

Сергей Константинов. API.

yatwirl@gmail.com · linkedin.com/in/twirl · twirl.substack.com

«API-first» подход — одна из самых горячих тем в разработке программного обеспечения в наше время. Многие компании начали понимать, что API выступает мультипликатором их возможностей — но также умножает и допущенные ошибки.

Эта книга написана для того, чтобы поделиться опытом и изложить лучшие практики разработки API. Книга состоит из шести разделов, посвящённых:

- проектированию API,
- паттернам дизайна API,
- поддержанию обратной совместимости,
- HTTP API и архитектурным принципам REST,
- SDK и UI-библиотекам,
- продуктовому управлению API.

Иллюстрации и вдохновение: Maria Konstantinova · art.mari ka.



Это произведение доступно по лицензии Creative Commons «Attribution-NonCommercial» («Атрибуция — Некоммерческое использование») 4.0 Всемирная.

Исходный код доступен на github.com/twirl/The-API-Book

Поделиться: [facebook](#) · [twitter](#) · [linkedin](#) · [reddit](#)

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

- Глава 1. О структуре этой книги
- Глава 2. Определение API
- Глава 3. Критерии качества API
- Глава 4. Выбор подхода к разработке API
- Глава 5. API-first подход
- Глава 6. Обратная совместимость
- Глава 7. О версионировании
- Глава 8. Условные обозначения и терминология

РАЗДЕЛ I. ПРОЕКТИРОВАНИЕ API

- Глава 9. Пирамида контекстов API
- Глава 10. Определение области применения
- Глава 11. Разделение уровней абстракции
- Глава 12. Разграничение областей ответственности
- Глава 13. Описание конечных интерфейсов
- Глава 14. Приложение к разделу I. Модельный API

РАЗДЕЛ II. ПАТТЕРНЫ ДИЗАЙНА API

- Глава 15. О паттернах проектирования в контексте API
- Глава 16. Аутентификация партнёров и авторизация вызовов API
- Глава 17. Стратегии синхронизации
- Глава 18. Слабая консистентность
- Глава 19. Асинхронность и управление временем
- Глава 20. Списки и организация доступа к ним
- Глава 21. Двунаправленные потоки данных. Push и poll-модели
- Глава 22. Мультиплексирование сообщений. Асинхронная обработка событий
- Глава 23. Атомарность массовых изменений
- Глава 24. Частичные обновления
- Глава 25. Деградация и предсказуемость

РАЗДЕЛ III. ОБРАТНАЯ СОВМЕСТИМОСТЬ

- Глава 26. Постановка проблемы обратной совместимости
- Глава 27. О ватерлинии айсберга
- Глава 28. Расширение через абстрагирование
- Глава 29. Сильная связность и сопутствующие проблемы
- Глава 30. Слабая связность
- Глава 31. Интерфейсы как универсальный паттерн
- Глава 32. Блокнот душевного покоя

РАЗДЕЛ IV. HTTP API И АРХИТЕКТУРНЫЕ ПРИНЦИПЫ REST

- Глава 33. О концепции HTTP API. Парадигмы разработки клиент-серверного взаимодействия

Глава 34. Преимущества и недостатки HTTP API в сравнении с альтернативными технологиями

Глава 35. Мифология REST

Глава 36. Составляющие HTTP запросов и их семантика

Глава 37. Организация HTTP API согласно принципам REST

Глава 38. Разработка номенклатуры URL ресурсов. CRUD-операции

Глава 39. Работа с ошибками в HTTP API

Глава 40. Заключительные положения и общие рекомендации

РАЗДЕЛ V. SDK И UI

Глава 41. Терминология. Обзор технологий разработки SDK

Глава 42. SDK: проблемы и решения

Глава 43. Проблемы встраивания UI-компонентов

Глава 44. Декомпозиция UI-компонентов

Глава 45. MV*-фреймворки

Глава 46. Backend-Driven UI

Глава 47. Разделяемые ресурсы и асинхронные блокировки

Глава 48. Вычисляемые свойства

Глава 49. Заключение

РАЗДЕЛ VI. API КАК ПРОДУКТ

Глава 50. Продукт API

Глава 51. Бизнес-модели API

Глава 52. Формирование продуктового видения

Глава 53. Взаимодействие с разработчиками

Глава 54. Взаимодействие с бизнес-аудиторией

Глава 55. Линейка сервисов API

Глава 56. Ключевые показатели эффективности API

Глава 57. Идентификация пользователей и борьба с фрэном

Глава 58. Технические способы борьбы с несанкционированным доступом к API

Глава 59. Поддержка пользователей API

Глава 60. Документация

Глава 61. Тестовая среда

Глава 62. Управление ожиданиями

ВВЕДЕНИЕ

Глава 1. О структуре этой книги

Книга, которую вы держите в руках, посвящена разработке API как отдельной инженерной задаче. Хотя многое из того, о чём мы будем говорить, применимо к написанию любых программ, наша цель состояла прежде всего в описании тех проблем и подходов к их решению, которые характерны именно для предметной области API.

Мы ожидаем, что читатель обладает навыками разработки программного обеспечения, и не даём подробных определений и объяснений в отношении понятий, которыми разработчик должен по нашему мнению владеть. Без такого рода навыков читать последний раздел вам будет достаточно некомфортно (а остальные разделы — ещё сложнее) — за что мы искренне просим прощения, но не видим другого способа написать эту книгу, не раздув её объём ещё в три раза.

Настоящая книга состоит из введения и шести больших разделов. Первые три из них («Проектирование API», «Паттерны дизайна API» и «Обратная совместимость») являются полностью абстрактными и не привязанными ни к каким конкретным технологиям — мы рассчитываем, что они окажутся полезными тем читателям, которые хотят выстроить системное понимание того, что такое архитектура API и как разрабатывать сложные иерархии интерфейсов. Предложенный подход, как мы надеемся, помогает спроектировать API сверху вниз, от сырой идеи до конкретной реализации.

Четвёртый и пятый раздел посвящены вполне конкретным технологиям — разработке HTTP API («REST») и SDK (в основном речь пойдёт о библиотеках визуальных компонент).

Наконец, в рамках шестого раздела, наименее технического из всех, мы обсудим API как продукт и сфокусируемся на не-разработческих аспектах существования API: изучение рынка, позиционирование сервиса, взаимодействие с потребителями, KPI команды и так далее. Мы настаиваем здесь, что последний раздел одинаково важен и для менеджеров, и для программистов, поскольку технические продукты можно развивать только в тесном взаимодействии продуктовой и технической команд.

На этом переходим к делу.

Глава 2. Определение API

Прежде чем говорить о разработке API, необходимо для начала договориться о том, что же такое API. Энциклопедия скажет нам, что API — это программный интерфейс приложений. Это точное определение, но бессмысленное. Примерно как определение человека по Платону: «двуногое без перьев» — определение точное, но никоим образом не дающее нам представление о том, чем на самом деле человек примечателен. (Да и не очень-то и точное: Диоген Синопский как-то ощипал петуха и заявил, что это человек Платона; пришлось дополнить определение уточнением «с плоскими ногтями».)

Что же такое API по смыслу, а не по формальному определению?

Вероятно, вы сейчас читаете эту книгу посредством браузера. Чтобы браузер смог отобразить эту страничку, должны корректно отработать: разбор URL согласно спецификации; служба DNS; соединение по протоколу TLS; передача данных по протоколу HTTP; разбор HTML-документа; разбор CSS-документа; корректный рендеринг HTML+CSS.

Но это только верхушка айсберга. Для работы HTTP необходима корректная работа всего сетевого стека, который состоит из 4-5, а то и больше, протоколов разных уровней. Разбор HTML-документа производится согласно сотням различных спецификаций. Рендеринг документа обращается к нижележащему API операционной системы, а также напрямую к API видеокарты. И так далее, и тому подобное — вплоть до того, что наборы команд современных CISC-процессоров имплементируются поверх API микрокоманд.

