям из примера 7.7.
3. Ваше следующее задание — прочитать Atom-фид с эндпоинта `forecast/feed`, который мы реализовали в этой главе. В GitHub Codespaces есть предустановленный веб-фид-ридер Newsboat (https://newsboat.org) — его вы и будете использовать. Чтобы выполнить упражнение, запустите проект Django с помощью команды **`docker compose up --detach --wait`**, измените видимость порта Codespaces на «Публичный» и настройте ридер `newsboat` на чтение фидов по URL проекта в Codespaces. Чтобы изменить видимость URL на «Публичный», на вкладке PORTS щелкните правой кнопкой мыши на порте `8000` и в меню Port Visibility выберите Public. Ответ к этому упражнению находится в репозитории кода книги (см. файл `src/django/docs/solutions/web-feeds-exercise.md`).

Резюме

В этой главе мы обсудили происхождение и эволюцию веб-фидов, форматы RSS и Atom, а также их плюсы и минусы. Кроме того, выполнив примеры и упражнения к этой главе, вы научились реализовывать Atom-фид и взаимодействовать с ним с помощью фид-ридера (агрегатора).

ГЛАВА 8

gRPC

RPC расшифровывается как *Remote Procedure Call («удаленный вызов процедур»)*. Как видно из названия, стиль RPC предполагает, что взаимодействие с API выполняется через активацию удаленных процедур (функций). Для API RPC характерны операционно-ориентированные интерфейсы, где операция описывает конкретное действие, например CreateUser («Создать пользователя») или GenerateReport («Сгенерировать отчет»). В этом отношении RPC противоположен стилю REST, который ориентирован на получение и изменение ресурсов[1], таких как User и Report, с использованием стандартных HTTP-методов `GET`, `POST`, `PUT` и `DELETE`.

Сейчас самой распространенной реализацией RPC является gRPC[2]. В спецификации gRPC характеризуется как «высокопроизводительный универсальный RPC-фреймворк с открытым исходным кодом»[3]. gRPC был создан компанией Google, а в 2017 году проект передали компании Cloud Native Computing Foundation, которая сейчас поддерживает его на сайте `https://grpc.io`.

В начале этой главы мы обсудим общие цели использования RPC. Далее речь пойдет о gRPC и его возможностях. Мы рассмотрим двоичный формат сериализации Protocol Buffers и протокол HTTP/2 (транспортный механизм, который использует gRPC; см. раздел «HTTP/2» в главе 4). На примере реализации службы ECHO мы разберем четыре типа RPC, которые поддерживает gRPC: унарный, серверный потоковый, клиентский потоковый и двунаправленный потоковый. Затем с помощью gRPC мы расширим веб-фид WFS, описанный

---

1 См. раздел «Модели зрелости API» в главе 5.

2 Значение буквы g в названии меняется в каждом релизе. Полный список значений можно найти в документе репозитория gRPC на GitHub: doc/g_stands_for.md (https://oreil.ly/QTdMM).

3 Помимо gRPC, существуют и другие технологии на основе RPC, среди которых Apache Thrift (https://oreil.ly/3llo2), tRPC (https://oreil.ly/A8n2b) и JSON-RPC (https://www.jsonrpc.org).

в главе 7, добавив контент, который генерирует локально запущенная LLM. В заключение мы обсудим безопасность gRPC, создание документации, а также его сильные и слабые стороны.

Удаленный вызов процедур

Язык программирования Fortran II, созданный в 1958 году[1], стал первым высокоуровневым языком, где использовалось понятие процедур. *Процедуры* (также называемые *подпрограммами*, *функциями* или *методами*) — это блоки кода, предназначенные для выполнения конкретных задач. Процедуры помогают разработчикам понимать и структурировать программы[2]. Из Fortran II процедуры распространились в область распределенных систем, которые в 1970-х годах работали в ранних компьютерных сетях, таких как ARPANET. Тогда же была введена концепция RPC. Она была подробно описана в докторской диссертации Брюса Джея Нельсона (Bruce Jay Nelson) в начале 1980-х годов[3].

Технология RPC не просто имела сходство с механизмом локального (на одном компьютере) вызова процедур — она позволяла решить две основные проблемы распределенных систем[4]:

- взаимодействие между сервисами, использующими разные форматы данных;
- взаимодействие по ненадежным сетям.

Структура RPC, предложенная Нельсоном, не затрагивала другие проблемы, такие как отказы сервисов или отсутствие глобального времени. Некоторые из этих проблем решаются в современных реализациях RPC, например путем использования тайм-аутов вызовов.

---

1 См. с. 78 в книге: *O'Regan G.* A Brief History of Computing. — Springer, 2008.

2 *White J. E.* A High-Level Framework for Network-Based Resource Sharing («Высокоуровневый фреймворк для сетевого совместного использования ресурсов») // AFIPS '76: Материалы международной конференции National Computer Conference and Exposition, 7–10 июня 1976. — P. 561–570. https://oreil.ly/B1mGW.

3 *Birrell A. D., Nelson B. J.* Implementing Remote Procedure Calls («Реализация удаленного вызова процедур») // ACM Transactions on Computer Systems (TOCS), 1984. — № 1. — P. 39–59. https://oreil.ly/vZUwd

4 См. раздел Remote Procedure Call («Удаленный вызов процедур») в книге Computer Networks: A Systems Approach (https://oreil.ly/CKCd5) Ларри Петерсона (Larry Peterson) и Брюса Дэви (Bruce Davie). (*Петерсон Л., Дэви Б.* Компьютерные сети. Системный подход. — М., 2026.)

Происхождение gRPC

Технология gRPC берет начало от Stubby — внутреннего фреймворка для удаленного вызова процедур, который был разработан компанией Google и использовался с начала 2000-х. Stubby был тесно связан с системой разрешения имен и балансировщиками нагрузки Google, что помешало компании создать открытый RPC-фреймворк на его основе. В 2013 году в Google начали разработку gRPC с нуля[1], и процесс шел параллельно с завершением спецификации HTTP/2. Первый стабильный релиз gRPC состоялся в 2016 году.

Цели проектирования gRPC схожи с теми, которые описаны Нельсоном, однако содержат и другие функции, упрощающие создание распределенных систем, такие как проверка работоспособности системы, балансировка нагрузки и отказоустойчивость[2]. В следующих разделах мы рассмотрим основные возможности gRPC.

Сериализация protobuf

Форматы данных могут различаться по таким признакам, как порядок байтов (endianness), размер данных (например, 32- или 64-битный) или кодировка символов (например, ASCII или Unicode). При работе с сетевыми сервисами, использующими разные форматы данных, используются три подхода: sender-makes-right («отправитель всегда прав»), receiver-makes-right («получатель всегда прав») и подход с использованием промежуточного представления[3].

Подход *sender-makes-right* заключается в том, что каждый отправляющий сервис знает все форматы, которые используются принимающими сервисами. Этот подход обязывает отправителя отправлять сообщение в формате, который поддерживается получателем.

При противоположном подходе сообщение отправляется в том формате, который поддерживает отправитель, а получатель вынужден преобразовывать

---

[1] Чтобы узнать больше об истории gRPC, см. видео Ten Years of gRPC («Десять лет gRPC») на YouTube (https://oreil.ly/RtehX) от Джун-Ю (Джины) Йе (Jung-Yu (Gina) Yeh) и Ричарда Белвилля (Richard Belleville).

[2] См. статью gRPC Motivation and Design Principles («Цели и принципы разработки gRPC») (https://oreil.ly/WwH61) Луи Райана (Louis Ryan).

[3] См. раздел Presentation Formatting («Форматирование представления данных») в книге Computer Networks: A Systems Approach (https://oreil.ly/y6Flc) Ларри Петерсона (Larry Peterson) и Брюса Дэви (Bruce Davie).

сообщение в свой внутренний формат. Этот подход, называемый *receiver-makes-right*, предполагает, что получатель знает все форматы, используемые отправителями.

В третьем подходе предполагается, что и отправитель, и получатель используют единый общий формат, называемый *промежуточным представлением*. При этом подходе операцию преобразования выполняют обе стороны, но от них требуется поддержка преобразования только одного формата. Практической реализацией подхода являются операции сериализации и десериализации (см. врезку «Кодирование и сериализация» в главе 2).

В gRPC используется подход с промежуточным представлением, а в качестве формата сериализации по умолчанию применяется Protocol Buffers (https://protobuf.dev), сокращенно — *protobuf*[1]. Кроме того, Protocol Buffers используется в качестве языка описания API (Interface Description Language, IDL): он определяет RPC-методы и их сообщения запроса/ответа. Определения Protobuf хранятся в файлах `.proto`, доступных для чтения человеком. Структуры данных из этих файлов компилируются в двоичный формат, который, в отличие от текстового, занимает меньше места в хранилище и ускоряет парсинг. Использование этого компактного формата для обмена сообщениями по сети повышает производительность gRPC.

В примере 8.1 показана полная процедура определения RPC в protobuf. Этот файл нужно написать самостоятельно: он содержит RPC из одного запроса и одного ответа. На языке gRPC такой вызов называется *унарным* — в нем и запрос, и ответ состоят из одного сообщения. Такой тип RPC соответствует модели «запрос — ответ», известной нам по API на основе HTTP. gRPC поддерживает и другие типы вызовов, основанные на потоковой передаче данных. Их мы рассмотрим в разделе «Четыре типа RPC» далее в этой главе.

Сервис gRPC содержит одно или несколько определений RPC-методов. В нашем случае в `EchoService` один RPC-метод `DemoUnary`. Клиент вызывает `DemoUnaryRPC`, используя сообщение `DemoUnaryRequest`, а сервер выполняет этот вызов. Сервер возвращает клиенту содержимое полученного запроса в виде сообщения `DemoUnaryResponse`[2]. На основе этого protobuf-определения gRPC генерирует заготовку кода, которая используется для реализации клиента и сервера gRPC. Генерацию кода в gRPC мы подробно рассмотрим в примере 8.4, а сейчас остановимся подробнее на формате protobuf.

---

[1] Основная цель Protocol Buffers — предоставить механизмы сериализации данных. gRPC также поддерживает другие форматы сериализации, включая JSON.

[2] Концепция службы ECHO обсуждалась в разделе «Реализация службы TCP ECHO» главы 3.

Пример 8.1. Файл protobuf для унарного RPC в gRPC

```
// src/grpc/src/echo/echo/proto/echo/v1/echo.proto

syntax = "proto3";  ❶

package echo.v1;  ❷

message DemoUnaryRequest {  ❸
  // Текстовое содержимое запроса.  ❹
  string content = 1;  ❺
}

message DemoUnaryResponse {  ❻
  // Текстовое содержимое ответа.
  string content = 1;
}

// Сервис возвращает клиенту полученное содержимое.  ❼
service EchoService {  ❽
  // Этот метод демонстрирует унарный RPC.
  rpc DemoUnary(DemoUnaryRequest) returns (DemoUnaryResponse);  ❾
}
```

❶ В этом файле protobuf используется последняя версия Protocol Buffers.

❷ Чтобы передавать информацию об изменениях, в protobuf-определении в Protocol Buffers есть встроенная поддержка версионирования. Здесь пакету `echo` присвоена версия `v1`.

❸ Этот блок определяет сообщение запроса. Формат protobuf спроектирован так, чтобы схемы данных могли изменяться со временем. Чтобы упростить процесс внесения изменений, рекомендуется отдельно определять сообщения запроса и ответа для каждого RPC[1]. Общее соглашение по именованию — добавлять к имени RPC суффиксы `Request` и `Response`.

❹ Строки, начинающиеся с `//`, protobuf интерпретирует как комментарии. Файлы protobuf выступают в качестве документации API, которая записывается в виде комментариев. Этот комментарий в примере избыточен, так как его смысл легко выводится из кода. Мы добавили его, чтобы напомнить о необходимости документировать protobuf-файлы[2].

---

[1] Подробнее об этой рекомендации см. в разделе Create Unique Protos per Method («Создавайте уникальные определения Protobuf для каждого метода») в документации Protocol Buffers (https://oreil.ly/6Ex2v). Кроме того, рекомендуется размещать каждое сообщение в отдельном файле (https://oreil.ly/Q6jWQ), но в данном случае для удобства чтения это правило не соблюдается.

[2] См. раздел Precisely, Concisely Document Most Fields and Messages («Обеспечьте точное и лаконичное документирование большинства полей и сообщений») в документации Protocol Buffers (https://oreil.ly/7ik1M).

❺ В качестве первого поля в сообщении запроса используется поле типа string с именем `content`. Номера полей, например `1`, нельзя повторно использовать в пределах одного сообщения.

❻ Этот блок определяет сообщение ответа. Сообщения запроса и ответа состоят из одного поля типа string, но определены как отдельные сообщения, чтобы упростить внесение изменений в будущем. В руководствах по gRPC иногда встречается альтернативный подход, не рекомендуемый с точки зрения эволюции protobuf, — определить одно сообщение `DemoUnaryMessage` вместо отдельных сообщений `DemoUnaryRequest` и `DemoUnaryResponse`.

❼ Строки комментариев, расположенные перед определениями сервиса и RPC-методов, будут добавлены в автоматически генерируемую документацию сервиса.

❽ Этот блок определяет gRPC-сервис. Сервис может содержать несколько RPC.

❾ Строки, начинающиеся с `rpc`, определяют RPC-методы. RPC представляет действие, и его имя обычно начинается с глагола. Любой RPC-метод принимает и возвращает только один аргумент (сообщение). Однако сообщение может содержать много полей: оно может иметь вложенную структуру и использовать другие типы сообщений в качестве полей.



Формат protobuf спроектирован так, чтобы схемы данных могли изменяться со временем. Здесь мы не будем углубляться в детали внесения обратно или прямо совместимых изменений в protobuf. Отметим только, что добавление нового поля в запрос не приведет к сбою сервера — тот проигнорирует любые неизвестные поля. Если в запросе нет ожидаемого поля, то ему присваивается значение по умолчанию. Чтобы удалить поле из сообщения protobuf, его сначала помечают как устаревшее. После того как поле перестает использоваться в клиентском коде, его можно удалить из protobuf, но номер удаленного поля необходимо зарезервировать, чтобы оно не использовалось повторно[1].

Узнать больше о внесении изменений в protobuf можно в материалах Language Guide («Руководство по языку») (https://oreil.ly/8Ywtm) и API Best Practices («Лучшие практики API») (https://oreil.ly/l9jq9), а также в докладе Эрика Андерсона (Eric Anderson) Modifying gRPC Services Over Time («Как gRPC-сервисы изменяются со временем») (https://oreil.ly/hjtgQ). Можно также использовать форматтер, линтер и детектор критических изменений, которые предоставляет buf (https://oreil.ly/D5EiI).

[1] См. раздел Deleting Fields («Удаление полей») в документации protobuf (https://oreil.ly/HqsyX).

Прежде чем продолжить обсуждение реализации RPC в gRPC, настроим лабораторную среду. Следуйте инструкциям из примера 8.2. Команды необходимо выполнять, начиная с корневого каталога репозитория с кодом книги.

Пример 8.2. Настройка лабораторной среды gRPC

```
cd src/grpc  ❶
docker compose build --build-arg UID=$(id -u) --build-arg GID=$(id -g)  ❷
docker compose up --detach –wait  ❸
```

❶ Перейдите в каталог `src/grpc`. Все команды в этой главе выполняются из этого каталога, если не указано другое.

❷ Соберите образы контейнеров.

❸ Запустите два контейнера: клиент и сервер. Кроме того, создайте ключи и сертификаты для TLS.

Реализация функциональности «запрос — ответ» унарного RPC, аналогичной модели «запрос — ответ» в стиле REST, показана в примере 8.3. Чтобы выполнить инструкции, потребуется окно с тремя панелями. Их можно открыть с помощью `tmux` (введение в `tmux` см. в примере B.2 во вступлении).

Пример 8.3. Последовательность выполнения демонстрации унарного RPC (требуются настройки из примера 8.2)

```
# Терминал  ❶
tmux new-session \; split-window -v \; split-window -v \;

# Верхняя панель  ❷
SECURITY=insecure
docker compose exec server bash -ci \
       "PYTHONPATH=./src/echo python src/echo/echo/server.py ${SECURITY}"

# Центральная панель  ❸
RPC=unary SECURITY=insecure
PCAP_FILE=/tmp/echo_rpc_${RPC}_${SECURITY}.pcap
docker compose exec client bash -c \
       "sudo rm -f ${PCAP_FILE} && \
       sudo tcpdump -w ${PCAP_FILE} 'port 50051' && \
       cp ${PCAP_FILE} tests"

# Нижняя панель  ❹
RPC=unary SECURITY=insecure
docker compose exec client bash -ci \
       "PYTHONPATH=./src/echo python src/echo/echo/client.py ${RPC} ${SECURITY}"

# Вывод:
Client request: Hello, Serialized message: 0a0548656c6c6f
Ответ сервера: Hello
```

❶ Создайте три панели с помощью `tmux`.

❷ Запустите gRPC-сервер. В этой демонстрации переменная окружения `SECURITY=insecure` отключает шифрование TLS для последующего анализа захваченного сетевого трафика. Код сервера подробно будет рассмотрен в примере 8.6.

❸ Запустите захват пакетов на клиенте с помощью `tcpdump`. Захваченный трафик сохранится в файл `echo_rpc_unary_insecure.pcap` в каталоге `tests`[1].

❹ Вызовите gRPC-клиент. Его код будет рассмотрен в примере 8.7. Клиент отправляет серверу текст `Hello`, а сервер возвращает его обратно клиенту. Шестнадцатеричное значение `0a0548656c6c6f` содержит байты сообщения запроса, сериализованного с помощью protobuf (см. подробное объяснение в разделе «Формат передачи данных protobuf» далее в этой главе).

Компилятор `protoc` protobuf принимает файл protobuf на входе и генерирует исходный код, который служит основой для реализации gRPC-клиента и сервера, используемых в демонстрации. Детали процесса генерации кода мы рассмотрим в следующем разделе.

Генерация кода

Автоматическая генерация кода (или создание его вручную) на основе имеющейся спецификации API характерна для подхода API-first[2]. В случае gRPC API спецификацией интерфейса служит файл protobuf. Разработчик не должен изменять код, сгенерированный из protobuf. Обычно код выступает в качестве основы, которую разработчик расширяет для реализации конкретной функциональности приложения.

В примере 8.4 показано, как компилятор protobuf (`protoc`) генерирует код из файла `echo.proto`. У конкретного языка программирования может быть своя обертка для `protoc`. Например, в Python есть пакет `grpcio-tools`, который предоставляет доступ к `protoc` через модуль `grpc_tools.protoc`[3]. Если такой обертки нет, то используется универсальный компилятор `protoc`.

---

[1] Сгенерированный захват пакетов пока не представляет интереса, но он будет использоваться в разделе «Формат передачи данных protobuf» далее в этой главе. В этом разделе речь пойдет о двоичном формате сообщений protobuf, которыми обмениваются клиент и сервер во время RPC-вызова.

[2] Подход API-first был описан в одноименной врезке в главе 1.

[3] Обратите внимание на разницу между именем пакета и именем модуля. Будьте внимательны при установке программного обеспечения из интернета, чтобы избежать тайпсквоттинга (атаки подмены доменных имен).



Имена и расположение генерируемых файлов различаются в зависимости от языка программирования[1]. Разработчики gRPC из Google рекомендуют рассматривать сгенерированные файлы как артефакты сборки, размещать их в отдельном каталоге и не хранить в системе контроля версий. Так действует инструмент Bazel (https://bazel.build) от Google. Хранить файлы в системе контроля версий вместе с остальным кодом приложения можно, если речь идет о небольших проектах на Python с gRPC. В этом случае файлы упаковываются как обычные Python-модули. Такой подход упрощает поддержку проекта[2]. Чтобы разместить сгенерированные файлы в одном каталоге с файлами protobuf, используется аргумент с необычным значением --proto_path.

Пример 8.4. Генерация кода с помощью компилятора protoc (требуются настройки из примера 8.2)

```
docker compose exec client bash -ic \
       "cd src/echo && \
       python -m grpc_tools.protoc \
       --proto_path=echo/proto/echo/v1=echo/proto/echo/v1 \
       --python_out=. --grpc_python_out=. \ echo/proto/echo/v1/echo.proto"
```

В процессе генерации кода будут созданы два файла: `echo_pb2.py` и `echo_pb2_grpc.py`. В нашем примере они расположены в том же каталоге, что и файл `echo.proto`.

Файл echo_pb2.py

Содержит данные в двоичном формате и метаданные для сообщений, сервиса и RPC-методов.

Файл echo_pb2_grpc.py

Содержит заглушку (stub code) для клиента и сервера gRPC, а также дополнительный код для упрощения создания сервера. Фрагмент этого файла показан в примере 8.5.

---

[1] Общий обзор см. в официальной документации protobuf в разделе What's Generated from Your .proto? («Что генерируется из .proto-файла?») (https://oreil.ly/M0EP3) и в документации gRPC (https://oreil.ly/Beo04). Обратите внимание: для некоторых языков можно избежать статической генерации кода и использовать файлы protobuf напрямую из клиентского и серверного приложений, но такой подход менее распространен.

[2] Этот подход позволяет избежать операционных накладных расходов, связанных с внедрением дополнительных инструментов сборки. Об этом говорил Алекс Игл (Alex Eagle) в докладе Building Monorepo gRPC Services with Bazel («Создание gRPC-сервисов в монорепозитории с использованием Bazel») (https://oreil.ly/QoArk): «Бермудский треугольник инструментов разработки Google: Bazel, Protobuf, gRPC. Выглядит как тропический рай. Вы залетаете туда, все ваши приборы сходят с ума, все превращается в ошибки, вы полностью дезориентированы, и разработчик оттуда уже никогда не вернется».



Заглушки gRPC выполняют сериализацию и десериализацию сообщений и предоставляют интерфейс для вызова удаленных процедур.

Пример 8.5. Автоматически сгенерированный файл echo_pb2_grpc.py

```
# src/grpc/src/echo/echo/proto/echo/v1/echo_pb2_grpc.py

import grpc  ❶
import echo_pb2  ❷

class EchoServiceStub(object):  ❸
  """Сервис возвращает клиенту содержимое полученного сообщения."""

  def __init__(self, channel):
    self.DemoUnary = channel.unary_unary(
      '/echo.v1.EchoService/DemoUnary',  ❹
      request_serializer=echo_pb2.DemoUnaryRequest.SerializeToString,  ❺
      response_deserializer=echo_pb2.DemoUnaryResponse.FromString)

class EchoServiceServicer(object):  ❻
  def DemoUnary(self, request, context):  ❼
    """Этот метод демонстрирует унарный RPC."""
    context.set_code(grpc.StatusCode.UNIMPLEMENTED)
    context.set_details('Method not implemented!')
    raise NotImplementedError('Method not implemented!')

def add_EchoServiceServicer_to_server(servicer, server):  ❽
  rpc_method_handlers = {
    'DemoUnary': grpc.unary_unary_rpc_method_handler(
       servicer.DemoUnary,
       request_deserializer=echo_pb2.DemoUnaryRequest.FromString,  ❾
       response_serializer=echo_pb2.DemoUnaryResponse.SerializeToString)
  }
```

❶ Модуль `grpc` поставляется с Python-пакетом `grpcio` (https://oreil.ly/SCGqw) и предоставляет основные функции gRPC, такие как поддержка каналов, серверов и потоков.

❷ Модуль `echo_pb2` был сгенерирован из файла определения protobuf с помощью компилятора `protoc`. Этот модуль содержит структуры данных, которые используются при сериализации и десериализации сообщений protobuf и вызове сервисов gRPC.

❸ Класс `EchoServiceStub` предназначен для создания заглушки на стороне клиента. В Python на это указывает суффикс Stub в имени класса. Слово *stub* есть в сгенерированном коде клиентской заглушки в большинстве языков, поддерживаемых gRPC.

❹ URI-путь этого RPC-сервиса содержит метод, обслуживаемый сервером gRPC и вызываемый клиентской заглушкой. gRPC скрывает от разработчика

детали сетевого вызова — URI жестко задается в автоматически сгенерированном коде.

❺ В клиентской заглушке запрос сериализуется с помощью метода `SerializeToString()`, а ответ десериализуется с помощью метода `FromString()`. Эти методы предоставляются protobuf и работают с целым сообщением запроса или ответа, а не с отдельными полями. Хотя название `String` подразумевает строковый тип данных, оно обозначает последовательность байтов (в Python — `<class bytes>`, а не `<class str>`). Подробнее об этой сериализации см. в примере 8.7 ниже.

❻ Класс сервиса (`servicer`) содержит все RPC-методы, принадлежащие одному сервису gRPC. Один gRPC-сервер может обслуживать несколько таких классов.

❼ Метод `DemoUnary`, отвечающий за формирование ответа на запрос клиента, реализуется разработчиком. В этом методе выполняется бизнес-логика приложения на стороне сервера.

❽ Реализованный класс сервиса регистрируется на сервере для обработки RPC-вызовов.

❾ Аналогично клиентской заглушке заглушка на стороне сервера, которую обобщенно называют *обработчиком*, десериализует входящие запросы и сериализует исходящие ответы.

Сервер и клиент gRPC ECHO

Используя код, сгенерированный компилятором `protoc`, мы приступаем к реализации сервера и клиента gRPC для `EchoService`. На рис. 8.1 представлена общая схема потока данных в gRPC. В *среде выполнения* gRPC ключевым компонентом на стороне клиента является *канал*, который запускается *клиентским приложением*. Канал — это абстракция сетевого взаимодействия с сервером gRPC. На стороне *gRPC-сервера* основным компонентом является *сервер*, запускаемый *серверным приложением*.

Клиентское приложение передает сообщение запроса *клиентской заглушке*, которая сериализует сообщение и инициирует RPC-вызов. Канал отвечает[1]

[1] Более подробное описание функциональности канала, в том числе балансировки нагрузки на стороне клиента, см. в докладах Google: Deep Dive: gRPC («Глубокое погружение: gRPC») Юйсюаня Ли (Yuxuan Li) (https://oreil.ly/RKc3e) и Load Balancing in gRPC («Балансировка нагрузки в gRPC») Исвара Сваминатана (Easwar Swaminathan) (https://oreil.ly/TNLn0). Обратите внимание на то, что gRPC поддерживает долгоживущие TCP-соединения для снижения задержек, что часто приводит к дисбалансу нагрузки на бэкендах балансировщика — например, на подах за сервисом Kubernetes ClusterIP.

за создание блока «Транспорт» и отправку RPC-вызова на соответствующий сервер gRPC. После того как первичное соединение установлено, прослушивающий сервер gRPC создает свой блок «Транспорт» и передает сообщение *серверной заглушке* (обработчику). Она десериализует сообщение и локально вызывает серверное приложение. Оно, в свою очередь, выполняет бизнес-логику на стороне сервера и передает сообщение ответа серверной заглушке для сериализации. Это сообщение проходит через транспорт, десериализуется клиентской заглушкой и возвращается клиентскому приложению.

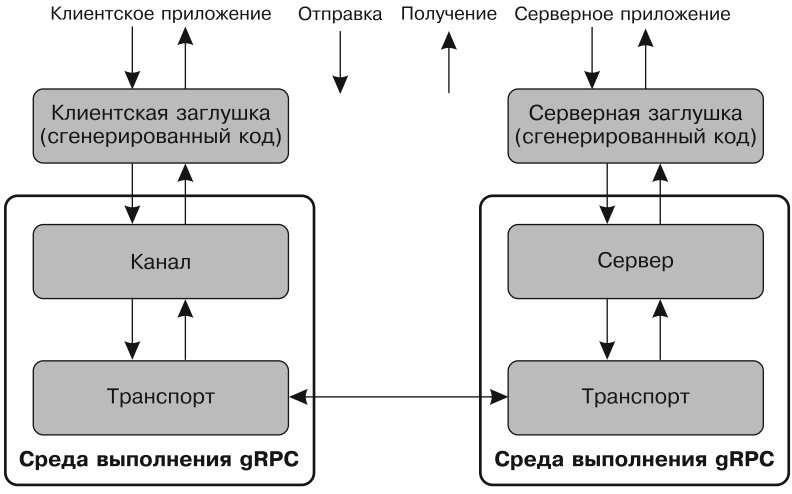


Рис. 8.1. Схема передачи данных в gRPC

В примере 8.6 показана полная реализация gRPC-сервера, предоставляющего унарный RPC-метод. Реализация клиента приведена в примере 8.7. И сервер, и клиент основаны на автоматически сгенерированном коде. Общая структура сервера и клиента одинакова, независимо от того, какой тип RPC используется — унарный или потоковый. Потоковые RPC рассматриваются далее, в разделе «Четыре типа RPC» этой главы.

Пример 8.6. Реализация унарного RPC для gRPC-сервера (ваш код)

```
# src/grpc/src/echo/echo/server.py

import concurrent
import sys

import grpc
from grpc_reflection.v1alpha import reflection

import echo.proto.echo.v1.echo_pb2 as pb2
import echo.proto.echo.v1.echo_pb2_grpc as pb2_grpc
```

```
class EchoServiceServicer(pb2_grpc.EchoServiceServicer):  ❶
    def DemoUnary(self, request, context):  ❷
        content = request.content  ❸
        return pb2.DemoUnaryResponse(content=content)  ❹

def get_ssl_server_credentials(use_custom_ca=True, use_mtls=True):
    private_key = open("src/server.key", "rb").read()
    certificate_chain = open("src/server.crt", "rb").read()
    root_certificates = open("src/ca.crt", "rb").read() if use_custom_ca else None
    return grpc.ssl_server_credentials(
        [(private_key, certificate_chain)],
        root_certificates=root_certificates,
        require_client_auth=use_mtls)

server = grpc.server(concurrent.futures.ThreadPoolExecutor())  ❺
pb2_grpc.add_EchoServiceServicer_to_server(  ❻
    EchoServiceServicer(), server)

SERVICE_NAMES = (reflection.SERVICE_NAME,  ❼
    pb2.DESCRIPTOR.services_by_name["EchoService"].full_name)
reflection.enable_server_reflection(SERVICE_NAMES, server)  ❼

credentials = get_ssl_server_credentials(use_mtls=True)
server.add_secure_port("0.0.0.0:50051", credentials)  ❽

server.start()  ❾
server.wait_for_termination()  ❿
```

❶ Класс сервиса основан на автоматически сгенерированном классе `pb2_grpc.EchoServiceServicer`. Класс сервиса содержит все RPC-методы, принадлежащие одному сервису gRPC. Один gRPC-сервер может обслуживать несколько таких классов.

❷ Унарный RPC-метод имеет доступ к бизнес-логике приложения. В данном случае в ответ возвращается содержимое запроса без изменений. Параметр `request` содержит десериализованный запрос от клиента, а `context` позволяет получать информацию о среде выполнения gRPC и взаимодействовать с ней. Десериализацию выполняет обработчик, который является частью автоматически сгенерированного кода серверной заглушки (см. пример 8.5 выше). gRPC поддерживает как синхронные, так и асинхронные типы взаимодействия, но в этой главе мы сосредоточимся на первом варианте.

❸ Как и ожидалось от службы ECHO, ответ сервера воспроизводит содержимое исходного запроса.

❹ Ответ возвращается клиенту. Обратите внимание, что ответ был сериализован обработчиком серверной заглушки.

❺ Эта строка определяет сервер, но еще не запускает его, так как его конфигурация не завершена. gRPC-серверы обычно используют несколько потоков.

❻ Класс сервиса добавляется на сервер.

❼ Переменная `SERVICE_NAMES` хранит имена сервисов, для которых включена функция рефлексии. Она позволяет клиенту на уровне кода запрашивать у сервера информацию о доступных сервисах, RPC-методах и форматах сообщений. Она включается отдельно для каждого сервиса и предоставляет документацию в формате, удобном для чтения человеком. `reflection.SERVICE_NAME` — это встроенное имя самого сервиса рефлексии gRPC.

❽ Конфигурация сервера завершается определением IPv4-адреса и TCP-порта, на котором сервер будет ожидать подключения. IP-адрес `0.0.0.0` указывает серверу прослушивать все локальные IPv4-адреса, в том числе адрес сервера в сети, созданной Docker, чтобы клиент мог обращаться к серверу через эту сеть.

Чтобы сократить объем кода, в этом примере показана только конфигурация защищенного TLS-соединения. В репозитории с кодом для книги также есть конфигурация незащищенного (незашифрованного) соединения, которая используется для анализа захвата пакетов.

Функция `get_ssl_server_credentials()` показывает настройку взаимной аутентификации TLS (mTLS) на стороне сервера. mTLS активируется параметром `require_client_auth=True`. Помимо шифрования и аутентификации сервера, как в обычном TLS, в mTLS добавляется этап аутентификации клиента.

❾ Сервер запускается в фоновом режиме и ожидает подключений от клиентов. Создание транспорта завершается только после того, как инициировано первое подключение.

❿ Метод `wait_for_termination()` блокирует выполнение основной программы, чтобы сервер продолжал работать. Программа завершается только при явной остановке сервера.

Пример 8.7. Реализация унарного RPC для gRPC-клиента (ваш код)

```
# src/grpc/src/echo/echo/client.py

import sys
import time

import grpc

import echo.proto.echo.v1.echo_pb2 as pb2
import echo.proto.echo.v1.echo_pb2_grpc as pb2_grpc

def unary(stub):  ❶
    request = pb2.DemoUnaryRequest(content="Hello")  ❷
    request_serialized = request.SerializeToString()  ❸
```

```
    print(f"Client request: {request.content}, "  ❹
          f"Serialized message: {request_serialized.hex()}")
    response = stub.DemoUnary(request)  ❺
    print(f"Server response: {response.content}")  ❻

def get_ssl_channel_credentials(use_custom_ca=True, use_mtls=True):
    root_certificates = open("src/ca.crt", "rb").read() if use_custom_ca else None
    private_key = open("src/client.key", "rb").read() if use_mtls else None
    certificate_chain = open("src/client.crt", "rb").read() if use_mtls else None
    return grpc.ssl_channel_credentials(
        root_certificates=root_certificates,
        private_key=private_key,
        certificate_chain=certificate_chain)

credentials = get_ssl_channel_credentials(use_mtls=True)
channel = grpc.secure_channel("server:50051", credentials)  ❼

stub = pb2_grpc.EchoServiceStub(channel)  ❽

try:
    unary(stub)  ❾
except grpc.RpcError as rpc_error:
    print(f"gRPC call error: {rpc_error}")
    raise
finally:
    channel.close()  ❿
```

❶ Вспомогательная функция `unary()` показывает, как gRPC-клиент получает ответ на запрос, отправленный серверу.

❷ Создается сообщение запроса protobuf, где полю `content` присваивается значение `Hello`.

❸ Чтобы вывести сообщение запроса protobuf в понятном для человека виде, мы осуществили сериализацию с помощью метода `SerializeToString()`, который предоставляет protobuf. Этот метод позволяет сериализовать сообщение в объект типа bytes в Python (последовательность байтов). Кроме того, метод `SerializeToString()` применяется к сообщению запроса неявно, так как входит в функциональность клиентской заглушки gRPC.

❹ Выводится содержимое сообщения запроса (строка `Hello`), а также тип и шестнадцатеричное представление сериализованного сообщения.

❺ Метод `DemoUnary()` в клиентской заглушке выполняет RPC-вызов.

❻ Выводится содержимое сообщения ответа (строка `Hello`).

❼ Создание gRPC-клиента начинается с определения канала, который представляет собой абстракцию соединения с gRPC-сервером (`server`) на TCP-порте `50051`. Речь идет об абстракции, поскольку gRPC поддерживает балансировку нагрузки на стороне клиента, и фактически `server` может

разрешаться в несколько IP-адресов. Каждое соединение между клиентом и фактическим сервером обрабатывается субканалом *(subchannel)*, но эта функциональность в книге не рассматривается.

В этом примере показана только конфигурация канала, защищенного с помощью TLS, но в репозитории кода книги есть пример и с незащищенным (незашифрованным) соединением.

❽ Следующий шаг — связать клиентскую унарную заглушку с определенным каналом. Для повышения эффективности один канал может обслуживать несколько клиентских заглушек.

❾ В этой строке используется вспомогательная функция `unary()` для выполнения RPC-вызова.

❿ Канал закрывается перед выходом из основной программы. Однако более реалистичный сценарий такой: канал бы не закрылся сразу, а использовался повторно в течение всего жизненного цикла клиента, чтобы избежать затрат на его повторное создание и выполнять множество вызовов через разные заглушки.

Четыре типа RPC

Основываясь на коде сервера и клиента для унарного RPC, в этом разделе мы обсудим другие типы RPC, используемые в gRPC: серверный потоковый, клиентский потоковый и двунаправленный потоковый.

Унарный RPC

Вызов содержит один клиентский запрос, за которым следует ответ сервера.