Иными словами, десятки, если не сотни различных API должны корректно отработать для выполнения базовых действий типа просмотра web-страницы; без надёжной работы каждого из них современные информационные технологии попросту не могли бы существовать.

API — это обязательство. Формальное обязательство связывать между собой различные программируемые контексты.

Когда автора этой книги просят привести пример хорошего API, он обычно показывает фотографию древнеримского акведука:



Древнеримский акведук Пон-дю-Гар. Построен в I веке н.э. Image Credit: igorelick @ pixabay

- он связывает между собой две области;
- обратная совместимость нарушена ноль раз за последние две тысячи лет.

Отличие древнеримского акведука от хорошего API состоит лишь в том, что API предлагает *программный* контракт. Для связывания двух областей необходимо написать некоторый код. Цель настоящей книги состоит в том, чтобы помочь вам разработать API, так же хорошо выполняющий свою задачу, как и древнеримский акведук.

Акведук хорошо иллюстрирует и другую проблему разработки API: вашими пользователями являются инженеры. Вы не поставляете воду напрямую потребителю: к вашей инженерной мысли подключаются заказчики путём пристройки к ней каких-то своих инженерных конструкций. С одной стороны, вы можете обеспечить водой гораздо больше людей, нежели если бы вы сами подводили трубы к каждому крану. С другой — качество инженерных решений заказчика вы не можете контролировать, и проблемы с водой, вызванные некомпетентностью подрядчика, неизбежно будут валить на вас.

Именно поэтому проектирование API налагает на вас несколько большую ответственность. API является как мультипликатором ваших возможностей, так и мультипликатором ваших ошибок.

Глава 3. Критерии качества API

Прежде чем переходить к изложению рекомендаций по проектированию архитектуры API, нам следует определиться с тем, что мы считаем качественным API, и какую пользу мы получаем от того, что наш API «качественный». Достаточно очевидно, что качество API определяется в первую очередь тем, насколько хорошо он помогает разработчикам решать стоящие перед ними задачи. (Оставим за скобками ситуации, когда вендор API преследует какие-то свои неявные цели, отличные от предоставления полезного продукта.)

Как же дизайн API может помочь разработчику? Очень просто: API должен решать задачи *максимально удобно и понятно*. Путь разработчика от формулирования своей задачи до написания работающего кода должен быть максимально коротким. Это, в том числе, означает, что:

- из структуры вашего API должно быть максимально очевидно, как решить ту или иную задачу:
 - разработчику должно быть достаточно одного взгляда на документацию, чтобы понять, с помощью каких сущностей следует решать поставленную задачу;
- API должен быть читаемым:
 - в идеале, разработчик, просто глядя в номенклатуру методов, сразу пишет правильный код, не углубляясь в детали (особенно — детали реализации!);
 - немаловажно уточнить, что из интерфейсов объектов должно быть понятно не только решение задачи, но и возможные ошибки и исключения;
- API должен быть консистентен:
 - при разработке новой функциональности, т.е. при обращении к каким-то незнакомым сущностям в API, разработчик может действовать по аналогии с уже известными ему концепциями API, и его код будет работать;
 - желательно при этом, чтобы API соответствовал принципам и правилам платформы и используемого фреймворка.

Однако статическое удобство и понятность API — это относительно простая часть. В конце концов, никто не стремится специально сделать API нелогичным и нечитаемым — всегда при разработке мы начинаем с каких-то понятных базовых концепций. При минимальном опыте проектирования сложно сделать ядро API, не удовлетворяющее критериям очевидности, читаемости и консистентности.

Проблемы возникают, когда мы начинаем API развивать. Добавление новой функциональности рано или поздно приводит к тому, что некогда простой и понятный API становится наслоением разных концепций, а попытки сохранить обратную совместимость приводят к нелогичным, неочевидным и попросту плохим решениям. Отчасти это связано так же и с тем, что невозможно обладать полным знанием о будущем: ваше понимание о «правильном» API тоже будет меняться со временем, как в объективной части (какие задачи и каким образом решает API), так и в субъективной — что такое очевидность, читабельность и консистентность для вашего API.

Принципы, которые мы будем излагать ниже, во многом ориентированы именно на то, чтобы API правильно развивался во времени и не превращался в нагромождение разнородных неконсистентных интерфейсов. Важно понимать, что такой подход тоже не бесплатен: необходимость держать в голове варианты развития событий и закладывать возможность изменений в API означает избыточность интерфейсов и возможно излишнее абстрагирование. И то, и другое, помимо прочего, усложняет и работу программиста, пользующегося вашим API. **Закладывание перспектив «на будущее» имеет смысл, только если это будущее у API есть, иначе это попросту оверинжиниринг.**

Глава 4. Выбор подхода к разработке API

Вернёмся к нашей аналогии API как акведука, соединяющего два различных контекста. Пытаясь сделать подключение к нашему сооружению удобным для потребителей, мы также сталкиваемся и с другой стороной вопроса: а как им (потребителям) было бы удобно? Как они привыкли? Нет ли в нашей предметной области каких-то распространённых стандартов подключения к водопроводу?

В абсолютном большинстве случаев такие стандарты есть: кто-то уже разрабатывал похожие API раньше. Чем дальше отстоят друг от друга два контекста, тем большее количество разнообразных абстракций и фреймворков работы с ними будет существовать.

Использование привычных механизмов — важная составляющая качества API. Там, где есть устоявшийся стандарт взаимодействия (скажем, протокол TCP/IP в компьютерных сетях), изобретать свой собственный стандарт рекомендуется только в тех случаях, когда вы стопроцентно уверены, что преимущества нового подхода будут настолько очевидны, что вам простят необходимость изучить новую технологию для работы с API.

Во многих областях, однако, подобной определённости нет; напротив, существуют многочисленные конкурирующие друг с другом парадигмы разработки API, и вам придётся сделать выбор в пользу одной из них (или в пользу разработки собственного подхода). Две такие области мы рассмотрим в разделах IV и V настоящей книги:

- выбор парадигмы организации клиент-серверного взаимодействия (REST API, RPC, GraphQL) — в главе «[Преимущества и недостатки HTTP API в сравнении с альтернативными технологиями](#)»;
- выбор подхода к написанию UI компонентов — в главе «[Терминология. Обзор технологий разработки SDK](#)».

Глава 5. API-first подход

На сегодняшний день всё больше и больше ИТ-компаний понимают и принимают важность концепции «API-first», т.е. парадигмы разработки ПО, в которой главной составляющей является разработка API.

Следует, однако, различать API-first подход в продуктовом и техническом смысле.

Первое означает, что при разработке некоторого сервиса сначала как первый (и иногда единственный) шаг разрабатывается API к нему, и мы обсудим этот подход в разделе «API как продукт».

Если же мы говорим об API-first подходе в техническом смысле, то мы имеем в виду следующее: **контракт, т.е. обязательство связывать программные контексты, предшествует реализации и определяет её**. Конкретнее, речь идёт о двух принципах:

- контракт разрабатывается и фиксируется в виде спецификации до того, как функциональность непосредственно реализована;
- если обнаруживается несоответствие контракта и его имплементации, изменения вносятся в имплементацию, а не в контракт.

Здесь под спецификацией мы понимаем формальное машиночитаемое описание контракта на одном из языков определения интерфейсов (Interface Definition Language, IDL) — например, в виде Swagger/OpenAPI спецификации или .proto-файла.

Оба вышеуказанных принципа фактически утверждают приоритет интересов партнёра-разработчика API:

- первый принцип позволяет партнёру разрабатывать код по формальной спецификации, не требуя согласования с провайдером API;
 - появляется возможность использовать генерацию кода по спецификации, что может существенно упрощать и автоматизировать разработку;
 - партнёрский код может быть написан вообще в отсутствие доступа к API;
- второй принцип позволяет не требовать изменений в коде партнёра, если обнаружен ошибка в реализации API.

Таким образом, API-first подход — это некоторая гарантия для ваших потребителей. Но, как легко заметить, работает эта гарантия только в условиях качественного дизайна API: если в фазе разработки спецификации были допущены неисправимые ошибки, то второй принцип придётся нарушить.

Глава 6. Обратная совместимость

Обратная совместимость — это некоторая *временная* характеристика качества вашего API. Именно необходимость поддержания обратной совместимости отличает разработку API от разработки программного обеспечения вообще.

Разумеется, обратная совместимость не абсолютна. В некоторых предметных областях выпуск новых обратно несовместимых версий API является вполне рутинной процедурой. Тем не менее, каждый раз, когда выпускается новая обратно несовместимая версия API, всем разработчикам приходится инвестировать какое-то ненулевое количество усилий, чтобы адаптировать свой код к новой версии. В этом плане выпуск новых версий API является некоторого рода «налогом» на потребителей — им нужно тратить вполне осязаемые деньги только для того, чтобы их продукт продолжал работать.

Конечно, крупные компании с прочным положением на рынке могут позволить себе такой налог взимать. Более того, они могут вводить какие-то санкции за отказ от перехода на новые версии API, вплоть до отключения приложений.

С нашей точки зрения, подобное поведение ничем не может быть оправдано. Избегайте скрытых налогов на своих пользователей. Если вы можете не ломать обратную совместимость — не ломайте её.

Да, безусловно, поддержка старых версий API — это тоже своего рода налог. Технологии меняются, и, как бы хорошо ни был спроектирован ваш API, всего предусмотреть невозможно. В какой-то момент ценой поддержки старых версий становится невозможность предоставлять новую функциональность и поддерживать новые платформы, и выпустить новую версию всё равно придётся. Однако вы по крайней мере сможете убедить своих потребителей в необходимости перехода.

Более подробно о жизненном цикле API и политиках выпуска новых версий будет рассказано в разделе II.

Глава 7. О версионировании

Здесь и далее мы будем придерживаться принципов версионирования Semantic Versioning (semver)¹.

1. Версия API задаётся тремя цифрами вида 1.2.3.
2. Первая цифра (мажорная версия) увеличивается при обратно несовместимых изменениях в API.
3. Вторая цифра (минорная версия) увеличивается при добавлении новой функциональности с сохранением обратной совместимости.
4. Третья цифра (патч) увеличивается при выпуске новых версий, содержащих только исправление ошибок.

Выражения «мажорная версия API» и «версия API, содержащая обратно несовместимые изменения функциональности» тем самым следует считать эквивалентными.

Обычно (но не обязательно) устанавливается, что на последнюю стабильную версию API можно сослаться как по полной версии (1.2.3), так и по усечённой (1.2 или просто 1). Некоторые системы поддерживают и более сложные схемы указания подключаемой версии (например, ^1.2.3 читается как «подключить последнюю стабильную версию, обратно совместимую с версией 1.2.3») или дополнительные шорткаты (например 1.2-beta для подключения бета-версии API семейства 1.2). В настоящей книге мы будем в основном использовать обозначения вида v1 (v2, v3 и так далее) для обозначения последнего стабильного релиза API семейства 1.x.x.

Более подробно о смысле и политиках такого версионирования читайте в главе [«Постановка проблемы обратной совместимости»](#).

Примечания

¹ Semantic Versioning

<https://semver.org/>

Глава 8. Условные обозначения и терминология

В мире разработки программного обеспечения существует множество различных парадигм разработки, adeptы которых зачастую настроены весьма воинственно по отношению к adeptам других парадигм. Поэтому при написании этой книги мы намеренно избегали слов «метод», «объект», «функция» и так далее, используя нейтральный термин «сущность», под которым понимается некоторая атомарная единица функциональности: класс, метод, объект, монада, прототип (нужное подчеркнуть).

Для составных частей сущности, к сожалению, достаточно нейтрального термина нам придумать не удалось, поэтому мы используем слова «поля» и «методы».

Большинство примеров API в общих разделах будут даны в виде абстрактных обращений по HTTP-протоколу к некоторой специфической именованной функции API («эндпойнту») с передачей данных в формате JSON. Это некоторая условность, которая помогает описать концепции, как нам кажется, максимально понятно. Вместо GET /v1/orders вполне может быть вызов метода orders.get(), локальный или удалённый; вместо JSON может быть любой другой формат данных. Смысл утверждений от этого не меняется.

Рассмотрим следующую запись:

```
// Описание метода
POST /v1/bucket/{id}/some-resource
/{resource_id}
X-Idempotency-Token: <токен идемпотентности>
{
    // Это односторонний комментарий
    "some_parameter": "value",
    ...
}
→ 404 Not Found
Cache-Control: no-cache
{
    /* А это многострочный
       комментарий */
    "error_message":
        "Длинное сообщение,
         которое приходится
         разбивать на строки"
}
```

Её следует читать так:

- клиент выполняет POST-запрос к ресурсу `/v1/bucket/{id}/some-resource`, где `{id}` заменяется на некоторый идентификатор bucket-а (при отсутствии уточнений подстановки вида `{something}` следует относить к ближайшему термину слева);
- запрос сопровождается (помимо стандартных заголовков, которые мы опускаем) дополнительным заголовком `X-Idempotency-Token`;
- фразы в угловых скобках (`<токен идемпотентности>`) описывают семантику значения сущности (поля, заголовка, параметра);
- в качестве тела запроса передаётся JSON, содержащий поле `some_parameter` со значением `value` и ещё какие-то поля, которые для краткости опущены (что показано многоточием);
- в ответ (индицируется стрелкой →) сервер возвращает статус `404 Not Found`; статус может быть опущен (отсутствие статуса следует трактовать как `200 OK`);
- в ответе также могут находиться дополнительные заголовки, на которые мы обращаем внимание;
- телом ответа является JSON, состоящий из единственного поля `error_message`; отсутствие значения поля означает, что его значением является именно то, что в этом поле и ожидается — в данном случае какое-то сообщение об ошибке
- если какой-то токен оказывается слишком длинным, мы будем переносить его на следующую строку, используя символ «`\` для индикации переноса.

Здесь термин «клиент» означает «приложение, установленное на устройстве пользователя, использующее рассматриваемый API». Приложение может быть как нативным, так и веб-приложением. Термины «агент» и «юзер-агент» являются синонимами термина «клиент».

Ответ (частично или целиком) и тело запроса могут быть опущены, если в контексте обсуждаемого вопроса их содержание не имеет значения.

Возможна сокращённая запись вида: `POST /some-resource {..., "some_parameter", ...} → { "operation_id" }`; тело запроса и/или ответа может опускаться аналогично полной записи.

Чтобы сослаться на это описание будут использоваться выражения типа «метод `POST /v1/bucket/{id}/some-resource`» или, для простоты, «метод `some-resource`» или «метод `bucket/some-resource`» (если никаких других `some-resource` в контексте главы не упоминается и перепутать не с чем).

Помимо HTTP API-нотации мы будем активно использовать С-подобный псевдокод — точнее будет сказать, JavaScript или Python-подобный, поскольку нотации типов мы будем опускать. Мы предполагаем, что подобного рода императивные конструкции достаточно читабельны, и не будем здесь описывать грамматику подробно. Примеры в виде HTTP API-нотации призваны иллюстрировать дизайн *контрактов*, т.е. показывать, как мы разрабатываем API. Примеры в псевдокоде обычно отражают, какой код напишут разработчики для работы с API, или как бы мы сами написали SDK на основе такого контракта.

РАЗДЕЛ I. ПРОЕКТИРОВАНИЕ API

Глава 9. Пирамида контекстов API

Подход, который мы используем для проектирования, состоит из четырёх шагов:

- определение области применения;
- разделение уровней абстракции;
- разграничение областей ответственности;
- описание конечных интерфейсов.

Этот алгоритм строит API сверху вниз, от общих требований и сценариев использования до конкретной номенклатуры сущностей; фактически, двигаясь этим путём, вы получите на выходе готовый API — чем этот подход и ценен.