Серверный потоковый RPC

Вызов содержит один клиентский запрос, за которым следует один или несколько ответов сервера[1]. Типичный вариант использования этого типа вызова — отправка уведомлений об обновлениях. Клиент отправляет запрос серверу, чтобы начать отслеживать изменения ресурса (например, конфигурации), и сервер отвечает потоком обновлений[2]. Чтобы лучше понять этот тип RPC, представим процесс загрузки/выгрузки файла. Используя

---

[1] С потоковыми RPC больше трудностей. В докладе Using gRPC for Long-Lived and Streaming RPCs («Использование gRPC для долгоживущих и потоковых RPC-вызовов») (https://oreil.ly/mHwxe) Эрик Андерсон (Eric Anderson) дает однозначную рекомендацию: «Большинство RPC должны быть унарными». Напомним, что потоковые RPC, как и унарные, могут быть реализованы и синхронно, и асинхронно.

[2] В HTTP/2 сервер использует флаг End Stream, чтобы указать, что больше данные отправляться не будут. Подробнее об использовании HTTP/2 в gRPC см. в разделе «Формат передачи данных protobuf» далее в этой главе.

серверную потоковую передачу, можно реализовать загрузку файла «из коробки», но возобновление загрузки не будет поддерживаться по умолчанию.

Клиентский потоковый RPC

Вызов содержит один или несколько клиентских запросов, за которыми следует один ответ сервера. Пример использования клиентской потоковой передачи — отправка логов на сервер. Можно использовать готовую реализацию клиентской потоковой передачи для загрузки файла по частям, но без поддержки возобновления.

Двунаправленный потоковый RPC

Вызов содержит произвольное количество клиентских запросов и ответов сервера. Этот тип подходит для таких сценариев, как приложения для онлайн-общения или загрузка/выгрузка файлов с поддержкой возобновления.

В примере 8.8 показан файл `.proto`, где определяется служба ECHO, реализующая четыре типа RPC. Обратите внимание, что потоковая часть RPC отмечена ключевым словом `stream`. Этот файл — расширенная версия примера 8.1, где был определен унарный RPC. Соответственно, файлы protobuf допускают расширение.

Пример 8.8. Четыре типа RPC, поддерживаемые gRPC

```
// src/grpc/src/echo/echo/proto/echo/v1/echo.proto

syntax = "proto3";

package echo.v1;

message DemoUnaryRequest {
  string content = 1;
}

message DemoUnaryResponse {
  string content = 1;
}

message DemoServerStreamingRequest {
  string content = 1;
}

message DemoServerStreamingResponse {
  string content = 1;
}

// Аналогичные сообщения Request и Response для оставшихся типов RPC
// пропущены для удобства чтения...

// Сервис возвращает клиенту полученное содержимое.
service EchoService {
```

```
  // Этот метод демонстрирует унарный RPC.
  rpc DemoUnary(DemoUnaryRequest) returns (DemoUnaryResponse);

  // Этот метод демонстрирует серверный потоковый RPC.
  rpc DemoServerStreaming(DemoServerStreamingRequest)
      returns (stream DemoServerStreamingResponse);

  // Этот метод демонстрирует клиентский потоковый RPC.
  rpc DemoClientStreaming(stream DemoClientStreamingRequest)
      returns (DemoClientStreamingResponse);

  // Этот метод демонстрирует двунаправленный потоковый RPC.
  rpc DemoBidirectionalStreaming(stream DemoBidirectionalStreamingRequest)
      returns (stream DemoBidirectionalStreamingResponse);
}
```

В примере 8.9 для всех четырех типов RPC показан вывод, который получает клиент в ответ на вызов (-ы) к `EchoService`.



Чтобы поэкспериментировать с кодом, выполните вызов каждого типа по аналогии с унарным вызовом, показанным в примере 8.3. Используйте следующие настройки переменных окружения: RPC=unary, RPC=server_streaming, RPC=client_streaming, RPC=bidirectional_streaming (изменяйте в соответствии с типом вызова).

Пример 8.9. Вывод клиента для четырех типов RPC, поддерживаемых gRPC

```
# RPC=unary
Запрос клиента: Hello
Ответ сервера: Hello

# RPC=server_streaming
Запрос клиента: Hello
Ответ сервера: H
Ответ сервера: e
Ответ сервера: l
Ответ сервера: l
Ответ сервера: o

# RPC=client_streaming
Запрос клиента: H
Запрос клиента: e
Запрос клиента: l
Запрос клиента: l
Запрос клиента: o
Ответ сервера: Hello

# RPC=bidirectional_streaming
Запрос клиента: H
Ответ сервера: H
Запрос клиента: e
```

```
Ответ сервера: e
Запрос клиента: l
Ответ сервера: l
Запрос клиента: l
Ответ сервера: l
Запрос клиента: o
Ответ сервера: o
```

Для удобства сравнения реализации сервера и клиента для четырех типов RPC представлены отдельно: реализации сервера — в примере 8.10, а клиента — в примере 8.11. Все четыре типа RPC реализованы в одной службе ECHO, причем потоковые RPC спроектированы для работы с отдельными символами строки `Hello`.

Пример 8.10. Реализации сервера для четырех типов RPC, поддерживаемых gRPC

```
# src/grpc/src/echo/echo/server.py

class EchoServiceServicer(pb2_grpc.EchoServiceServicer):
    def DemoUnary(self, request, context):
        content = request.content
        return pb2.DemoUnaryResponse(content=content)

    def DemoServerStreaming(self, request, context):
        for content in request.content:
            yield pb2.DemoServerStreamingResponse(content=content)

    def DemoClientStreaming(self, request_iterator, context):
        content = ""
        for request in request_iterator:
            content += request.content
        return pb2.DemoClientStreamingResponse(content=content)

def DemoBidirectionalStreaming(self, request_iterator, context):
    for request in request_iterator:
        content = request.content
        yield pb2.DemoBidirectionalStreamingResponse(content=content)
```

Пример 8.11. Реализации клиента для четырех типов RPC, поддерживаемых gRPC

```
# src/grpc/src/echo/echo/client.py

def unary(stub):
    request = pb2.DemoUnaryRequest(content="Hello")
    print((f"Client request: {request.content}")
    response = stub.DemoUnary(request)
    print(f"Server response: {response.content}")

def server_streaming(stub):
    request = pb2.DemoServerStreamingRequest(content="Hello")
    print(f"Client request: {request.content}")
    responses = stub.DemoServerStreaming(request)
```

```
    for response in responses:
        print(f"Server response: {response.content}")

def client_streaming(stub):
    def requests():
        for content in "Hello":
            yield pb2.DemoClientStreamingRequest(content=content)
            print(f"Client request: {content}")
    response = stub.DemoClientStreaming(requests())
    print(f"Server response: {response.content}")

def bidirectional_streaming(stub):
    def requests():
        for content in "Hello":
            yield pb2.DemoBidirectionalStreamingRequest(content=content)
            print(f"Client request: {content}")
            time.sleep(1)
    responses = stub.DemoBidirectionalStreaming(requests())
    for response in responses:
        print(f"Server response: {response.content}")
```

Уточним некоторые моменты, связанные с реализацией клиента и сервера. Во-первых, для потоков gRPC в Python необходимо использовать итераторы. В Python итератор реализуется с помощью ключевого слова `yield`, которое используется для создания генератора (любой генератор является итератором). Во-вторых, метод `bidirectional_streaming()` использует `time.sleep(1)`, чтобы делать паузы длиной 1 секунда между обработкой запросов в клиентском потоке. Эта пауза нужна, чтобы замедлить отправку запросов клиентом и обеспечить последовательный вывод отдельных символов в клиентском и серверном потоках. Посимвольная синхронизация становится возможной, поскольку двунаправленная потоковая передача работает в полнодуплексном режиме (см. рис. 1.4).

Формат передачи данных protobuf

В этом разделе мы рассмотрим основные особенности gRPC на сетевом уровне, используя образец унарного запроса, который был показан в примере 8.3. Понимать работу gRPC на этом уровне полезно, если вы только планируете внедрить gRPC, проводите сравнительные тесты с другими реализациями RPC[1] или занимаетесь отладкой сбоев соединения[2].

---

[1] Connect RPC (https://connectrpc.com) — это альтернативная реализация, разработчики которой стремятся сделать ее совместимой с gRPC.

[2] Основной способ диагностики проблем в сервисах gRPC (https://oreil.ly/Bj0JW) — увеличение объема логирования.

На рис. 8.2 показан захват пакетов, загруженный в графический интерфейс Wireshark (https://www.wireshark.org). В качестве примера для объяснения мы будем использовать выделенный пакет запроса под номером 10. Кроме того, важные детали работы протокола содержатся в областях, помеченных цифрами от *1* до *4*.

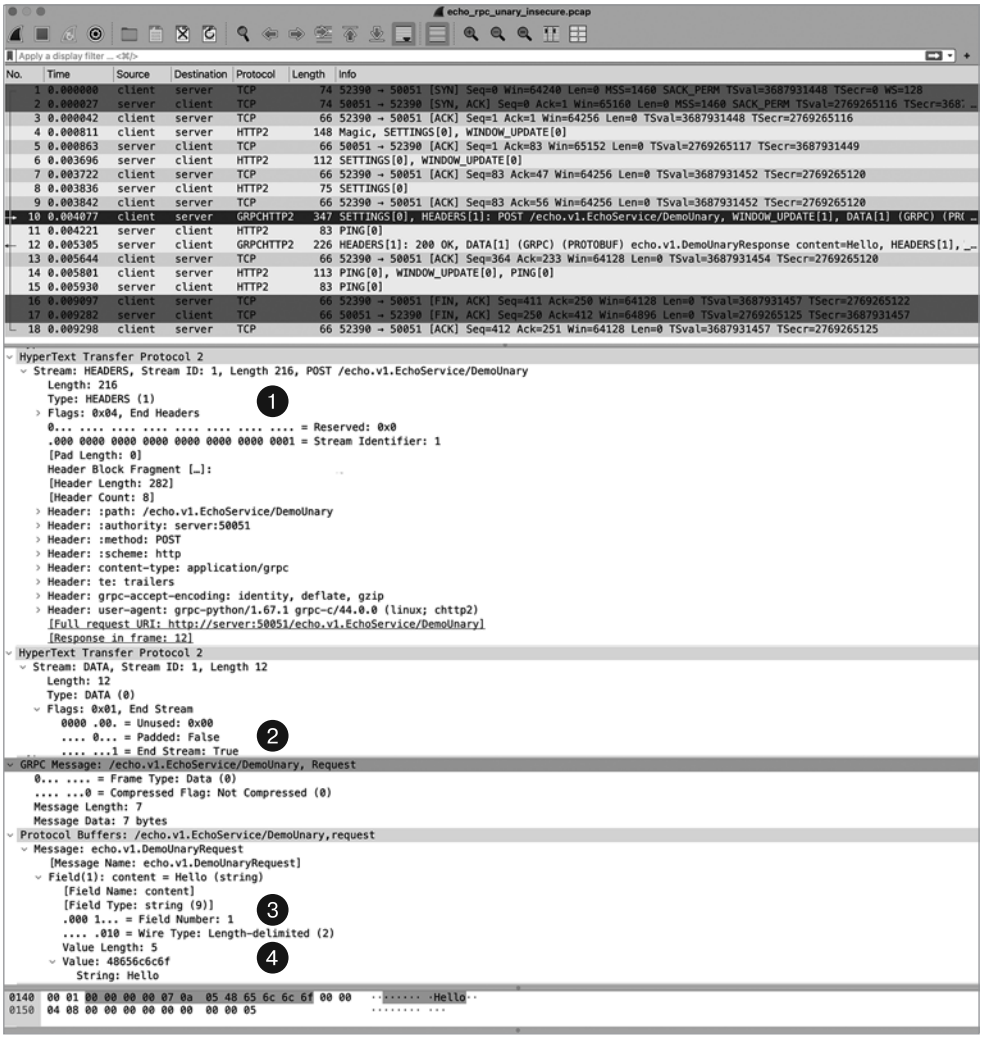


Рис. 8.2. Транспортный механизм gRPC, работающий поверх HTTP/2[1]

[1] Полноразмерный снимок экрана см. в репозитории кода книги, в файле src/grpc/README.md.

1. В качестве транспортного механизма в gRPC используется HTTP/2[1]. Транспортный механизм gRPC, работающий поверх HTTP/2, следует отличать от базового протокола TCP, который сам HTTP/2 использует в качестве транспорта. gRPC-запрос отправляется на сервер в виде HTTP-запроса `POST`. Формат сжатия заголовков HPACK, используемый в HTTP/2, позволяет уменьшить размер gRPC-запросов по сравнению с реализациями на HTTP/1. Кроме того, gRPC заимствует из HTTP/2 механизм управления потоком и несколько других функций. Например, независимые двунаправленные потоки позволяют реализовать потоковую передачу данных в gRPC, а двоичное кодирование фреймов в HTTP/2 хорошо интегрируется с двоичным форматом protobuf.
2. В HTTP/2 клиент использует флаг `End Stream`, чтобы уведомить сервер о том, что данных от него больше не ожидается. Так клиент переходит в *«полузакрытое»* состояние: сервер продолжит отправлять ответ, но будет отклонять последующие данные от клиента. Поскольку gRPC-запрос в примере является унарным, то клиент отправляет флаг `End Stream` сразу. В случае клиентского или двунаправленного потокового взаимодействия клиент отправил бы `End Stream` только в конце своего потока данных. Вдобавок сервер использует флаг `End Stream`, чтобы уведомить клиент о том, что отправка ответа завершена (на рисунке не показано). Кроме того, сервер использует HTTP-*трейлеры* для передачи клиенту информации о статусе gRPC-ответа (тоже не показано на рисунке). Трейлеры похожи на HTTP-заголовки, но отправляются после тела HTTP-сообщения[2].
3. Номер поля и тип кодирования (шестнадцатеричное значение `0a`) вместе с длиной значения в байтах (`05`) и самим значением (`48656c6c6f`) составляют *формат передачи данных* protobuf. Оставшаяся часть раздела посвящена этому формату.
4. Значение `48656c6c6f` — это шестнадцатеричное представление 7-битной ASCII-строки `Hello` (совпадает с ее кодировкой в формате UTF-8). Для строк в protobuf используется кодирование с префиксом длины, где перед строкой, закодированной в UTF-8, указывается ее длина.

---

[1] Введение в HTTP/2 см. в разделе «HTTP/2» главы 4. Предложение о поддержке gRPC поверх HTTP/3 (https://oreil.ly/QNQ8k) завершено, но на момент написания книги реализаций существует немного.

[2] Трейлеры являются частью спецификации Hypertext Transfer Protocol — HTTP/1.1 («Протокол передачи гипертекста — HTTP/1.1») (https://oreil.ly/gWBpi). Отсутствие реализации трейлеров в JavaScript fetch API — одна из причин, по которой gRPC не поддерживается браузерами. См. статью Карла Мастранджело (Carl Mastrangelo) Why Does gRPC Insist on Trailers? («Почему в gRPC обязательно используются трейлеры?») (https://oreil.ly/V4Qlb) и обсуждение этой проблемы на GitHub MDN (https://oreil.ly/y7JFU).

Пакет под номером 10 содержит битовый поток, который в шестнадцатеричном формате имеет вид `0a0548656c6c6f` и является примером формата передачи данных protobuf. Этот формат можно декодировать с помощью компилятора `protoc` (пример 8.12). Вывод в примере подтверждает, что формат передачи соответствует полю с номером `1` и строке `Hello`.



В формате protobuf вместо имен полей используются их номера, что позволяет сократить объем данных, передаваемых по сети с помощью gRPC.

Пример 8.12. Формат передачи данных protobuf, декодированный с помощью xxd и protocol (требуются настройки из примера 8.2)

```
docker compose exec client bash -ci \
       "echo -n 0a0548656c6c6f | \
       xxd -plain -revert | \
       python -m grpc_tools.protoc --decode_raw"
1: "Hello"
```

Чтобы обсудить формат кодирования[1] подробнее, разделим шестнадцатеричное представление строки `Hello` из примера пробелами: `0a 05 48656c6c6f`. Такой тип представления называется «тег — длина — значение» (https://oreil.ly/RhUQD). Тег в protobuf рассчитывается на основе *номера поля* и *типа кодирования* по формуле `(field_number << 3) | wire_type`, где `<<` обозначает битовый сдвиг влево на три позиции. Это значит, что три младших (крайних справа) бита обозначают тип кодирования (табл. 8.1), а остальные биты — номер поля.

Таблица 8.1. Типы кодирования в protobuf

ID	Название	Цель использования
0	VARINT	int32, int64, uint32, uint64, sint32, sint64, bool, enum
1	I64	fixed64, sfixed64, double
2	LEN	string, bytes, вложенные сообщения, упакованные повторяющиеся поля
3	SGROUP	Начало группы (устарело)
4	EGROUP	Конец группы (устарело)
5	I32	fixed32, sfixed32, float

В нашем примере тег имеет длину 1 байт и равен `0a` в шестнадцатеричной системе (`10` в десятичной). В примере 8.13 мы получим это значение по формуле

[1] Дополнительную информацию см. в разделе Encoding («Кодирование») документации protobuf (https://oreil.ly/r5rEL).

тега protobuf, используя Python-цикл «чтение — вычисление — вывод» (read — eval — print loop, REPL). Нам понадобятся следующие значения номера поля и типа кодирования.

1. В нашем protobuf из примера 8.1 полю `content` присвоен номер `1`, а его тип — string.
2. Из табл. 8.1 мы знаем, что типу string соответствует числовой идентификатор типа кодирования `2`.



Чтобы запустить цикл REPL, выполните команду docker compose exec client python, а для завершения — введите exit() и нажмите Enter.

Пример 8.13. Тег формата кодирования protobuf, вычисленный через Python

```
>>> field_number = 1
>>> wire_type = 2
>>> tag = (field_number << 3) | wire_type
>>> print(tag)
10
>>> print(f"{tag:02x}")
0a
```

В табл. 8.1 представлены и другие типы кодирования protobuf, в частности *VARINT*. Как видно из названия, этот тип представляет целые числа переменной длины. По размеру такое представление эффективно для небольших целых чисел. VARINT позволяет кодировать значение сообщения, тег и длину. В нашем примере десятичное число 10, закодированное в формате VARINT, занимает 1 байт вместо стандартных 8 (64 бита) в представлении фиксированной длины. Тег `0a` и длина `05` строки `Hello` из нашего примера тоже закодированы в формате VARINT и имеют длину 1 байт.

Реализация

Основной сценарий использования gRPC — взаимодействие между сервисами. В этом разделе мы реализуем такой сценарий и создадим сервис обогащения информацией для Atom-ленты WFS. Мы расширим Atom-ленту из главы 7, добавив сводку о текущих погодных условиях в городе и рекомендации для туристов. Эти данные будет генерировать LLM. Кроме того, мы разберем следующие дополнительные возможности gRPC:

- использование других типов сообщений в качестве типов полей protobuf (https://oreil.ly/cE261);

- серверное кэширование ответов с помощью перехватчиков (https://oreil.ly/ozsUP);
- использование кодов состояния для обозначения ошибок (https://oreil.ly/liYsd);
- установление дедлайнов на стороне клиента (https://oreil.ly/Qg2wG).

Информацию о других возможностях можно найти в официальных руководствах по gRPC (https://oreil.ly/tsE2b) и в примерах кода из репозитория gRPC (https://oreil.ly/i7D3V).

Сервис обогащения Atom-ленты gRPC

Если вы начали чтение главы с этого места и параллельно выполняете практические задания, то убедитесь, что вы инициализировали лабораторную среду, как было показано в примере 8.2. Кроме того, ознакомьтесь с использованием `tmux` в примере 8.3 и создайте окно с двумя панелями. Далее запустите gRPC-сервер обогащения, следуя инструкциям в примере 8.14. Если на одной из панелей все еще работает ECHO-сервер, то его можно остановить, нажав Ctrl+C.

Пример 8.14. Запуск gRPC-сервера обогащения (требуются настройки из примера 8.2)

```
docker compose exec server bash -ci \
       "PYTHONPATH=./src/enricher python src/enricher/enricher/server.py"
```

После запуска сервера рекомендуется выполнить подготовительный запрос для прогрева на любой доступной панели (пример 8.15). Для этого запроса можно использовать любой город.

Пример 8.15. Прогрев LLM (требуются настройки из примера 8.14)

```
docker compose exec client grpcurl -plaintext \
       -d '{"weather_forecast": {"city": {"name": "A city"}}}' \
       server:50051 enricher.v1.EnricherService/Enrich
```

Рекомендуем выполнить этот запрос прямо сейчас, поскольку первый вызов gRPC-клиента (например, grpcurl (https://oreil.ly/fdBUa)) к gRPC-серверу обогащения может занять несколько минут. Время ожидания зависит от скорости интернет-соединения компьютера, на котором запущена лабораторная среда. При первом использовании LLM потребуется загрузить веса. Кроме того, модели понадобится время для генерации ответа на промпт. Выполнение этого запроса также позволит проверить качество ответа LLM для выбранного города.

Теперь, когда сервер запущен и LLM загружена, перейдем к обсуждению реализации сервера. Чтобы увидеть реализуемую функциональность gRPC, при взаимодействии с сервером мы будем использовать grpcurl в качестве клиента.

В примере 8.16 показан файл protobuf сервиса `EnricherService`. Унарный RPC-метод `Enrich()` принимает пользовательский тип сообщения `WeatherForecast` в качестве запроса и возвращает строку `content`, которая содержит сводку прогноза погоды и рекомендации для туристов, сгенерированные LLM.

Пример 8.16. Protobuf-определение gRPC-сервиса Enricher

```
// src/grpc/src/enricher/enricher/proto/enricher/v1/enricher.proto

syntax = "proto3";
package enricher.v1;

import "google/protobuf/timestamp.proto";  ❶

message City {  ❷
  // UUID города.
  string uuid = 1;
  // Название города.
  string name = 2;
  // Название страны, где находится город.
  string country = 3;
  // Название континента, где находится страна.
  string region = 4;
}

message WeatherForecast {
  // Город, для которого составляется прогноз погоды.
  City city = 1;  ❸
  // Количество секунд UTC-времени, прошедших с эпохи Unix 1970-01-01T00:00:00Z.
  google.protobuf.Timestamp timestamp = 2;  ❹
  // Прогнозируемое количество осадков за сегодня (дождь).
  double rain_sum_mm = 3;
  // Прогнозируемое количество осадков за сегодня (ливни).
  double showers_sum_mm = 4;
  // Прогнозируемое количество осадков за сегодня (снег).
  double snowfall_sum_cm = 5;
  // Максимальная прогнозируемая температура на сегодня.
  double temperature_max_celsius = 6;
  // Минимальная прогнозируемая температура на сегодня.
  double temperature_min_celsius = 7;
}

message EnrichRequest {
  // Детали прогноза погоды для города.
  WeatherForecast weather_forecast = 1;
}

message EnrichResponse {
  // Содержимое обогащенного прогноза погоды в текстовом формате.
  string content = 1;
}
```

```
// Сервис обогащает информацию из запроса.
service EnricherService {
  // Этот метод возвращает обогащенное содержимое запроса.
  rpc Enrich(EnrichRequest) returns (EnrichResponse);
}
```

❶ Эта строка содержит функцию импорта (`import`) protobuf-определений. Определения импортируются из файла `google/protobuf/timestamp.proto`. Файл находится в официальном репозитории protobuf (https://oreil.ly/7LR1m), но в Python поставляется как часть пакета `grpcio-tools` (https://oreil.ly/mF4DS). Функциональность импорта позволяет строить новые protobuf-определения на основе имеющихся и использовать их повторно.

❷ Этот блок определяет сообщение `City`.

❸ Пользовательское сообщение `City`, определенное выше, используется в качестве поля в сообщении `WeatherForecast`.

❹ Импортированное сообщение `google.protobuf.Timestamp` используется в качестве поля сообщения. Само сообщение `Timestamp` состоит из двух полей: `message Timestamp {int64 seconds = 1; int32 nanos = 2;}`, где его разрешение показано в наносекундах. Обратите внимание: у полей protobuf есть значения по умолчанию (для `int64` и `int32` это значение равно `0`). В нашем приложении поле `nanos` остается со значением по умолчанию, поэтому в поле `timestamp` мы увидим количество секунд UTC-времени, прошедших с эпохи Unix 1970-01-01T00:00:00Z.

В отличие от gRPC-сервера для `EchoService`, описанного в примере 8.6, в реализации gRPC-сервера для `EnricherService` (пример 8.17) добавляется перехватчик `CachingInterceptor`. Перехватчик gRPC схож с промежуточным ПО, которое мы обсудим в примере 10.7. Как видно из названия, он перехватывает каждый RPC-вызов, чтобы выполнить определенную функциональность, а затем передает вызов дальше по цепочке перехватчиков. Они могут быть реализованы на стороне клиента (на канале) и на стороне сервера. Чаще всего перехватчики используются для кэширования, логирования или аутентификации.

Пример 8.17. Реализация gRPC-сервера и перехватчика

```
# src/grpc/src/enricher/enricher/server.py

class CachingInterceptor(grpc.ServerInterceptor):
    def __init__(self):
        self.cache = {}  ❶

    def intercept_service(self, continuation, handler_call_details):  ❷
        rpc_method = handler_call_details.method
        logger.info("Перехват", extra={"rpc_method": rpc_method})
```

```
        if rpc_method.startswith("/grpc.reflection.v1"):  ❸
            logger.info("Пропуск перехватчика")
            return continuation(handler_call_details)

        next_handler = continuation(handler_call_details)  ❹
        def return_cached_response(request, context):
            cache_key = request.SerializeToString()  ❺
            if cache_key not in self.cache:  ❻
                response = next_handler.unary_unary(request, context)
                self.cache[cache_key] = response
            return self.cache[cache_key]  ❼

        return grpc.unary_unary_rpc_method_handler(  ❽
            return_cached_response,
            request_deserializer=next_handler.request_deserializer,
            response_serializer=next_handler.response_serializer)

server = grpc.server(
    concurrent.futures.ThreadPoolExecutor(),
    interceptors=(CachingInterceptor(),),)
pb2_grpc.add_EnricherServiceServicer_to_server(
    EnricherServiceServicer(), server)

SERVICE_NAMES = (reflection.SERVICE_NAME,
    pb2.DESCRIPTOR.services_by_name["EnricherService"].full_name)
reflection.enable_server_reflection(SERVICE_NAMES, server)

server.add_insecure_port("0.0.0.0:50051")
server.start()
logger.info("gRPC server is ready to serve requests")
server.wait_for_termination()
```

❶ Для кэширования ответов сервера в качестве атрибута экземпляра класса (`self.cache`) используется словарь Python. Он инициализируется как пустой кэш, хранящийся в памяти. В более реалистичном сценарии кэш имел бы ограничение по размеру и функцию TTL (time-to-live — «время жизни») для удаления устаревших записей.

❷ Метод `intercept_service()` должен быть реализован в любом серверном перехватчике.

❸ Это условие используется, чтобы пропустить перехватчик, если RPC-запрос клиента предназначен для рефлексии. Рефлексия — это дополнительная функция gRPC-сервера, которая включается для сервиса, чтобы предоставить клиентам protobuf-определения сервисов и сообщений как часть gRPC API.

❹ Функция `continuation()` возвращает значение `next_handler` (следующий обработчик в цепочке). Это может быть другой перехватчик или фактический обработчик RPC-метода. У нас есть только один перехватчик, поэтому `next_handler` относится к фактическому обработчику RPC-метода.

❺ Метод protobuf `SerializeToString()` сериализует запрос в Python-тип `bytes` (не в строку, как можно предположить из названия). Результат используется в качестве ключа кэша.

❻ Если ответ еще не был кэширован, то для его получения вызывается соответствующий RPC-метод обработчика. В нашем случае вызывается метод `unary_unary()` объекта `next_handler`. Затем ответ сохраняется в кэше.

❼ Возвращается кэшированный ответ.

❽ Возвращается RPC-обработчик, который использует метод `return_cached_response()` для кэширования ответа.

`CachingInterceptor` можно написать как универсальный, чтобы кэшировать ответы разных унарных RPC-методов и повторно использовать его для разных gRPC-сервисов. Для нашего сервиса из примера 8.18 `next_handler` — это обработчик для `EnricherServiceServicer`, а метод `unary_unary()` — это унарный метод `Enrich()`. Кроме того, в `EnricherServiceServicer` используются еще две возможности gRPC: коды состояния для обозначения ошибок и дедлайны.

Пример 8.18. Реализация EnricherServicer в gRPC

```
# src/grpc/src/enricher/enricher/server.py

class EnricherServiceServicer(pb2_grpc.EnricherServiceServicer):
    def Enrich(self, request, context):
        if not (-90 <= request.weather_forecast.temperature_max_celsius <= 60):
            context.abort(  ❶
                grpc.StatusCode.INVALID_ARGUMENT,
                "Доступный диапазон temperature_max_celsius [-90, 60].")

        if not (-90 <= request.weather_forecast.temperature_min_celsius <= 60):
            rich_status = create_field_validation_error_status(
                field="temperature_min_celsius",
                description="Allowed range is [-90, 60].")
            context.abort_with_status(rpc_status.to_status(rich_status))  ❷

        deadline = context.time_remaining()  ❸
        logger.info("Время до дедлайна", extra={"дедлайн": deadline})

        weather_forecast = get_weather_forecast(request)  ❹
        summary = summarize_weather_forecast(weather_forecast)  ❺
        enriched_content = weather_forecast + (
            f"<br>\n"
            f"<br>\n"
            f"\nLanguage Model says:\n"
            f"<br>\n"
            f"{summary}")
        return pb2.EnrichResponse(content=enriched_content)  ❻
```

❶ Мы запустили проверку диапазона максимальной температуры, чтобы продемонстрировать базовый тип кода состояния ошибки. Для прерывания

RPC gRPC-сервер возвращает код состояния `INVALID_ARGUMENT` и указанное сообщение. Ошибка возникает, когда переменной максимальной температуры передается недопустимое значение.

```
docker compose exec client grpcurl -plaintext \
       -d '{"weather_forecast": {"temperature_max_celsius": 100.0}}' \
       server:50051 enricher.v1.EnricherService/Enrich
ERROR:
  Code: InvalidArgument
  Message: Allowed temperature_max_celsius range is [-90, 60].
```

❷ Аналогичным образом запускается проверка диапазона минимальной температуры, но при прерывании RPC выдается расширенный статус ошибки. В нем дополнительно содержится поле `Details`. Ниже показан только пример вывода расширенного статуса ошибки. Реализация доступна в репозитории кода книги.

```
docker compose exec client grpcurl -plaintext \
       -d '{"weather_forecast": {"temperature_min_celsius": 100.0}}' \
       server:50051 enricher.v1.EnricherService/Enrich
ERROR:
  Code: InvalidArgument
  Message: Field validation error
  Details:
  1) { "@type": "type.googleapis.com/google.rpc.BadRequest",
       "field_violations": [
       { "field": "temperature_min_celsius",
         "description": "Allowed range is [-90, 60]."
       }]
}
```

❸ Из объекта `context` получаем время, оставшееся до дедлайна, который задал клиент. Дедлайн — это момент времени, после которого клиент перестает ожидать ответ от сервера и завершает RPC с кодом состояния `DEADLINE_EXCEEDED`.

В микросервисных архитектурах gRPC-сервер может быть клиентом для другого gRPC-сервера, поэтому gRPC поддерживает распространение исходного дедлайна. Даже если изначальный RPC-запрос по цепочке запускает другие RPC, исходный дедлайн должен соблюдаться (в том числе в случаях повторных попыток обработки запроса). Распространение дедлайна помогает избежать бесполезного удержания ресурсов, например памяти, в ожидании ответов, которые, скорее всего, не поступят. Подробнее о дедлайнах в gRPC можно прочитать в руководстве (https://oreil.ly/kCwd7).

```
docker compose exec client grpcurl -plaintext \
       --max-time 1 server:50051 enricher.v1.EnricherService/Enrich
ERROR:
  Code: DeadlineExceeded
  Message: context deadline exceeded
```

❹ Функция `get_weather_forecast()` позволяет представить прогноз погоды в текстовом формате, чтобы передать его в LLM. Реализацию этой функции мы здесь подробно не рассматриваем. Если кратко, то она объединяет поля из сообщения `WeatherForecast` в `EnrichRequest`, чтобы получить строку вида: «Это прогноз погоды на сегодня для города Токио в Японии. Сезон — начало июня. Без осадков. Без ливней. Без снегопада. Максимальная температура воздуха — 24 градуса Цельсия. Минимальная температура воздуха — 17 градусов Цельсия».

❺ Получив текст прогноза, LLM представляет его в виде краткой сводки и дополняет рекомендациями для туристов, которые собираются посетить город. Реализацию этой функции мы также здесь не обсуждаем, но результат ее работы показан на рис. 8.3.

❻ Обогащенный контент возвращается клиенту.

Прежде чем перейти к реализации обогащения Atom-ленты, посмотрим, как этот механизм работает на практике. Убедитесь, что gRPC-сервер обогащения запущен на одной из панелей (как было показано в примере 8.14). На другой панели настроим и запустим ридер `newsboat`, используя команды из примера 8.19.

Пример 8.19. Чтение обогащенной Atom-ленты с помощью Newsboat (требуются настройки из примера 8.14)

```
# Запуск приложения Django  ❶
cd src/django
docker compose build --build-arg UID=$(id -u) --build-arg GID=$(id -g)
docker compose up --detach --wait

# Настройка ридера newsboat  ❷
docker compose exec app bash -c \
       "mkdir --parents ~/.newsboat && \
       echo 'download-timeout 600' > ~/.newsboat/config && \
       echo http://app:8000/forecast/feed_enriched > ~/.newsboat/urls"

# Чтение ленты  ❸
docker compose exec app bash -c "newsboat --refresh-on-start"
```

❶ Перейдите в каталог приложения Django, соберите Docker-образы и запустите контейнеры.

❷ Настройте фид-ридер `newsboat`. Увеличьте время загрузки ленты до значения, достаточного для генерации вывода LLM, и укажите URL обогащенной ленты.

❸ Запустите ридер `newsboat` в интерактивном режиме.



При первом использовании загружаются веса модели, поэтому первый вызов LLM может занять несколько минут — в зависимости от скорости интернет-соединения компьютера, на котором выполняются команды. Кроме того, от производительности компьютера зависит количество времени, необходимое для генерации ответа локально работающей моделью. На более старом оборудовании генерация может занять до нескольких минут.

Через некоторое время (несколько секунд или минут, в зависимости от скорости работы LLM) нажмите Enter в `newsboat`, чтобы открыть ленту *City Weather Forecast Enriched*. Далее еще раз нажмите Enter, чтобы выбрать ленту для нужного города. Вы увидите прогноз погоды, дополненный сводкой, которую сгенерировала модель (рис. 8.3). Нажмите клавишу Q, чтобы вернуться в меню выбора городов, или Enter — чтобы просмотреть другие города. Для полного выхода из `newsboat` нажмите и удерживайте Q.

```
newsboat 2.21.0 - Article 'Tokyo' (5 unread, 6 total)
Feed: City Weather Forecast Enriched
Title: Tokyo
Date: Fri, 06 Jun 2025 23:43:00 +0000
Link: http://app:8000/forecast/cf945356-8cf0-4baf-9177-bcc9b3e4a561

This is today's weather forecast for Tokyo in Japan in the beginning of June:
No rain.
No showers.
No snowfall.
Maximum temperature 24 degrees Celsius.
Minimum temperature 17 degrees Celsius.

Language Model says:
Yes, this weather is typical for early June in Tokyo, Japan. The absence of
precipitation suggests clear skies with mild temperatures suitable for outdoor
activities. For tourists interested in exploring traditional Japanese culture during
this time, consider visiting Ueno Park or Asakusa to experience the vibrant atmosphere
and historical sites.
q:Quit s:Save n:Next Unread o:Open in Browser e:Enqueue ?:Help                  Top
```

Рис. 8.3. Atom-лента прогноза погоды, обогащенная с помощью LLM

В оставшейся части этого раздела мы рассмотрим интеграцию gRPC-клиента в Atom-ленту, которую обслуживает приложение Django (пример 8.20).