Может показаться, что наиболее полезные советы приведены в последнем разделе, однако это не так; цена ошибки, допущенной на разных уровнях весьма различна. Если исправить плохое именование довольно просто, то исправить неверное понимание того, зачем вообще нужен API, практически невозможно.

NB. Здесь и далее мы будем рассматривать концепции разработки API на примере некоторого гипотетического API заказа кофе в городских кофейнях. На всякий случай сразу уточним, что пример является синтетическим; в реальной ситуации, если бы такой API пришлось проектировать, он, вероятно, был бы совсем не похож на наш выдуманный пример.

Глава 10. Определение области применения

Ключевой вопрос, который вы должны задать себе четыре раза, выглядит так: какую проблему мы решаем? Задать его следует четыре раза с ударением на каждом из четырёх слов.

1. *Какую* проблему мы решаем? Можем ли мы чётко описать, в какой ситуации гипотетическим потребителям-разработчикам нужен наш API?
2. *Какую* *проблему* мы решаем? А мы правда уверены, что описанная выше ситуация — проблема? Действительно ли кто-то готов платить (в прямом и переносном смысле) за то, что ситуация будет как-то автоматизирована?
3. *Какую* проблему *мы* решаем? Действительно ли решение этой проблемы находится в нашей компетенции? Действительно ли мы находимся в той позиции, чтобы решить эту проблему?
4. *Какую* проблему мы *решаем*? Правда ли, что решение, которое мы предлагаем, действительно решает проблему? Не создаём ли мы на её месте другую проблему, более сложную?

Итак, предположим, что мы хотим предоставить API автоматического заказа кофе в городских кофейнях. Попробуем применить к нему этот принцип.

1. Зачем кому-то может потребоваться API для приготовления кофе? В чём неудобство заказа кофе через интерфейс, человек-человек или человек-машина? Зачем нужна возможность заказа машина-машина?
 - Возможно, мы хотим решить проблему выбора и знания? Чтобы человек наиболее полно знал о доступных ему здесь и сейчас опциях.
 - Возможно, мы оптимизируем время ожидания? Чтобы человеку не пришлось ждать, пока его заказ готовится.
 - Возможно, мы хотим минимизировать ошибки? Чтобы человек получил именно то, что хотел заказать, не потеряв информацию при разговорном общении либо при настройке незнакомого интерфейса кофемашины.

Вопрос «зачем» — самый важный из тех вопросов, которые вы должны задавать себе. Не только глобально в отношении целей всего проекта, но и локально в отношении каждого кусочка функциональности. **Если вы не можете коротко и понятно ответить на вопрос «зачем эта сущность нужна» — значит, она не нужна.**

Здесь и далее предположим (в целях придания нашему примеру глубины и некоторой упоротости), что мы оптимизируем все три фактора в порядке убывания важности.

2. Правда ли решаемая проблема существует? Действительно ли мы наблюдаем неравномерную загрузку кофейных автоматов по утрам? Правда ли люди страдают от того, что не могут найти поблизости нужный им латте с ореховым сиропом? Действительно ли людям важны те минуты, которые они теряют, стоя в очередях?
3. Действительно ли мы обладаем достаточным ресурсом, чтобы решить эту проблему? Есть ли у нас доступ к достаточному количеству кофемашин и клиентов, чтобы обеспечить работоспособность системы?
4. Наконец, правда ли мы решим проблему? Как мы поймём, что оптимизировали перечисленные факторы?

На все эти вопросы, в общем случае, простого ответа нет. В идеале ответы на эти вопросы должны даваться с цифрами в руках. Сколько конкретно времени тратится неоптимально, и какого значения мы рассчитываем добиться, располагая какой плотностью кофемашин? Заметим также, что в реальной жизни просчитать такого рода цифры можно в основном для проектов, которые пытаются влезть на уже устоявшийся рынок; если вы пытаетесь сделать что-то новое, то, вероятно, вам придётся ориентироваться в основном на свою интуицию.

Почему API?

Поскольку наша книга посвящена не просто разработке программного обеспечения, а разработке API, то на все эти вопросы мы должны взглянуть под другим ракурсом: а почему для решения этих задач требуется именно API, а не просто программное обеспечение? В нашем вымышленном примере мы должны спросить себя: зачем нам нужно предоставлять сервис для других разработчиков, чтобы они могли готовить кофе своим клиентам, а не сделать своё приложение для конечного потребителя?

Иными словами, должна иметься веская причина, по которой два домена разработки ПО должны быть разделены: есть оператор(ы), предоставляющий API; есть оператор(ы), предоставляющий сервисы пользователям. Их интересы в чём-то различны настолько, что объединение этих двух ролей в одном лице нежелательно. Более подробно мы изложим причины и мотивации делать именно API в разделе III.

Заметим также следующее: вы должны браться делать API тогда и только тогда, когда в ответе на второй вопрос написали «потому что в этом состоит наша экспертиза». Разрабатывая API, вы занимаетесь некоторой мета-разработкой: вы пишете ПО для того, чтобы другие могли разрабатывать ПО для решения задачи пользователя. Не обладая экспертизой в обоих этих доменах (API и конечные продукты) написать хороший API сложно.

Для нашего умозрительного примера предположим, что в недалеком будущем произошло разделение рынка кофе на две группы игроков: одни предоставляют само железо, кофейные аппараты, а другие имеют доступ к потребителю — примерно как это произошло, например, с рынком авиабилетов, где есть собственно авиакомпании, осуществляющие перевозку, и сервисы планирования путешествий, где люди выбирают варианты перелётов. Мы хотим агрегировать доступ к железу, чтобы владельцы приложений могли встраивать заказ кофе.

Что и как

Закончив со всеми теоретическими упражнениями, мы должны перейти непосредственно к дизайну и разработке API, имея понимание по двум пунктам.

1. Что конкретно мы делаем.
2. Как мы это делаем.

В случае нашего кофе-примера мы:

1. Предоставляем сервисам с большой пользовательской аудиторией API для того, чтобы их потребители могли максимально удобно для себя заказать кофе.
2. Для этого мы абстрагируем за нашим HTTP API доступ к «железу» и предоставим методы для выбора вида напитка и места его приготовления и для непосредственно исполнения заказа.

Глава 11. Разделение уровней абстракции

«Разделите свой код на уровни абстракции» — пожалуй, самый общий совет для разработчиков программного обеспечения. Однако будет вовсе не преувеличением сказать, что изоляция уровней абстракции — самая сложная задача, стоящая перед разработчиком API.

Прежде чем переходить к теории, следует чётко сформулировать, *зачем* нужны уровни абстракции и каких целей мы хотим достичь их выделением.

Вспомним, что программный продукт — это средство связи контекстов, средство преобразования терминов и операций одной предметной области в другую. Чем дальше друг от друга эти области отстоят — тем большее число промежуточных передаточных звеньев нам придётся ввести. Вернёмся к нашему примеру с кофейнями. Какие уровни сущностей мы видим?

1. Мы готовим с помощью нашего API *заказ* — один или несколько стаканов кофе — и взимаем за это плату.
2. Каждый стакан кофе приготовлен по определённому *рецепту*, что подразумевает наличие разных ингредиентов и последовательности выполнения шагов приготовления.
3. Напиток готовится на конкретной физической *кофемашине*, располагающейся в какой-то точке пространства.

Каждый из этих уровней задаёт некоторый срез нашего API, с которым будет работать потребитель. Выделяя иерархию абстракций мы, прежде всего, стремимся снизить связность различных сущностей нашего API. Это позволит нам добиться нескольких целей.

1. Упрощение работы разработчика и легкость обучения: в каждый момент времени разработчику достаточно будет оперировать только теми сущностями, которые нужны для решения его задачи; и наоборот, плохо выстроенная изоляция приводит к тому, что разработчику нужно держать в голове множество концепций, не имеющих прямого отношения к решаемой задаче.
2. Возможность поддерживать обратную совместимость; правильно подобранные уровни абстракции позволяют нам в дальнейшем добавлять новую функциональность, не меняя интерфейс.

3. Поддержание интероперабельности. Правильно выделенные низкоуровневые абстракции позволяют нам адаптировать наш API к другим платформам, не меняя высокоуровневый интерфейс.

Допустим, мы имеем следующий интерфейс:

```
// возвращает рецепт лунго  
GET /v1/recipes/lungo
```

```
// размещает на указанной кофемашине  
// заказ на приготовление лунго  
// и возвращает идентификатор заказа  
POST /v1/orders  
{  
    "coffee_machine_id",  
    "recipe": "lungo"  
}
```

```
// возвращает состояние заказа  
GET /v1/orders/{id}
```

И зададимся вопросом, каким образом разработчик определит, что заказ клиента готов. Допустим, мы сделаем так: добавим в рецепт лунго эталонный объём, а в состояние заказа — количество уже налитого кофе. Тогда разработчику нужно будет проверить совпадение этих двух цифр, чтобы убедиться, что кофе готов.

```
GET /v1/recipes/lungo  
→  
{  
    "volume": "100ml"  
}
```

```
GET /v1/orders/{id}  
→  
{  
    "volume": "80ml"  
}
```

Такое решение выглядит интуитивно плохим, и это действительно так: оно нарушает все вышеперечисленные принципы.

Во-первых, для решения задачи «заказать лунго» разработчику нужно обратиться к сущности «рецепт» и выяснить, что у каждого рецепта есть объём. Далее, нужно принять концепцию, что приготовление кофе заканчивается в тот момент, когда объём сравнялся с эталонным. Нет никакого способа об этой конвенции догадаться: она неочевидна и её нужно найти в документации. При этом никакой пользы для разработчика в этом знании нет.

Во-вторых, мы автоматически получаем проблемы, если захотим варировать размер кофе. Допустим, в какой-то момент мы захотим предоставить пользователю выбор, сколько конкретно миллилитров лунго он желает. Тогда нам придётся проделать один из следующих трюков.

Вариант 1: мы фиксируем список допустимых объёмов и заводим фиктивные рецепты типа `/recipes/small-lungo`, `recipes/large-lungo`. Почему фиктивные? Потому что рецепт один и тот же, меняется только объём. Нам придётся либо тиражировать одинаковые рецепты, отличающиеся только объёмом, либо вводить какое-то «наследование» рецептов, чтобы можно было указать базовый рецепт и только переопределить объём.

Вариант 2: мы модифицируем интерфейс, объявляя объём кофе, указанный в рецепте, значением по умолчанию; при размещении заказа мы разрешаем указать объём, отличный от эталонного:

```
POST /v1/orders
{
  "coffee_machine_id",
  "recipe": "lungo",
  "volume": "800ml"
}
```

Для таких кофе произвольного объёма нужно будет получать требуемый объём не из `GET /v1/recipes`, а из `GET /v1/orders`. Сделав так, мы сразу получаем клубок из связанных проблем:

- разработчик, которому придётся поддерживать эту функциональность, имеет высокие шансы сделать ошибку: добавив поддержку произвольного объёма кофе в код, работающий с `POST /v1/orders` нужно не забыть переписать код проверки готовности заказа;

- мы получим классическую ситуацию, когда одно и то же поле (объём кофе) значит разные вещи в разных интерфейсах. В GET /v1/recipes поле «объём» теперь значит «объём, который будет запрошен, если не передать его явно в POST /v1/orders»; переименовать его в «объём по умолчанию» уже не получится, с этой проблемой теперь придётся жить.

Мы получим:

```
GET /v1/orders/{id}
→
{
  ...
  // Это текущий объём кофе,
  // который наследует имя поля
  // из старого интерфейса
  "volume": "80ml",
  // а это объём кофе, запрошенный
  // пользователем
  "volume_requested": "800ml"
}
```

В-третьих, вся эта схема полностью неработоспособна, если разные модели кофемашин производят лунго разного объёма. Для решения задачи «объём лунго зависит от вида машины» нам придётся сделать совсем неприятную вещь: сделать рецепт зависимым от id машины. Тем самым мы начнём активно смешивать уровни абстракции: одной частью нашего API (рецептов) станет невозможно пользоваться без другой части (информации о кофемашине). Что немаловажно, от разработчиков потребуется изменить логику своего приложения: если раньше они могли предлагать сначала выбрать объём, а потом кофемашину, то теперь им придётся полностью изменить этот шаг.

Хорошо, допустим, мы поняли, как сделать плохо. Но как же тогда сделать *хорошо*? Разделение уровней абстракции должно происходить вдоль трёх направлений:

1. От сценариев использования к их внутренней реализации: высокоравневые сущности и номенклатура их методов должны напрямую отражать сценарии использования API; низкоравневый — отражать декомпозицию сценариев на составные части.

2. От терминов предметной области пользователя к терминам предметной области исходных данных — в нашем случае от высокоуровневых понятий «рецепт», «заказ», «кофейня» к низкоуровневым «температура напитка» и «координаты кофемашины»
3. Наконец, от структур данных, в которых удобно оперировать пользователю к структурам данных, максимально приближенных к «сырым» — в нашем случае от «лунго» и «сети кофеен "Ромашка"» — к сырьем байтовым данным, описывающим состояние кофемашины марки «Доброе утро» в процессе приготовления напитка.

Чем дальше находятся друг от друга программные контексты, которые соединяет наш API — тем более глубокая иерархия сущностей должна получиться у нас в итоге.

В нашем примере с определением готовности кофе мы явно пришли к тому, что нам требуется промежуточный уровень абстракции:

- с одной стороны, «заказ» не должен содержать информацию о датчиках и сенсорах кофемашины;
- с другой стороны, кофемашина не должна хранить информацию о свойствах заказа (да и вероятно её API такой возможности и не предоставляет).

Наивный подход в такой ситуации — искусственно ввести некий промежуточный уровень абстракции, «передаточное звено», который переформулирует задачи одного уровня абстракции в другой. Например, введём сущность `task` вида:

```
{  
    "volume_requested": "800ml",  
    "volume_prepared": "200ml",  
    "readiness_policy": "check_volume",  
    "ready": false,  
    "coffee_machine_id": "  
    "operation_state": {  
        "status": "executing",  
        "operations": [  
            // описание операций, запущенных на  
            // физической кофемашине  
        ]  
    }  
}
```

Таким образом, сущность «заказ» будет только хранить ссылки на рецепт и исполняемую задачу и не вторгаться в «чужие» уровни абстракции:

```
GET /v1/orders/{id}
→
{
  "recipe": "lungo",
  "task": {
    "id": <task id>
  }
}
```

Мы называем этот подход «наивным» не потому, что он неправильный; напротив, это вполне логичное решение «по умолчанию», если вы на данном этапе ещё не знаете или не понимаете, как будет выглядеть ваш API. Проблема его в том, что он умозрительный: он не добавляет понимания того, как устроена предметная область.

Хороший разработчик в нашем примере должен спросить: хорошо, а какие вообще говоря существуют варианты? Как можно определять готовность напитка? Если вдруг окажется, что сравнение объёмов — единственный способ определения готовности во всех без исключения кофемашинах, то почти все рассуждения выше — неверны: можно совершенно спокойно включать в интерфейсы определение готовности кофе по объёму, т.к. никакого другого и не существует. Прежде, чем что-то абстрагировать — надо представлять, что мы, собственно, абстрагируем.

Для нашего примера допустим, что мы сели изучать спецификации API кофемашин и выяснили, что существует принципиально два класса устройств:

- кофемашины с предустановленными программами, которые умеют готовить заранее прошитые N видов напитков, и мы можем управлять только какими-то параметрами напитка (скажем, объёмом напитка, вкусом сиропа и видом молока); у таких машин отсутствует доступ к внутренним функциям и датчикам, но зато машина умеет через API сама отдавать статус приготовления напитка;
- кофемашины с предустановленными функциями типа «смолоть такой-то объём кофе», «пролить N миллилитров воды», «взбить молочную пену» и т.д.: у таких машин отсутствует понятие «программа приготовления», но есть доступ к микрокомандам и датчикам.