Пример 8.20. Реализация обогащенной Atom-ленты прогноза погоды

```
# src/django/app/core/api/atom/feed_enriched.py

class CityAtomFeedEnrichedView(FeedView):
    feed_type = Atom1Feed
    title = "City Weather Forecast Enriched"
    link = "/forecast/feed_enriched"
```

```
channel = grpc.insecure_channel("grpc-server:50051")  ❶
stub = pb2_grpc.EnricherServiceStub(channel)  ❶

def item_description(self, city):
    forecast = WeatherForecastRepository.filter(city=city).first()  ❷
    city_proto = pb2.City(  ❸
        uuid=str(city.uuid),
        name=city.name,
        country=city.country,
        region=city.region,
    )
    forecast_timestamp = datetime.datetime.combine(  ❹
        forecast.date, datetime.datetime.min.time()
    )
    timestamp_proto = Timestamp()  ❹
    timestamp_proto.FromDatetime(forecast_timestamp)  ❹
    weather_forecast_proto = pb2.WeatherForecast(  ❺
        city=city_proto,
        timestamp=timestamp_proto,
        rain_sum_mm=forecast.rain_sum_mm,
        showers_sum_mm=forecast.showers_sum_mm,
        snowfall_sum_cm=forecast.snowfall_sum_cm,
        temperature_max_celsius=forecast.temperature_max_celsius,
        temperature_min_celsius=forecast.temperature_min_celsius,
    )
    request = pb2.EnrichRequest(weather_forecast=weather_forecast_proto)  ❻
    try:
        response = self.stub.Enrich(request, wait_for_ready=True)  ❼
    except grpc.RpcError as rpc_error:
        logger.exception(f"gRPC call error: {rpc_error}")
    raise

    return response.content  ❽
```

❶ В этом блоке создаются канал и заглушка для gRPC-клиента. Канал gRPC создается для подключения к gRPC-серверу по адресу `grpc-server:50051`. Чтобы ограничить объем кода, обсуждаемого в этом разделе, устанавливается небезопасное (незашифрованное) соединение. Заглушка для `EnricherService` создается на канале. Чтобы снизить потребление ресурсов, рекомендуется повторно использовать каналы и заглушки, поэтому здесь они созданы как атрибуты класса.

❷ Из базы данных извлекается последний прогноз погоды для заданного города.

❸ Создается protobuf-сообщение `City`. Функция `str(city.uuid)` преобразует поле `uuid` в строковый тип, ожидаемый protobuf-сообщением.

❹ Заполняется значение protobuf-поля `timestamp`. Поскольку `forecast.date` содержит только дату, то значение расширяется: в него добавляется временная метка — полночь текущего дня. Далее значение переменной `forecast_timestamp` используется для создания экземпляра protobuf-сообщения встроенного типа `Timestamp`.

❺ На основе сообщений, созданных выше, и других значений примитивных типов из модели базы данных `forecast` создается protobuf-сообщение `WeatherForecast`.

❻ На данном этапе можно создать protobuf-сообщение запроса, используя сообщение `WeatherForecast`. Напомним, что в gRPC RPC-методы принимают только один аргумент.

❼ RPC-вызов выполняется с помощью метода `Enrich()`, являющегося частью клиентской заглушки. gRPC-функция wait-for-ready (https://oreil.ly/b68ML) не дает заглушке отправить запрос, пока сервер не будет доступен. Если клиент использует дедлайн (https://oreil.ly/HUJB0, в нашем примере не используется), это время ожидания также учитывается в дедлайне. В этом блоке перехватывается исключение `grpc.RpcError`. Предположим, что речь идет о неустранимой ошибке, — тогда информация о ней логируется, после чего исключение вызывается повторно (распространяется вверх для обработки на более высоких уровнях). В реальной системе мы могли бы попытаться обработать устранимые ошибки[1].

❽ Возвращается строковое значение поля `content`, десериализованное из RPC-ответа.

Эта реализация RPC дает представление об удобстве локальных вызовов. Если приложить дополнительные усилия при разработке[2], то можно значительно упростить и процесс эксплуатации системы, в частности создание документации API, реализацию шифрования и аутентификации, поддержание отказоустойчивости и прозрачности системы. Однако инструменты для реализации этих функций недоступны в gRPC «из коробки».

---

[1] Введение в обработку исключений в Python см. в статье Мигеля Гринберга (Miguel Grinberg) The Ultimate Guide to Error Handling in Python («Полное руководство по обработке ошибок в Python») (https://oreil.ly/IbTFa).

[2] См. доклад Бенджамина Федорки (Benjamin Fedorka) How Netflix Makes gRPC Easy to Serve, Consume and Operate («Как Netflix упрощает работу с gRPC: обслуживание, потребление и эксплуатация») (https://oreil.ly/NqgA9). Многие улучшения, упомянутые в этом докладе, существуют как проприетарные расширения gRPC, созданные внутри Netflix.

Безопасность

gRPC основан на протоколе HTTP/2, поэтому наследует связанные с ним риски безопасности, о которых мы уже говорили в подразделе «Проект OWASP API Security Top 10» главы 2. Эти риски делятся на три категории: аутентификацию и авторизацию, управление системой и управление ресурсами.

gRPC чаще всего используется при взаимодействии между сервисами. Обычно это происходит в частных сетях, поэтому у разработчиков есть соблазн использовать менее безопасные практики, которые стандартно применяются для периметровой защиты компьютерных сетей.



При обеспечении периметровой безопасности сети контроль трафика выполняется на границе сети с глобальной сетью, а трафик внутри периметра проверяется менее тщательно. Противоположный подход основан на концепции нулевого доверия (zero-trust), где любой трафик по умолчанию считается недоверенным, пока не подтверждено обратное.

Например, при обеспечении периметровой безопасности аутентификация часто ограничивается стандартным шифрованием TLS и аутентификацией сервера в сочетании с сертификатом, имеющим большой срок действия. В результате система становится уязвимой для атак со стороны внутренних клиентов.

Эту проблему частично решает внедрение mTLS (mutual TLS), однако возникает необходимость дополнительно управлять ключами и сертификатами, что усложняет эксплуатацию системы. Способ автоматизации mTLS предлагается в проекте SPIRE (https://oreil.ly/ukU61), однако он ориентирован главным образом на облачные системы. Помимо mTLS, gRPC поддерживает аутентификацию клиентов на основе токенов — это повышает уровень безопасности, но при этом усложняет процесс эксплуатации[1].

Главный риск безопасности связан с функцией рефлексии (https://oreil.ly/Wi1hF), которую можно включить на серверах gRPC. Рефлексия позволяет клиентам получать от сервера информацию об API. Включенная рефлексия представляет угрозу, так как она позволяет разведывать внутреннюю структуру сервера, которая содержит сервисы, RPC-методы, типы сообщений. Раскрытие этой информации может привести к целенаправленным атакам.

---

[1] Подробнее об аутентификации см. в руководстве по аутентификации gRPC (https://oreil.ly/pLFeC).

Документация

На момент написания книги нет официального инструмента gRPC для генерации HTML-документации, аналогичного Swagger UI (https://oreil.ly/70qwo) или GraphiQL (https://oreil.ly/oW3JN). Инструменты, предназначенные для этой цели, появляются и исчезают. В крупных проектах на gRPC обычно используются неофициальные инструменты или поддерживаются собственные генераторы (как в Kubernetes).

Для небольших проектов сам файл protobuf может выступать в качестве статической документации по gRPC. Создание интерактивной документации обеспечивает функция рефлексии (пример 8.21), которая позволяет обнаруживать сервисы, RPC-методы и типы сообщений. По умолчанию рефлексия отключена, но ее можно включить для каждого сервиса отдельно.

Пример 8.21. Использование рефлексии gRPC в качестве интерактивной документации (требуется настройка окружения из примера 8.2)

```
# Перечислить доступные сервисы
docker compose exec client grpcurl -plaintext server:50051 list

# Вывод:
enricher.v1.EnricherService
grpc.reflection.v1alpha.ServerReflection

# Описать сервис
docker compose exec client grpcurl -plaintext \
       server:50051 describe enricher.v1.EnricherService

# Вывод:
enricher.v1.EnricherService is a service:
service EnricherService {
  rpc Enrich ( .enricher.v1.EnrichRequest )
    returns ( .enricher.v1.EnrichResponse );
}

# Предоставить шаблон сообщения со значениями по умолчанию
docker compose exec client grpcurl -plaintext \
       -msg-template server:50051 describe enricher.v1.EnrichRequest

# Вывод:
enricher.v1.EnrichRequest is a message:
message EnrichRequest {
  .enricher.v1.WeatherForecast weather_forecast = 1;
}

Message template:
{ "weather_forecast": {
    "city": { "uuid": "", "name": "", "country": "", "region": "" },
```

```
    "timestamp": "1970-01-01T00:00:00Z",
    "rain_sum_mm": 0, "showers_sum_mm": 0, "snowfall_sum_cm": 0,
    "temperature_max_celsius": 0, "temperature_min_celsius": 0
  }
}
```

gRPC: преимущества и недостатки

gRPC обладает следующими преимуществами.

Производительность

Использование двоичного формата сериализации Protocol Buffers в сочетании со сжатием заголовков HTTP/2 позволяет передавать по сети меньший объем данных по сравнению с API на основе JSON/XML и HTTP/1. gRPC предоставляет единый, согласованный стек реализации для клиента и для сервера, что упрощает внедрение оптимизаций. Основная библиотека gRPC, которая используется при реализации на Python, написана на C++. Это позволяет обойти ограничения производительности, вызванные Global Interpreter Lock (GIL) в Python. По этой же причине gRPC используется в бэкендах систем машинного обучения.

Поддержка распределенных систем

Исторически одной из целей RPC было решение проблем, возникающих из-за ненадежности сетей. В gRPC реализуются все современные концепции, используемые для построения распределенных систем: сроки выполнения запросов, отмена запросов, балансировка нагрузки на стороне клиента, управление потоком, проверки работоспособности сервера, кэширование и повторные попытки обработки запроса. Благодаря этому gRPC становится подходящим решением для систем, которые состоят из множества сервисов, взаимодействующих по сети (как в случае с микросервисной архитектурой).

Контракт API

Сообщения и сервисы Protocol Buffers определяют типизированный контракт API между клиентом и сервером. Соблюдение контракта гарантируется, поскольку основой для реализации бизнес-логики пользовательского приложения является автогенерируемый код, создаваемый компилятором из `.proto`-файлов. Кроме того, файлы protobuf можно расширять с помощью плагинов `protoc`, таких как Protovalidate (https://oreil.ly/Z9nI2), которые добавляют логику валидации полей. Работа с gRPC — это реализация подхода API-first, который требует планирования критических изменений перед их фактической реализацией.

Поддержка потоковой передачи

Помимо унарного RPC, gRPC поддерживает три типа потоковой передачи: *серверную*, *клиентскую* и *двунаправленную*. Последний тип позволяет выполнять передачу в полнодуплексном режиме, где и клиент, и сервер имеют полный контроль над сообщениями, которые передаются по сети. Благодаря этому разработчики могут создавать собственные протоколы прикладного уровня с помощью gRPC.

Поддержка многих языков программирования

Новые функции gRPC в первую очередь отражаются в реализациях для Java (https://oreil.ly/TFXbN) и Go (https://oreil.ly/HvNa2). Затем эти функции появляются в реализации на C++ (https://oreil.ly/EMlr4), которая является базовой библиотекой для поддержки других языков: C++, Python, Ruby, Objective-C или PHP. Есть реализации и для .NET (https://oreil.ly/5Zwme), Node.js (https://oreil.ly/lwyNL) и некоторых других языков. Благодаря protobuf клиенты и серверы, написанные на разных языках, могут взаимодействовать, хотя их реализации имеют функциональные различия.

Однако у gRPC есть и несколько слабых сторон.

Ограниченная поддержка в браузерах

Браузеры не поддерживают gRPC нативно[1]. Ограниченная поддержка возможна, например, через `grpc-web` (https://oreil.ly/gbSmS) или `grpc-Gateway` (https://oreil.ly/Xx4PX) с использованием прокси. Можно также использовать *Connect Protocol* (https://oreil.ly/DBnAB) — модифицированную версию HTTP для RPC, которая не требует HTTP-трейлеров (подробнее об HTTP-трейлерах мы говорили в разделе «Формат передачи данных protobuf» этой главы).

Сложности с генерацией кода

Необходимость генерации кода усложняет процесс эксплуатации системы: код нужно генерировать с помощью правильной версии `protoc` (этот подход рекомендуют разработчики gRPC из Google) или распространять его между клиентами и серверами. В обоих случаях структура каталогов protobuf и определения сообщений и сервисов должны быть организованы так, чтобы проект мог развиваться. Для развертывания проектов на gRPC стандартных пайплайнов CI/CD недостаточно — как правило, требуются дополнительные этапы. Кроме того, при использовании сгенерированного кода может потребоваться соблюдение стандартов оформления кода, которые изначально не предполагались в проекте.

---

[1] См. раздел The State of gRPC in the Browser («Состояние gRPC в браузере») на сайте gRPC (https://oreil.ly/TrM6u).

Ограниченная поддержка инструментов для создания документации

На момент написания книги нет официального инструмента gRPC для генерации HTML-документации, аналогичного Swagger UI (https://oreil.ly/lX7UK) или GraphiQL (https://oreil.ly/eMaz7). Файлы protobuf служат статической документацией по gRPC, а функция рефлексии предоставляет документацию в режиме онлайн. Использование рефлексии в качестве документации было показано в примере 8.21.

Отсутствие официальных инструментов

Проблемы с поддержкой инструментов возникают из-за особенностей Protocol Buffers и самого gRPC. При использовании protobuf в gRPC рекомендуется применять специализированные инструменты для управления зависимостями кода, такие как Bazel (https://bazel.build). Редакторы и IDE часто не имеют полноценной поддержки gRPC. Функции, связанные с эксплуатацией, предоставляются сторонними инструментами, например `buf` (https://oreil.ly/qwsTr) для форматирования, линтинга и обнаружения критических изменений или `ghz` (https://oreil.ly/Br8rL) для нагрузочного тестирования.

Меньшее распространение среди разработчиков

По сравнению с REST API-стиль RPC, который используется в gRPC, менее привычен для разработчиков. Использование двоичного формата Protocol Buffers требует освоения новых навыков при разработке и отладке сервисов на основе gRPC.

Ознакомиться с другими плюсами и минусами gRPC можно в серии постов *gRPC: The Good and the Bad* («gRPC: плюсы и минусы») в блоге Кевина Макдональда (Kevin McDonald) (https://oreil.ly/qjcld).

Когда использовать gRPC

gRPC подходит для реализации взаимодействия между сервисами, например в микросервисных архитектурах. Однако перед внедрением gRPC следует проанализировать его сильные и слабые стороны.

В некоторых ситуациях эффективное взаимодействие между сервисами, которое обеспечивается двоичной передачей сообщений и типизированным API-контрактом, может быть более значимым, чем сложности в эксплуатации, которые вызывает gRPC. Кроме того, если у команды есть опыт работы с микросервисными архитектурами, то она может иметь инструменты для преодоления этих сложностей. Наконец, gRPC эффективен в тех случаях, когда требуется одновременное использование сервисов, написанных на разных языках программирования.

gRPC полезен и для разработки приложений, где есть общение через чат и потоковая передача данных, мобильных приложений и систем, требующих обновления данных в реальном времени. Поскольку gRPC использует протокол HTTP/2 и поддерживает потоковую передачу данных, то время задержки сокращается.

Упражнения

1. Вручную закодируйте десятичное число `150` в формат VARINT согласно спецификации protobuf, чтобы получить `9601` в шестнадцатеричном формате. Для справки используйте пример в документации (https://oreil.ly/v4IYX). Решение находится в файле `src/grpc/README.md` в репозитории кода книги.
2. Измените код реализации сервера из примера 8.6 так, чтобы вместо mTLS (mutual TLS) применялся стандартный TLS. Используйте предоставленную команду `grpcurl`, чтобы подтвердить, что связь по-прежнему защищена (зашифрована) и нет необходимости использовать клиентский сертификат и ключ (`--cert src/client.crt --key src/client.key`), которые требуются для mTLS. Подсказка: вспомните роль аргумента `require_client_auth`.

```
docker compose exec client bash -ci \
       "SSLKEYLOGFILE=tests/SSLKEYLOGFILE.client grpcurl \
       --cacert src/ca.crt \
       -d '{ \"content\": \"Hello\" }' \
       server:50051 echo.v1.EchoService/DemoUnary"
```

3. Использование перехватчиков требует понимания того, как они взаимодействуют с RPC. Проанализируйте следующую реализацию базового кода состояния ошибки в примере 8.18. Изменит ли она поведение сервера? Подсказка: вспомните о кэшировании.

```
if not (-90 <= request.weather_forecast.temperature_max_celsius <= 60):
  context.set_code(grpc.StatusCode.INVALID_ARGUMENT)
  context.set_details("Allowed temperature_max_celsius range is [-90, 60].")
  return pb2.EnrichResponse()
```

Резюме

В этой главе мы рассмотрели gRPC как пример API-стиля RPC. Чтобы увидеть основные возможности gRPC, такие как protobuf, генерация кода и четыре типа RPC (унарный RPC, а также серверная, клиентская и двунаправленная потоковая передача), мы реализовали службу ECHO на gRPC и защитили

клиент и сервер с помощью mTLS, где требуется взаимная аутентификация и шифрование.

Кроме того, мы реализовали некоторые более продвинутые функции gRPC, например кэширование на стороне сервера, коды состояния и клиентские дедлайны. Для этого мы расширили реализацию Atom-ленты WFS, добавив сводку, сгенерированную LLM.

Надеемся, что краткое обсуждение преимуществ и недостатков gRPC, а также ссылки на дополнительные материалы станут хорошей базой для более глубокого изучения этой технологии.

ГЛАВА 9

Вебхуки

Прежде чем говорить о вебхуках, разберемся с понятием обратного вызова (callback). Пример работы механизма обратного вызова в реальной жизни — звонок другу, который занят, но обещает перезвонить. Для API обратный вызов — это интерфейс приложения (например, HTTP-эндпоинт), который используется для приема сообщений (вызовов).

Процесс коммуникации в API с обратными вызовами выглядит так: целевой сервис, который хочет получать уведомления, регистрирует свой обратный вызов в сервисе-источнике, ответственном за генерацию событий. В API с обратными вызовами каждое сообщение, отправленное сервисом-источником, должно быть подтверждено ответом целевого сервиса.

На рис. 9.1 показана архитектура API с обратными вызовами.

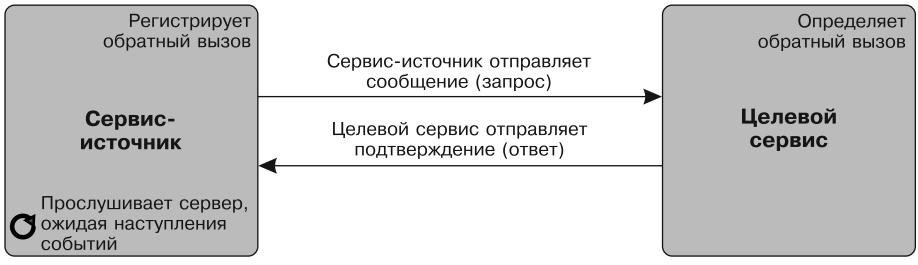


Рис. 9.1. Архитектура API с обратными вызовами

Вебхук — это HTTP-запрос, который инициируется событием в сервисе-источнике и отправляется на URL обратного вызова (callback URL), определенный в целевом сервисе. Данные в теле вебхука содержат информацию о событии. Целевой сервис обрабатывает эти данные и реагирует на произошедшее событие.

В этой главе мы обсудим вебхуки, их происхождение, классификацию, а также преимущества и недостатки. Изучив эту главу, вы поймете, подходят ли вебхуки для вашего API. Помимо знания теоретических концепций, вы получите практические навыки реализации, защиты и документирования API с вебхуками.

Что такое вебхуки

Вебхуки известны под разными названиями, например *HTTP push API*, *HTTP-коллбэки* или *обратные API*. Цель вебхука — отправить сообщение из сервиса-источника в целевой сервис без явного запроса с его стороны. Целевой сервис предоставляет эндпоинт (обратный вызов), который действует как открытая дверь, «впускающая» входящие сообщения. В наиболее распространенной реализации вебхуков используется HTTP-метод `POST`. Возможны вебхуки и с HTTP-методом `GET`, хотя такие встречаются реже.

После регистрации на событие, известное сервису-источнику (после подписки на источник), целевые сервисы получают уведомления о событиях через вебхуки. Регистрацию можно настроить вручную через конфигурацию (вне HTTP-потока) или автоматически с помощью HTTP-запроса `POST`, который отправлен в сервис-источник и содержит URL обратного вызова целевого сервиса.

Сообщения вебхуков сериализуются (обычно в текстовом формате — JSON или XML) и отправляются после возникновения события в сервисе-источнике. Целевой сервис получает сообщение и подтверждает его, выдавая HTTP-ответ с кодом состояния «Успешно». Обратите внимание: целевой сервис не запрашивает сообщения у источника (за исключением возможного первого запроса на регистрацию). Вместо этого сообщение передается сервисом-источником по принципу пуш-уведомления.



Сервис-источник и целевой сервис вебхука, между которыми происходит взаимодействие, — это компьютеры (серверы). Прямое участие человека в этом взаимодействии не предполагается (за исключением настройки). Если исходить из упрощенного понимания того, как устроены веб-приложения, то серверы часто отождествляются с компьютерами, поэтому коммуникацию через вебхуки иногда называют межсерверной.

Однако это представление не совсем точное. Согласно определению архитектуры «клиент — сервер» (см. раздел «Socket API» главы 3), сервер — это сторона, ожидающая соединения. Чтобы избежать неточностей, мы будем использовать более общие термины — «сервис-источник» и «целевой сервис».

Происхождение вебхуков

Считается, что термин *web hook* ввел Джефф Линдсей (Jeff Lindsay) в статье *Web Hooks to Revolutionize the Web* («Вебхуки, которые произведут революцию в вебе») в своем блоге в 2007 году (https://oreil.ly/VhF8d). Идея вебхуков возникла из концепции сайтов как источников данных, которые можно повторно использовать и комбинировать (подобно тому как программы объединяются с помощью конвейеров в Unix)[1]. В Unix конвейер позволяет связывать последовательность программ через их стандартные потоки, где вывод одной программы (программы А) становится вводом для другой (программы Б). Джефф Линдсей представил передачу веб-данных между сервисами аналогично тому, как конвейеры Unix передают данные, используя потоки `stdout` и `stdin`. В вебхуках HTTP-запрос `POST` может действовать как конвейер, который отправляет данные из одной системы в другую при наступлении определенного события. Таким образом, коммуникация между системами становится событийно-ориентированной (event-driven).

Вебхуки стали альтернативой опросу (https://oreil.ly/op4yf), при котором клиент периодически отправляет запросы на сервер, чтобы проверить наличие новых данных. Сервер отвечает, предоставляя новые данные, если они есть, и клиент получает ответ от сервера. Далее цикл повторяется.

Входящие и исходящие вебхуки

Вебхуки делятся на входящие и исходящие. *Входящие вебхуки* позволяют целевому сервису получать сообщения от сервиса-источника, а *исходящие* — отправлять сообщения целевому сервису. Чтобы определить, какой из вебхуков используется, нужно выбрать один из сервисов в качестве точки отсчета.

На рис. 9.2 поток сообщений показан с точки зрения сервиса Б *(в центре)*. Сервис Б интегрируется с сервисом А через входящий вебхук (Б получает сообщение от А), а с сервисом В — через исходящий (Б отправляет сообщение в В).

Чтобы отличить входящий вебхук от исходящего, посмотрите на направление потока сообщений относительно сервиса. Если сообщение поступает в сервис — это входящий вебхук, если покидает сервис — исходящий.

---

[1] В статье Annotate the Web, Then Rewire It («Аннотируйте веб, затем перестраивайте его заново») (https://oreil.ly/1NEnd) Джон Уделл (Jon Udell) высказывал идею о том, что сайты — это источники данных, которые можно использовать повторно.



Рис. 9.2. Входящие и исходящие вебхуки

Интеграция и поток данных

Вернемся к схеме на рис. 9.2 и применим концепцию вебхука к вымышленному интернет-магазину books.example.com, который интегрирован с вымышленным поставщиком книг warehouse.example.com. Соответственно, books.example.com продает книги онлайн, а распределительный центр warehouse.example.com является его стратегическим партнером. На рис. 9.3 показана интеграция между этими двумя сервисами через вебхуки и поток сообщений.

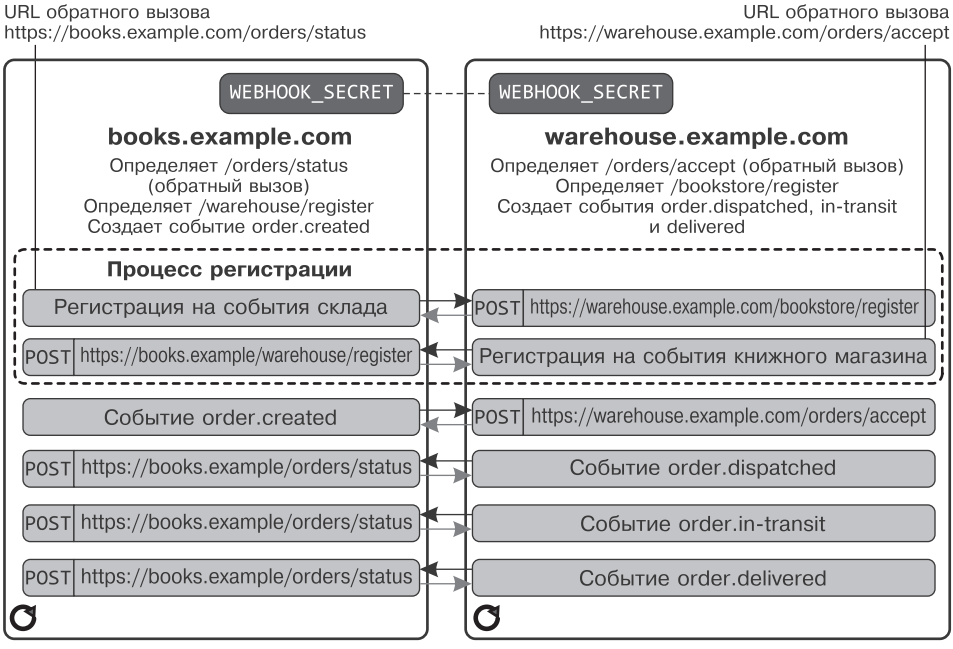


Рис. 9.3. Интеграция между двумя сервисами через вебхуки

Как видите, каждый сервис определяет свой URL обратного вызова, который будет использоваться для получения вебхуков. Сервис `books.example.com` определяет

обратный вызов `/orders/status`, а сервис `warehouse.example.com` — `/orders/accept`. Интеграция начинается с того, что оба сервиса регистрируются друг у друга, отправляя URL обратного вызова на соответствующий эндпоинт регистрации. Сервис `books.example.com` регистрируется в сервисе `warehouse.example.com`, чтобы получать вебхуки о статусе заказа со склада, а сервис `warehouse.example.com` регистрируется в сервисе `books.example.com`, чтобы получать уведомления о вновь созданных заказах в книжном магазине. Когда в сервисе books.example.com происходит событие `order.created`, оно отправляет вебхук (`POST`-запрос) в сервис `warehouse.example.com`, который принимает заказ и начинает его обработку. Пока заказ обрабатывается на складе, генерируются события `order.dispatched`, `order.in-transit` и `order.delivered`, которые отправляются в виде вебхука в сервис `books.example.com`. Кроме того, в обоих сервисах есть секрет `WEBHOOK_SECRET`, который используется для подписи и проверки полезной нагрузки сообщения. Секрет гарантирует подлинность и целостность сообщения (подробнее об использовании секретов см. в разделе «Безопасность» далее в этой главе).

Вне зависимости от того, с каким сервисом вы будете работать, процедура интеграции вебхуков состоит из следующих этапов.

1. Целевой сервис:
   а) реализует URL обратного вызова (HTTP-эндпоинт), через который он будет получать вебхук (запрос) от сервиса-источника;
   б) содержит общий секрет (симметричный ключ), который также известен сервису-источнику[1];
   в) регистрирует свой URL обратного вызова в сервисе-источнике (выполняет операцию подписки).
2. Сервис-источник:
   а) интегрирует URL обратного вызова целевого сервиса;
   б) содержит общий секрет, который также известен целевому сервису. Этот общий секрет используется для подписи полезной нагрузки запроса;
   в) инициирует вебхук (HTTP-запрос) к URL обратного вызова целевого сервиса.
3. Целевой сервис:
   а) проверяет полученный вебхук, анализируя его полезную нагрузку с использованием общего секрета;
   б) обрабатывает вебхук;
   в) возвращает подтверждение (HTTP-ответ) сервису-источнику.

---

[1] В этом примере мы рассматриваем только случай симметричного шифрования с использованием HMAC. Напомним, что существуют также цифровые подписи, основанные на асимметричной криптографии.

Реализация

В этом разделе мы интегрируем наш WFS с внешним сервисом с помощью вебхука. Для этого необходимы два сервиса. Первый будет выступать в роли источника, который генерирует события, а второй — в роли потребителя, который получает вебхук.

На рис. 9.4 показана общая схема сервиса WFS, использующего вебхуки для интеграции с внешним сервисом. Слева находится сервис-источник — это CLI-программа WFS `app_event`, которая создает событие `dummy.event`. Создание события инициирует HTTP-запрос `POST` к URL обратного вызова целевого сервиса. Справа находится целевой сервис (поставщик API), который реализует эндпоинты URL обратного вызова (`/webhooks/v1/echo` и `/webhooks/v2/echo`).

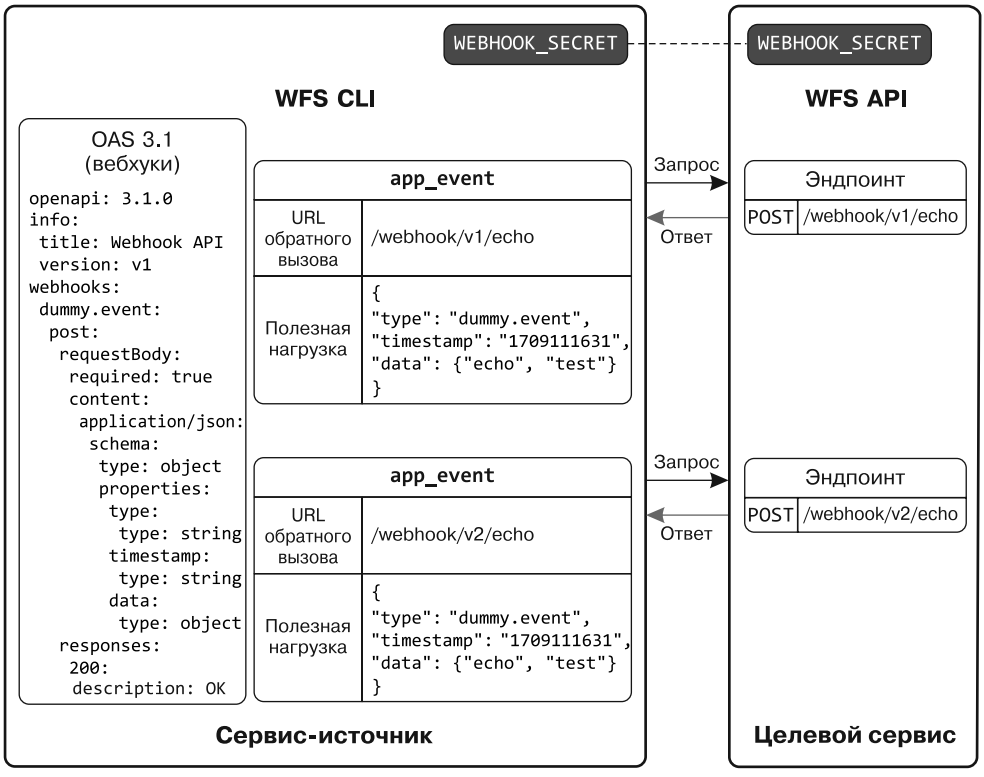


Рис. 9.4. Вебхуки WFS

В документе OpenAPI (показан на рисунке *слева*) описаны запрос и ответ вебхука. Чтобы интеграция была успешной, структуру запроса и ответа должны соблюдать оба сервиса. Документ гарантирует, что сервис-источник и целевой

сервис, участвующие в обмене данными, понимают структуру и поведение вебхука. В документе описывается возможное поведение запроса вебхука, инициированного сервисом-источником, и ожидания от него, а также предполагаемые ответы целевого сервиса.

Обе реализации вебхуков (v1 и v2) соответствуют спецификации, предложенной проектом Standard-Webhooks (https://www.standardwebhooks.com). В обоих случаях эндпоинт обратного вызова будет выполнять одну и ту же задачу — получать вебхук от сервиса-источника и возвращать ответ-подтверждение. Цель двух примеров, приведенных ниже, — отработать реализацию спецификации Standard-Webhooks, используя ее библиотеку, а также реализовать вебхук вручную, чтобы понять концепции, лежащие в его основе.

> **СПЕЦИФИКАЦИЯ STANDARD-WEBHOOKS**
>
> Standard-Webhooks (https://oreil.ly/137po) — это проект, цель которого — стандартизация реализации вебхуков в ИТ-индустрии путем определения спецификации вебхуков (https://oreil.ly/pO9P6). В рамках проекта предлагается набор руководств, которым необходимо следовать при реализации вебхуков. Согласно этим руководствам, при разработке следует учитывать такие факторы, как безопасность, надежность, совместимость, простота, а также обратная и прямая совместимость. Кроме того, в проекте предоставляются эталонные реализации руководств для разных языков программирования, таких как C#, Elixir, Go, Java/Kotlin, JavaScript/TypeScript, PHP, Python, Ruby и Rust.

В примере 9.1 определены URL обратного вызова (эндпоинты вебхуков).

Пример 9.1. Определение эндпоинтов вебхуков (URL обратного вызова)

```
# src/django/app/config/urls.py

from core.api.webhook.v1.views import WebhookStandardView
from core.api.webhook.v2.views import WebhookCustomView

...
urlpatterns = [
    ...
    path("webhook/v1/echo", WebhookStandardView.as_view()),  ❶
    path("webhook/v2/echo", WebhookCustomView.as_view()),  ❷
    ...
]
```

❶ Класс `WebhookStandardView` упрощает реализацию вебхуков с помощью пакета `standardwebhooks` для Python (https://oreil.ly/ksYiD). Этот пакет соответствует спецификации Standard-Webhooks и содержит встроенную функциональность HMAC (https://oreil.ly/mxw5n).

❷ Работа с классом `WebhookCustomView` более сложная, поскольку спецификацию Standard-Webhooks приходится реализовывать вручную.

Определив эндпоинты, реализуем логику обработки входящих HTTP-запросов. Реализация `WebhookStandardView` показана в примере 9.2.

Пример 9.2. Реализация вебхука ECHO с использованием пакета standardwebhooks для Python

```
# src/django/app/core/api/webhook/v1/views.py

from standardwebhooks import Webhook
from django.utils.decorators import method_decorator
from django.views import View
from django.views.decorators.csrf import csrf_exempt

from config.constants import WEBHOOK_SECRET_B64


@method_decorator(csrf_exempt, name="dispatch")  ❶
class WebhookStandardView(View):
    wh = Webhook(WEBHOOK_SECRET_B64)  ❷

    def post(self, request):  ❸
        try:
            payload = self.wh.verify(request.body, request.headers)  ❹
        except Exception as e:
            return JsonResponse({"errors": [str(e)]}, status=400)

        return JsonResponse(payload)  ❺
```

❶ Класс `WebhookStandardView` — это представление на основе классов (https://oreil.ly/QjxI3). По умолчанию фреймворк Django проверяет входные данные, содержащиеся в таких представлениях, поскольку они обрабатываются как HTML-формы. Django проверяет данные во всех входящих HTTP-запросах `POST` на наличие рисков CSRF-атаки. Если CSRF-токен отсутствует в запросе, Django отклоняет его. Чтобы отключить проверку CSRF-токена при получении HTTP-запросов, добавим декоратор `csrf_exempt`. Это действие нежелательно, поскольку снижает безопасность системы, однако оно необходимо при интеграции со сторонними сервисами, которые не находятся под вашим контролем.

❷ Из пакета `standardwebhooks` инициализируется класс `Webhook` с секретом в формате Base64 (`WEBHOOK_SECRET_B64`). Этот секрет используется для проверки полезной нагрузки сообщения. Экземпляр вебхука сохраняется в переменную `wh`, которая является атрибутом класса.

❸ Метод `post()` содержит логику, которая выполняется каждый раз, когда целевой сервис принимает HTTP-запрос `POST`.

❹ Метод вебхука `verify()` проверяет полезную нагрузку сообщения. Если сообщение не проходит проверку, то возникает исключение и сервису-источнику возвращается JSON-ответ с сообщением об ошибке и кодом состояния `400` (`Bad Request`) (Неправильный запрос).

❺ Если сообщение успешно проходит проверку, то сервису-источнику возвращается подтверждение в виде HTTP-ответа с полезной нагрузкой JSON и кодом состояния `200` (`OK`).

Безопасность

Для защиты вебхуков обычно применяют следующие меры безопасности: TLS (https://oreil.ly/u9rEh) для шифрования, обеспечения целостности и подлинности данных при передаче, а также метод HMAC (https://oreil.ly/uMOPR) для дополнительной проверки целостности и подлинности сообщений. В этом разделе показано, как использовать HMAC, с помощью которого обеспечивается безопасность в спецификации Standard-Webhooks.



HMAC — универсальный метод, который применяется не только в вебхуках. В отличие от стандартной процедуры аутентификации в API, которая основана на HTTP и реализуется с помощью токенов или API-ключей, вебхуки часто используют HMAC. Этот метод нужен, чтобы HTTP-сервер мог убедиться, что он получает POST-запросы от правильного источника. HMAC обеспечивает целостность и подлинность сообщений, но не шифрование, поскольку предполагается, что связь уже зашифрована с помощью TLS.