Предположим, для большей конкретности, что эти два класса устройств поставляются вот с таким физическим API.

- Машины с предустановленными программами:

```
// Возвращает список
// предустановленных программ
GET /programs
→
{
    // идентификатор программы
    "program": 1,
    // вид кофе
    "type": "lungo"
}
```

```
// Запускает указанную
// программу на исполнение
// и возвращает статус исполнения
POST /execute
{
    "program": 1,
    "volume": "200ml"
}
→
{
    // Уникальный идентификатор задания
    "execution_id": "01-01",
    // Идентификатор
    // исполняемой программы
    "program": 1,
    // Запрошенный объём напитка
    "volume": "200ml"
}
```

```
// Отменяет текущую программу
POST /cancel
```

```
// Возвращает статус исполнения
// Формат аналогичен
// формату ответа `POST /execute`
GET /execution/status
```

NB. На всякий случай отметим, что данный API нарушает множество описанных нами принципов проектирования, начиная с отсутствия версионирования; он приведен в таком виде по двум причинам: (1) чтобы мы могли показать, как спроектировать API более удачно; (2) скорее всего, в реальной жизни вы получите именно такой API от производителей

кофемашин, и это ещё довольно вменяемый вариант.

- Машины с предустановленными функциями:

```
// Возвращает список
// доступных функций
GET /functions
→
{
  "functions": [
    {
      // Тип операции
      // * set_cup – поставить стакан
      // * grind_coffee – смолоть кофе
      // * pour_water – пролить воду
      // * discard_cup – утилизировать стакан
      "type": "set_cup",
      // Допустимые аргументы
      // для каждой операции
      // Для простоты ограничимся
      // одним аргументом:
      // * volume
      //   – объём стакана,
      //   кофе или воды
      "arguments": [ "volume" ]
    },
    ...
  ]
}
```

```
// Запускает на исполнение функцию
// с передачей указанных
// значений аргументов
POST /functions
{
  "type": "set_cup",
  "arguments": [
    {
      "name": "volume",
      "value": "300ml"
    }
  ]
}
```

```

// Возвращает статусы датчиков
GET /sensors
→
{
  "sensors": [
    {
      // Допустимые значения
      // * cup_volume
      //   - объём установленного стакана
      // * ground_coffee_volume
      //   - объём смолотого кофе
      // * cup_filled_volume
      //   - объём напитка в стакане
      "type": "cup_volume",
      "value": "200ml"
    },
    ...
  ]
}

```

NB. Пример нарочно сделан умозрительным для моделирования ситуации, описанной в начале главы: для определения готовности напитка нужно сличить объём налитого с эталоном.

Теперь картина становится более явной: нам нужно абстрагировать работу с кофемашиной так, чтобы наш «уровень исполнения» в API предоставлял общие функции (такие, как определение готовности напитка) в унифицированном виде. Важно отметить, что с точки зрения разделения абстракций два этих вида кофемашин сами находятся на разных уровнях: первые предоставляют API более высокого уровня, нежели вторые; следовательно, и «ветка» нашего API, работающая со вторым видом машин, будет более «развесистой».

Следующий шаг, необходимый для отделения уровней абстракции — необходимо понять, какую функциональность нам, собственно, необходимо абстрагировать. Для этого нам необходимо обратиться к задачам, которые решает разработчик на уровне работы с заказами, и понять, какие проблемы у него возникнут в случае отсутствия нашего слоя абстракции.

1. Очевидно, что разработчику хочется создавать заказ унифицированным образом — перечислить высокоуровневые параметры заказа (вид напитка, объём и специальные требования, такие как вид сиропа или молока) — и не думать о том, как на конкретной машине исполнить этот заказ.
2. Разработчику надо понимать состояние исполнения — готов ли заказ или нет; если не готов — когда ожидать готовность (и надо ли её ожидать вообще в случае ошибки исполнения).

3. Разработчику нужно уметь соотносить заказ с его положением в пространстве и времени — чтобы показать потребителю, когда и как нужно заказ забрать.
4. Наконец, разработчику нужно выполнять атомарные операции — например, отменять заказ.

Заметим, что API первого типа гораздо ближе к потребностям разработчика, нежели API второго типа. Концепция атомарной «программы» гораздо ближе к удобному для разработчика интерфейсу, нежели работа с сырьими наборами команд и данными сенсоров. В API первого типа мы видим только две проблемы:

- отсутствие явного соответствия программ и рецептов; идентификатор программы по-хорошему вообще не нужен при работе с заказами, раз уже есть понятие рецепта;
- отсутствие явного статуса готовности.

С API второго типа всё гораздо хуже. Главная проблема, которая нас ожидает — отсутствие «памяти» исполняемых действий. API функций и сенсоров полностью *stateless*; это означает, что мы даже не знаем, кем, когда и в рамках какого заказа была запущена текущая функция.

Таким образом, нам нужно внедрить два новых уровня абстракции.

1. Уровень управления исполнением, предоставляющий унифицированный интерфейс к атомарным программам. «Унифицированный интерфейс» в данном случае означает, что, независимо от того, на какого рода кофемашине готовится заказ, разработчик может рассчитывать на:

- единую номенклатуру статусов и других высокоуровневых параметров исполнения (например, ожидаемого времени готовности заказа или возможных ошибок исполнения);
- единую номенклатуру доступных методов (например, отмены заказа) и их одинаковое поведение.

2. Уровень программы исполнения. Для API первого типа он будет представлять собой просто обёртку над существующим API программ; для API второго типа концепцию «рантайма» программ придётся полностью имплементировать нам.

Что это будет означать практически? Разработчик по-прежнему будет создавать заказ, оперируя только высокоуровневыми терминами:

```
POST /v1/orders
{
  "coffee_machine_id",
  "recipe": "lungo",
  "volume": "800ml"
}
→
{ "order_id" }
```

Имплементация функции POST /orders проверит все параметры заказа, заблокирует его стоимость на карте пользователя, сформирует полный запрос на исполнение и обратится к уровню исполнения. Сначала необходимо подобрать правильную программу исполнения:

```
POST /v1/program-matcher
{ "recipe", "coffee-machine" }
→
{ "program_id" }
```

Получив идентификатор программы, нужно запустить её на исполнение:

```
POST /v1/programs/{id}/run
{
  "order_id",
  "coffee_machine_id",
  "parameters": [
    {
      "name": "volume",
      "value": "800ml"
    }
  ]
}
→
{ "program_run_id" }
```

Обратите внимание, что во всей этой цепочке вообще никак не участвует тип API кофемашины — собственно, ровно для этого мы и абстрагировали. Мы могли бы сделать интерфейсы более конкретными, разделив функциональность run и match для разных API, т.е. ввести раздельные endpoint-ы:

- POST /v1/program-matcher/{api_type}
- POST /v1/{api_type}/programs/{id}/run

Достоинством такого подхода была бы возможность передавать в `match` и `run` не унифицированные наборы параметров, а только те, которые имеют значение в контексте указанного типа API. Однако в нашем дизайне API такой необходимости не прослеживается. Обработчик `run` сам может извлечь нужные параметры из мета-информации о программе и выполнить одно из двух действий:

- вызвать `POST /execute` физического API кофемашины с передачей внутреннего идентификатора программы — для машин, поддерживающих API первого типа;
- инициировать создание рантайма для работы с API второго типа.

Уровень рантаймов API второго типа, исходя из общих соображений, будет скорее всего непубличным, и мы плюс-минус свободны в его имплементации. Самым простым решением будет реализовать виртуальную `state`-машину, которая создаёт «рантайм» (т.е. `stateful` контекст исполнения) для выполнения программы и следит за его состоянием.

```
POST /v1/runtimes
{
  "coffee_machine",
  "program",
  "parameters"
}
→
{ "runtime_id", "state" }
```

Здесь `program` будет выглядеть примерно так:

```
{
  "program_id",
  "api_type",
  "commands": [
    {
      "sequence_id",
      "type": "set_cup",
      "parameters"
    },
    ...
  ]
}
```

A `state` вот так:

```

{
    // Статус рантайма
    // * "pending" – ожидание
    // * "executing" – исполнение команды
    // * "ready_waiting" – напиток готов
    // * "finished" – все операции завершены
    "status": "ready_waiting",
    // Текущая исполняемая команда
    // (необязательное)
    "command_sequence_id",
    // Чем закончилось исполнение программы
    // (необязательное)
    // * "success"
    //   – напиток приготовлен и выдан
    // * "terminated"
    //   – исполнение остановлено
    // * "technical_error"
    //   – ошибка при приготовлении
    // * "waiting_time_exceeded"
    //   – готовый заказ был
    //     утилизирован, его не забрали
    "resolution": "success",
    // Значения всех переменных,
    // включая состояние сенсоров
    "variables"
}

```

NB: в имплементации связки `orders → match → run → runtimes` можно пойти одним из двух путей:

- либо обработчик POST `/orders` сам обращается к доступной информации о рецепте, кофемашине и программе и формирует stateless-запрос, в котором указаны все нужные данные (тип API кофемашины и список команд в частности);
- либо в запросе содержатся только идентификаторы, и следующие обработчики в цепочке сами обращаются за нужными данными через какие-то внутренние API.

Оба варианта имеют право на жизнь; какой из них выбрать зависит от деталей реализации.

Изоляция уровней абстракции

Важное свойство правильно подобранных уровней абстракции, и отсюда требование к их проектированию — это требование изоляции: **взаимодействие возможно только между сущностями соседних уровней абстракции**. Если при проектировании выясняется, что для выполнения того или иного действия требуется «перепрыгнуть» уровень абстракции, это явный признак того, что в проекте допущены ошибки.

Вернёмся к нашему примеру. Каким образом будет работать операция получения статуса заказа? Для получения статуса будет выполнена следующая цепочка вызовов:

- пользователь вызовет метод GET /v1/orders;
- обработчик orders выполнит операции своего уровня ответственности (проверку авторизации, в частности), найдёт идентификатор program_run_id и обратится к API программ runs/{program_run_id};
- обработчик runs в свою очередь выполнит операции своего уровня (в частности, проверит тип API кофемашины) и в зависимости от типа API пойдёт по одной из двух веток исполнения:
 - либо вызовет GET /execution/status физического API кофемашины, получит объём кофе и сличит с эталонным;
 - либо обратится к GET /v1/runtimes/{runtime_id}, получит state.status и преобразует его к статусу заказа;
- в случае API второго типа цепочка продолжится: обработчик GET /runtimes обратится к физическому API GET /sensors и произведёт ряд манипуляций: сопоставит объём стакана / молотого кофе / налитой воды с запрошенным и при необходимости изменит состояние и статус.

NB: слова «цепочка вызовов» не следует воспринимать буквально. Каждый уровень может быть технически организован по-разному:

- можно явно проксировать все вызовы по иерархии;
- можно кэшировать статус своего уровня и обновлять его по получении обратного вызова или события. В частности, низкоуровневый цикл исполнения рантайма для машин второго рода очевидно должен быть независимым и обновлять свой статус в фоне, не дожинаясь явного запроса статуса.

Обратите внимание, что здесь фактически происходит следующее: на каждом уровне абстракции есть какой-то свой статус (заказа, рантайма, сенсоров), который сформулирован в терминах соответствующей этому уровню абстракции предметной области. Запрет «перепрыгивания» уровней приводит к тому, что нам необходимо независимо дублировать статус на каждом уровне.

Рассмотрим теперь, каким образом через наши уровни абстракции «прорастёт» операция отмены заказа. В этом случае цепочка вызовов будет такой:

- пользователь вызовет метод POST /v1/orders/{id}/cancel;
- обработчик метода произведёт операции в своей зоне ответственности:
 - проверит авторизацию;
 - решит денежные вопросы — нужно ли делать рефанд;
 - найдёт идентификатор program_run_id и обратится к обработчику runs/{program_run_id}/cancel;
- обработчик runs/cancel произведёт операции своего уровня (в частности, установит тип API кофемашины) и в зависимости от типа API пойдёт по одной из двух веток исполнения:
 - либо вызовет POST /execution/cancel физического API кофемашины;
 - либо вызовет POST /v1/runtimes/{id}/terminate;
- во втором случае цепочка продолжится, обработчик terminate изменит внутреннее состояние:
 - изменит resolution на "terminated"
 - запустит команду "discard_cup".

Имплементация модифицирующих операций (таких, как cancel) требует более продвинутого обращения с уровнями абстракции по сравнению с немодифицирующими вызовами типа GET /status. Два важных момента, на которые здесь стоит обратить внимание:

1. На каждом уровне абстракции понятие «отмена заказа» переформулируется:

- на уровне orders это действие фактически распадается на несколько «отмен» других уровней: нужно отменить блокировку денег на карте и отменить исполнение заказа;

- при этом на физическом уровне API второго типа «отмена» как таковая не существует: «отмена» — это исполнение команды `discard_cup`, которая на этом уровне абстракции ничем не отличается от любых других команд.

Промежуточный уровень абстракции как раз необходим для того, чтобы переход между «отменами» разных уровней произошёл гладко, без необходимости перепрыгивания через уровни абстракции.

2. С точки зрения верхнеуровневого API отмена заказа является терминальным действием, т.е. никаких последующих операций уже быть не может; а с точки зрения низкоуровневого API обработка заказа продолжается, т.к. нужно дождаться, когда стакан будет утилизирован, и после этого освободить кофемашину (т.е. разрешить создание новых рантаймов на ней). Это вторая задача для уровня исполнения: связывать оба статуса, внешний (заказ отменён) и внутренний (исполнение продолжается).

Может показаться, что соблюдение правила изоляции уровней абстракции является избыточным и заставляет усложнять интерфейс. И это в действительности так: важно понимать, что никакая гибкость, логичность, читабельность и расширяемость не бывает бесплатной. Можно построить API так, чтобы он выполнял свою функцию с минимальными накладными расходами, по сути — дать интерфейс к микроконтроллерам кофемашины. Однако пользоваться им будет крайне неудобно, и расширяемость такого API будет нулевой.

Выделение уровней абстракции, прежде всего, *логическая* процедура: как мы объясняем себе и разработчику, из чего состоит наш API. **Абстрагируемая дистанция между сущностями существует объективно**, каким бы образом мы ни написали конкретные интерфейсы. Наша задача состоит только лишь в том, чтобы эта дистанция была разделена на уровни *явно*. Чем более неявно разведены (или, хуже того, перемешаны) уровни абстракции, тем сложнее будет разобраться в вашем API, и тем хуже будет написан использующий его код.

Потоки данных

Полезное упражнение, позволяющее рассмотреть иерархию уровней абстракции API — исключить из рассмотрения все частности и построить — в голове или на бумаге — дерево потоков данных: какие данные протекают через объекты вашего API и как видоизменяются на каждом этапе.

Это упражнение не только полезно для проектирования, но и, помимо прочего, является единственным способом развивать большие (в смысле номенклатуры объектов) API. Человеческая память не безгранична, и любой активно развивающийся проект достаточно быстро станет настолько большим, что удержать в голове всю иерархию сущностей со всеми полями и методами станет невозможно. Но вот держать в уме схему потоков данных обычно вполне возможно — или, во всяком случае, получается держать в уме на порядок больший фрагмент дерева сущностей API.

Какие потоки данных мы имеем в нашем кофейном API?