Целостность и подлинность в HMAC обеспечивается с помощью криптографической хеш-функции и общего секретного ключа для подписи и проверки полезной нагрузки сообщения. Общий секретный ключ известен как сервису-источнику, так и целевому сервису. На рис. 9.3 продемонстрировано использование `WEBHOOK_SECRET` обоими сервисами.

В примере 9.3 показано, как реализовать верификацию вебхука по схеме подписи Standard-Webhooks, в которую входит вычисление HMAC.

Пример 9.3. Реализация вебхука ECHO с верификацией запроса вручную

```
# src/django/app/core/api/webhook/v2/views.py

...
class WebhookCustomView(View):
    def post(self, request):
        """
        Метод возвращает тело запроса
        """
        if len(request.body) > 20480:  ❶
            return JsonResponse({"errors": ["Размер полезной нагрузки
              слишком большой."]}, status=413)

        wh_id = request.headers.get("Webhook-Id")  ❶
        if not wh_id:
            err = "Отсутствует заголовок Webhook-Id."
            return JsonResponse({"errors": [err]}, status=400)

        wh_received_signature = request.headers.get("Webhook-Signature")  ❶
        if not wh_received_signature:
            err = "Отсутствует заголовок Webhook-Signature."
            return JsonResponse({"errors": [err]}, status=400)

        wh_timestamp = request.headers.get("Webhook-Timestamp")  ❶
        if not wh_timestamp:
            err = "Отсутствует заголовок Webhook-Timestamp."
            return JsonResponse({"errors": [err]}, status=400)

        timestamp = datetime.fromtimestamp(int(wh_timestamp), tz=timezone.utc)
        offset = (datetime.now(timezone.utc) - timestamp).total_seconds()
        wh_expired = (int(offset) >= 5)
        if wh_expired:  ❷
            return JsonResponse({"errors": ["Запрос устарел."]}, status=400)

        # Создание подписи в соответствии со спецификацией Standard-Webhooks  ❸
        payload = request.body.decode("utf-8")
        secret = WEBHOOK_SECRET.encode("utf-8")
        sig_scheme = f"{wh_id}.{int(wh_timestamp)}.{payload}".encode("utf-8")
        signature = hmac.new(secret, msg=sig_scheme, digestmod="SHA256").digest()
        signature_b64 = base64.b64encode(signature).decode("utf-8")
        wh_expected_signature = f"v1,{signature_b64}"

        # Сравнение подписей
        if wh_received_signature != wh_expected_signature:  ❹
            return JsonResponse({"errors": ["HMAC не совпадает."]}, status=401)
        ...
```

❶ Эти утверждения позволяют проверить соответствие размера полезной нагрузки рекомендованным 20 Кбайт (https://oreil.ly/SG6FZ), а также наличие заголовков `Webhook-Id`, `Webhook-Signature` и `Webhook-Timestamp`,

обязательных для спецификации Standard-Webhooks. Если сообщение не проходит проверку, то возвращается соответствующий HTTP-ответ с кодом состояния `413 (Content Too Large)` (Размер сообщения слишком большой) или `400 (Bad Request)` (Неправильный запрос).

❷ Проверяется срок действия входящего запроса. Если метка времени запроса старше 5 секунд, то он считается просроченным и возвращается ответ с HTTP-кодом состояния `400 (Bad Request)` (Неправильный запрос). Проверка срока действия предотвращает атаку повторного воспроизведения (https://oreil.ly/D2IuK).

❸ В этом блоке кода генерируется подпись запроса согласно схеме подписи из спецификации Standard-Webhooks (https://oreil.ly/dPaGl). В процедуру входит вычисление HMAC на основе схемы подписи (`sig_scheme`) и секрета (`sectret`). `sig_scheme` — ключевого элемента этой процедуры, содержащего идентификатор вебхука, метку времени и полезную нагрузку. Метка времени в подписи используется, чтобы предотвратить обход проверки срока действия, а идентификатор вебхука — чтобы проверить его подлинность. Подпись далее преобразуется в строку Base64 (`signature_b64`) и объединяется с номером версии в `wh_expected_signature`.

❹ Сверка полученной и ожидаемой подписей гарантирует, что полезная нагрузка сообщения не была изменена. Если подписи не совпадают, то возвращается ответ с HTTP-кодом состояния `401 (Unauthorized)` (Доступ запрещен).

БЕЗОПАСНОСТЬ ВЕБХУКОВ

При написании кода мы исходили из нескольких установок, влияющих на реализацию.

Во-первых, мы рассматривали эндпоинт вебхука как внутренний эндпоинт организации, поэтому выводились подробные сообщения об ошибках. Но если бы вебхук предназначался для публичного использования, то мы бы не раскрывали детали возникших ошибок.

Во-вторых, чтобы предотвратить изменение сообщения, мы использовали общий секрет для подписи и проверки полезной нагрузки сообщения. Обмен секретами, выполняемый некорректно, может представлять угрозу безопасности.

Наконец, мы использовали метку времени для проверки срока действия сообщения. Она предотвращает повторное использование сообщения в атаке повторного воспроизведения (https://oreil.ly/H7K1i). В нашей реализации метка времени вычисляется после проверки наличия соответствующего заголовка, однако отсчет должен начинаться с момента обработки запроса.

Теперь протестируем интеграцию сервисов, используя обе реализации вебхуков.

В примере 9.4 имитируется отправка вебхуков из сервиса-источника к целевому сервису. Конкретная реализация выбирается на основе аргумента командной строки `--callback-url`. Эта строка предоставляет полезную нагрузку, которая отправляется на URL обратного вызова вебхука с помощью HTTP-метода `POST`. Если тест проходит успешно, то в консоль выводится полезная нагрузка ответа — это подтверждение, полученное от целевого сервиса.



Детали реализации CLI app_event представлены в файле src/django/app/core/management/commands/app_event.py.

Пример 9.4. Тестирование реализаций вебхуков

```
# Настройка лабораторной среды  ❶
cd src/django
docker compose build --build-arg UID=$(id -u) --build-arg GID=$(id -g)
docker compose up --detach --wait

# Установка переменных окружения  ❷
EPOCH=$(date +%s)
PAYLOAD='{"type":"dummy.event", "timestamp":'$EPOCH', "data":{"echo":"test"}}'

# Выполнение теста для эндпоинта v1  ❸
docker compose exec app bash -c \
       "python manage.py app_event \
       --callback-url webhook/v1/echo \
       --payload '$PAYLOAD'"

POST webhook/v1/echo
{"type":"dummy.event","timestamp":1709111631,"data":{"echo":"test"}}

# Выполнение теста для эндпоинта v2  ❹
docker compose exec app bash -c \
       "python manage.py app_event \
       --callback-url webhook/v2/echo \
       --payload '$PAYLOAD'"

# Вывод:
POST webhook/v2/echo
{"type":"dummy.event","timestamp":1709111631,"data":{"echo":"test"}}
```

❶ Выполните настройку лабораторной среды. Перейдите в соответствующий каталог, соберите Docker-образы и запустите контейнеры.

❷ Установите переменную окружения `EPOCH`, которая является меткой времени произошедшего события. Она измеряется в UTC-секундах с начала Unix-эпохи 1970-01-01T00:00:00Z. Эта метка времени отличается от метки

времени отправки вебхука. Создайте переменную окружения `PAYLOAD`, которая использует `EPOCH`.

❸ Запустите первый тест (`/webhook/v1/echo`), в котором используется вебхук, реализованный с помощью библиотеки Standard-Webhooks.

❹ Запустите второй тест (`/webhook/v2/echo`), в котором используется вебхук, реализованный вручную в соответствии со спецификацией Standard-Webhooks.

Документация

В примере 5.19 мы представили спецификацию OpenAPI (OAS). В этом разделе мы продолжим работу с OAS, но уже в контексте вебхуков.



Для поддержки вебхуков в словаре OAS есть специальные поля webhooks и callbacks.

Самая ранняя версия OAS, которую можно использовать для документирования вебхуков, — 3.0, поскольку в ней появилось поле `callbacks`. Оно содержится в поле `paths` и инкапсулирует объект OAS `path item`. Этот объект можно использовать, чтобы задокументировать процесс регистрации вебхуков внутри канала связи. Регистрация предполагает, что целевой сервис отправил HTTP-запрос к сервису-источнику, при этом запрос содержит URL обратного вызова целевого сервиса.

В версии OAS 3.1 добавилось поле `webhooks`. Это поле используется для описания событий, запускающих вебхуки, в том случае, если регистрация происходит вне канала связи (например, когда URL обратного вызова задается на уровне исходного кода или указывается через веб-интерфейс).

Наш инструмент командной строки (`app_event`), выступающий в качестве сервиса-источника для отправки вебхуков, не является API, поэтому не поддерживает HTTP-регистрацию вебхуков внутри канала связи. Однако для документирования функциональности этого инструмента все равно можно использовать OAS.

В примере 9.5 показан документ OpenAPI, в котором описан наш вебхук.

Пример 9.5. Спецификация OpenAPI для вебхука ECHO v1

```
# src/django/docs/api/webhook/v1-schema.yaml

openapi: 3.1.0  ❶
info:
  title: Вебхуки WFS CLI
```

```
  description: Вебхуки (события), запускаемые CLI WFS.
  version: v1
webhooks:  ❷
  dummy.event:  ❷
    post:  ❷
      summary: Тестовое событие, созданное в сервисе-источнике
      description: |
        Вебхук, инициированный сервисом-источником (поставщиком API),
        и ожидаемые ответы от целевого сервиса (потребителя API).

        **Предупреждение:** Ответ от целевого сервиса с кодом состояния,
        отличающимся от 2XX,считается неудачным вебхуком.
      parameters:  ❸
        - $ref: '#/components/parameters/WebhookId'
        - $ref: '#/components/parameters/WebhookSignature'
        - $ref: '#/components/parameters/WebhookTimestamp'
      requestBody:  ❹
        $ref: '#/components/requestBodies/Event'
      responses:  ❺
        200:
          $ref: '#/components/responses/OK'
        400:
          $ref: '#/components/responses/BadRequest'
components:
  parameters:  ❸
    WebhookId:
      name: Webhook-Id
      in: header
      required: true
      schema:
        type: string
    WebhookSignature:
      name: Webhook-Signature
      in: header
      required: true
      schema:
        type: string
    WebhookTimestamp:
      name: Webhook-Timestamp
      in: header
      required: true
      schema:
        type: string
  requestBodies:  ❹
    Event:
      description: JSON-полезная нагрузка события
      required: true
      content:
        application/json:
          schema:
            $ref: '#/components/schemas/Event'
          examples:
```

```
          dummy.event:
            value:
              type: dummy.event
              timestamp: 1709111631
              data:
                echo: test
responses:  ❺
  OK:
    description: Вебхук принят целевым сервисом
    content:
      application/json:
        schema:
          $ref: '#/components/schemas/Event'  ❻
  BadRequest:
    description: Вебхук отклонен целевым сервисом
    content:
      application/json:
        schema:
          $ref: '#/components/schemas/Error'
schemas:  ❻
    Event:
      type: object
      properties:
        type:
          type: string
          description: Тип события
        timestamp:
          type: string
          description: Метка времени события
        data:
          type: object
          description: Данные, связанные с событием
    Error:
      type: object
      properties:
        errors:
          type: array
          description: Сообщение об ошибке!
          items:
            type: string
```

❶ Обязательное поле `openapi` содержит версию OAS, используемую для API.

❷ В поле `webhooks` перечислены вебхуки поставщика API (у каждого из них уникальное имя). В нашем случае поле содержит один вебхук, соответствующий `dummy.event`. Он описывает HTTP-запрос `POST` (поле `post`) и ожидаемые ответы от целевого сервиса.

❸ В поле `parameters` используется ссылка (`$ref.`), указывающая на повторно используемые заголовки: `Webhook-Id`, `Webhook-Signature` и `Webhook-`

`Timestamp`. Символ `#` (решетка) используется, чтобы OAS начала искать нужное поле в разделе `components` (например, `#/components/parameters/`) с корня документа OpenAPI.

❹ Поле `requestBodies` описывает тело запроса. В данном случае это JSON-объект, содержащий ключи `type`, `timestamp` и `data`. Ключ `data` относится к объектному типу, а остальные ключи — к строковому.

❺ Поле `responses` описывает ответы вебхука. В данном случае он может возвращать `OK` или `BadRequest`.

❻ Поле `schemas` описывает полезную нагрузку сообщения, которая содержит поля `type` (строка), `timestamp` (строка) и `data` (объект).

Как видно из этого примера, OAS позволяет разработчикам документировать API с использованием машиночитаемых форматов, таких как YAML или JSON. Однако с точки зрения пользователя API важно и то, насколько спецификация удобна для чтения человеком.

Такие инструменты, как Swagger Editor, позволяют создавать и проверять документацию OAS в режиме онлайн. Одно из главных преимуществ Swagger Editor — функция предпросмотра спецификации в режиме реального времени. Протестируйте возможности инструмента самостоятельно: вставьте в редактор спецификацию из примера 9.5. Ожидаемый результат представлен на рис. 9.5.

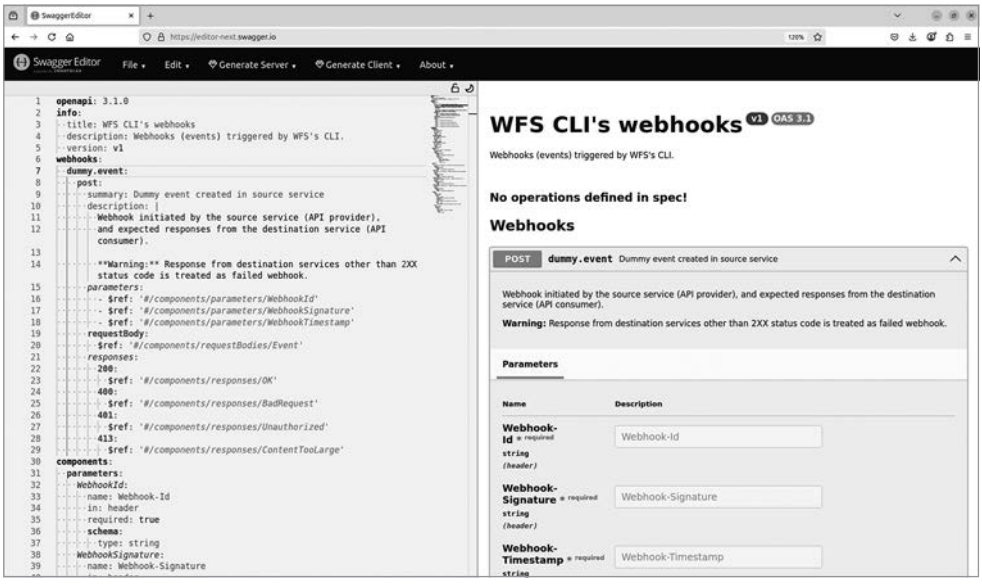


Рис. 9.5. Онлайн-редактор Swagger Editor



Соблюдайте осторожность при использовании онлайн-инструментов. Они удобны, но могут быть небезопасными. Мы не рекомендуем использовать онлайн-инструменты для обработки конфиденциальных данных. Сначала всегда рассматривайте локальные альтернативы.

Вебхуки: преимущества и недостатки

Начнем с преимуществ.

Подтверждение получения сообщений

В отличие от API на основе брокеров (см. главу 11), которые позволяют отправлять сообщения без подтверждения, в вебхуках целевой сервис подтверждает получение каждого сообщения. Это подтверждение укрепляет интеграцию сервисов, так как сообщения, которые не были получены, можно отправить повторно. Благодаря этому повышается устойчивость вебхуков к сбоям: если целевой сервис недоступен, то позже будет выполнена повторная попытка доставить сообщение.

Отсутствие механизма опроса

При опросе сервера сервис периодически отправляет запросы для получения данных. Иными словами, сервис регулярно запрашивает новые данные, что увеличивает объем данных, передаваемых по сети, и приводит к задержке между возникновением события и моментом, когда целевому сервису приходит уведомление о нем. В вебхуках сообщения отправляются целевому сервису по факту наступления события, без необходимости опроса.

Событийно-ориентированный подход

Отправка сообщения инициируется наступлением события. Благодаря этому целевой сервис мгновенно получает сообщения через вебхуки.

Асинхронная обработка данных

Коммуникация через вебхуки иногда описывается как «асинхронная по своей сути». Так говорят те, кто отождествляет HTTP-запрос `POST`, через который выполняется регистрация URL обратного вызова, с запросом, а отправляемые вебхуки — с ответами. Однако это отождествление некорректно, если речь идет о внешней конфигурации URL обратного вызова, — такая конфигурация не является «запросом» в привычном нам смысле. Полезно знать о наличии такой трактовки асинхронности. Тем не менее мы рекомендуем придерживаться определения асинхронной коммуникации, которое было дано в подразделе «Синхронная и асинхронная коммуникация» в главе 1.

Широковещательная рассылка сообщений

Вебхуки — это HTTP-запросы `POST` типа «точка — точка», которые работают в конфигурациях *«один к одному»* (один источник и один получатель) или *«один ко многим»* (один источник и много получателей). API на основе брокеров, напротив, обычно работают по схеме *«многие ко многим»*, где есть несколько сервисов-источников (поставщиков) и целевых сервисов (потребителей). Реализация вебхуков по схеме «многие ко многим» возможна, но такая конфигурация создаст дополнительную нагрузку как на сервисы-источники, так и на целевые сервисы. В API на основе брокеров эта нагрузка ложится на самого брокера.

Целостность полезной нагрузки

В полезную нагрузку сообщения можно внедрить нежелательные компоненты, а само сообщение может быть отправлено из ненадежного источника. Чтобы снизить эти риски, в вебхуках используется HMAC с общим секретным ключом для подписи и проверки полезной нагрузки сообщения, а также временная метка для проверки срока действия сообщения. Наконец, можно использовать список разрешенных IP-адресов, чтобы принимать сообщения только от доверенных источников.

Однако у вебхуков есть и недостатки.

Сложности с версионированием

Сервисы в вебхуках связаны через адрес сообщения и его полезную нагрузку, и изменение любого из этих элементов может нарушить интеграцию. Если требуется сохранить URL обратного вызова неизменным, то выполнять версионирование вебхуков можно двумя способами: с помощью HTTP-заголовков или тела сообщения (см. табл. 2.6). В примере ниже рассматривается версионирование через тело сообщения.

Для каждого имеющегося события сервис-источник создает отдельное сообщение, содержащее старую и новую полезную нагрузку. Например, `{"version": "v1", "first_name": "Джон", "last_name": "Доу"}` для старого сообщения и `{"version": "v2", "full_name": "Джон Доу"}` для нового. Преимущество этого подхода в том, что целевому сервису отправляются обе версии — он может выбрать подходящее сообщение и постепенно перейти на новый формат. Недостаток заключается в том, что целевой сервис вынужден принимать два сообщения для одного события, а сервис-источник должен поддерживать две версии полезной нагрузки и отправлять два запроса.

При альтернативном подходе мы не версионируем полезную нагрузку сообщения, а добавляем к ней новое поле. Например, к нагрузке `{"first_name": "Джон", "last_name": "Доу"}` добавляется поле `full_name`, и она становится

`{"first_name": "Джон", "last_name": "Доу", "full_name": "Джон Доу"}`. После этого отправляется только сообщение с новой нагрузкой. Когда новое поле только добавилось, его использование обычно считается необязательным. В сообщении будут доступны и новое, и старое поля, а целевой сервис решит, какое из них использовать.

Часто в обоих указанных подходах заранее согласовывают дату перехода на новую версию и уведомляют о ней целевой сервис. Эта дата указывает, когда сервис-источник прекратит отправку старой полезной нагрузки. Период до даты переключения — это время, за которое целевой сервис должен адаптироваться к новой нагрузке.

Отсутствие гарантий доставки сообщений

Вебхуки используются для интеграции между сервисами, где требуется подтверждение получения каждого сообщения. Если получение не подтверждено из-за того, что целевой сервис недоступен или возникли проблемы с сетью, то сообщение необходимо отправить повторно. Вследствие этого усложняется архитектура сервиса-источника, поскольку требуется добавить компоненты для хранения сообщений, организации очередей и планирования отправки. Кроме того, целевой сервис должен уметь обрабатывать потенциальные дубликаты.

Отсутствие стандартов

Хотя вебхуки появились в 2007 году, для них до сих пор нет единого стандарта. Попытка создать единый протокол предпринималась в проекте Standard-Webhooks, где были предложены спецификация и библиотеки для популярных языков программирования.

Когда использовать вебхуки

Вебхуки используются, когда требуется, чтобы один сервис реагировал на события, происходящие в другом (внешнем) сервисе. Например, при создании нового заказа в интернет-магазине (см. пример с книжным магазином, приведенный в этой главе) вебхук может уведомить складской сервис о том, что необходимо начать обработку заказа.

Еще один случай применения вебхуков — автоматизация процессов. Например, вебхук может запускать конвейер развертывания каждый раз, когда в репозиторий отправляется новый коммит.

Кроме того, вебхуки используются для отправки уведомлений, когда одному сервису нужно сообщить другому о событиях. Например, вебхук может

оповещать пользователей об изменениях при обновлении страницы мониторинга сервиса.

Наконец, вебхуки подходят для длительных асинхронных процессов, когда сервис уведомляет о ходе выполнения задачи — например, о том, что завершилось архивирование данных, конвертация видео или синхронизация файлов между системами.

Упражнения

1. В разделе «Интеграция и поток данных» выше в этой главе мы продемонстрировали процесс интеграции с помощью вебхуков. Теперь ваша очередь. Вам нужно реализовать вебхуки, поступающие из GitHub (https://github.com). Создайте форк репозитория с кодом из этой книги и вебхук, который уведомляет целевой эндпоинт (`/webhook/v3/echo callback`) о любом событии в репозитории, например об отправке коммита или обновлении описания репозитория. Подсказка: при выполнении этого упражнения используйте программу командной строки ngrok (https://ngrok.com), чтобы сделать локальный сервер, являющийся получателем вебхука, доступным через интернет. Ответ к этому упражнению находится в репозитории кода книги (см. файл `src/django/docs/solutions/callbacks-exercise.md`).
2. В нашей реализации вебхуков мы вручную создавали HMAC-подпись на Python. Теперь ваша задача — реализовать HMAC-подпись для вебхука в виде shell-скрипта. После создания подписи составьте команду `curl`, которая отправляет `POST`-данные на эндпоинты `/webhook/v1/echo` и `/webhook/v2/echo`. Ответ к этому упражнению находится в репозитории кода книги (см. файл `src/django/docs/solutions/callbacks-exercise.md`).

Резюме

В этой главе вы познакомились с вебхуками, которые позволяют сервисам обмениваться информацией через механизм обратного вызова. Взаимодействие между сервисами заключается в том, что сервис-источник регистрирует обратный вызов и отправляет сообщение целевому сервису, который определяет обратный вызов.

Вебхуки — это обратные вызовы, в которых используется HTTP-метод `POST`. С точки зрения сервиса вебхуки могут быть входящими или исходящими.

Обычно их применяют для интеграции между сервисами, автоматизации процессов, а также для настройки уведомлений о событиях или логирования.

В этой главе мы обсудили происхождение вебхуков и процесс их интеграции. В практической части главы мы реализовали вебхуки в соответствии со спецификацией Standard-Webhooks, использовали HMAC для обеспечения целостности и подлинности сообщений, а также задокументировали вебхук с помощью OpenAPI.

В заключение мы поговорили об основных преимуществах и недостатках вебхуков, знание которых поможет понять, подходят ли вебхуки для решения ваших задач.

ГЛАВА 10

WebSocket

API на основе HTTP работают по модели «запрос — ответ»: клиент отправляет запрос серверу и ожидает соответствующий ответ. В HTTP сервер не может инициировать соединение или отправить ответ без предварительного запроса от клиента. С помощью методов server-sent events («события, отправляемые сервером», SSE) (https://oreil.ly/sDPnx) и HTTP-стриминга (https://oreil.ly/ETuMx) сервер *может* многократно инициировать отправку данных клиенту, однако для установления соединения все равно требуется запрос от клиента. С другой стороны, сам клиент может периодически опрашивать сервер (отправлять запросы), чтобы получить новую информацию. Для этого клиент использует методы *short-polling* («короткий опрос», где для каждого запроса устанавливается новое TCP-соединение) или *long-polling* («длинный опрос», где одно долгоживущее TCP-соединение используется повторно для отправки нескольких запросов).

Цель WebSocket — обеспечить одновременную двунаправленную коммуникацию. В отличие от модели «запрос — ответ» в WebSocket и клиент, и сервер могут отправлять друг другу сообщения независимо. Эта возможность появляется благодаря переключению с HTTP на WebSocket — специальный протокол, основанный на TCP и поддерживающий полнодуплексный режим передачи данных.

При обсуждении WebSocket часто упоминается асинхронная коммуникация[1], однако этот протокол не является асинхронным по умолчанию[2]. Есть синхронные реализации клиентов и серверов WebSocket[3], однако его асинхронное ис-

---

[1] Подробнее о синхронном и асинхронном типах коммуникации см. в подразделе «Синхронная и асинхронная коммуникация» в главе 1.

[2] Стандарт WebSocket (https://oreil.ly/VEGhC) не предполагает обязательного использования синхронной или асинхронной модели коммуникации.

[3] См., например, простой сервер и клиент WebSocket для Python (https://oreil.ly/W3E1D).

пользование имеет ряд преимуществ, среди которых эффективное управление множеством соединений и повышение отзывчивости системы[1].

В этой главе мы поговорим о протоколе WebSocket, процессе установления соединения (процедуре рукопожатия) и деталях реализации WebSocket API на стороне сервера и клиента. Кроме того, мы расширим систему WFS, добавив в нее оповещения о погоде, и покажем, как установить защищенное соединение через WebSocket. В заключение обсудим, как документировать WebSocket API и каковы его преимущества и недостатки.

WebSocket и WebSocket API

WebSocket — это протокол, созданный для организации двунаправленной коммуникации без использования методов, которые требуют отправки большого количества HTTP-запросов, например `XMLHttpRequest` (https://oreil.ly/OePtl) или long-polling (https://oreil.ly/JYxQX)[2]. Протокол WebSocket, стандартизированный в 2011 году в RFC 6455 (https://oreil.ly/PSNLg), предоставляет канал для двунаправленной коммуникации через одно долгоживущее TCP-соединение.

На рис. 10.1 показаны различия в передаче данных между клиентом и сервером для протоколов HTTP и WebSocket. В HTTP/1 сообщения одновременно передаются только в одном направлении — от отправителя (клиента) к получателю (серверу) или наоборот. В WebSocket, напротив, сообщения между клиентом и сервером могут передаваться в обоих направлениях одновременно.

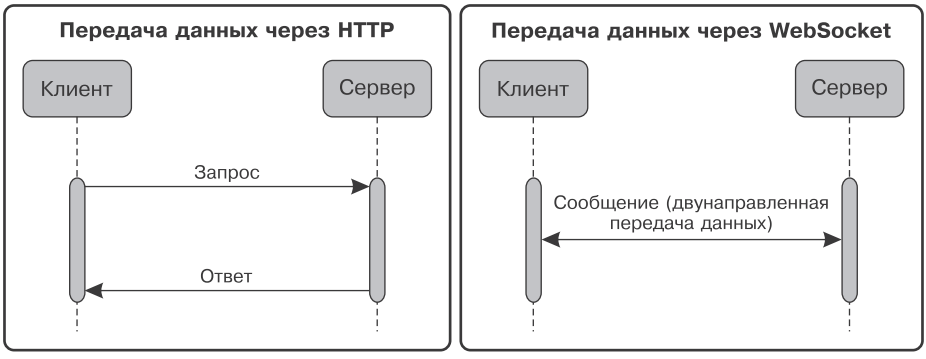


Рис. 10.1. Сравнение передачи данных через HTTP и WebSocket

[1] Система называется отзывчивой, если своевременно реагирует на запросы пользователя. См. определение отзывчивости в документе The Reactive Manifesto («Манифест реактивного программирования») (https://www.reactivemanifesto.org).

[2] Узнать больше о применении WebSocket можно в книге Алекса Диакону (Alex Diaconu) The WebSocket Handbook (https://oreil.ly/4o8tV).

Обратите внимание: в WebSocket HTTP может использоваться только для первичного установления соединения (процедуры рукопожатия). При этом допускаются и альтернативные способы выполнения рукопожатия, не основанные на HTTP.

В браузерах для использования WebSocket реализуется WebSocket API в соответствии со спецификацией *языка описания веб-интерфейсов WebSocket* (WebSockets Web IDL) (https://oreil.ly/a5quQ)[1].

Установка соединения WebSocket

Для установления двусторонней связи между клиентом и сервером по протоколу WebSocket выполняется трехэтапный процесс переключения протокола (https://oreil.ly/611PH). Этот процесс, известный как рукопожатие WebSocket, показан на рис. 10.2.

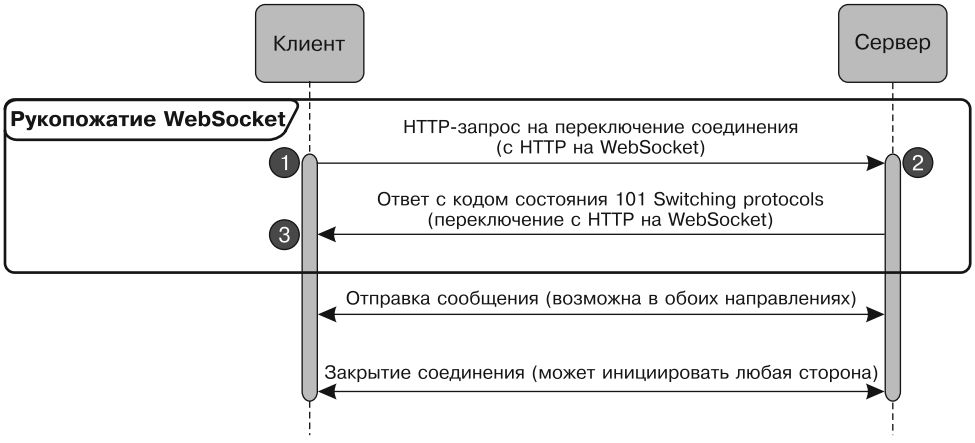


Рис. 10.2. Процесс установления соединения через WebSocket

Установка соединения происходит следующим образом.

1. Клиент отправляет HTTP-запрос `GET`, содержащий заголовки `Upgrade`, `Connection` и `Sec-WebSocket-Key`.
2. Сервер получает запрос клиента и возвращает HTTP-ответ с заголовками `Upgrade`, `Connection` и `Sec-WebSocket-Accept`, а также с кодом состояния `101`

[1] Web IDL (https://oreil.ly/0qz3P) — это язык описания интерфейсов, используемый для решения проблемы совместимости программных веб-интерфейсов. Он позволяет создавать описания веб-API, совместимые с JavaScript. В примере 10.4 будет показана реализация API WebSocket в браузере с использованием Web IDL.

(`Switching Protocols`) (Переключение протоколов), сигнализируя о готовности переключить базовое HTTP-соединение на WebSocket.

3. Клиент получает ответ сервера и переключает соединение с HTTP на WebSocket.

После успешного установления соединения клиент и сервер получают возможность отправлять фреймы данных WebSocket.

Рассмотрим процесс рукопожатия WebSocket на практике. Для этого настроим лабораторную среду по инструкциям из примера 10.1.



В этой главе при работе в GitHub Codespaces нужно изменить видимость порта 8001 сервера WebSocket на Public (Общедоступный) и отключить TLS, присвоив переменной TLS_ENABLE значение 0 (см. файл src/django/compose.yaml). GitHub Codespaces поддерживает TLS по умолчанию.

Пример 10.1. Настройка лабораторной среды WebSocket

```
cd src/Django  ❶
docker compose build --build-arg UID=$(id -u) --build-arg GID=$(id -g)  ❷
docker compose up --detach –wait  ❸
```

❶ Перейдите в каталог для настройки лабораторной среды.

❷ Соберите Docker-образы.

❸ Запустите Docker-контейнеры.

Далее выполните следующие шаги, чтобы увидеть процесс рукопожатия WebSocket в действии.

1. Откройте новую вкладку в браузере.
2. Откройте инструменты разработчика (используйте сочетание клавиш Ctrl+Shift+I в Windows/Linux или Option+Cmd+I в macOS) и перейдите на вкладку Network.
3. Перейдите по адресу http://localhost:8000/websocket/echo[1].

Выполнив эти действия, вы должны увидеть такую страницу (рис. 10.3).

На рис. 10.3 показано приложение ECHO WebSocket Application (EWA). Если в главном окне браузера отображается сообщение `Hello WebSocket!`,

[1] При работе с GitHub Codespaces используется уникальный URL, содержащий имя Codespace вместо адреса localhost. Этот URL привязан к порту 8000, и его можно найти на вкладке PORTS в VS Code.

значит, приложение работает. В разделах Response Headers и Request Headers описаны заголовки сервера и клиента, которые используются при процедуре рукопожатия. В консоли браузера мы видим лог клиента, содержащий такую последовательность шагов.

1. Клиент пытается установить соединение WebSocket с эндпоинтом `ws://localhost:8001/ws/v1/echo`. Клиент отправляет HTTP-запрос с заголовками, необходимыми для рукопожатия со стороны клиента.
2. Сервер принимает и проверяет запрос клиента. Затем он возвращает HTTP-ответ с кодом состояния `101 (Switching Protocols)` и с заголовками, необходимыми для рукопожатия со стороны сервера. Код состояния `101 (Switching Protocols)` в ответе указывает клиенту переключить соединение с HTTP на WebSocket.
3. Клиент получает ответ сервера и переключает соединение на WebSocket:
   а) далее клиент отправляет серверу текстовый фрейм данных, содержащий сообщение `Hello WebSocket!` в формате JSON;
   б) сервер получает текстовый фрейм данных и возвращает его обратно клиенту;
   в) получив текстовый фрейм данных, клиент выводит сообщение в консоль и закрывает соединение WebSocket.

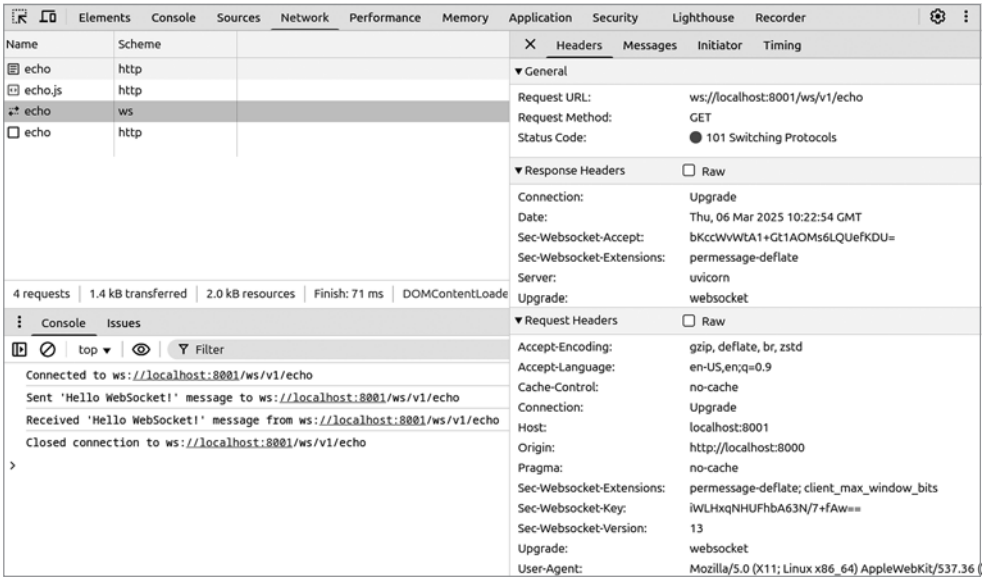


Рис. 10.3. Рукопожатие WebSocket, проанализированное через браузер



Термин «сообщение» связан с данными прикладного уровня в модели OSI или в TCP/IP. В протоколе WebSocket сообщение называют фреймом данных (не путать с фреймом Ethernet). Такой фрейм может содержать текстовые или двоичные данные.

Теперь, когда мы разобрали механизм установления соединения в EWA, рассмотрим подробнее заголовки рукопожатия WebSocket, показанные на рис. 10.3.

В примере 10.2 приведены обязательные HTTP-заголовки, которые необходимы для рукопожатия со стороны клиента. Сервер должен понимать HTTP, поскольку будет использовать HTTP-заголовки для проверки клиента и переключения протокола с HTTP на WebSocket.

В EWA используется JavaScript-клиент (сгенерированный сайт Django), который автоматически отправляет запрос серверу при загрузке. Для удобства мы сократили примеры 10.2 и 10.3, поэтому они содержат только обязательные заголовки; в браузере их будет больше.

Пример 10.2. Рукопожатие WebSocket (запрос клиента)

```
GET /ws/v1/echo HTTP/1.1                    ❶
Origin: http://localhost:8000               ❷
Host: localhost:8001                        ❸
Connection: keep-alive, Upgrade             ❹
Upgrade: websocket                          ❺
Sec-WebSocket-Key: zLsL2bD4QARLOTXv0vI1Ig== ❻
Sec-WebSocket-Version: 13                   ❼
```

❶ Клиент отправляет HTTP-запрос `GET` к эндпоинту сервера `/ws/v1/echo`.

❷ Заголовок `Origin` идентифицирует клиента. Клиент определяется по схеме запроса, имени хоста и номеру порта.

❸ Заголовок `Host` указывает расположение сервера WebSocket, имя его хоста и номер порта.

❹ Заголовок `Connection` с директивой `keep-alive` указывает, что клиент хочет повторно использовать TCP-соединение, установленное при первом HTTP-запросе, и держать его открытым для передачи последующих сообщений. Директива `Upgrade` сообщает серверу, что клиент хочет переключить текущий протокол на протокол, указанный в заголовке `Upgrade`.

❺ Заголовок `Upgrade` сообщает серверу, что клиент хочет переключить протокол прикладного уровня на `websocket`.

❻ Заголовок `Sec-WebSocket-Key` (https://oreil.ly/YW5W5) отправляется клиентом и используется сервером для проверки рукопожатия. Заголовку присваивается случайное 16-байтовое значение в кодировке Base64.

❼ Заголовок `Sec-WebSocket-Version` (https://oreil.ly/taO5a) должен иметь значение `13`. Если сервер не поддерживает эту версию протокола WebSocket, то соединение не будет установлено.



Обратите внимание на то, что порт 8001 в заголовке Host (сервер) отличается от порта 8000 в заголовке Origin (клиент). Причина в том, что в нашей конфигурации сервер обрабатывает все запросы WebSocket с помощью фреймворка Starlette (https://oreil.ly/T0BkU) на порте 8001, тогда как серверное приложение Django (https://www.djangoproject.com), отправляющее запрос к серверу WebSocket, работает на порте 8000. Другими словами, у нас есть два относительно независимых экземпляра приложения, каждый из которых привязан к конкретному порту в зависимости от используемого стека технологий и способа его развертывания. Если бы сервер поддерживал обработку соответствующих протоколов (HTTP, TCP и WebSocket) на одном порте, то такая конфигурация не потребовалась бы.

Проверив заголовки в запросе клиента, сервер отправляет ответ, содержащий код состояния `101 (Switching Protocols)` и заголовки, которые ожидает клиент. В ответе сервера на рукопожатие есть заголовки, связанные с WebSocket, — они перечислены в примере 10.3.

Пример 10.3. Рукопожатие WebSocket (ответ сервера)

```
HTTP/1.1 101 Switching Protocols                    ❶
Upgrade: websocket                                  ❷
Connection: Upgrade                                 ❸
Sec-WebSocket-Accept: +KhiFov/sM2iO0A+xSQebkTa+0E=  ❹
```

❶ Сервер сообщает клиенту о готовности переключить протоколы, отправляя HTTP-ответ с кодом состояния `101 (Switching Protocols)`.

❷ Заголовок `Upgrade` говорит о том, что протокол будет переключен на `websocket`.

❸ Заголовок `Connection` указывает, что сервер переключится на протокол, указанный в заголовке `Upgrade`.

❹ Заголовок `Sec-WebSocket-Accept` (https://oreil.ly/5f8wY) указывает, что сервер принял соединение. На основе ключа `Sec-WebSocket-Key`, который отправил клиент, генерируется одноразовый хеш.

После успешного завершения рукопожатия устанавливается соединение WebSocket, по которому клиент и сервер могут передавать данные в обоих направлениях.

Реализация

Наш API WebSocket содержит два основных компонента: клиент и сервер. На рис. 10.3 была показана процедура рукопожатия при открытии соединения WebSocket и обмен одним фреймом данных между клиентом и сервером. В этом разделе мы рассмотрим реализацию API WebSocket для клиента и сервера EWA.

Клиент и сервер WebSocket ECHO

Теперь, когда мы рассмотрели основы протокола WebSocket и процедуру рукопожатия, мы можем реализовать API WebSocket для клиента и сервера. В примере 10.4 показана реализация API WebSocket на стороне JavaScript-клиента.

Пример 10.4. Реализация клиента EWA

```
// src/django/app/core/www/static/app/js/echo.js

...
const socket = new WebSocket('ws://localhost:8001/ws/v1/echo');  ❶

socket.onopen = function(e) {  ❷
  console.info(`Connected to ${socket.url}`);
  const msg = {'message': 'Hello WebSocket!'};
  socket.send(JSON.stringify(msg));
  console.info(`Sent '${msg.message}' message to ${socket.url}`);
}

socket.onmessage = function(e) {  ❸
  const message = JSON.parse(e.data).message;
  document.querySelector('h1').innerText = message;
  console.info(`Received '${message}' message from ${socket.url}`);
  socket.close();
}

socket.onclose = function(e) {  ❹
  console.info(`Closed connection to ${socket.url}`);
}

socket.onerror = function(e) {  ❺
  console.error(`Encountered an error when talking to ${socket.url}`);
};
...
```

❶ Клиент `WebSocket` пытается установить новое соединение с сервером. Эта строка инициирует процедуру рукопожатия при открытии соединения WebSocket.

❷ Функция `onopen()` срабатывает при успешном установлении соединения WebSocket. Клиент отправляет на сервер сообщение в формате JSON, а затем записывает его в консоль.

❸ Функция `onmessage()` срабатывает, когда клиент получает сообщение от сервера. Клиент выполняет четыре действия: читает сообщение, обновляет тег `h1` на веб-странице, записывает сообщение в консоль и отправляет запрос на закрытие соединения WebSocket. Обычно клиент не закрывает соединение после получения сообщения — мы делаем это, чтобы продемонстрировать механизм закрытия.

❹ Функция `onclose()` срабатывает после закрытия соединения WebSocket. Клиент записывает в лог информацию о закрытии соединения.

❺ Функция `onerror()` отвечает за запись сообщения об ошибке в лог, если ошибка обнаруживается в уже установленном соединении.



При выводе логов в консоли клиента может оказаться ненужная информация. Кроме того, вывод логов может привести к снижению производительности приложения. Мы делаем это только для того, чтобы показать алгоритм выполнения программы. В продакшен-среде логи следует отправлять в агрегатор логов, а не выводить в консоль.

В примере 10.5 показана реализация сервера EWA.

Пример 10.5. Реализация сервера EWA

```
# src/django/app/core/api/websocket/v1/endpoints.py

from starlette.endpoints import WebSocketEndpoint

...
class EchoEndpoint(WebSocketEndpoint):  ❶
    encoding = "json"

    async def on_connect(self, websocket):  ❷
        await websocket.accept()

    async def on_receive(self, websocket, data):  ❸
        await websocket.send_json(data)
```

❶ Класс `EchoEndpoint` обрабатывает входящие соединения WebSocket. Он наследуется от `WebSocketEndpoint` из фреймворка Starlette (https://oreil.ly/JwqFN). `WebSocketEndpoint` предоставляет три переопределяемых метода для обработки соединений: `on_connect()`, `on_receive()` и `on_disconnect()`. На нашем эндпоинте используются только `on_connect()` и `on_receive()`. Атрибут `encoding` проверяет, имеют ли данные в `on_receive()` формат JSON.

❷ Метод `on_connect()` вызывается, когда клиент устанавливает соединение с WebSocket-эндпоинтом. В данном случае соединение принимается, но при необходимости его можно и отклонить.

❸ Метод `on_receive()` вызывается, когда клиент отправляет на сервер фрейм данных (сообщение). Сервер ECHO отвечает клиенту теми же данными, которые получил (то есть возвращает их «эхом»).

Протокол WebSocket

В этом подразделе мы рассмотрим пример обмена данными по протоколу WebSocket и его фреймы данных. С помощью анализа пакетов EWA мы сможем реконструировать логику работы приложения, описанную в разделе «Установка соединения WebSocket» выше в этой главе. Для анализа сетевого трафика будем использовать анализатор сетевых протоколов Wireshark (https://www.wireshark.org).

Проанализируем трафик WebSocket, захваченный при обращении к http://localhost:8000/websocket/echo. Захват пакетов, сохраненный в файле `src/django/docs/assets/websocket-echo-unencrypted.pcap`, загрузим в Wireshark (рис. 10.4).

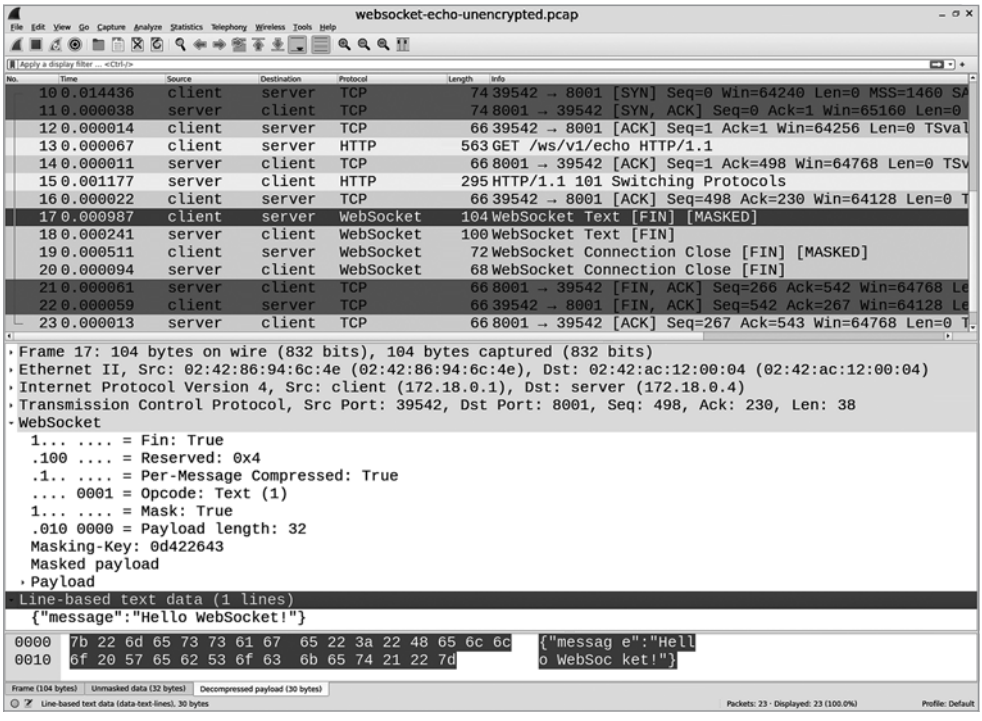


Рис. 10.4. Незашифрованный трафик WebSocket приложения EWA

- Фреймы 10–12 — клиент устанавливает TCP-соединение с сервером, выполняя трехэтапное рукопожатие TCP.
- Фрейм 13 — клиент инициирует рукопожатие WebSocket, отправляя HTTP-запрос `GET`.
- Фрейм 15 — сервер отвечает на рукопожатие, сообщая о готовности переключить протокол на WebSocket (код состояния 101).
- Фрейм 17 — после установления соединения WebSocket с сервером клиент отправляет текстовый фрейм данных. Его полезная нагрузка содержит сообщение `Hello WebSocket!` в формате JSON. В протоколе WebSocket флаг `FIN` (Final Fragment Indicator) (Индикатор последнего фрагмента) со значением `1` сообщает серверу, что других фреймов данных не будет. Иными словами, сообщение полностью содержится в одном фрейме.
- Фрейм 18 — сервер возвращает полученную полезную нагрузку («эхо»), см. нижнюю часть рисунка.
- Фрейм 19 — клиент отправляет фрейм закрытия, чтобы завершить соединение WebSocket («заключительное рукопожатие»). Для этого полю `Opcode` во фрейме присваивается значение `Connection Close` (Закрытие соединения).
- Фрейм 20 — сервер подтверждает намерение клиента и отвечает фреймом закрытия.
- Фреймы 21–23 — сервер и клиент закрывают базовое TCP-соединение.

Мы рассмотрели особенности протокола WebSocket, процедуры рукопожатия для установления и завершения соединения, а также проанализировали сетевые пакеты, которыми обмениваются клиент и сервер. Теперь мы можем перейти к реализации более сложного примера с WebSocket.

Сервер оповещений о погоде на основе WebSocket

В этом подразделе мы расширим функциональность WFS, добавив оповещения о погоде. Программа, которую мы реализуем, позволит клиентам получать оповещения для заданного города, который определяется по UUID. Этот идентификатор будет использоваться, чтобы определить, на канал какого города отправлять сообщения с оповещениями. После запуска рассылки через командную строку (CLI) клиенты, подписанные на канал, получат сообщения. Подобные оповещения будут получать все клиенты WebSocket, подключенные к API.

На рис. 10.5 показана система оповещений, которую мы создадим, и движение сообщений между ее компонентами.

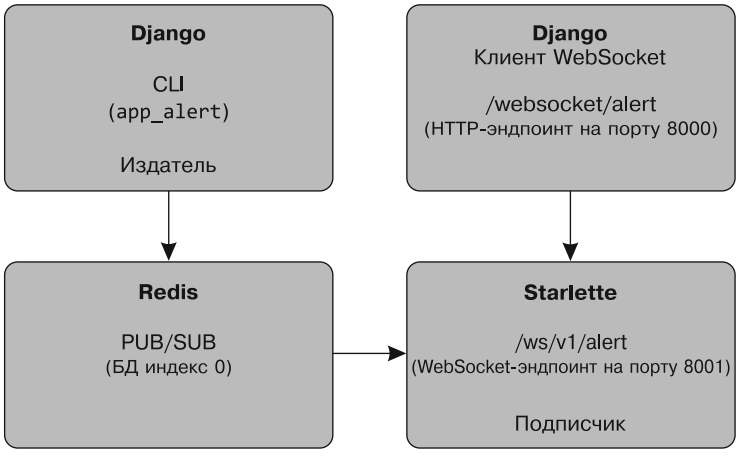


Рис. 10.5. Обзор системы оповещений WFS и движение сообщений между ее компонентами

WFS архитектурно разделена на две независимые программы: приложение Django (клиент WebSocket) на порте `8000` и приложение Starlette (сервер WebSocket) на порте `8001`. Соответственно, нам нужно было найти способ передачи сообщений от первого приложения ко второму. Наиболее подходящим оказался паттерн обмена сообщениями «издатель — подписчик» (publish-subscribe, pub/sub) (подробнее о нем см. в подразделе «Паттерн "Издатель — подписчик (pub/sub)"» в главе 11). Этот паттерн позволит передавать данные из приложения Django в Starlette.

Мы реализуем паттерн pub/sub с помощью Redis — базы данных типа «ключ — значение», работающей в оперативной памяти. Redis предоставляет легковесную реализацию доставки сообщений по схеме pub/sub. Кроме того, в Django мы реализуем интерфейс командной строки (CLI) `app_alert`, который будет создавать сообщение с оповещением и публиковать его в канале pub/sub. Затем приложение Starlette подпишется на этот канал, чтобы получать поступающие в него сообщения и отправлять их в качестве оповещений всем подключенным клиентам WebSocket.



Изначально Redis относился к резидентным (in-memory) базам данных типа «ключ — значение», однако со временем начал выполнять функции инструмента кэширования и NoSQL базы данных типа «ключ — значение». Сейчас Redis поддерживает и такие функции, как векторный поиск, транзакции и обмен сообщениями по модели pub/sub.

Прежде чем обсуждать реализацию системы оповещений, протестируем ее, отправив тестовое оповещение. В примере 10.6 пошагово показано, как отправить

оповещение. Предполагается, что вы находитесь в каталоге для настройки лабораторной среды, как было показано в примере 10.1.

Пример 10.6. Отправка оповещения WFS с помощью CLI (требуются настройки из примера 10.1)

```
# Получение UUID города  ❶
CITY_UUID=$(docker compose exec app python manage.py app_cities | \
  head -n 1 | \
  cut -d' ' -f 2)

# Отправка сообщения с оповещением  ❷
docker compose exec app python manage.py app_alert --city-uuid=$CITY_UUID

# Вывод:
Оповещение 'Ожидается ливень с градом!' отправлено в канал 'd86962f4-fb28-
4b98-bbeb-c9fbec0d0fca'.
```

❶ Получаем UUID первого города (`CITY_UUID`), который используется в качестве канала pub/sub в Redis.

❷ Отправляем стандартное сообщение с оповещением («Ожидается ливень с градом!») всем клиентам, которые прослушивают канал pub/sub в Redis, указанный в параметре `--city-uuid`.

На рис. 10.6 показано движение сообщений с оповещениями от сервера WebSocket к клиенту. В нижней части изображения виден терминал, где пользователь выполняет команду `app_alert` из примера 10.6. В верхней части изображения показан браузер, получающий стандартное сообщение с оповещением. Обратите внимание на порядок сообщений в консоли (в центре изображения): сначала было установлено соединение с эндпоинтом WebSocket, а потом получено сообщение.

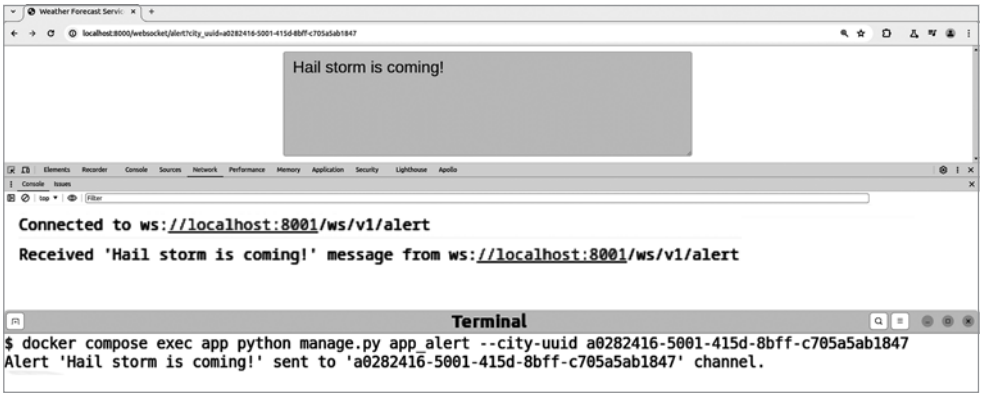


Рис. 10.6. Сквозное тестирование оповещений WFS

Теперь мы можем перейти к реализации сервера WebSocket.

Начнем с выбора технологии для реализации сервера WebSocket. Мы будем использовать фреймворк Starlette (https://www.starlette.io), который поддерживает WebSocket API (https://oreil.ly/82kVk) и может обрабатывать запросы асинхронно с помощью протокола ASGI (Asynchronous Server Gateway Interface) (https://oreil.ly/USGcA).

В примере 10.7 показана конфигурация приложения Starlette.



Интерфейс ASGI позволяет веб-серверам на Python обрабатывать запросы асинхронно.

Пример 10.7. Конфигурация Starlette

```
# src/django/app/config/asgi.py

...
from starlette.applications import Starlette
from starlette.middleware import Middleware
from starlette.middleware.authentication import AuthenticationMiddleware
from core.helpers.middleware import WebSocketOriginValidatorMiddleware
from core.helpers.middleware import WebSocketTrustedHostMiddleware
...
from config.constants import APP_STARLETTE_AUTHENTICATE_ENDPOINTS
from config.urls import routes
from core.helpers.auth import auth_error
from core.helpers.auth import StarletteJWTAuthBackend
...

middleware = [  ❶
  Middleware(WebSocketOriginValidatorMiddleware, allow_
origins=["localhost:8000"]),
  Middleware(WebSocketTrustedHostMiddleware, allowed_
hosts=["localhost:8001"]),
  Middleware(AuthenticationMiddleware,
    backend=StarletteJWTAuthBackend(APP_STARLETTE_AUTHENTICATE_ENDPOINTS),
    on_error=auth_error
  ),
]

application_starlette = Starlette(  ❷
  debug=True, routes=routes, middleware=middleware
)
```

❶ Конфигурация приложения начинается с определения трех объектов `Middleware`: `WebSocketOriginValidatorMiddleware`, `WebSocketTrustedHostMiddleware` и `AuthenticationMiddleware`. Функциональность

последнего расширяется пользовательским классом `StarletteJWTAuthBackend` для аутентификации клиентов. Этот класс описан в примере 10.11 далее.

Действие класса `Middleware` распространяется на весь цикл «запрос — ответ» в приложении. Прежде чем запрос будет обработан конкретной функцией, он проходит через стек промежуточного ПО в порядке, заданном приложением. Ответ, сгенерированный приложением, проходит тот же стек в обратном порядке. Список `allow_origins` содержит источники клиентов, которым разрешено подключаться к серверу, а список `allowed_hosts` указывает разрешенные хосты сервера.

❷ Переменная `application_starlette` создается из объекта `Starlette`. В этот объект передается список промежуточного ПО вместе с маршрутами WebSocket, определенными в примере 10.8.

Серверное приложение Starlette настроено. Следующий шаг — определить два эндпоинта. Первый, предоставляемый серверным приложением Django, будет использоваться браузером для получения веб-страницы. Второй эндпоинт, предоставляемый Starlette, будет использоваться кодом JavaScript на этой веб-странице, чтобы установить соединение WebSocket и получать оповещения. Определение эндпоинтов показано в примере 10.8.

Пример 10.8. Эндпоинты WebSocket

```
# src/django/app/config/urls.py

...
from rest_framework_simplejwt.views import TokenObtainPairView
from starlette.routing import WebSocketRoute
from core.api.websocket.v1.endpoints import AlertEndpoint
from core.www import ws_alert_view
...

urlpatterns = [  ❶
    ...
    path("websocket/alert", ws_alert_view),
    path("api/jwt/obtain", TokenObtainPairView.as_view()),
    ...
]
routes = [  ❷
    WebSocketRoute("/ws/v1/alert", AlertEndpoint),
    ...
]
```

❶ Список `urlpatterns` содержит эндпоинты приложения WFS. Эндпоинт `/websocket/alert` — это представление Django, которое генерирует клиент WebSocket (в виде HTML-страницы). Клиент получает токен JWT,

устанавливает соединение WebSocket с сервером и управляет этим соединением. Эндпоинт `/api/jwt/obtain` позволяет получить токен JWT. Токен генерируется с помощью пакета Simple JWT (https://oreil.ly/rDMcm), предназначенного для работы с DRF (Django REST Framework) (см. пример 5.17).

❷ Объект `WebSocketRoute` определяет эндпоинт `/ws/v1/alert`, который обрабатывается классом `AlertEndpoint`. Этот класс реализует логику серверного API WebSocket.

Теперь рассмотрим детали реализации класса `AlertEndpoint`, который обрабатывает маршрут `/ws/v1/alert`, в примере 10.9.

Пример 10.9. Сервер оповещений WebSocket

```
# src/django/app/core/api/websocket/v1/endpoints.py

import asyncio

from asgiref.sync import sync_to_async
from redis import asyncio as redisasyncio
from starlette.endpoints import WebSocketEndpoint

from core.helpers.utils import to_bytes
from core.helpers.utils import to_str
from core.models import CityRepository

class AlertEndpoint(WebSocketEndpoint):  ❶
    clients = dict()
    encoding = "json"
    lock = asyncio.Lock()
    pubsub = redisasyncio.Redis(host="redis", port=6379, db=0).pubsub()  ❷

    async def on_connect(self, websocket):  ❸
        await websocket.accept()
        city_uuid = websocket.query_params.get("city_uuid")
        if not city_uuid:
            reason = "В запросе отсутствует или содержится недопустимый
              параметр запроса city_uuid. "
            return await websocket.close(code=1008, reason=reason)  ❹

        city = await sync_to_async(CityRepository.get_by_id)(city_uuid)
        if not city:
            reason = f"Город с {city_uuid} не существует."
            return await websocket.close(code=1008, reason=reason)  ❹

        await self.pubsub.subscribe(city_uuid)  ❻

        if city_uuid not in self.clients:  ❼
            self.clients[city_uuid] = set()
        self.clients[city_uuid].add(websocket)
```

```
        asyncio.create_task(
            self.__msg_watch_and_broadcast(websocket, city_uuid)  ❽
        )

    async def on_disconnect(self, websocket, close_code):  ❺
        city_uuid = websocket.query_params.get("city_uuid")
        clients = self.clients.get(city_uuid)
        if city_uuid and clients:
            for client in clients.copy():
                if websocket == client:
                    self.clients[city_uuid].remove(client)

    async def __msg_watch_and_broadcast(self, websocket, city_uuid):  ❽
        while to_bytes(str(city_uuid)) in self.pubsub.channels:
            async with self.lock:
                message = await self.pubsub.get_message(
                    ignore_subscribe_messages=False
                )
                msg = message and message["type"] == "message"
                msg_matches_channel = (
                    message and to_str(message["channel"]) == city_uuid
                )
                if msg and msg_matches_channel:
                    data = {"message": to_str(message["data"])}
                    await self.__broadcast(data, city_uuid)  ❾
            await asyncio.sleep(1)

    async def __broadcast(self, city_uuid, data):  ❾
        for client in self.clients.get(city_uuid).copy():
            try:
                await client.send_json(data)
            except Exception as e:
                print(f"Сообщение не было отправлено по причине {e}")
```

❶ Сообщение WebSocket (фрейм данных) проходит по маршруту `/ws/v1/alert`, который обрабатывается классом `AlertEndpoint`. Этот класс наследуется от класса `WebSocketEndpoint` из фреймворка Starlette.

❷ Атрибут класса `pubsub` — это объект Redis pub/sub (https://oreil.ly/tS1K9), который используется для приема и обработки сообщений, публикуемых в соответствующий канал.

❸ Метод `on_connect()` вызывается, когда клиент устанавливает соединение с эндпоинтом WebSocket. Метод принимает объект соединения WebSocket и считывает UUID города из параметра запроса.

❹ Если в базе данных нет UUID города или город не существует, то соединение закрывается с кодом состояния `WebSocket 1008` (`Policy Violation`) (Нарушение политики WebSocket). Таким образом, инициируется событие закрытия[1].

[1] О кодах состояния WebSocket подробнее см. в RFC 6455 (https://oreil.ly/eJFCp).

❺ Метод `on_disconnect()` вызывается, когда соединение WebSocket закрылось. Он предпринимает попытку удалить подключенного клиента WebSocket, убирая его данные из атрибута `clients`.

❻ Если UUID города есть и город имеется в базе данных, то для подписки на канал и прослушивания сообщений используется объект `pubsub`. UUID города применяется для создания канала Redis pub/sub, на который подписывается клиент WebSocket[1].

❼ Этот блок кода создает словарь Python, в котором будут храниться все клиенты WebSocket, подписанные на канал определенного города.

❽ Приватный метод `__msg_watch_and_broadcast()` отвечает за прослушивание новых сообщений в канале pub/sub. Паттерн обмена сообщениями pub/sub будет подробно описан в подразделе «Паттерн "Издатель — подписчик" (pub/sub)» в главе 11. Если сообщение имеет тип `message` и его свойство `channel` совпадает с UUID города, то сообщение транслируется всем подключенным клиентам WebSocket. Метод `self.pubsub.get_message()` непрерывно прослушивает сообщения, поступающие в канал, так как он выполняется внутри неблокируемого цикла с интервалом 1 секунда. Метод выполняется внутри асинхронной задачи, которую обрабатывает библиотека Python *asyncio* (https://oreil.ly/2HPNj). `asyncio.Lock` не допускает одновременного доступа нескольких асинхронных задач при обработке сообщений из канала Redis pub/sub.

❾ Приватный метод `__broadcast()` отвечает за отправку оповещения всем подключенным клиентам WebSocket.

Реализовав эндпоинт для оповещений, перейдем к реализации консольной программы оповещений. Программа `alert` — это интерфейс командной строки Django, который публикует сообщение-оповещение в Redis pub/sub. В примере 10.10 показано, как опубликовать сообщение в Redis pub/sub.

Пример 10.10. Консольная программа для оповещений в приложении

```
# src/django/app/core/management/commands/app_alert.py

from asgiref.sync import async_to_sync
from django.core.management.base import BaseCommand
from redis import asyncio
from core.models import CityRepository
...
```

[1] Канал Redis используется подписчиками для получения сообщений. См. паттерн обмена сообщениями pub/sub (https://oreil.ly/411ub).

```
class Command(BaseCommand):
    city = CityRepository.get_first()
    ...
    redis = asyncio.Redis(host="redis", port=6379, db=0)  ❶

    def add_arguments(self, parser):  ❷
        parser.add_argument("--city-uuid", default=f"{self.city.uuid}", ...)
        parser.add_argument("--message", default="Ожидается ливень с градом!", ...)

    @async_to_sync
    async def handle(self, *args, **options):
        city_uuid = options["city_uuid"]
        message = options["message"]
        ...
        await self.redis.publish(city_uuid, message)  ❸
        await self.redis.close()

        return f"Оповещение '{message}' отправлено в канал '{city_uuid}'."
```

❶ Атрибут класса `redis` хранит экземпляр `asyncio.Redis`, который подключается к базе данных Redis с индексом `0`. Базу данных с этим же индексом использует сервер в примере 10.9.

❷ Метод `add_arguments()` содержит два аргумента: `--city-uuid` и `--message`. Каждому аргументу присвоено значение по умолчанию. Пример использования флага `--city-uuid`, переданного в CLI `app_event`, был показан в примере 10.6.

❸ Метод `publish()` публикует сообщение-оповещение в канал Redis pub/sub, который идентифицируется через UUID города.

Клиент WebSocket для оповещений о погоде

В примерах 10.9 и 10.10 был представлен весь код бэкенда для API WebSocket. Однако в этом подразделе нет реализации клиентского API — ее можно найти в файле `src/django/app/core/www/static/app/js/alert-ws.js`. Здесь мы не будем описывать клиент, так как его код содержит более 100 строк и его сложно объяснить в печатной версии, не выполняя. Тем не менее мы рекомендуем ознакомиться с упомянутым файлом, поскольку в нем содержится реализация алгоритма экспоненциального откладывания для повторного подключения клиента при разрыве соединения. В этом подразделе мы объясним только алгоритм выполнения программы клиентского API WebSocket.

1. Программа начинается с того, что клиент получает токен доступа (JWT) для пользователя приложения по умолчанию. Получение токена выполняется на эндпоинте `/api/jwt/obtain`.

2. Параметры `access_token` и `city_uuid` передаются эндпоинту `/ws/v1/alert` как параметры запроса. `access_token` используется для аутентификации пользователя, а `city_uuid` — для подписки на канал Redis pub/sub. Токен доступа не передается в заголовке, поскольку API WebSocket не позволяет передавать заголовки[1].
3. После установления соединения WebSocket клиент ожидает сообщений с оповещениями. Если такое сообщение приходит, оно отображается пользователю.
4. Если программа проверяет наличие кода состояния Connection close в функции `onclose()` и обнаруживает, что соединение было неожиданно прервано, то пытается подключиться повторно, вызывая функцию `webSocketReconnect()`.

Безопасность

В этом разделе мы рассмотрим несколько методов повышения безопасности WebSocket. Эти методы также применимы к большинству API на основе HTTP.

Проверка источника

Согласно спецификации WebSocket, одна из мер безопасности — использование клиентского заголовка `Origin`. На его основе сервер решает, принимать или отклонять запрос на установление соединения.

Безопасность доверенного хоста

Суть этого метода в том, что сервер разрешает подключение только тем клиентам, которые обращаются к нему через доверенный хост.

Аутентификация на основе JWT-токенов

Токены, в частности JWT, позволяют выполнять аутентификацию и авторизацию клиентов без сохранения состояний. В WebSocket токены передаются как часть строки запроса во время установления соединения и проверяются сервером.

Шифрование TLS

Шифрование TLS обеспечивает конфиденциальность, целостность и подлинность данных, которыми обмениваются по сети. В WebSocket для безопасного обмена данными используется схема `wss://` вместо `ws://`.

---

[1] Дополнительная информация о реализации клиентов WebSocket содержится в документации MDN для протокола WebSocket (https://oreil.ly/p6ToW).

Проверка источника

Модель безопасности, описанная в спецификации WebSocket, основана на проверке источника запроса и похожа на *правило ограничения домена* (same-origin policy, SOP) (https://oreil.ly/YEqyt). Однако в WebSocket проверку источника выполняет сервер, что позволяет предотвратить несанкционированный доступ к ресурсам из разных источников. В модели проверки источника клиент может общаться с сервером, только если у них совпадает источник (схема, имя хоста и номер порта). Сервер проверяет источник, анализируя заголовок `Origin` в HTTP-запросе на установление соединения от клиента. В нашей реализации проверка источника обеспечивается промежуточным ПО `WebSocketOriginValidatorMiddleware`, как было показано в примере 10.7.

Безопасность доверенного хоста

Еще одна мера безопасности в WebSocket, похожая на модель проверки источника, — это проверка заголовка `host` в HTTP-запросе на установление соединения от клиента. Проверка реализована в промежуточном ПО `WebSocketTrustedHostMiddleware` (см. пример 10.7). Сервер принимает только те клиентские запросы, в которых значение заголовка `host` входит в список доверенных хостов сервера. Использование списка `allowed_hosts`, содержащего допустимые имена хостов или IP-адреса, снижает риск атаки через HTTP-заголовок `host`.

JSON Web Token

Реализация JWT подробно была описана в подразделе «JSON Web Token» в главе 5. Здесь же мы реализуем серверную проверку JWT, полученного при рукопожатии WebSocket.

За аутентификацию запросов в соединении WebSocket отвечает промежуточное ПО `AuthenticationMiddleware` из примера 10.7. Его функциональность расширяется с помощью кастомного класса `StarletteJWTAuthBackend`, который проверяет JWT и предоставляет идентификатор пользователя. Детали реализации механизма аутентификации показаны в примере 10.11.

Пример 10.11. Реализация аутентификации на бэкенде в приложении Starlette

```
# src/django/app/core/helpers/auth/permissions.py

from starlette.authentication import AuthenticationBackend
...

class StarletteJWTAuthBackend(AuthenticationBackend):  ❶

    def __init__(self, authenticate_endpoints):
        self.authenticate_endpoints = authenticate_endpoints
```

```
async def authenticate(self, conn):
    if conn.scope["path"] not in self.authenticate_endpoints:  ❷
        return

    access_token_b64 = conn.query_params.get("access_token")
    if not access_token_b64:  ❸
        raise AuthenticationError("Отсутствует параметр
          запроса access_token.")
    try:  ❹
        jwt_token = base64.b64decode(access_token_b64).decode("ascii")
        payload = jwt.decode(
            jwt_token,
            SECRET_KEY,
            algorithms=["HS256"],
            options={"verify_signature": True}
        )
    except jwt.ExpiredSignatureError:
        raise AuthenticationError("Если срок действия
          токена access_token истек.")
    except (Exception, json.JSONDecodeError):
        msg = "Если токен access_token закодирован в формате,
          отличающемся от base64 или JSON."
        raise AuthenticationError(msg)

    token_expired = datetime.now() >= datetime.fromtimestamp(payload["exp"])
    if token_expired:  ❺
        raise AuthenticationError("Срок действия
          предоставленного access_token истек.")

    query_set = await sync_to_async(User.objects.filter)(pk=payload["id"])
    user = await sync_to_async(query_set.first)()
    if not user:  ❻
        raise AuthenticationError("Пользователь не найден.")

    return AuthCredentials(scopes=["authenticated"]), user  ❼
```

❶ Класс `StarletteJWTAuthBackend` расширяет функциональность класса `AuthenticationBackend` из Starlette.

❷ Сначала сервер проверяет, входит ли эндпоинт в список эндпоинтов, требующих аутентификации. Если эндпоинт отсутствует в списке, то соединение не требует аутентификации и запрос обрабатывается без нее.

❸ Наличие параметра запроса `access_token` проверяется во время процедуры рукопожатия. Проверка выполняется один раз при установлении соединения, поскольку после завершения процедуры рукопожатия соединение WebSocket по умолчанию считается доверенным. Если параметр `access_token` отсутствует, то запрос отклоняется.

❹ Программа предпринимает попытку декодировать полезную нагрузку JWT. Если в процессе декодирования возникает ошибка, запрос отклоняется.

❺ Дата истечения срока действия токена извлекается из полезной нагрузки и сравнивается с текущей датой. Если срок действия токена истек, запрос отклоняется.

❻ Данные о пользователе извлекаются из базы данных на основе идентификатора, указанного в полезной нагрузке JWT. Если пользователь не найден, запрос отклоняется.

❼ Если аутентификация прошла успешно, возвращается объект с учетными данными и объект пользователя.

Шифрование TLS

При работе с протоколом прикладного уровня, таким как WebSocket, необходимо шифрование передаваемых данных. Протокол TLS обеспечивает безопасность передачи данных между клиентами и серверами с помощью шифрования.

В этом подразделе приведен алгоритм защиты приложения Starlette (WebSocket API) с помощью TLS. Кроме того, мы будем использовать самоподписанный SSL-сертификат. Подробности можно найти в скрипте `src/django/app/scripts/startup.sh`. Рассмотрим алгоритм защиты.

1. Внесите изменения в файл `src/django/compose.yaml`, установив переменную окружения `TLS_ENABLE=1`. Она является защитным механизмом, который запускает генерацию самоподписанного SSL-сертификата. Сгенерированный сертификат находится в файле `/etc/wfs/ssl/certs/wfs.crt`, а закрытый ключ — в файле `/etc/wfs/ssl/private/wfs.key`. С помощью флагов `--ssl-certfile` и `--ssl-keyfile` соответственно эти файлы передаются веб-серверу `uvicorn`, который обслуживает приложение Starlette, предоставляющее WebSocket API.
2. Запустите сервис WFS командой **`docker compose up --detach --wait`**.



Самоподписанные сертификаты можно использовать на этапе тестирования, но применять их в продакшен-среде не рекомендуется. В продакшене при шифровании соединения WebSocket (или любого другого) с помощью TLS используйте сертификаты, выпущенные доверенным центром сертификации (CA).

На рис. 10.7 показано зашифрованное приложение EWA. Обратите внимание, что схема в протоколе WebSocket — `wss://`, а не `ws://`.



Рис. 10.7. Зашифрованное приложение EWA в браузере

Документация

Большинство реализаций WebSocket относятся к категории асинхронных API, где обмен данными происходит в неблокирующем режиме. Мы задокументируем WebSocket API сервиса WFS с использованием спецификации AsyncAPI (https://www.asyncapi.com). AsyncAPI поддерживает WebSocket API, хотя основная область ее применения — документирование приложений на основе событийно-ориентированных архитектур (event-driven architectures, EDA).

AsyncAPI — это проект, дополняющий спецификацию OpenAPI (https://www.openapis.org), однако стандарт AsyncAPI чаще всего используется для документирования, тестирования и визуализации асинхронных API. Контракты AsyncAPI передаются пользователям API (разработчикам, тестировщикам, архитекторам и т. д.) в переносимом формате (YAML или JSON).

СПЕЦИФИКАЦИЯ ASYNCAPI

AsyncAPI — это проект с открытым исходным кодом под эгидой Linux Foundation, который упрощает работу с событийно-ориентированными архитектурами (EDA).

Спецификация AsyncAPI не привязана к конкретной технологии — она предоставляет способ описания асинхронного взаимодействия приложений. Кроме того, AsyncAPI предоставляет инструменты для тестирования и мониторинга событийно-ориентированных систем, а также для управления ими.

Спецификация AsyncAPI создавалась на основе спецификации OpenAPI (OAS), ранее известной как Swagger.

В примере 10.12 показана спецификация AsyncAPI для WebSocket API сервиса WFS. Приведенный документ — фрагмент полной спецификации, однако в нем есть вся информация, необходимая для описания WebSocket API.

Пример 10.12. Спецификация AsyncAPI для WebSocket API

```
# src/django/docs/api/websocket/v1-schema.yaml

asyncapi: 3.0.0  ❶
defaultContentType: application/json  ❷
info:
  title: Weather Forecast Service - WebSocket API
  version: v1
operations:  ❸
  receiveAlert:
    action: receive
    summary: Receive message from alert channel
    channel:  ❹
      $ref: '#/channels/alert'
    bindings:  ❺
      ws:
        query:
          type: object
          required:
            - access_token
            - city_uuid
          properties:
            access_token:
              type: string
              description: JWT
            city_uuid:
              type: string
              description: City's UUID
channels:
  alert:
    address: /ws/v1/alert
    messages:  ❻
      genericMessage:
        $ref: '#/components/messages/genericMessage'
components:
  messages:
    genericMessage:
      contentType: application/json
      payload:
        type: object
        required:
          - message
        properties:
          message:
            type: string
```

❶ Поле `asyncapi` содержит версию спецификации AsyncAPI.

❷ Поле `defaultContentType` указывает, что сообщения по умолчанию кодируются в формате JSON.

❸ Поле `operations` описывает операции, которые можно выполнять с эндпоинтом WebSocket API. Операция с идентификатором `receiveAlert` отвечает за получение сообщений из канала оповещений. Получение сообщений описывается действием `action: receive`. В терминах AsyncAPI канал — это развертываемый на стороне сервера компонент API, к которому можно отправлять запросы (не путать с аналогичным понятием в Redis).

❹ Поле `channel` содержит эндпоинты WebSocket, которые получают сообщения. В этом примере мы создаем канал оповещений на эндпоинте `/ws/v1/alert`.

❺ Поле `bindings` описывает привязки, которые используются для установления соединения. В нашем случае ожидаются параметры запроса `access_token` и `city_uuid`.

❻ Поле `messages` описывает схему сообщения, получаемого с эндпоинта оповещений. В нашем случае поле `message` должно быть в полезной нагрузке JSON.



Чтобы перейти в режим предварительного просмотра HTML-документации, сгенерированной на основе спецификации AsyncAPI, скопируйте содержимое примера 10.12 и вставьте его в AsyncAPI Studio (https://studio.asyncapi.com). Помните, что выполнять вставку информации можно только в тех онлайн-сервисах, которым вы доверяете. Всегда предпочтительнее использовать локальные инструменты, как будет показано в примере 10.13.

Если вы хотите повысить видимость API и облегчить его изучение, то можете преобразовать схему AsyncAPI в HTML-документацию и предоставить ее пользователям. В примере 10.13 показано, как создать HTML-документацию на основе спецификации AsyncAPI и разместить ее на веб-сервере Nginx. Страница со сгенерированной документацией показана на рис. 10.8.

Пример 10.13. Генерация документации для WebSocket API с помощью asyncapi/cli

```
# Переход в целевой каталог  ❶
cd src/django/docs/api/websocket

# Преобразование документации AsyncAPI из YAML в HTML  ❷
docker run --rm -it --user=root \
           --volume ${PWD}/output:/app/output \
           --volume ${PWD}/v1-schema.yaml:/app/asyncapi.yaml \
           asyncapi/cli generate fromTemplate /app/asyncapi.yaml \
           @asyncapi/html-template@3.0.0 \
           --force-write --use-new-generator --output /app/output

# Размещение документации в виде HTML-страницы  ❸
docker run --rm --detach --publish 127.0.0.1:8888:80 \
           --volume ${PWD}/output:/usr/share/nginx/html:ro nginx
```

❶ Перейдите в каталог, содержащий схему AsyncAPI.

❷ Сгенерируйте HTML-документацию с использованием Docker-образа `asyncapi/cli` (https://oreil.ly/7V6O4).

❸ Разместите сгенерированную HTML-документацию на веб-сервере Nginx. Документация будет доступна по адресу http://localhost:8888[1].

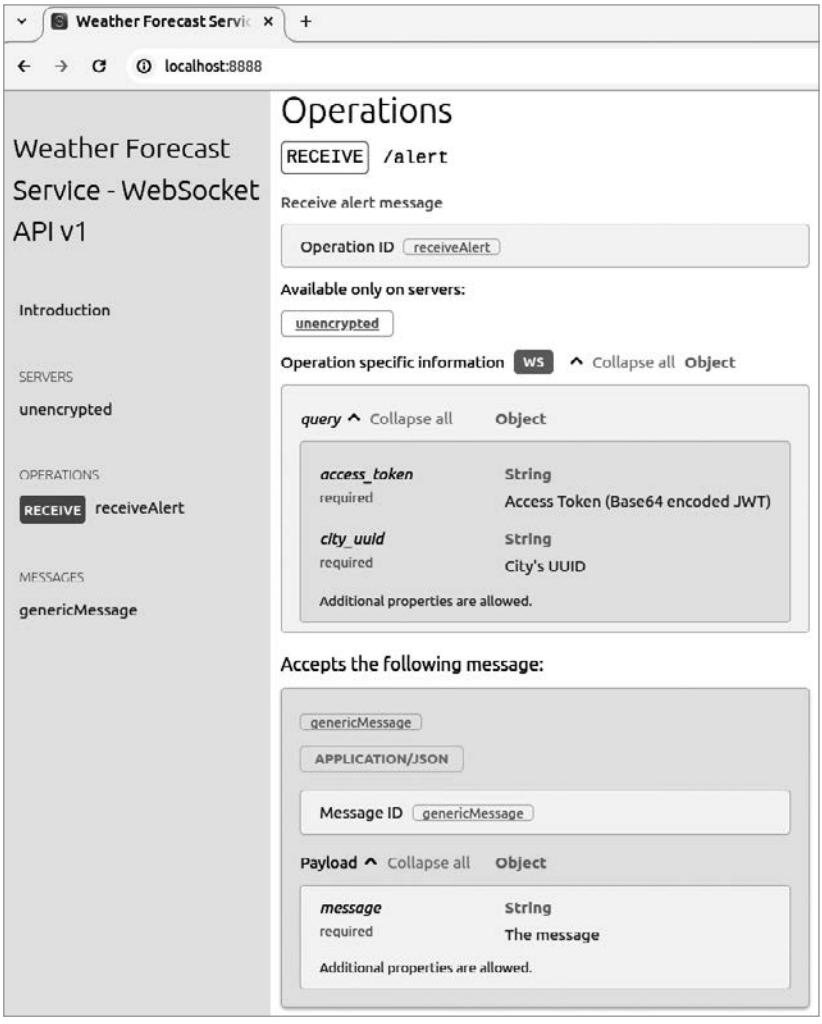


Рис. 10.8. Документация WebSocket, созданная через AsyncAPI

[1] При работе с GitHub Codespaces используется уникальный URL, содержащий имя Codespace вместо адреса localhost. Этот URL привязан к порту 8888, и его можно найти на вкладке PORTS в VS Code.

Итак, мы обсудили процедуру рукопожатия WebSocket, реализацию API WebSocket, методы обеспечения безопасности, а также создание документации. В следующем разделе мы поговорим о преимуществах и недостатках WebSocket.

WebSocket: преимущества и недостатки

Начнем с преимуществ.

Двунаправленная коммуникация

WebSocket работает в режиме полнодуплексной передачи: клиент и сервер могут одновременно отправлять и получать данные. Более того, отправка данных не ограничена требованиями модели «запрос — ответ» — каждая сторона может отправлять и получать данные, не дожидаясь ответа от другой. В протоколе WebSocket для передачи данных используется одно TCP-соединение.

Отзывчивость

Протокол WebSocket позволяет создавать «отзывчивые» приложения, в которых время ожидания ответа минимально. К таким приложениям относятся чаты, интерактивные дашборды, удаленные терминалы, встроенные в браузеры, онлайн-игры или системы оповещений.

Передача больших объемов данных

Протокол WebSocket поддерживает двоичные и текстовые фреймы данных, при этом интерпретация данных выполняется на прикладном уровне протокола. Двоичный формат используется при передаче разных видов мультимедийного контента (аудио, видео или изображений), поскольку этот формат не требует кодирования и декодирования данных. Тем не менее при передаче больших объемов данных через WebSocket увеличивается расход памяти сервера. В качестве альтернативы WebSocket для передачи большого количества данных можно использовать WebRTC (https://webrtc.org).

Минимальное время ожидания ответа

Как мы помним, в HTTP 1.0 для каждого запроса требуется новое TCP-соединение, а в HTTP 1.1 соединение поддерживается в течение ограниченного промежутка времени. В отличие от протоколов HTTP в WebSocket поддерживается одно постоянное TCP-соединение, созданное в результате первого HTTP-запроса. Поскольку для отправки новых данных не нужно создавать новое соединение, то время ожидания ответа сокращается. Кроме того, уменьшается размер сообщений, так как WebSocket не отправляет HTTP-заголовки с каждым сообщением (фреймом данных). Наконец,

WebSocket может работать поверх HTTP/2 и использовать преимущества мультиплексирования[1].

Совместимость с HTTP

WebSocket создан для работы через HTTP-порты `80` и `443`, поэтому трафик может проходить через промежуточные узлы, такие как прокси (обратные прокси, балансировщики нагрузки). Однако он может блокироваться специально настроенными промежуточными устройствами, например корпоративными брандмауэрами с поддержкой проверки пакетов (см. раздел «HTTP/2» главы 4).

Расширение функциональности

Возможности протокола WebSocket могут быть расширены функциями других протоколов (https://oreil.ly/ykp9D), например мультиплексированием HTTP/2. Добавление расширений происходит во время процедуры рукопожатия, когда клиент и сервер могут согласовать поддерживаемые расширения.

Однако у WebSocket есть и несколько недостатков.

Отсутствие поддержки кэширования

WebSocket не поддерживает кэширование. Протокол создан для одновременной двунаправленной передачи данных, поэтому каждое сообщение может отличаться от предыдущего, а состояние приложения быстро меняется. В этих условиях кэширование непрактично.

Сложности управления состояниями

Соединения WebSocket сохраняют предыдущие состояния. Однако отслеживание информации о состоянии клиента может перегрузить сервер. Сервер поддерживает и синхронизирует состояние каждого клиента, поэтому потребление памяти увеличивается и производительность системы снижается. Кроме того, прерывание сеанса клиента с высокой вероятностью приведет к выходу пользователя из приложения. Далее пользователю потребуется повторно войти и заново воссоздать состояние.

Ненадежность соединений

Приложения WebSocket подвержены заблуждениям о распределенных вычислениях (https://oreil.ly/HGOpB), в том числе заблуждению «Сеть надежна». На случай неожиданного разрыва соединения необходимо реализовывать механизм повторного подключения, который обнаруживает

[1] См. RFC 8441, Bootstrapping WebSockets with HTTP/2 («Запуск WebSockets поверх HTTP/2») (https://oreil.ly/6SL0d).

разрыв и пытается переподключить клиент, чтобы избежать ситуации, когда множество клиентов одновременно пытаются получить доступ к ресурсу (о проблеме *thundering herd* мы говорили в подразделе «Повторная отправка запроса» в главе 2).

Проблемы аутентификации

Аутентификация в WebSocket происходит во время процедуры рукопожатия. Когда клиент аутентифицирован и соединение установлено, сервер считает аутентифицированными все последующие сообщения. Если после установления соединения потребуется дополнительная защита, то аутентификацию нужно будет реализовать на прикладном уровне, где будет проверяться каждый фрейм WebSocket. Модель безопасности, используемая в WebSocket, основана на проверке источника запроса. В этой модели сервер проверяет заголовок `Origin` и, в зависимости от источника клиента, разрешает или отклоняет соединение. Однако злоумышленники могут обойти эту модель, изменив заголовок `Origin` во время процедуры рукопожатия.

Ненадежная доставка сообщений

WebSocket наследует надежность от TCP, гарантируя доставку сообщений в нужном порядке и без искажений, пока соединение открыто. Однако на прикладном уровне в WebSocket отсутствуют встроенные механизмы подтверждения получения сообщений или повторной передачи. При разрыве соединения есть риск, что сообщения, которые находились в процессе передачи, не будут доставлены. Тогда приложению придется самостоятельно обрабатывать повторные подключения и повторную отправку сообщений.

Ограниченная масштабируемость

Приложения на WebSocket могут потреблять много ресурсов, из-за чего возникает проблема масштабирования (система должна сохранять работоспособность в условиях растущего объема работы). WebSocket сохраняет предыдущие состояния, поэтому и сервер, и клиент обязаны хранить данные о состоянии соединения и постоянно их актуализировать. Сложность масштабирования при работе с таким соединением заключается в том, чтобы не нанести ущерб уже установленным подключениям. Возможное решение — установить ограничение на количество соединений, которое обрабатывает каждый сервер, и убедиться, что есть достаточное количество серверов для обработки всех подключений. Для распределения нагрузки между серверами можно использовать балансировщик нагрузки.

Отсутствие обратного давления на прикладном уровне

Для обработки обратного давления на прикладном уровне можно использовать такие методы, как агрегирование данных (объединение нескольких сообщений), сжатие (уменьшение размера данных за счет кодирования),

дедупликация (устранение дублирующихся данных) и обрезка (удаление ненужных данных). Благодаря этим методам объем передаваемых данных сокращается[1]. Этот процесс обозначается термином *«уплотнение»* (compaction).

Когда использовать WebSocket

WebSocket подойдет, если для работы вашего приложения требуется двунаправленная коммуникация с минимальным временем ожидания ответа. Помимо систем оповещения, WebSocket часто применяется для реализации систем уведомлений, онлайн-чатов, многопользовательских игр, визуализации данных на дашбордах и интерактивных сервисов для совместной работы. Например, компания Miro (https://miro.com), популярная среди миллионов пользователей, использует WebSocket для создания интерактивных дашбордов.

Упражнения

1. Расширьте функциональность нашего сервиса прогноза погоды, добавив функцию чата. В чате пользователь должен иметь возможность отправлять и получать сообщения. В этом упражнении вам нужно реализовать чат на стороне клиента и сервера. Клиент чата загружается через браузер с HTTP-эндпоинта `http://localhost:8000/websocket/chat`, а сервер WebSocket расположен на эндпоинте `ws://localhost:8001/ws/v1/chat`. Для подключения к последнему эндпоинту нужно передать токен аутентификации в виде параметра запроса `access_token`.

   Чтобы получить токен, отправьте `POST`-запрос в формате JSON на эндпоинт `http://localhost:8000/api/jwt/obtain`. Поля `username` и `password` в полезной нагрузке запроса должны иметь значение `admin`. При работе с клиентом WebSocket подумайте о реализации механизма повторного подключения на случай разрыва соединения на стороне сервера. При реализации сервера подумайте о механизме рассылки сообщений всем подключенным клиентам.

   Выполнение этого упражнения потребует усилий, так как вам нужно реализовать обе стороны: клиент (фронтенд) и сервер (бэкенд). При работе с клиентом сосредоточьтесь на следующих функциях: `onopen()`, `onmessage()`, `onclose()` и `onerror()`. Ответ к этому упражнению находится в репозитории кода книги (см. файл `src/django/docs/solutions/websocket-exercise.md`).

---

[1] WebSocketStream (https://oreil.ly/IpUO7) решает проблему обратного давления в WebSocket с помощью потоков (https://oreil.ly/ytstS), однако на данный момент технология является экспериментальной и имеет ограниченную поддержку в браузерах.

2. В подразделе «Шифрование TLS» выше в этой главе мы показали, как зашифровать приложение Starlette, предоставив сертификат TLS и закрытый ключ веб-серверу ASGI `uvicorn`. В этом упражнении вы проверите работу TLS, используя трафик HTTP и WebSocket для взаимодействия с сайтом WFS.

   Чтобы включить или отключить TLS для WFS, в файле `src/django/compose.yaml` измените значение переменной окружения: `TLS_ENABLE=1` (для зашифрованного трафика) или `TLS_ENABLE=0` (для незашифрованного трафика). Далее запустите (или перезапустите) проект, используя команду **`docker compose up --detach --wait`**.

   Если вы открываете https://localhost:8000 в локальной среде разработки (не в GitHub Codespaces), то браузер покажет предупреждение «Ваше подключение не защищено» в Chrome или «Потенциальная угроза безопасности» в Firefox. Чтобы взаимодействовать с веб-страницей WFS, добавьте самоподписанный SSL-сертификат приложения в список исключений браузера. Для этого посетите https://localhost:8000 и https://localhost:8001. Далее, в зависимости от используемого браузера, нажмите Перейти на сайт localhost (небезопасно) в Chrome или Принять риск и продолжить в Firefox. Затем перейдите по адресу https://localhost:8000/websocket/echo, чтобы увидеть сообщение `Hello WebSocket!`.

Резюме

API WebSocket обеспечивает одновременную двунаправленную коммуникацию между клиентом и сервером. В этой главе мы рассмотрели протокол WebSocket, а также клиентские и серверные API WebSocket.

Мы реализовали приложение WebSocket ECHO и с помощью Wireshark проанализировали процедуру рукопожатия на этапе открытия (и закрытия) соединения WebSocket. Используя знания о протоколе WebSocket и API WebSocket, мы также реализовали систему оповещений. Для этого в WFS мы реализовали паттерн pub/sub, задействуя Redis и серверный API WebSocket на основе Starlette. Далее мы подключились к этому API с помощью клиента WebSocket, написанного на JavaScript. Кроме того, мы защитили WFS с помощью промежуточного ПО и зашифровали трафик WebSocket с помощью TLS. В разделе, посвященном документации, мы поговорили о спецификации AsyncAPI. Наконец, в заключительном разделе рассмотрели преимущества и недостатки API WebSocket.

ГЛАВА 11

API на основе обмена сообщениями

В этой главе речь пойдет об API для брокерских систем обмена сообщениями, известных как *брокерские API*. Наша цель — познакомить вас с распространенными паттернами обмена сообщениями и помочь вам развить навыки, необходимые для реализации и документирования этих паттернов. В частности, мы будем работать с брокером сообщений RabbitMQ.

За последние десятилетия было разработано несколько архитектур приложений, которые предполагают обмен сообщениями. В начале 2000-х получила распространение сервисно-ориентированная архитектура (Service-Oriented Architecture, SOA). Она позволяла обмениваться сообщениями через сервисную шину предприятия (Enterprise Service Bus, ESB) — промежуточное ПО, обеспечивающее взаимодействие между программными приложениями. В середине 2000-х стала набирать популярность событийно-ориентированная архитектура (Event-Driven Architecture, EDA). В ней событие в одной системе инициировало сообщение к другой системе, чтобы та могла отреагировать на событие. Для передачи данных между приложениями и в ESB, и в EDA использовалось промежуточное программное обеспечение для обмена сообщениями (message-oriented middleware, MOM). Это же ПО использовалось в микросервисных архитектурах, оказавшихся на пике популярности после 2010 года. К таким микросервисным архитектурам относятся, например, ActiveMQ (https://activemq.apache.org), Amazon SQS (https://oreil.ly/DeI6R), Azure Service Bus (https://oreil.ly/U0743), Google Cloud Pub/Sub (https://oreil.ly/QJ36Q), Kafka (https://oreil.ly/2knCH), RabbitMQ (https://www.rabbitmq.com) или Redis (https://redis.io).



Термин «промежуточное ПО для обмена сообщениями» (message-oriented middleware, MOM) появился в конце XX века[1]. Хотя сейчас его используют реже, он отражает идею применения отдельного компонента-посредника в обмене сообщениями (не обязательно брокера).

[1] Внутреннее устройство MOM подробно описано в книге Enterprise Service Bus (O'Reilly, 2004) Дэвида А. Чаппелла (David A. Chappell).

Что такое обмен сообщениями

Обмен сообщениями — это метод коммуникации, который позволяет приложениям обмениваться данными в форме сообщений. В подразделе «Сообщение» в главе 1 мы определили сообщение как дискретную единицу данных (самодостаточную запись), которая движется от отправителя к одному или нескольким получателям.

В архитектуре «клиент — сервер» сообщение передается напрямую между клиентом и сервером, причем коммуникацию инициирует клиент, отправляя первое сообщение (запрос), а сервер отвечает на него встречным сообщением (ответом).

MOM — это программное или аппаратное решение, которое упрощает отправку и получение сообщений между приложениями. В отличие от архитектуры «клиент — сервер», в MOM сообщения никогда не передаются напрямую между приложениями. MOM выступает в роли посредника, гарантируя доставку сообщений от производителя (*издателя*, *отправителя* или *эмитента*) к потребителю (*подписчику* или *получателю*). Сообщения обычно отправляются и принимаются асинхронно (см. подраздел «Синхронная и асинхронная коммуникация» в главе 1).

На рис. 11.1 показана схема, иллюстрирующая концепцию MOM. Система содержит шесть приложений (отправителей и получателей), которые используют API MOM (клиентскую библиотеку) для отправки или получения сообщений. Сообщения, которыми обмениваются приложения, проходят через MOM. Каждое приложение отправляет или получает одно сообщение, которое движется от отправителя к получателю. Номера на конвертах необходимы для отслеживания сообщений.

В брокерском MOM сообщения не передаются напрямую между приложениями, а отправляются через посредник (брокер). Основная задача брокера — маршрутизация сообщений. Кроме того, он выполняет дополнительные задачи, например аутентификацию, мониторинг и ведение логов. Приложение, которое подключается к брокерскому MOM, называется *клиентом* и может выступать в роли производителя или потребителя сообщений. *Брокер* — это сервер, который ожидает подключения клиентов. Пример брокера из реального мира — почтовое отделение, которое принимает посылки от отправителей и организовывает их доставку получателям.

Есть несколько технологий для реализации доставки сообщений через брокер. В этой главе мы уделим внимание RabbitMQ.



Рис. 11.1. Архитектура MOM

КОНФИГУРАЦИИ БРОКЕРА RABBITMQ

Помимо конфигурации с одним брокером, RabbitMQ поддерживает кластеризацию и федерацию брокеров.

При *кластеризации* несколько узлов работают в качестве единого брокера в локальной сети, объединяя ресурсы, чтобы повысить доступность и масштабируемость системы. Для такой конфигурации необходимы соединения с наименьшим временем ожидания, поскольку метаданные и состояние очередей являются общими для всех узлов.

Федерация брокеров, в свою очередь, позволяет пересылать сообщения между независимыми брокерами, которые находятся в разных сетях, регионах или даже принадлежат разным облачным провайдерам. Такой подход позволяет выполнять гибридные облачные развертывания, поддерживать связь между регионами и объединять отдельные административные домены.

На рис. 11.2 показан стандартный маршрут движения сообщений в RabbitMQ.

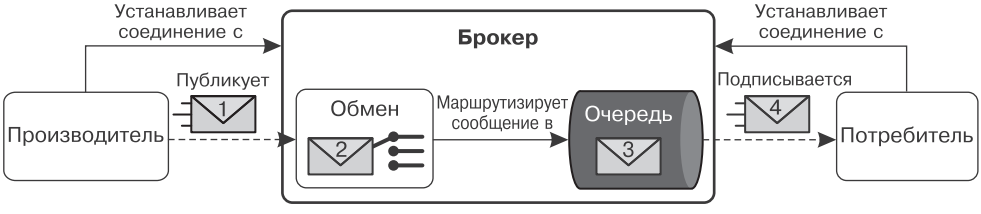


Рис. 11.2. Маршрут движения сообщений в RabbitMQ

Маршрут состоит из четырех этапов:

1) производитель отправляет сообщение на обменник;
2) сообщение поступает на обменник и маршрутизируется в очередь на основе типа и конфигурации обменника;
3) очередь, которая играет роль буфера, хранит сообщение до востребования потребителем;
4) потребитель забирает сообщение из очереди.



Обратите внимание: брокерами сообщений могут быть не только очереди. Например, в Apache Kafka используются логи с партициями, в MQTT — топики, а в Redis pub/sub — каналы. Эти механизмы отличаются от традиционных очередей, но выполняют похожую задачу.

Прежде чем переходить к паттернам обмена сообщениями, определим базовые термины RabbitMQ.

Производитель

Клиент, который отправляет сообщения брокеру.

Потребитель

Клиент, который получает сообщения от брокера.

Канал

Специальный путь для установки соединения на прикладном уровне протокола AMQP (Advanced Message Queuing Protocol) в RabbitMQ. Несколько каналов используют одно TCP-соединение.

Очередь

Буферное хранилище, в котором хранятся сообщения для потребителей, организованные по методу FIFO (first in first out, «первый пришел, первый ушел»). Очередь можно сравнить с почтовым ящиком: с одной стороны почтальон (производитель) опускает письмо, а с другой — получатель (потребитель) забирает его.

Обменник

Маршрутизатор, который принимает сообщения от производителя и направляет их в одну или несколько очередей. Обменник сопоставим с ретранслятором или диспетчером. В RabbitMQ есть четыре типа обменников: прямой (`direct`), широковещательный (`fanout`), основанный на топике (`topic`) и основанный на заголовках (`headers`).

Ключ маршрутизации

Свойство сообщения, которое обменник использует для маршрутизации сообщения в одну или несколько очередей. Ключ маршрутизации можно сопоставить с адресом на письме.

Топик

Тип обменника, который маршрутизирует сообщения в очереди на основе шаблонов — специальных символов (* или #). Эти символы определяются в привязках между обменником и одной или несколькими очередями. Ключ маршрутизации сообщения — это строка, которую обменник сопоставляет с шаблонами.

Привязка

Связь между обменником и очередью или между двумя обменниками. Привязка используется для маршрутизации сообщений в очередь. Сообщение движется от обменника к очереди.

Идентификатор корреляции (Correlation ID, CID)

Уникальный идентификатор, который передается вместе с сообщением. Он часто используется для сопоставления запроса с ответом.

Подтверждение

Сигнал, который отправляется от потребителя к брокеру или от брокера к производителю[1]. Потребитель должен отправить брокеру явное подтверждение, чтобы показать, что сообщение было успешно получено или обработано. Сигнал о том, что сообщение было получено, может отправиться и от брокера к производителю, но механизм отправки такого сигнала зависит от брокера сообщений и его конфигурации. Например, в RabbitMQ производителю нужно включить функцию «подтверждение издателя» (publisher confirms).

Что такое очередь

В компьютерных науках *очередь* (https://oreil.ly/5jC3q) — это коллекция элементов данных, которые хранятся в порядке FIFO, где первый добавленный элемент должен быть удален первым. На рис. 11.3 показана очередь. Данные попадают в конец очереди (операция добавления в очередь, enqueue) и удаляются, когда оказываются в ее начале (операция извлечения из очереди, dequeue).

---

[1] Существует также автоматическое подтверждение, которое обрабатывается внутри брокера; см. врезку «Подтверждение получения сообщений в RabbitMQ» далее в этой главе.

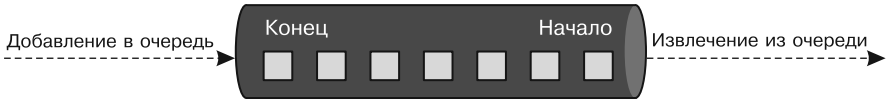


Рис. 11.3. Очередь

В контексте обмена сообщениями *очередь* — это хранилище, в котором производитель публикует сообщения, а потребители их обрабатывают. Брокер RabbitMQ хранит очереди и управляет ими. Пока брокер не удалит сообщение, оно остается в очереди. Сообщение удаляется лишь после того, как брокер получил подтверждение, и чем быстрее это произойдет — тем лучше. У специалистов по брокерам сообщений даже есть поговорка: «Лучшая очередь — пустая очередь» (A happy queue is an empty queue).

После того как сообщение попало в очередь, его нельзя изменить. Можно только создать новое, то есть опубликовать его заново. Производители не отслеживают ответы потребителей, работая по принципу «отправил и забыл» (fire and forget). Отправив сообщение в очередь, производитель не ждет ответа от потребителя. Исключением из правила являются случаи использования паттерна «запрос — ответ» (см. подраздел «Паттерн “Запрос — ответ”» далее в этой главе).



Подробное описание всех паттернов обмена сообщениями, которые упомянуты в этой главе, находится в репозитории книги, в каталоге src/rabbitmq.

Очередь помогает ослабить зависимость между производителем и потребителем, чтобы им не приходилось постоянно отслеживать наличие друг друга. Безусловно, речь не идет о том, что производитель и потребитель в брокерах сообщений совсем не связаны друг с другом[1]. Например, они могут быть связаны через протокол передачи сообщений (AMQP или MQTT) или формат сообщений. Если полезная нагрузка в сообщении производителя имеет формат JSON, то потребитель должен уметь его декодировать. Более того, если сообщение изменится (например, поле будет переименовано или удалено), то клиенты также должны уметь обрабатывать это изменение.



Связанность — это зависимость между компонентами системы, при которой изменение одного компонента влияет на другой.

[1] Лучше понять идею связанности поможет книга Влада Хононова: *Khononov V.* Balancing Coupling in Software Design. — Addison-Wesley, 2023. Подробнее о связанности в системах на основе обмена сообщениями см. запись в блоге Грегора Хопе (Gregor Hohpe) The Many Facets of Coupling («Многогранность связанности») (https://oreil.ly/X2doC).

Системы с очередями обычно более надежны и устойчивы к сбоям. Например, если потребитель недоступен, то сообщение остается в очереди до тех пор, пока он не вернется в сеть и не сможет его получить. Более того, в случае внезапного наплыва сообщений систему можно масштабировать, добавив больше потребителей.



В RabbitMQ очереди и обменники могут быть постоянными (durable) или временными (transient). Постоянные очереди и обменники сохраняются после перезапуска брокера, а временные — нет. Временные очереди и обменники повторно создаются вручную, когда брокер возвращается в сеть.

Постоянная очередь в RabbitMQ не делает постоянными сообщения. Гарантируется только непрерывность существования самой очереди. После перезапуска брокера будет повторно создана очередь, но не сообщения, которые в ней содержатся. Чтобы сделать сообщение постоянным, необходимо установить свойство delivery_mode, как будет показано в примере 11.4.

Паттерны обмена сообщениями

В этом разделе описаны распространенные паттерны обмена сообщениями, которые используются в брокерских системах. Примеры составлены на основе руководств к RabbitMQ (https://oreil.ly/rI0Ip).

Паттерн «Рабочая очередь» (Work Queue)

Рабочая очередь (или *«Конкурирующие потребители»*, Competing Consumers) — это паттерн обмена сообщениями, который позволяет системе обрабатывать несколько сообщений параллельно. В нем сообщения доставляются в одну очередь, а брокер передает их доступному потребителю. В среднем каждый потребитель получает одинаковое количество сообщений. Поскольку очередь одна, то потребители конкурируют за каждое сообщение. Наличие очереди гарантирует, что каждое сообщение доставляется только одному потребителю, поэтому потребителям не нужно координировать действия.

На рис. 11.4 показан принцип работы этого паттерна. В очереди есть три сообщения, и каждое доставляется только одному потребителю.

Сообщения, показанные на рис. 11.4, распределяются между потребителями по кругу (принцип round-robin), и каждый потребитель получает по одному сообщению. Однако такое поведение характерно именно для RabbitMQ — другие системы могут распределять сообщения иначе.

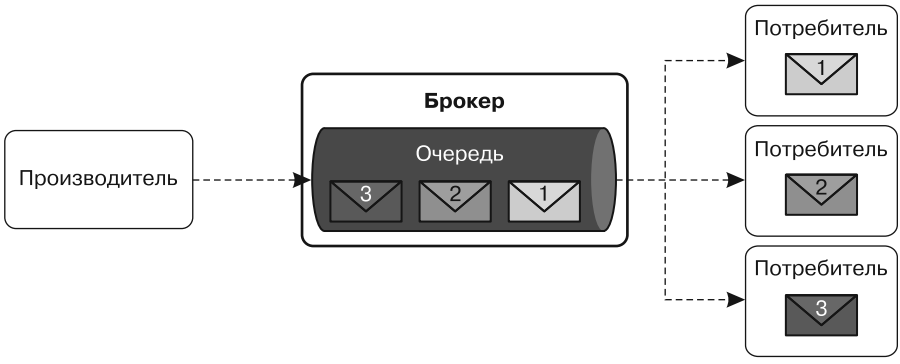


Рис. 11.4. Паттерн «Рабочая очередь»: каждое сообщение доставляется только одному потребителю



Реализация паттерна находится в репозитории кода книги в каталоге src/rabbitmq/2.WorkQueue.

Паттерн «Издатель — подписчик» (pub/sub)

«Издатель — подписчик» (pub/sub) — это паттерн обмена сообщениями, который работает по принципу «один ко многим», рассылая каждое сообщение нескольким потребителям. Его часто применяют при отправке оповещений или уведомлений, когда каждое сообщение должны получить несколько потребителей. Когда в очередь pub/sub добавляют нового потребителя, он получает все сообщения, которые публикуются в этой очереди с момента его подключения. На рис. 11.5 показан принцип работы паттерна «Издатель — подписчик». Обратите внимание: на схеме обменник отправляет одно и то же сообщение в три отдельные очереди.

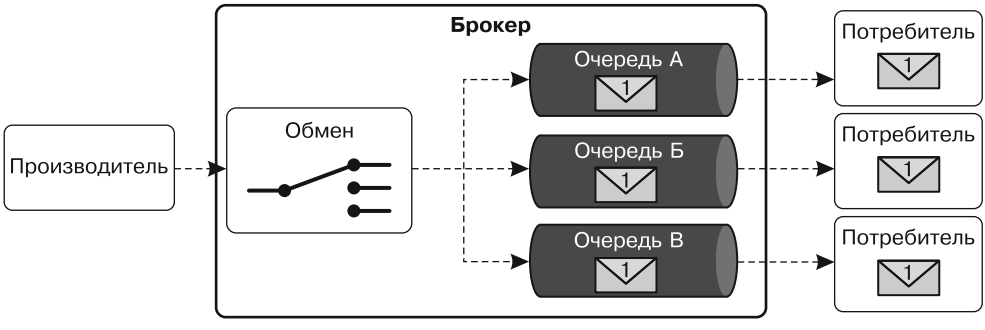


Рис. 11.5. Паттерн «Издатель — подписчик»: каждое сообщение доставляется всем потребителям



Реализация паттерна находится в репозитории кода книги в каталоге src/rabbitmq/3.PubSub.

Паттерн «Маршрутизация»

Этот паттерн представляет собой вариацию «Издателя — подписчика». При маршрутизации потребители получают сообщения выборочно на основе ключа маршрутизации. На рис. 11.6 показаны два сообщения, отправленные производителем: они проходят через обменник (маршрутизатор), который перенаправляет их в соответствующую очередь. Первое сообщение получают два потребителя, а второе — один.

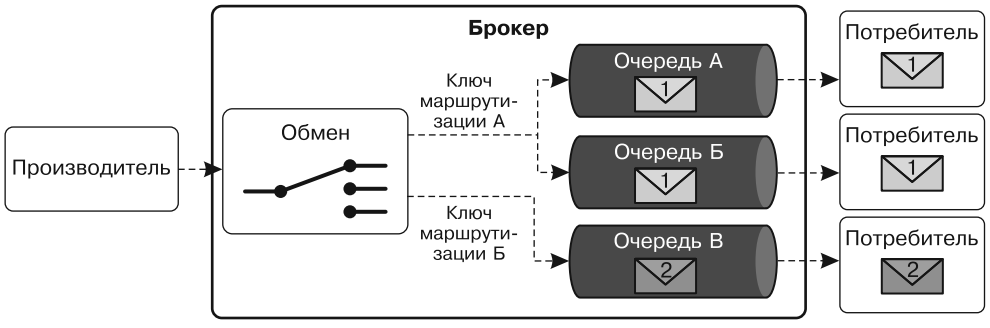


Рис. 11.6. Паттерн «Маршрутизация»: доставка сообщения на основе ключа маршрутизации



Реализация паттерна находится в репозитории кода книги в каталоге src/rabbitmq/4.Routing.

Паттерн «Топики»

Паттерн «Маршрутизация» имеет ограничение: в нем сообщения могут быть доставлены только по конкретному ключу маршрутизации. Паттерн *«Топики»* устраняет это ограничение, позволяя потребителям подписываться на сообщения на основе шаблона ключа маршрутизации. На рис. 11.7 показаны два сообщения, отправленные производителем: они проходят через обменник (маршрутизатор), который перенаправляет их в соответствующую очередь. Первое сообщение получают два потребителя, а второе — один. Обратите внимание на разницу между паттернами «Маршрутизация» и «Топики». При маршрутизации ключ — это одно слово, а в топиках ключ маршрутизации содержит шаблон, включающий в себя точку и специальный знак (*).



Паттерны «Маршрутизация» и «Топики» — это частные случаи паттерна маршрутизации на основе содержания сообщения.

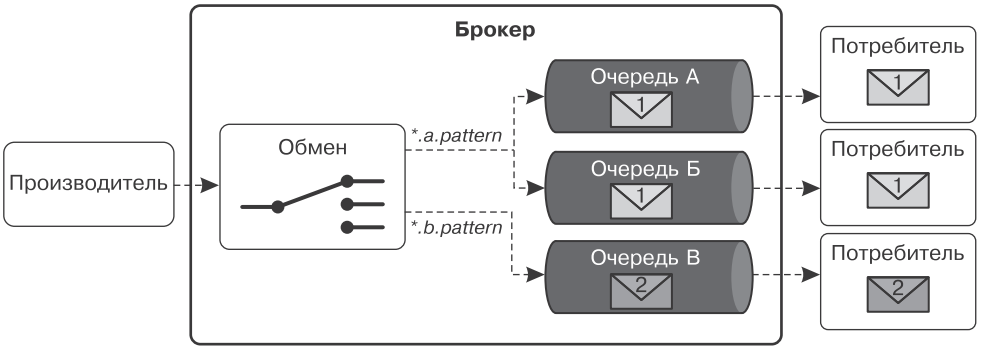


Рис. 11.7. Паттерн «Топики»: доставка сообщения на основе шаблона в ключе маршрутизации



Реализация паттерна находится в репозитории кода книги в каталоге src/rabbitmq/4.Topics.

Паттерн «Запрос — ответ»

Этот паттерн основан на архитектуре «клиент — сервер». На рис. 11.8 показано, как реализовать его с помощью RabbitMQ. Обратите внимание, что оба клиентских приложения выступают одновременно в качестве производителя и потребителя сообщений. В паттерне создаются две очереди: одна для запросов, другая для ответов.

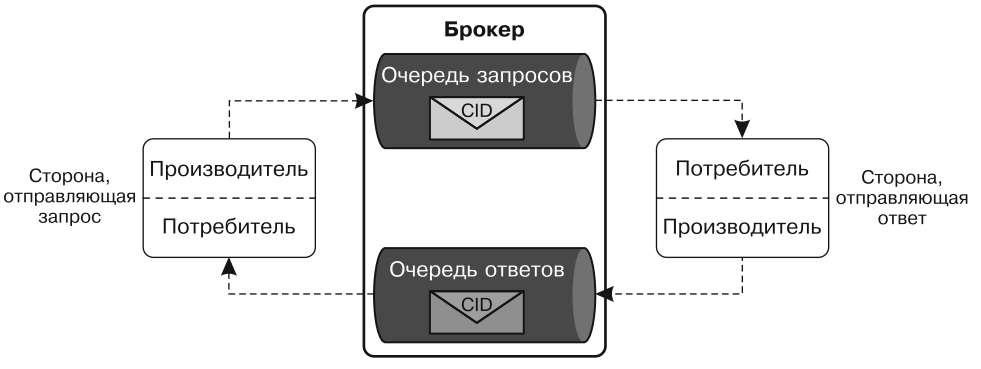


Рис. 11.8. Паттерн «Запрос — ответ»

В нашей конфигурации сторона, отправляющая ответ, создает очередь запросов, а сторона, отправляющая запросы, — очередь ответов. Сообщение-запрос, которое отправляется в очередь запросов, содержит CID (индикатор корреляции), который используется для сопоставления запроса с ответом.

Сторона, отправившая запрос, ожидает ответ, прослушивая очередь ответов. Сторона, отправляющая ответы, получает запрос и отправляет ответ вместе с CID в очередь ответов. Далее сторона, отправляющая запросы, получает ответ. Если CID запроса и ответа совпадают, то сторона, отправляющая запросы, понимает, что получила ожидаемый ответ. Если же CID не совпадают, то сторона, отправляющая запросы, может проигнорировать ответ, зафиксировать несоответствие в логе как ошибку или отправить ответ в очередь недоставленных сообщений (Dead Letter Queue, DLQ) (https://oreil.ly/J3GNH) для дальнейшего анализа, чтобы не блокировать основную очередь ответов.



Реализация паттерна находится в репозитории книги в каталоге src/rabbitmq/6.RequestResponse.

Реализация

В этом разделе мы рассмотрим реализацию паттерна «Рабочая очередь». Все паттерны, описанные в книге, имеют схожую структуру каталогов. Изучив этот раздел, вы сможете самостоятельно реализовать другие паттерны.



Реализация паттерна «Рабочая очередь» и других паттернов находится в репозитории книги в каталоге src/rabbitmq.

Паттерн «Рабочая очередь»

Для знакомства с паттерном будем использовать пример с броском кубика. Общая схема представлена на рис. 11.9.

Производитель имитирует бросок кубика. Результатом одного броска становится случайное число от 1 до 6, и каждый такой результат создает новое сообщение. Оно отправляется в очередь `die`. Потребитель извлекает из этой очереди по одному сообщению за раз.

Обратите внимание: компоненты на схеме содержат IP-адреса. Эти адреса статичны, чтобы было легче их идентифицировать и экспериментировать с решением.

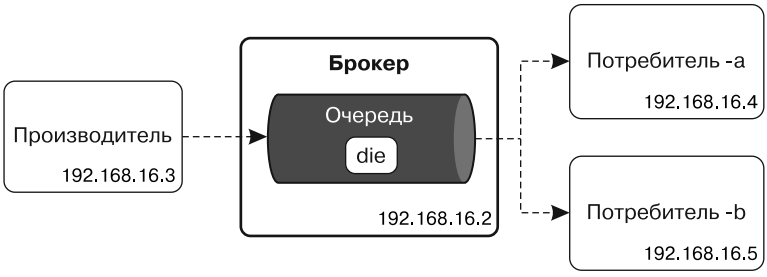


Рис. 11.9. Общая схема функционирования паттерна «Рабочая очередь»

Начните с выполнения команд из примера 11.1. Получившийся вывод должен быть похож на тот, который представлен на рис. 11.10.

```
$ for i in {1..6}; do docker compose exec producer python /usr/src/producer.py; done
[x] Sent '{'data': {'message': 'Rolling 1', 'count': 1}, 'version': '1.0.0'}' message to 'die' queue.
[x] Sent '{'data': {'message': 'Rolling 3', 'count': 3}, 'version': '1.0.0'}' message to 'die' queue.
[x] Sent '{'data': {'message': 'Rolling 5', 'count': 5}, 'version': '1.0.0'}' message to 'die' queue.
[x] Sent '{'data': {'message': 'Rolling 3', 'count': 3}, 'version': '1.0.0'}' message to 'die' queue.
[x] Sent '{'data': {'message': 'Rolling 4', 'count': 4}, 'version': '1.0.0'}' message to 'die' queue.
[x] Sent '{'data': {'message': 'Rolling 2', 'count': 2}, 'version': '1.0.0'}' message to 'die' queue.
```

```
$ docker compose exec consumer-a python /usr/src/consumer.py
[*] Waiting for messages from 'die' queue.
[x] Received 'Rolling 1' message.
[x] Received 'Rolling 5' message.
[x] Received 'Rolling 4' message.
```

```
$ docker compose exec consumer-b python /usr/src/consumer.py
[*] Waiting for messages from 'die' queue.
[x] Received 'Rolling 3' message.
[x] Received 'Rolling 3' message.
[x] Received 'Rolling 2' message.
```

Рис. 11.10. Ожидаемый вывод при реализации примера с рабочей очередью

Пример 11.1. Последовательность команд для примера с рабочей очередью

```
# Настройка лабораторной среды ❶
cd src/rabbitmq/2.WorkQueue
docker compose up --detach --wait
tmux new-session \; split-window -v \; split-window -h \; select-pane -U \;

# Нижняя левая панель ❷
docker compose exec consumer-a python /usr/src/consumer.py

# Нижняя правая панель ❷
docker compose exec consumer-b python /usr/src/consumer.py

# Верхняя панель ❸
for i in {1..6}; do docker compose exec producer python /usr/src/producer.py; done

# Верхняя панель ❹
docker compose down --volumes && docker network rm -f rabbitmq
```

❶ Выполните настройку лабораторной среды. Перейдите в каталог лаборатории, запустите контейнеры Docker и создайте новую сессию `tmux`, которая разделит окно терминала на три панели.

❷ Выполните команду, заставляющую двух потребителей прослушивать сообщения.

❸ Выполните команду, которая заставит производитель отправить шесть сообщений в цикле `for`. Очередь распределяет сообщения между потребителями по кругу (round-robin), и каждый потребитель получает по одному сообщению.

❹ Завершите выполнение примера, остановив и удалив контейнеры, тома и сеть.

В примерах 11.2 и 11.3 показана реализация производителя и потребителя. Порядок объяснений соответствует порядку выполнения программы: сначала мы создаем экземпляры классов производителя и потребителя, далее подключаемся к брокеру и создаем канал и очередь. После этого начинается процесс отправки или получения сообщений.

Пример 11.2. Реализация производителя сообщений

```
# src/rabbitmq/2.WorkQueue/src/producer.py

...
class RabbitMQProducer:
  ...
  queue = os.getenv("RABBITMQ_QUEUE", "die")
  ...

def __init__(self):  ❸
  self.__connect()  ❹
  self.__declare_channel()  ❺
  self.__declare_queue()  ❻

def __connect(self):  ❹
      ...
      self.connection = BlockingConnection(params)
      ...

def __declare_channel(self):  ❺
  self.channel = self.connection.channel()

def __declare_queue(self, durable=True):  ❻
  self.channel.queue_declare(queue=self.queue, durable=durable)
```

```
def publish(self, msg: dict):  ❼
  ...
  self.channel.basic_publish(
    exchange="",
      routing_key=self.queue,
      body=to_json(msg),
      ...
    )

if __name__ == "__main__":  ❶
  ...
  number = random.randrange(1, 6)
  msg = {
    "data": {
      "message": f"Rolling {number}",
      "outcome": number
    },
    "version": "1.0.0"
  }

  producer = RabbitMQProducer()  ❷
  producer.publish(msg)  ❼
```

Пример 11.3. Реализация потребителя сообщений

```
# src/rabbitmq/2.WorkQueue/src/consumer.py

...
class RabbitMQConsumer:
  ...
  queue = os.getenv("RABBITMQ_QUEUE", "die")
  ...

  def __init__(self):  ❸
    self.__connect()  ❹
    self.__declare_channel()  ❺
    self.__declare_queue()  ❻

  def __connect(self):  ❹
        ...
        self.connection = BlockingConnection(params)
        ...

  def __declare_channel(self):  ❺
    self.channel = self.connection.channel()

  def __declare_queue(self, durable=True):  ❻
    self.channel.queue_declare(queue=self.queue, durable=durable)
```

```
  def __callback(self, channel, method, properties, msg):  ❽
    msg = json_to_dict(msg, properties.content_encoding)["data"]["message"]
    print(f"[x] Received '{msg}' message.")
    channel.basic_ack(delivery_tag=method.delivery_tag)

  def consume(self):  ❼
    self.channel.basic_qos(prefetch_count=1)
    self.channel.basic_consume(
      queue=self.queue,
      auto_ack=False,
      on_message_callback=self.__callback
    )
    self.channel.start_consuming()

if __name__ == "__main__":  ❶
  ...
  consumer = RabbitMQConsumer()  ❷
  consumer.consume()  ❼
```

❶ `__main__` — это точка входа в программу, которая позволяет выполнять код только в том случае, если скрипт запущен из командной строки (недостаточно просто импортировать его как модуль). Здесь начинается выполнение программы.

❷ Экземпляры `producer` и `consumer` создаются из классов `RabbitMQProducer` и `RabbitMQConsumer` соответственно.

❸ Программа переходит к методу `__init__()` (конструктору класса), который отвечает за подключение к брокеру RabbitMQ.

❹ Метод `__connect()` создает два TCP-соединения. Одно соединяет производитель с брокером, а другое — потребитель с брокером.

❺ Метод `__declare_channel()` создает канал сообщений (`channel`) — виртуальное соединение внутри TCP-соединения, через которое проходят все публикуемые или потребляемые сообщения AMQP.

❻ Метод `__declare_queue()` создает очередь. Значение `True` для параметра `durable` гарантирует, что очередь сохранится даже после перезапуска брокера RabbitMQ.

Обратите внимание, что и `producer.py`, и `consumer.py` выполняют метод `__declare_queue()`. Он гарантирует, что очередь будет существовать до того, как потребители начнут получать из нее сообщения. Это позволяет избежать ситуаций, когда потребитель не может прочитать сообщения из несуществующей очереди.

❼ Метод `publish()` помещает сообщение в очередь, а метод `consume()` извлекает сообщение из очереди.

❽ У потребителя есть дополнительный метод `__callback()`, который выполняется при получении сообщения. Он позволяет обработать сообщение и отправить явное подтверждение получения (`basic_ack(delivery_tag=delivery_tag)`) брокеру.

ПОДТВЕРЖДЕНИЕ ПОЛУЧЕНИЯ СООБЩЕНИЙ В RABBITMQ

В RabbitMQ есть разные типы подтверждений. Подтверждение может отправляться от потребителя к брокеру или от брокера к производителю. Есть и автоматическое подтверждение, которое работает локально на брокере и не отправляется клиентам.

Потребитель может подтверждать получение сообщения RabbitMQ двумя способами: *явно* (в любой момент) или *автоматически* — тогда брокер по умолчанию считает каждое сообщение доставленным успешно.

При явном подтверждении потребитель может отправить подтверждение или в момент получения сообщения, или после завершения операции, которую инициировало это сообщение. Отправка подтверждения от потребителя к брокеру дает гарантию, что брокер не удалит сообщения из очереди, пока не получит подтверждение от потребителя. Обычно подтверждение потребителя указывает на успешное завершение операции и поэтому является последним шагом в выполнении программы. Отправлять подтверждение слишком рано опасно, поскольку в случае возникновения ошибки выполнение операции может прерваться.

Если вы выбираете явное подтверждение, то должны вызывать метод `basic_ack()`. Если сообщение не подтвердить, то брокер будет хранить все неподтвержденные сообщения, расходуя оперативную память и место в хранилище, что в итоге может привести к сбою брокера. Важно помнить: если потребитель не отправляет подтверждение (при работе в режиме явного подтверждения с `auto_ack=False`), то брокер присвоит сообщению пометку для повторной доставки. На уровне AMQP подтверждение соответствует фрейму AMQP, содержащему флаг `Basic.Ack`.

Недостаток явного подтверждения — снижение пропускной способности системы. Чтобы увеличить количество сообщений, обрабатываемых системой за единицу времени, потребитель может использовать автоматическое подтверждение, работающее по принципу «отправил и забыл». Этот метод обеспечивает максимальную пропускную способность, но не гарантирует, что все сообщения будут доставлены.

Автоматическое подтверждение (`auto_ack=True`) небезопасно, поскольку сообщение удаляется из очереди сразу после отправки брокером, без

согласования с потребителем[1]. В этом режиме брокер не знает и не отслеживает, что происходит с сообщением после его удаления из очереди.

Кроме того, брокер может отправлять подтверждение производителю, чтобы уведомить его о получении сообщения. Этот метод называется «подтверждение издателя» (publisher confirms). Он должен быть явно активирован на стороне производителя.

Теперь опубликуем сообщение (добавим его в очередь). Для этого выполним метод `publish()` в скрипте `producer.py` (пример 11.4).

Пример 11.4. Реализация публикации сообщения на стороне производителя

```
# src/rabbitmq/2.WorkQueue/src/producer.py

...
from pika import BasicProperties
from pika.spec import PERSISTENT_DELIVERY_MODE

class RabbitMQProducer:
    ...
    queue = os.getenv("RABBITMQ_QUEUE", "die")
    properties = BasicProperties(
      ...
      delivery_mode=PERSISTENT_DELIVERY_MODE,  ❺
      ...
    )
    ...

    def publish(self, msg: dict):   ❶
      ...
      self.channel.basic_publish(   ❶
        exchange="",                ❷
        routing_key=self.queue,     ❸
        body=to_json(msg),          ❹
        properties=self.properties  ❺
      )
      self.disconnect()  ❻

    def disconnect(self):  ❻
        if self.channel.is_open:
            self.channel.close()

        if self.connection.is_open:
            self.connection.close()
```

[1] В RabbitMQ в режиме автоматического подтверждения сообщение считается доставленным, когда оно записано в TCP-соединение потребителя, и AMQP отправляет фрейм Basic.Deliver, не дожидаясь TCP-подтверждения.

❶ В методе `publish()` сообщение публикуется в очередь с помощью метода `basic_publish()`.

❷ Параметр `exchange`, значение которого установлено как пустая строка, по умолчанию отправляет сообщение в обменник.

❸ `routing_key` указывает имя привязки, которое используется для маршрутизации сообщения в очередь.

❹ Сообщение `msg` кодируется в строку JSON.

❺ Свойства сообщения (`properties`) задаются путем их чтения из атрибутов класса. В данном случае выбран режим доставки сообщения `PERSISTENT_DELIVERY_MODE`. Этот режим гарантирует, что в случае перезапуска брокера сообщение сохранится в файловой системе и будет восстановлено.

❻ Метод `disconnect()` закрывает канал и соединение.

Если очередь помечена как постоянная (`durable`), а для сообщений выбран режим доставки `PERSISTENT_DELIVERY_MODE`, то в случае перезапуска брокера очередь сохранится. После восстановления брокера сообщения будут доставлены.

ДОСТАВКА СООБЩЕНИЙ

В распределенных системах предусмотрены три вида гарантии доставки сообщения.

Доставка «как минимум один раз» (at-least-once)
: Гарантируется, что сообщение будет доставлено — возможно, несколько раз.

Доставка «не более одного раза» (at-most-once)
: Сообщение или доставляется один раз, или не доставляется вообще. Этот метод предотвращает дублирование сообщений, но повышает риск их потери.

Доставка «ровно один раз» (exactly-once)
: Сообщение доставляется строго один раз[1].

[1] Доставка сообщений ровно один раз (exactly-once delivery) — сложная тема для обсуждения. Более подробно ознакомиться с ней можно в публикациях You Cannot Have Exactly-Once Delivery («Вы не можете гарантировать доставку “ровно один раз”») (https://oreil.ly/bjDo7) и Exactly-Once Message Delivery («Доставка сообщения ровно один раз») (https://oreil.ly/lfz2g).

Важно понимать, что доставку «ровно один раз» сложно реализовать нативно на уровне протокола[1]. В большинстве случаев для этого требуется координация отправки и получения сообщений между производителями и потребителями на уровне приложения[2]. Чтобы достичь эффекта доставки «ровно один раз», можно реализовать доставку «как минимум один раз» с последующей дедупликацией сообщений.

При использовании ручного подтверждения (см. врезку «Подтверждение получения сообщений в RabbitMQ» выше в этой главе) RabbitMQ гарантирует доставку «как минимум один раз», поскольку сообщения остаются в очереди, пока брокер не получит подтверждение от потребителя. Если потребитель завершает работу до отправки подтверждения, то неподтвержденные сообщения возвращаются в очередь для повторной доставки.

При использовании автоматического подтверждения сообщение, напротив, удаляется из очереди сразу после отправки потребителю, независимо от того, было ли оно фактически доставлено. В этом случае гарантируется доставка «не более одного раза» и повышается риск потери сообщений.

Кроме того, если не реализован механизм «подтверждение издателя», то сообщения могут теряться на стороне производителя во время публикации, что тоже приводит к доставке «не более одного раза». Таким образом, чтобы получить гарантию доставки «как минимум один раз», необходимы ручные подтверждения от потребителя и «подтверждения издателя» от производителя. При отсутствии этих механизмов есть риск потери сообщений во время публикации или потребления, что позволяет реализовать только доставку «не более одного раза».



Помните: пометка persistent для очереди и сообщений не дает гарантии, что сообщение не будет потеряно. Например, есть риск, связанный с временным окном, когда брокер уже принял сообщение, но еще не сохранил его на диск. Если в этот момент брокер аварийно завершит работу, то сообщения могут быть потеряны. Гарантию доставки сообщений от производителя обеспечивает метод «подтверждение производителя» (https://oreil.ly/TaEqy).

---

[1] Разработчики Kafka утверждают, что им удалось реализовать логику передачи «ровно один раз» с помощью транзакционных API (https://oreil.ly/CKiRE).

[2] Сравнение доставки сообщений в RabbitMQ и Kafka см. в записи в блоге Джека Ванлайтли (Jack Vanlightly) RabbitMQ versus Kafka — Message Delivery Semantics and Guarantees («RabbitMQ и Kafka — семантика и гарантии доставки сообщений») (https://oreil.ly/L4QE3).

Обратите внимание и на тот факт, что сообщения могут быть потеряны во время их восстановления. Когда работа брокера будет восстановлена после вынужденного завершения работы, он начнет восстанавливать очередь и сообщения из файловой системы. Эта процедура требует времени. Представим, что в процессе восстановления брокер перезапустили снова. В этом случае сообщения, очереди и обменники могут потеряться, так как процесс восстановления данных, существовавших до отключения, еще не завершился. Следовательно, помечая компоненты системы как постоянные, мы снижаем риск потери сообщений при сбое брокера, но не гарантируем сохранение всех сообщений, особенно в тех случаях, когда при восстановлении брокера происходят повторные сбои.

Рассмотрим процесс потребления сообщений в скрипте `consumer.py` (пример 11.5).

Пример 11.5. Реализация потребления сообщений

```
# src/rabbitmq/2.WorkQueue/src/consumer.py

...
class RabbitMQConsumer:
    ...

    def __callback(self, channel, method, properties, msg):  ❻
      ...
      delivery_tag = method.delivery_tag
      channel.basic_ack(delivery_tag=delivery_tag)
      ...
    ...

    def consume(self):  ❶
      ...
      self.channel.basic_qos(prefetch_count=1)  ❷
      self.channel.basic_consume(               ❸
        queue=self.queue,                       ❹
        auto_ack=False,                         ❺
        on_message_callback=self.__callback     ❻
      )
      self.channel.start_consuming()  ❼
    ...
...
```

❶ Метод `consume()` отвечает за потребление сообщений из очереди.

❷ Метод `basic_qos()` с параметром `prefetch_count=1` ограничивает потребление сообщений одним сообщением на канал. Благодаря этому сообщения равномерно распределяются между потребителями, и каждый из них обрабатывает только одно сообщение за раз, не получая новое, пока не подтвердит

предыдущее. Такой подход позволяет более сбалансированно распределить нагрузку между потребителями.

❸ Метод `basic_consume()` настраивает параметры для потребления сообщений.

❹ Аргумент `queue` определяет, из какой очереди будет потребляться сообщение.

❺ Аргумент `auto_ack` управляет автоматическим подтверждением сообщений. В данном случае мы отключаем автоматическое подтверждение и выбираем явное (см. шаг 6). Подтверждением сообщений на стороне потребителя управляете вы. Можно выставить настройки так, чтобы одни потребители использовали явное подтверждение, а другие — автоматическое, однако рекомендуется применять один тип подтверждения для всех потребителей одной очереди.

❻ Аргумент `on_message_callback` связывает метод потребителя `__callback()` с событием получения нового сообщения. Сообщение получает уникальный идентификатор в виде параметра `delivery_tag`, который автоматически назначается RabbitMQ. Подтверждение сообщения выполняется вызовом метода `basic_ack()`.

❼ Метод `start_consuming()` запускает потребление сообщений из очереди.

Безопасность

В этом разделе мы защитим обмен данными RabbitMQ с помощью TLS и реализуем аутентификацию клиентов с помощью фреймворка Simple Authentication and Security Layer (SASL) (https://oreil.ly/VMMdb).

Прежде чем мы перейдем к шифрованию в RabbitMQ, кратко обсудим протокол Advanced Message Queuing Protocol (AMQP) (https://oreil.ly/10LlL). Это двоичный протокол прикладного уровня, который используется в TCP в качестве транспортного протокола. AMQP обеспечивает поддержку приложений для обмена сообщениями, где используются разные паттерны коммуникации и доставки сообщений.

В примерах в этом разделе мы сравним незашифрованный и зашифрованный трафик AMQP, захваченный во время выполнения примера 11.1. Управление шифрованием выполняется через переменную окружения `TLS_ENABLE` в файле `compose.yaml`. По умолчанию переменной присвоено значение `1` для шифрования сообщений при передаче — как между производителем и брокером, так и между брокером и потребителем.

Инструкции по захвату сетевого трафика в этой главе не приводятся (примеры можно найти в главе 3). Вместо этого мы откроем готовые файлы `.pcap` с помощью анализатора сетевых протоколов Wireshark (https://www.wireshark.org).



Чтобы в Wireshark вместо IP-адресов отображались имена хостов, добавьте следующие строки в начало файла /etc/hosts на компьютере, где запущен Wireshark (чтобы измененить этот файл, требуются права sudo)[1]:

```
192.168.16.2    broker
192.168.16.3    producer
192.168.16.4    consumer-a
192.168.16.5    consumer-b
```

Затем в Wireshark перейдите в меню Edit ▸ Preferences ▸ Name Resolution и выберите пункт Resolve network (IP) addresses.

Передача незашифрованных сообщений

На рис. 11.11 показан снимок экрана с незашифрованным трафиком AMQP, загруженным в Wireshark. При работе в терминале также можно использовать команду **`tshark -r src/rabbitmq/2.Work Queue/docs/assets/amqp-unencrypted.pcap`**, чтобы вывести список сетевых пакетов. Анализируя порядок пакетов в файле `amqp-unencrypted.pcap`, можно вывести этапы работы программы и изучить содержимое передаваемых сообщений.



Сообщения AMQP называются фреймами и отличаются от фреймов Ethernet, о которых шла речь в разделе «TCP/IP и модель OSI» главы 3.

- Фреймы 4–6: производитель устанавливает TCP-соединение с брокером (`broker`). Подробнее о трехстороннем рукопожатии TCP см. в примере 3.5 или в коротком обзорном видеоролике TCP Connection Walkthrough («Пошаговый разбор TCP-соединения») (https://oreil.ly/b2CNg) Бена Итера (Ben Eater).
- Фреймы 7–16: после установки TCP-соединения производитель (`producer`) открывает AMQP-соединение с брокером.
- Фреймы 17–18: после создания AMQP-соединения создается канал RabbitMQ.
- Фреймы 19–20: производитель создает очередь `die`, а брокер подтверждает ее создание.
- Фрейм 21: производитель (`producer`) задает параметры сообщения, которое собирается отправить. Присвоение `x=` пустого значения позволяет выбирать обменник по умолчанию, а `rk=die` (ключ маршрутизации) доставляет сообщение в нужную очередь.

[1] Расположение файла hosts может различаться в разных операционных системах.

- Фреймы 22–23: производитель публикует сообщение в очередь. Сообщение разделено на два AMQP-фрейма: один с заголовком сообщения, а другой — с телом сообщения размером 66 байт, содержащим текст `Rolling 4`. Второй AMQP-фрейм содержит незашифрованную полезную нагрузку в формате JSON. Полезная нагрузка представлена в виде простого текста, который могут прочитать все, кто имеет доступ к сетевому трафику.
- Фреймы 25–28: AMQP-канал и соединение закрываются.

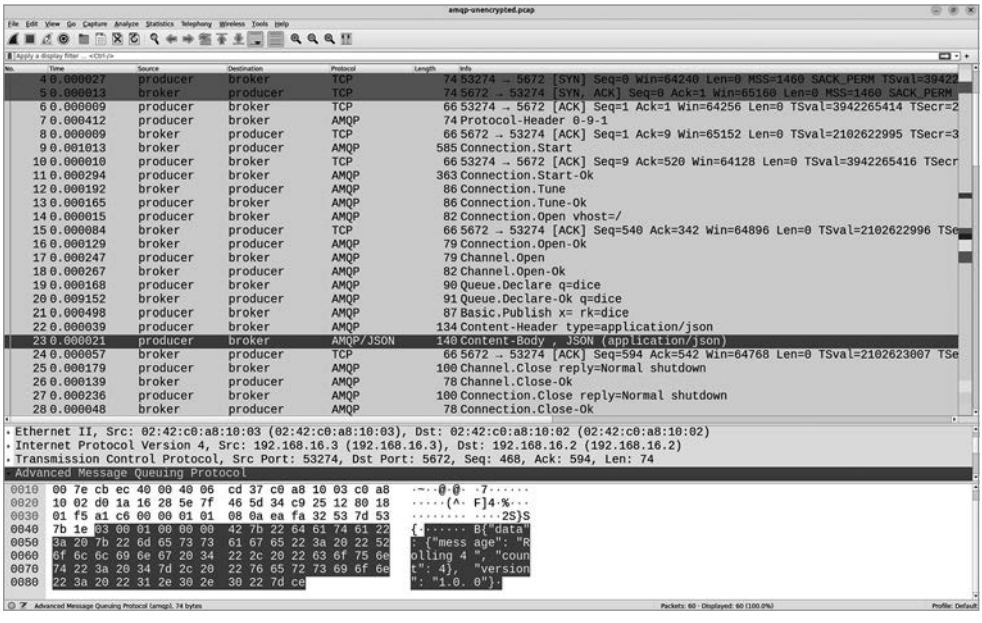


Рис. 11.11. Незашифрованный трафик AMQP

Процесс потребления сообщений схож с процессом их публикации. Разница в том, что при потреблении сообщений потребитель (например, `consumer-a`) читает сообщение от брокера. Этот процесс зафиксирован во фреймах 52 и 53 (на рис. 11.11 не показаны).

Передача зашифрованных сообщений

Когда соединение по протоколу AMQP шифруют с помощью TLS, протокол называют *AMQPS*. Обратите внимание: AMQPS не является самостоятельным официальным протоколом. Термин AMQPS обозначает AMQP поверх TLS аналогично тому, как HTTPS обозначает HTTP поверх TLS. Подробнее работа TLS была описана в примере 3.12.

Сравним незашифрованный AMQP-трафик, показанный на рис. 11.11, с зашифрованным AMQPS-трафиком на рис. 11.12 (файл `src/rabbitmq/2.WorkQueue/docs/assets/amqp-encrypted.pcap`). Разница в том, что в этот раз обмен данными между брокером и производителем (и потребителем) зашифрован с помощью TLS.



Подробнее изучить настройку TLS для RabbitMQ поможет файл src/rabbitmq/README.md в репозитории кода книги.

После завершения TLS-рукопожатия, начиная с фрейма 18, в TCP-сегментах отображаются зашифрованные данные приложения (`Application Data`). Выделенный текст в правом нижнем углу снимка экрана (`E^.i.m…X….[cf….t?ag`) — это зашифрованная полезная нагрузка.

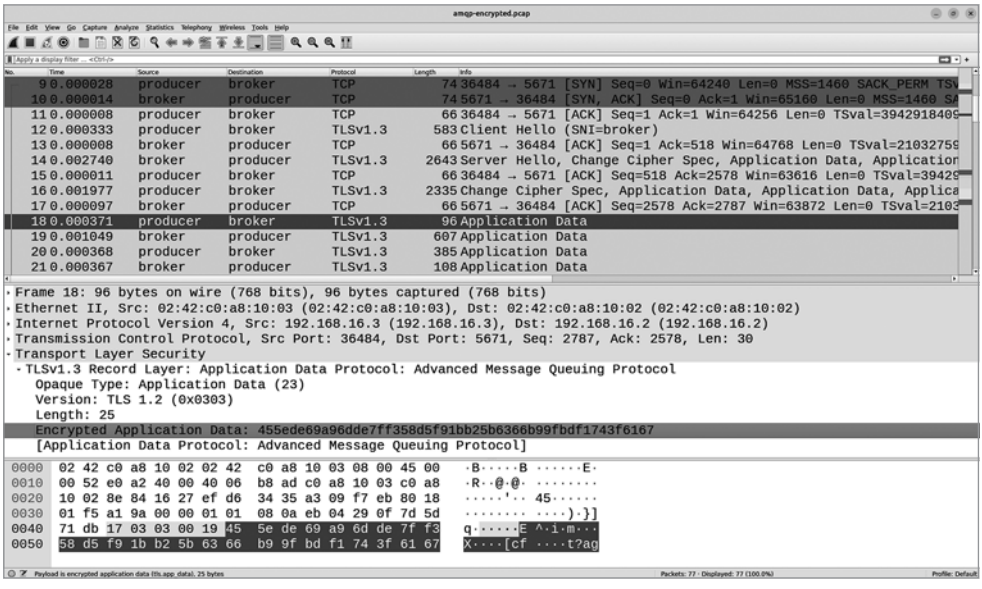


Рис. 11.12. Зашифрованный трафик AMQPS

Рассмотрим конфигурацию TLS для брокера RabbitMQ (пример 11.6).

Пример 11.6. Конфигурация TLS для брокера RabbitMQ

```
# src/rabbitmq/2.WorkQueue/src/config/etc/rabbitmq/conf.d/10-defaults.conf

# Установка портов для незашифрованного и зашифрованного обмена данными ❶
listeners.tcp.default = 5672
listeners.ssl.default = 5671
```

```
# Включение поддержки SSL/TLS  ❷
ssl_options.cacertfile = /etc/rabbitmq/ssl/certs/ca_certificate.pem
ssl_options.certfile = /etc/rabbitmq/ssl/certs/server_certificate.pem
ssl_options.fail_if_no_peer_cert = true
ssl_options.keyfile = /etc/rabbitmq/ssl/private/server_key.pem
ssl_options.verify = verify_peer
...
```

❶ Настройки `listeners` определяют порты приложения для незашифрованного (`5672`) и зашифрованного (`5671`) обмена данными.

❷ В этом блоке описана конфигурация TLS для брокера. Брокер использует серверные сертификаты путем задания свойств `ssl_options`, среди которых расположение файлов `ca_certificate.pem`, `server_certificate.pem` и `server_key.pem`. Опция `verify=verify_peer` обязывает сервер (`broker`) проверять сертификат клиента, если тот предоставляет свой TLS-сертификат.

В конфигурации mTLS клиент и брокер (сервер) проверяют друг у друга сертификаты для взаимной аутентификации.

Теперь мы можем перейти к реализации безопасного обмена данными на стороне клиента (производителя и потребителя). Рассмотрим реализацию, общую для `producer.py` и `consumer.py` (пример 11.7).

Пример 11.7. Реализация TLS на стороне производителя (и потребителя)

```
# src/rabbitmq/2.WorkQueue/src/producer.py

...
broker = os.getenv("RABBITMQ_BROKER")
user = os.getenv("RABBITMQ_USER")
password = os.getenv("RABBITMQ_PASS")
tls_enable = to_bool(os.getenv("TLS_ENABLE", True))  ❸
...

def __init__(self):
    self.__connect()  ❶

def __connect(self):  ❶
    """Установка соединения с брокером RabbitMQ."""
    credentials = PlainCredentials(self.user, self.password)  ❷
    if self.tls_enable:  ❸
        port = 5671
        context = create_default_context(
          cafile="/etc/rabbitmq/ssl/certs/ca_certificate.pem"
        )
        context.load_cert_chain(  ❹
            "/etc/rabbitmq/ssl/certs/client_certificate.pem",
            "/etc/rabbitmq/ssl/private/client_key.pem"
        )
```

```
    ssl_options = SSLOptions(context)  ❺
    params = ConnectionParameters(  ❻
      self.broker, port, ssl_options=ssl_options, credentials=credentials
    )

...
try:
  self.connection = BlockingConnection(params)  ❼
except Exception:
    print(f"Ошибка: невозможно подключиться к хосту '{self.broker}'")
    exit(1)
```

❶ Метод `__connect()` содержит настройки для установки соединения со службой `broker`.

❷ Класс `PlainCredentials` создает объект учетных данных для аутентификации клиентов (производителей и потребителей) в службе брокера. При создании объекта используются имя пользователя и пароль.

❸ Настройка `tls_enable` управляет шифрованием. Если ей присвоено значение `true`, то зашифрованное соединение будет использовать порт `5671` и создаст контекст безопасности с помощью метода `create_default_context()`, который считывает файл `ca_certificate.pem`. Этот сертификат CA используется на стороне клиента для проверки сертификата брокера. В конфигурации mTLS брокер использует такой же сертификат CA для проверки клиентского сертификата.

❹ Метод `load_cert_chain()` загружает клиентский сертификат (`client_certificate.pem`) и ключ (`client_key.pem`) в переменную `context`.

❺ Переменная `ssl_options` содержит параметры SSL, сформированные на основе данных из переменной `context`.

❻ `ConnectionParameters` создает объект `params`, который принимает в качестве аргументов `broker`, `port`, `ssl_options` и `credentials`.

❼ `BlockingConnection` устанавливает блокирующее соединение со службой брокера, используя настройки, содержащиеся в `params`.

Аутентификация и авторизация

В контексте обеспечения безопасности RabbitMQ необходимо также обсудить аутентификацию и авторизацию. В этом подразделе мы рассмотрим аутентификацию в RabbitMQ с помощью SASL и использование списков контроля доступа (access control lists, ACL) (https://oreil.ly/S1dGy) для управления авторизацией.

SASL — это фреймворк для аутентификации, который позволяет отделить прикладные протоколы (например, AMQP, LDAP, SMTP) от методов аутентификации. Цель этого фреймворка заключается в том, чтобы любой прикладной

протокол, поддерживающий SASL, мог использовать методы аутентификации, поддерживаемые SASL. RabbitMQ поддерживает следующие механизмы аутентификации SASL:

- `PLAIN`, `AMQPLAIN`, `ANONYMOUS` (не предназначен для использования в продакшен-среде);
- `RABBIT-CR-DEMO` (используется для демонстрации метода аутентификации «запрос — ответ»);
- `EXTERNAL` (доступен в виде плагина).

В этом подразделе мы будем использовать механизм `PLAIN`, который по умолчанию включен в клиентах RabbitMQ (потребителях и производителях) и на сервере (брокере). Для аутентификации `PLAIN` использует учетные данные (имя пользователя и пароль). Этот метод необходимо комбинировать с шифрованием TLS, поскольку учетные данные `PLAIN` передаются в открытом виде.

Рассмотрим код для реализации аутентификации, разделенный на три файла, содержание которых представлено в примерах 11.8, 11.9 и 11.10.

Пример 11.8. Настройки аутентификации в файле compose.yaml

```
# src/rabbitmq/2.WorkQueue/compose.yaml

services:
  broker:
    ...
    environment:
      - RABBITMQ_USER=bugs  ❶
      - RABBITMQ_PASS=${RABBITMQ_PASS:-bunny}  ❷
    ...

  producer:
    ...
    environment:
      - RABBITMQ_USER=bugs  ❶
      - RABBITMQ_PASS=${RABBITMQ_PASS:-bunny}  ❷

  consumer-a:
    ...
    environment:
      - RABBITMQ_USER=bugs  ❶
      - RABBITMQ_PASS=${RABBITMQ_PASS:-bunny}  ❷
```

Пример 11.9. Конфигурация SASL для брокера RabbitMQ

```
# src/rabbitmq/2.WorkQueue/src/config/etc/rabbitmq/conf.d/10-defaults.conf

default_user = $(RABBITMQ_USER)  ❶
default_pass = $(RABBITMQ_PASS)  ❷
```

Пример 11.10. Реализация SASL на стороне производителя (и потребителя)

```
# src/rabbitmq/2.WorkQueue/src/producer.py

...
broker = os.getenv("RABBITMQ_BROKER")
user = os.getenv("RABBITMQ_USER")      ❶
password = os.getenv("RABBITMQ_PASS")  ❷
...

def __init__(self):
    self.__connect()
    ...

def __connect(self):
    credentials = PlainCredentials(self.user, self.password)  ❸
    ...
    params = ConnectionParameters(self.broker, port, credentials=credentials)
    ...
    self.connection = BlockingConnection(params)
    ...
```

❶ Переменная окружения `RABBITMQ_USER` хранит имя пользователя.

❷ Переменная окружения `RABBITMQ_PASS` хранит пароль.

❸ Клиент (производитель или потребитель) проходит процедуру аутентификации при подключении к серверу (брокеру). Если учетные данные неверны, то брокер отклоняет соединение.



Не забудьте изменить учетные данные по умолчанию на данные ваших сервисов. Использование учетных данных по умолчанию приводит к рискам, указанным в рейтинге угроз безопасности OWASP Top 10 (https://oreil.ly/lqDLe).

RabbitMQ — это мультитенантная система. Она позволяет большому количеству пользователей (тенантов) совместно использовать одну инфраструктуру, при этом обеспечивая безопасность и строгую изоляцию данных между ними. Для поддержки мультитенантности в RabbitMQ используется механизм виртуальных хостов. Такой хост позволяет логически группировать и разделять ресурсы RabbitMQ (например, обменники, привязки и очереди).

Пользователю RabbitMQ доступны только три категории прав: чтение (`read`), запись (`write`) и настройка (`configure`). Право на чтение (`read`) позволяет пользователям потреблять сообщения из ресурсов. Право на запись (`write`) позволяет создавать сообщения, которые будут распространяться через ресурсы.

Наконец, право на настройку (`configure`) позволяет создавать, удалять или задавать права доступа к ресурсам. Для описания прав используются шаблоны регулярных выражений. Например, шаблон `.*` означает, что пользователь имеет доступ ко всем ресурсам в определенной категории (`read`, `write` и `configure`).

С помощью команд, показанных в примере 11.11, проверим права доступа по умолчанию, которые RabbitMQ назначает пользователю `bugs`.

Пример 11.11. Проверка прав пользователей RabbitMQ

```
# Перейдите в каталог упражнения  ❶
cd src/rabbitmq/2.WorkQueue

# Запустите контейнеры  ❷.
docker compose up --detach --wait

# Выведите список пользователей RabbitMQ  ❸
docker compose exec broker rabbitmqctl list_users --formatter pretty_table

Listing users ...
┌──────┬───────────────┐
│ user │ tags          │
├──────┼───────────────┤
│ bugs │ administrator │
└──────┴───────────────┘

# Выведите список виртуальных хостов RabbitMQ  ❹
docker compose exec broker rabbitmqctl list_vhosts --formatter pretty_table
Listing vhosts ...
┌──────┐
│ name │
├──────┤
│ /    │
└──────┘

# Выведите список прав пользователей  ❺
docker compose exec broker rabbitmqctl list_permissions --formatter pretty_
table
Listing permissions for vhost "/" ...
┌──────┬───────────┬───────┬──────┐
│ user │ configure │ write │ read │
├──────┼───────────┼───────┼──────┤
│ bugs │ .*        │ .*    │ .*   │
└──────┴───────────┴───────┴──────┘
```

❶ Перейдите в каталог упражнения.

❷ Ожидайте запуска контейнеров в фоновом режиме.

❸ Выведите список пользователей RabbitMQ.

❹ Выведите список виртуальных хостов RabbitMQ.

❺ Выведите список прав пользователей. Как вы можете видеть, пользователь `bugs` имеет права на чтение (`read`), запись (`write`) и настройку (`configure`) для всех ресурсов (`.*`) в виртуальном хосте `/`, что позволяет ему решать административные задачи. Такой набор прав нужен администратору, но не обычному пользователю. Чтобы контролировать доступ к отдельным ресурсам, нужно создать пользователя без прав администратора и назначить ему только необходимые права.

Документация

Мы уже рассмотрели реализацию производителя и потребителя в RabbitMQ, а также способы обеспечения безопасности системы. Теперь пора взглянуть на систему глазами пользователя. В этом разделе речь пойдет о создании документации к API на основе обмена сообщениями с помощью AsyncAPI (https://www.asyncapi.com).

Спецификация AsyncAPI

В примере 11.12 представлена спецификация AsyncAPI, которая описывает API на основе обмена сообщениями по паттерну «Рабочая очередь», показанному на рис. 11.9.



Если вы уже знакомы с OpenAPI (OAS) (https://oreil.ly/Ke7WN), но еще не знакомы с AsyncAPI, то обзор различий между этими двумя спецификациями сможете найти в разделе Coming from OpenAPI («Для тех, кто знаком с OpenAPI») на сайте AsyncAPI (https://oreil.ly/33mBQ).

Пример 11.12. Спецификация AsyncAPI для рабочей очереди

```
# src/rabbitmq/2.WorkQueue/docs/api/v1.0.0/work_queue.yaml

asyncapi: 3.0.0  ❶
defaultContentType: application/json  ❷
info:  ❸
  title: Work Queue
  version: 1.0.0
  description: Work queue lab documentation.
channels:  ❹
  Die:
    description: Die Queue
    address: die
```

```
    bindings:
      amqp:
        is: queue
        queue:
          contentEncoding: utf-8
          durable: true
          name: die
          vhost: /
    messages:
      RollDie:
        $ref: '#/components/messages/RollDie'
operations:  ❺
  Roll:
    action: send
    channel:
      $ref: '#/channels/Die'
    bindings:
      amqp:
        deliveryMode: 2 # 1 - transient, 2 - persistent
  Pick:
    action: receive
    channel:
      $ref: '#/channels/Die'
    bindings:
      amqp:
        ack: true
components:  ❻
  messages:
    RollDie:
      name: Roll Die
      title: Roll Die
      headers:
        type: object
        properties:
          version:
            type: string
            default: 1.0.0
      payload:
        type: object
        properties:
          data:
            type: object
            properties:
              outcome:
                type: string
              message:
                type: string
          version:
            type: string
            default: 1.0.0
```

❶ Обязательное поле `asyncapi` содержит версию спецификации AsyncAPI.

❷ Поле `defaultContentType` описывает MIME-тип, который по умолчанию используется для кодирования полезной нагрузки сообщения (JSON).

❸ Обязательное поле `info` содержит метаданные о документируемом API.

❹ Обязательное поле `channels` описывает маршруты для отправки и получения сообщений. В зависимости от используемой системы маршрут может быть топиком, ключом маршрутизации, очередью, типом события или URL-путем. В нашем примере канал связи реализован в виде очереди с соответствующими свойствами (`durability` и др.). Именем очереди становится уникальный идентификатор (`Die` ID).

❺ Поле `operations` описывает операции, которые должны быть реализованы в приложении для работы с каналами. В нашем примере есть две операции: `Roll` (`send`) и `Pick` (`receive`). Они описывают намерение отправить и получить сообщение через канал `Die`. Операция `Roll` указывает, что производитель использует `deliveryMode` со значением `2` (постоянная доставка). Операция `Pick` указывает, что потребитель должен подтвердить получение сообщения. При этом не уточняется, какой именно тип подтверждения должен использовать потребитель (автоматическое или явное).

Имена операций `Roll` и `Pick`, используемые в нашем примере, — это условные названия. Имя `Roll` означает, что бросок кубика публикует сообщение в очередь `die`, а имя `Pick` указывает на потребление сообщения из этой очереди.

❻ Поле `components` определяет повторно используемые структуры (или определения), которые применяются ко всему документу. В данном случае это спецификация, которая описывает структуру сообщения `Roll Die`, отправляемого и получаемого через каналы. Поле `properties` определяет допустимые заголовки (`headers`) и данные в полезной нагрузке сообщения (`payload`).



При изучении спецификации можно заметить, что для одних полей используется стиль наименования lowerCamelCase (например, defaultContentType), а для других — UpperCamelCase (RollDie). Названия полей в нижнем регистре характерны для самой спецификации AsyncAPI, а названия в верхнем регистре — это идентификаторы, используемые в вашем приложении. В примере выше для идентификаторов приложения мы выбрали стиль именования с заглавной буквы, но вы можете использовать и стиль наименования со строчной буквы.

Документация AsyncAPI

Рассмотрим инструменты, которые предоставляет AsyncAPI. В примере 11.13 показана утилита `asyncapi/cli`, которая используется для преобразования спецификации AsyncAPI в HTML. `asyncapi/cli` запускается в контейнере, созданном из Docker-образа `asyncapi/cli`. В качестве альтернативы можно выполнить программу `asyncapi` как автономный интерфейс командной строки (CLI).



С помощью онлайн-сервисов можно также сгенерировать документ для предварительного просмотра документации AsyncAPI в формате HTML. Пример такого сервиса — AsyncAPI Studio (https://studio.asyncapi.com). Однако рекомендуем сначала рассмотреть локальные альтернативы, позволяющие избежать раскрытия конфиденциальных данных в сети.

Пример 11.13. Генерация документации для рабочей очереди с помощью asyncapi/cli

```
# Перейдите в каталог упражнения  ❶
cd src/rabbitmq/2.WorkQueue/docs/api/v1.0.0

# Сгенерируйте HTML-документацию  ❷
docker run --rm -it --user=root \
       --volume ${PWD}/output:/app/output \
       --volume ${PWD}/work_queue.yaml:/app/asyncapi.yaml \
       asyncapi/cli generate fromTemplate /app/asyncapi.yaml \
       @asyncapi/html-template@3.0.0 \
       --use-new-generator --force-write --output /app/output

# Запустите сервер для документации с Nginx  ❸
docker run --rm --detach --publish 127.0.0.1:8888:80 \
       --volume ${PWD}/output:/usr/share/nginx/html:ro nginx
```

❶ Перейдите в каталог, содержащий файл спецификации AsyncAPI `work_queue.yaml`.

❷ Преобразуйте спецификацию AsyncAPI в HTML-документацию. В качестве входных данных программа `asyncapi` принимает файл `work_queue.yaml` вместе с шаблоном `@asyncapi/html-template`. Созданная HTML-документация сохраняется в файловой системе контейнера в каталоге, заданном в `--output`. Флаг `--force-write` гарантирует, что документация будет сгенерирована, даже если аналогичные файлы уже есть в каталоге.

❸ Запустите сервер Nginx для предоставления доступа к сгенерированной HTML-документации. Флаг `--volume` монтирует документацию из каталога

`output` на хосте Docker в расположение `/usr/share/nginx/html` в файловой системе контейнера. Флаг `--publish` перенаправляет трафик с порта контейнера (`80`) на порт хоста Docker (`8888`)[1].

При открытии http://localhost:8888 в браузере отобразится документация (рис. 11.13).

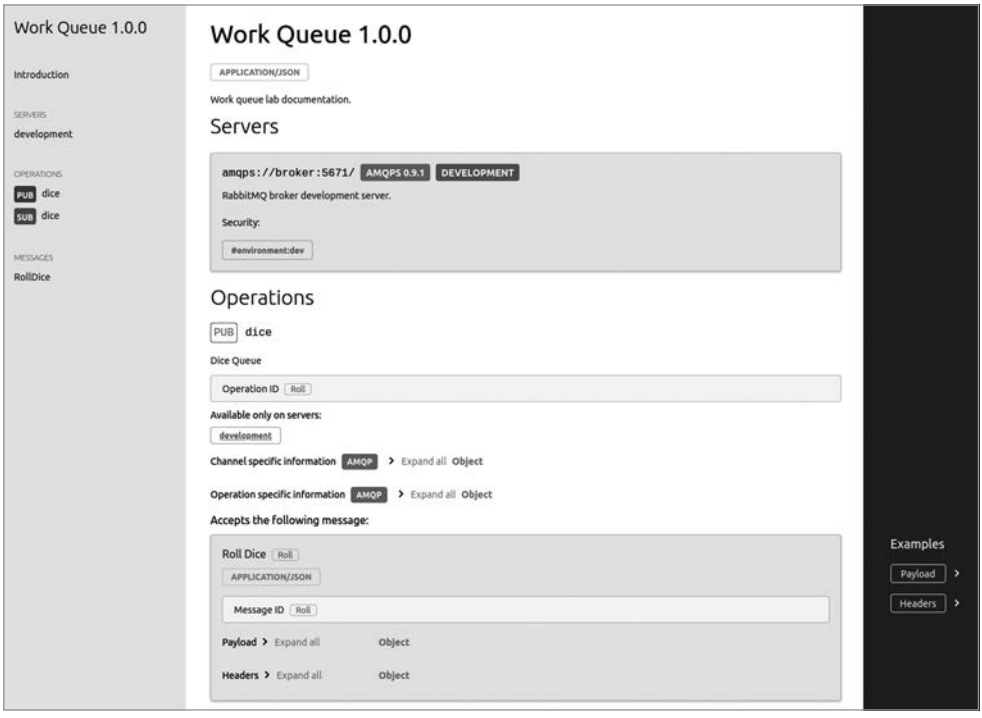


Рис. 11.13. HTML-документация, сгенерированная из спецификации AsyncAPI

Версионирование сообщений

В REST API версионирование выполняется на уровне эндпоинта или ресурса API. В API на основе обмена сообщениями можно версионировать само сообщение или ресурс (обменник или очередь) путем добавления версии к его имени.

---

[1] При работе с GitHub Codespaces используется уникальный URL, содержащий имя Codespace вместо адреса localhost. Этот URL привязан к порту 8888, и его можно найти на вкладке PORTS в VS Code.

Версионирование сообщений реализуется через добавление поля версии в заголовок сообщения и/или в его полезную нагрузку (пример 11.14).



Поскольку пример используется в учебных целях, то версия сообщения указана и в заголовках, и в полезной нагрузке. В реальности полезную нагрузку сообщения следует отделять от его метаданных, поэтому нужно будет убрать поле version из msg и оставить его только в заголовке. Но если заголовок не виден в теле сообщения, то процесс отладки версионирования усложняется.

Пример 11.14. Версионирование полезной нагрузки сообщения

```
# src/rabbitmq/2.WorkQueue/src/producer.py

...
class RabbitMQProducer:
    ...
    properties = BasicProperties(  ❷
        content_type='application/json',
        ...
        headers={"version": "1.0.0"}
    )
    ...

    def publish(self, msg: dict):
        self.properties.headers["version"] = msg["version"]  ❸
        self.channel.basic_publish(
            exchange="",
            routing_key=self.queue,
            body=to_json(msg)
            properties=self.properties
        )
        ...

if __name__ == "__main__":
    ...
    msg = {  ❶
        "data": {
            "message": f"Rolling {number}",
            "outcome": number
        },
        "version": "1.0.1"
    }
    ...
    publisher = RabbitMQProducer()  ❷
    publisher.publish(msg)  ❸
```

❶ Python-словарь `msg` содержит сообщение с данными и полем `version` (`1.0.1`).

❷ Объект `producer` содержит класс `BasicProperties`, а также тип данных в сообщении `content_type` (application/json) и Python-словарь заголовков `headers` с полем `version`, для которого по умолчанию установлено значение `1.0.0`.

❸ Версия по умолчанию в заголовке `properties` переопределяется значением, указанным в полезной нагрузке сообщения.

API на основе обмена сообщениями: преимущества и недостатки

Начнем с обсуждения преимуществ.

Доставка сообщений

Системы обмена сообщениями допускают разные способы доставки сообщений в зависимости от выбранной технологии. Сообщение может быть доставлено как минимум один раз, не более одного раза или ровно один раз.

Высокая пропускная способность и минимальное время ожидания

Системы обмена сообщениями спроектированы для обработки большого потока сообщений с минимальным временем ожидания. Например, разработчики RabbitMQ утверждают, что за секунду система может опубликовать в брокер или получить из него 26 тысяч сообщений размером 1024 байта (1 Кбайт). При проведении неофициального тестирования на ноутбуке Lenovo ThinkPad P17 G1 (16-ядерный процессор, 32 Гбайт памяти) нам удалось достичь пропускной способности 67 тысяч сообщений в секунду. Разработчики Apache Kafka утверждают, что система может создать 2 миллиона записей в секунду. В целом тестирование производительности любой системы — сложная задача, зависящая от множества параметров. Однако эти тесты дают примерное представление о количестве сообщений, с которым такие системы могут работать.

Масштабируемость

Системы обмена сообщениями допускают масштабирование, так как изначально спроектированы с учетом возможного увеличения объема работы. Если система не справляется с потоком входящих сообщений от производителей, то можно выполнять масштабирование на стороне потребителей вертикально или горизонтально.

Когда брокер не справляется с нагрузкой, его можно масштабировать вертикально, добавив ресурсы, или горизонтально, используя кластеризацию[1]. Кроме того, можно применить гибридный подход и использовать оба способа одновременно. Наконец, можно упростить масштабирование, физически разделив приложения-производители и приложения-потребители.

Важно помнить, что каждый подход к масштабированию имеет свои преимущества и недостатки. Например, горизонтальное масштабирование увеличивает сложность системы, поскольку требует настройки большого количества приложений и управления ими. Вертикальное масштабирование требует остановки системы и в итоге может быть ограничено возможностями сервера.

Отказоустойчивость

Очереди служат буфером между производителями и потребителями сообщений. Если потребитель выходит из строя, то брокер продолжает получать сообщения от производителя и сохранять их. Таким образом, потребитель сможет обработать накопившиеся сообщения после восстановления. Чтобы предотвратить потерю сообщений при сбое производителя, можно включить режим транзакции для канала связи, что гарантирует доставку каждого сообщения брокеру. Однако важно учитывать, что это решение снижает пропускную способность системы.

Чтобы обеспечить доставку сообщений брокеру, не снижая производительность транзакциями, можно использовать паттерн «Подтверждение издателя» (https://oreil.ly/KMwTg). Указанные механизмы снижают риски в случае сбоя производителя или потребителя, но в случае сбоя брокера поддержание отказоустойчивости системы требует дополнительных мер. Так, последствия отказа узла брокера RabbitMQ будут менее критичными, если развертывать брокер в конфигурации высокой доступности (high-availability, HA), например путем кластеризации с зеркалированием очереди.

Слабая связанность клиентов

В архитектуре на основе брокера клиенты слабо связаны, поскольку взаимодействуют только с брокером, а не друг с другом напрямую. Такое разделение позволяет независимо развертывать, масштабировать и поддерживать клиенты. Например, клиенты могут быть развернуты на разных компьютерах, в отдельных дата-центрах или облачных средах (таких как AWS, Azure

[1] Рекомендуем также прочесть статьи Best Practices for Scaling RabbitMQ («Лучшие практики масштабирования RabbitMQ») в блоге ScaleGrid (https://oreil.ly/G69UV) и Cluster Sizing and Other Considerations («Выбор размера кластера и другие соображения») в блоге RabbitMQ (https://oreil.ly/ll0Ge).

или Google Cloud), а также написаны на разных языках программирования. Пока у каждого клиента есть возможность подключиться к брокеру, они могут обмениваться сообщениями.

Отзывчивость

Системы обмена сообщениями проектируются под задачу доставки сообщений с минимальной задержкой. Например, разработчики Kafka утверждают, что задержка при доставке сообщений составляет всего 2 миллисекунды[1].

Теперь обсудим недостатки API на основе обмена сообщениями.

Сложность

Эти API сложны в реализации. Архитекторам и разработчикам нужно понимать детали системы обмена сообщениями: что такое очереди, топики и сообщения; какие методы подтверждения использовать и когда; какие гарантии доставки сообщений выбрать. Кроме того, интеграция сервисов усложняется, если в других сервисах используется альтернативная технология обмена сообщениями.

Связанность

Слабая связанность позволяет развертывать системы независимо, но не означает, что компоненты полностью автономны. Производители и потребители связаны через сообщение, так что если в схеме сообщения происходит критическое изменение, то в системе случается сбой, препятствующий развертыванию. Производитель и потребитель связаны и через брокера, который выступает в роли единой точки отказа. Без брокера связь между производителем и потребителем невозможна.

Дублирование сообщений и отсутствие гарантий порядка

Дублирование — распространенная проблема в системах обмена сообщениями, которая может замедлить обработку сообщений и увеличить потребление памяти, если в приложениях-потребителях не реализована идемпотентная обработка[2]. Дублирование сообщений может возникать, например, из-за неожиданных перезапусков приложений.

Еще один недостаток систем обмена сообщениями — проблемы с порядком сообщений. По умолчанию очереди обеспечивают порядок FIFO («первый пришел, первый ушел»), однако некоторым приложениям требуются другие

---

[1] Официальное заявление, взятое с сайта Kafka (https://kafka.apache.org).

[2] Для решения этой проблемы в RabbitMQ есть плагин для дедупликации сообщений (https://oreil.ly/sK65V) и механизм дедупликации потоков сообщений (https://oreil.ly/9as6b).

гарантии порядка — например, LIFO («последний пришел, первый ушел») или более сложный порядок, соответствующий логике приложения. Это несоответствие дополнительно усложняет обработку сообщений в нужном порядке, особенно при перезапуске потребителей или возникновении дублирующихся сообщений.

Нарастание времени ожидания

Передача сообщения от отправителя через брокер к получателю требует времени. Если сообщение должно пройти несколько промежуточных узлов, то задержка увеличивается с каждым таким узлом. Например, для обработки онлайн-платежей лучше подходят синхронные API, где запрос блокируется до получения ответа (например, подтверждения успешной транзакции). Отправку уведомлений о покупках (например, по электронной почте или SMS), напротив, лучше обрабатывать асинхронно, так как ответ можно обработать без блокировки клиента.

Сложности отладки

Сообщения, которые проходят через систему, состоящую из нескольких компонентов, сложно отслеживать. Распространенное решение — создание уникального идентификатора корреляции (CID) для каждого сообщения и его поиск в логах системы. Другая сложность — порядок событий, который в асинхронных системах может быть недетерминированным.

Сложности в обработке ошибок

При синхронном взаимодействии об ошибке становится известно немедленно — либо о ней сообщает само приложение (ошибки `4xx`, `5xx`), либо истекает максимальное время ожидания. При асинхронном взаимодействии момент возникновения ошибки неизвестен, поэтому логику ее обработки приходится реализовывать дополнительно. Например, если ошибка возникает в приложении, то ее сложно обнаружить в последовательности событий, связанных с вызовом. Для хранения недоставленных сообщений можно реализовать специальную очередь (Dead Letter Queue, DLQ). Кроме того, после некоторых ошибок можно попытаться восстановить работу системы, отправив сообщение повторно. RabbitMQ также предоставляет шаблон обменника для обработки недоставленных сообщений (Dead Letter Exchange, DLX, https://oreil.ly/MQJhf).

Высокая стоимость

Реализация систем обмена сообщениями требует затрат, связанных с обучением персонала, созданием инфраструктуры, эксплуатацией, тестированием и интеграцией с другими системами. Однако представители некоторых компаний, где были внедрены такие системы, утверждают, что преимущества перевешивают затраты (https://oreil.ly/eSUgB).

Когда использовать API на основе обмена сообщениями

Теперь, когда мы обсудили преимущества и недостатки API на основе обмена сообщениями, перейдем к вопросу о том, когда его следует применять.

Этот API подходит для сценариев, в которых пользователь должен гарантированно получить сообщение. Речь идет, например, об уведомлениях о финансовых транзакциях или о коммуникации в сфере здравоохранения, где потеря даже одного сообщения может привести к серьезным убыткам или поставить под угрозу жизнь человека. Поэтому архитектура системы должна учитывать возможные сбои, чтобы обеспечить надежную передачу сообщений даже при самом худшем варианте развития событий.

API на основе обмена сообщениями позволяют создавать отказоустойчивые и масштабируемые системы, которые предназначены для обработки слабосвязанных, асинхронных рабочих процессов. Такие системы работают с нестабильной и с нарастающей нагрузкой, сохраняют высокий уровень доступности и гарантируют надежную доставку сообщений даже в периоды пиковых нагрузок или сбоев. Гибкость и стабильность работы системы при колебаниях нагрузки обеспечиваются путем масштабирования — горизонтального (через добавление экземпляров) или вертикального (через увеличение мощности ресурсов).

API на основе обмена сообщениями — хорошая альтернатива в тех случаях, когда синхронное взаимодействие с системой ухудшает пользовательский опыт (например, когда пользователю приходится ждать запуска рабочего процесса в системе, где нет индикатора выполнения). Так, если реализовать конвертацию видео через REST, то пользователь будет ждать в заблокированном состоянии, пока обработка видео не завершится. Если эту же задачу реализовать в системе на основе обмена сообщениями, то пользовательский опыт значительно улучшится. На весь процесс потребуется два сообщения: одно будет создано для запуска конвертации, а второе пользователь получит по завершении процесса. При этом, пока видео конвертируется, пользователь не блокируется и продолжает работу с приложением.

Кроме того, API на основе обмена сообщениями можно использовать, когда требуется высокая отзывчивость системы (https://oreil.ly/l5IBE). К системам, которые должны быстро реагировать на вводимые данные, относятся, например, системы высокочастотного трейдинга.

Наконец, API на основе обмена сообщениями полезны при реализации межсервисного взаимодействия. Для запуска определенного рабочего процесса можно использовать синхронные вызовы REST API. Как только сервис получает

REST-запрос, он помещает сообщение в брокер, который затем асинхронно взаимодействует с другими сервисами. Таким образом, синхронное действие трансформируется в асинхронное межсервисное взаимодействие.

Упражнения

1. Используйте Shodan (https://shodan.io), чтобы создать запрос для поиска серверов RabbitMQ, использующих механизм SASL `PLAIN`. Какое количество серверов потенциально может быть скомпрометировано? Подсказка: создайте учетную запись в Shodan и определите порт RabbitMQ по умолчанию.

> **SHODAN**
>
> Shodan (https://www.shodan.io/dashboard) — это поисковая машина для систем, доступных через интернет (например, маршрутизаторов, серверов и веб-камер). Поскольку система сканирует и индексирует публично открытые системы, то ее называют «поисковой системой хакеров».
>
> С помощью Shodan можно написать запрос, который просканирует интернет и проверит наличие систем с открытым определенным портом. Shodan может обнаружить системы, использующие конкретную технологию или даже принадлежащие определенной организации. Такой функционал возможен благодаря подробному списку фильтров (https://oreil.ly/VKZsw).
>
> Например, чтобы найти веб-камеры в Дании, можно использовать запрос `webcam country:DK`. Обнаружить веб-серверы по всему миру позволяет запрос с фильтром по порту (например, `port:80`).

2. С помощью утилиты `rabbitmqctl` создайте нового пользователя с именем `roger`, назначьте ему тег `developer` (метаданные, описывающие роль пользователя) и ограничьте ему доступ только к очереди `die` на виртуальном хосте `/` с правами на запись (`write`) и чтение (`read`). Подсказка: для выполнения этого упражнения используйте подкоманды `rabbitmqctl`: `add_user`, `set_user_tags` и `set_permissions`.
3. Запустите пример с очередью задач из этой главы и протестируйте пропускную способность брокера RabbitMQ с помощью программы `perf-test` (https://oreil.ly/w7JzM). Подсказка: какую пропускную способность вы получите при выполнении следующей команды?

```
docker run -it --rm --name perf-test --network rabbitmq \
       pivotalrabbitmq/perf-test:latest \
       --uri amqp://bugs:bunny@broker \
       --time 10 --json-body --size 1024
```

Резюме

Мы начали эту главу с обсуждения основных понятий: сообщение, очередь и маршрутизация на основе содержания. С помощью технологии обмена сообщениями RabbitMQ мы реализовали очередь задач. Кроме того, мы рассмотрели другие распространенные паттерны обмена сообщениями: «Издатель — подписчик», «Маршрутизация», «Топики» и «Запрос — ответ». Далее с помощью AsyncAPI мы задокументировали API и выполнили версионирование сообщений. Затем мы разобрали способы обеспечения безопасности RabbitMQ, защитили сообщения с помощью TLS и SASL и ограничили доступ пользователей с помощью списков контроля доступа.

В заключение мы обсудили преимущества и недостатки API на основе обмена сообщениями.

Об авторах

Лукаш Дыновски — независимый консультант; опыт работы в ИТ — более десяти лет. Занимал различные должности, среди которых фулстек-разработчик, DevOps-инженер и архитектор ПО. Кроме того, имеет опыт работы в академической среде и является соавтором ряда научных публикаций. Побеждал в конкурсе O'Reilly Software Architectural Katas, а позже входил в состав жюри. Входит в топ 0,1 % разработчиков на Stack Overflow. Лукаш — Docker Captain и заядлый техноэнтузиаст. Наконец, он любящий муж и отец двух прекрасных детей.

Марцин Дулак более 20 лет работает в сфере ИТ. Он создавал алгоритмы квантового моделирования, в том числе портирование кода для высокопроизводительных вычислительных архитектур. Кроме того, имеет опыт внедрения и эксплуатации локальной и облачной инфраструктуры. На данный момент сфера интересов Марцина — веб-приложения. Любит проводить время дома с семьей и кошками.

Иллюстрация на обложке

Птица на обложке — патагонский певчий пересмешник *(Mimus patagonicus)*, обитающий в Патагонии, на территории Аргентины и Чили.

Хотя название вида явно указывает на способность к звукоподражанию, у патагонских пересмешников есть свои уникальные напевы (с элементами подражания). Эти птицы называются пересмешниками из-за того, что принадлежат к семейству *Mimidae (пересмешниковые)*, представители которого известны своими вокальными способностями.

Яркая внешняя черта патагонского пересмешника — его светлые «брови». Их роль в эволюции вида не установлена, однако можно предположить, что светлые полосы помогают птицам распознавать сородичей и маскироваться от хищников. Патагонские пересмешники умеют летать, но, в отличие от других видов, бóльшую часть времени проводят на земле и здесь же добывают пищу. В брачный сезон состав рациона меняется с фруктов и ягод на насекомых, богатых белком, который необходим птенцам. Семьи патагонских пересмешников отличаются крепкими связями и остаются вместе гораздо дольше, чем семьи других видов. По этой же причине патагонские пересмешники проявляют выраженное территориальное поведение.

Многие животные на обложках книг O'Reilly находятся под угрозой исчезновения, и все они важны для нашей планеты. Иллюстрация на обложке создана Карен Монтгомери на основе старинной гравюры из издания Lydekker's Royal Natural History. Дизайн серии разработали Эди Фридман, Элли Фолькхаузен и Карен Монтгомери.

Лукаш Дыновски, Марцин Дулак

Стили API. Проектирование и внедрение

Перевел с английского Е. Бартов

Изготовитель: ТОО «Спринт Бук». Место нахождения и фактический адрес:
010000, Казахстан, город Астана, район Алматы, проспект Ракымжан Кошкарбаев, д. 10/1, н. п. 18.

Дата изготовления: 07.2026. Наименование: книжная продукция. Срок годности: не ограничен.

Подписано в печать 18.06.26. Формат 70x100/16. Бумага офсетная. Усл. п. л. 33,540. Тираж 700. Заказ 0000.

Read IT Club

Комьюнити рецензентов
и переводчиков ИТ-литературы

Миссия участников клуба — обеспечить высокое качество профессиональной переводной литературы в России. «Книжные дебагеры» проверяют корректность терминологии и подписей на схемах и иллюстрациях, чтобы сделать книги более понятными русскоязычному читателю. Стать участником Read IT Club может любой ИТ-специалист, готовый поделиться опытом с сообществом.



Майк Амундсен

RESTFUL WEB API ПАТТЕРНЫ И ПРАКТИКИ



В настоящее время многие организации создают и поддерживают приложения, основанные на сторонних сервисах. Разработчикам программного обеспечения и архитекторам часто приходится заниматься координацией и поддержкой приложений на основе API существующих сервисов, работающих за пределами их экосистемы. Найдите проверенные рецепты, которые помогут объединить множество разрозненных компонентов в вашей сети.

Майк Амундсен предлагает пошаговые решения для поиска, связывания и поддержки приложений, созданных сторонними разработчиками. Неважно, делаете ли вы мобильные приложения, ориентированные на людей, или мощные решения, основанные на межмашинных взаимодействиях, вы найдете правила, процедуры, команды и протоколы объединения отдельных API, способствующие их безопасной, масштабируемой и надежной совместной работе.

SPRINT book



Билл Любанович

FASTAPI: ВЕБ-РАЗРАБОТКА НА PYTHON

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

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

SPRINT book