1. Данные с сенсоров — объёмы кофе / воды / стакана. Это низший из доступных нам уровней данных, здесь мы не можем ничего изменить или переформулировать.
2. Непрерывный поток данных сенсоров мы преобразуем в дискретные статусы исполнения команд, вводя в него понятия, не существующие в предметной области. API кофемашины не предоставляет нам понятий «кофе наливается» или «стакан ставится» — это наше программное обеспечение трактует поступающие потоки данных от сенсоров, вводя новые понятия: если наблюдаемый объём (кофе или воды) меньше целевого — значит, процесс не закончен; если объём достигнут — значит, необходимо сменить статус исполнения и выполнить следующее действие.

Важно отметить, что мы не просто вычисляем какие-то новые параметры из имеющихся данных сенсоров: мы сначала создаём новый кортеж данных более высокого уровня — «программа исполнения» как последовательность шагов и условий — и инициализируем его начальные значения. Без этого контекста определить, что собственно происходит с кофемашиной невозможно.

3. Обладая логическими данными о состоянии исполнения программы, мы можем (вновь через создание нового, более высокоуровневого контекста данных!) свести данные от двух типов API к единому формату исполнения операции создания напитка и её логических параметров: целевой рецепт, объём, статус готовности.

Таким образом, каждый уровень абстракции нашего API соответствует какому-то обобщению и обогащению потока данных, преобразованию его из терминов нижележащего (и вообще говоря бесполезного для потребителя) контекста в термины вышестоящего контекста.

Дерево можно развернуть и в обратную сторону.

1. На уровне заказа мы задаём его логические параметры: рецепт, объём, место исполнения и набор допустимых статусов заказа.
2. На уровне исполнения мы обращаемся к данным уровня заказа и создаём более низкоуровневый контекст: программа исполнения в виде последовательности шагов, их параметров и условий перехода от одного шага к другому и начальное состояние.
3. На уровне рантайма мы обращаемся к целевым значениям (какую операцию выполнить и какой целевой объём) и преобразуем их в набор микрокоманд API кофемашины и набор статусов исполнения каждой команды.

Если обратиться к описанному в начале главы «плохому» решению (предполагающему самостоятельное определение факта готовности заказа разработчиком), то мы увидим, что и с точки зрения потоков данных происходит смешение понятий:

- с одной стороны, в контексте заказа оказываются данные (объём кофе), «просочившиеся» откуда-то с физического уровня; тем самым, уровни абстракции непоправимо смешиваются без возможности их разделить;
- с другой стороны, сам контекст заказа неполноценный: он не задаёт новых мета-переменных, которые отсутствуют на более низких уровнях абстракции (статус заказа), не инициализирует их и не предоставляет правил работы.

Более подробно о контекстах данных мы поговорим в разделе II. Здесь же ограничимся следующим выводом: потоки данных и их преобразования можно и нужно рассматривать как некоторый срез, который, с одной стороны, помогает нам правильно разделить уровни абстракции, а с другой — проверить, что наши теоретические построения действительно работают так, как нужно.

Глава 12. Разграничение областей ответственности

Исходя из описанного в предыдущей главе, мы понимаем, что иерархия абстракций в нашем гипотетическом проекте должна выглядеть примерно так:

- пользовательский уровень (те сущности, с которыми непосредственно взаимодействует пользователь, сформулированы в понятных для него терминах; например, заказы и виды кофе);
- уровень исполнения программ (те сущности, которые отвечают за преобразование заказа в машинные термины);
- уровень рантайма для API второго типа (сущности, отвечающие за state-машину выполнения заказа).

Теперь нам необходимо определить ответственность каждой сущности: в чём смысл её существования в рамках нашего API, какие действия можно выполнять с самой сущностью, а какие — делегировать другим объектам. Фактически, нам нужно применить «зачем-принцип» к каждой отдельной сущности нашего API.

Для этого нам нужно пройти по нашему API и сформулировать в терминах предметной области, что представляет из себя каждый объект. Напомню, что из концепции уровней абстракции следует, что каждый уровень иерархии — это некоторая собственная промежуточная предметная область, ступенька, по которой мы переходим от описания задачи в терминах одного связываемого контекста («заказанный пользователем лунго») к описанию в терминах второго («задание кофемашине на выполнение указанной программы»).

В нашем умозрительном примере получится примерно так:

1. Сущности уровня пользователя (те сущности, работая с которыми, разработчик непосредственно решает задачи пользователя).
 - Заказ `order` — описывает некоторую логическую единицу взаимодействия с пользователем. Заказ можно:
 - создавать;
 - проверять статус;
 - получать;
 - отменять.

- Рецепт `recipe` — описывает «идеальную модель» вида кофе, его потребительские свойства. Рецепт в данном контексте для нас неизменяемая сущность, которую можно только просмотреть и выбрать.
 - Кофемашина `coffee-machine` — модель объекта реального мира. Из описания кофемашины мы, в частности, должны извлечь её положение в пространстве и предоставляемые опции (о чём подробнее поговорим ниже).
2. Сущности уровня управления исполнением (те сущности, работая с которыми, можно непосредственно исполнить заказ).
- Программа `program` — описывает некоторый план исполнения для конкретной кофемашины. Программы можно только просмотреть.
 - Селектор программ `programs/matcher` — позволяет связать рецепт и программу исполнения, т.е. фактически выяснить набор данных, необходимых для приготовления конкретного рецепта на конкретной кофемашине. Селектор работает только на выбор нужной программы.
 - Запуск программы `programs/run` — конкретный факт исполнения программы на конкретной кофемашине. Запуски можно:
 - инициировать (создавать);
 - проверять состояние запуска;
 - отменять.
3. Сущности уровня программ исполнения (те сущности, работая с которыми, можно непосредственно управлять состоянием кофемашины через API второго типа).
- Рантайм `runtime` — контекст исполнения программы, т.е. состояние всех переменных. Рантаймы можно:
 - создавать;
 - проверять статус;
 - терминировать.

Если внимательно посмотреть на каждый объект, то мы увидим, что, в итоге, каждый объект оказался в смысле своей ответственности составным. Например, `program` будет оперировать данными высшего уровня (рецепт и кофемашина), дополняя их терминами своего уровня (идентификатор запуска). Это совершенно нормально: API должен связывать контексты.

Сценарии использования

На этом уровне, когда наш API уже в целом понятно устроен и спроектирован, мы должны поставить себя на место разработчика и попробовать написать код. Наша задача: взглянуть на номенклатуру сущностей и понять, как ими будут пользоваться.

Представим, что нам поставили задачу, пользуясь нашим кофейным API, разработать приложение для заказа кофе. Какой код мы напишем?

Очевидно, первый шаг — нужно предоставить пользователю возможность выбора, чего он, собственно хочет. И первый же шаг обнажает неудобство использования нашего API: никаких методов, позволяющих пользователю что-то выбрать в нашем API нет. Разработчику придётся сделать что-то типа такого:

- получить все доступные рецепты из GET /v1/recipes;
- получить список всех кофемашин из GET /v1/coffee-machines;
- самостоятельно выбрать нужные данные.

В псевдокоде это будет выглядеть примерно вот так:

```
// Получить все доступные рецепты
let recipes =
    api.getRecipes();
// Получить все доступные кофемашины
let coffeeMachines =
    api.getCoffeeMachines();
// Построить пространственный индекс
let coffeeMachineRecipesIndex =
    buildGeoIndex(recipes, coffeeMachines);
// Выбрать кофемашины,
// соответствующие запросу пользователя
let matchingCoffeeMachines =
    coffeeMachineRecipesIndex.query(
        parameters,
        { "sort_by": "distance" }
    );
// Наконец, показать
// предложения пользователю
app.display(matchingCoffeeMachines);
```

Как видите, разработчику придётся написать немало лишнего кода (это не упоминая о сложности имплементации геопространственных индексов!). Притом, учитывая наши наполеоновские планы по покрытию нашим API всех кофемашин мира, такой алгоритм выглядит заведомо бессмысленной тратой ресурсов на получение списков и поиск по ним.

Напрашивается добавление нового эндпойнта поиска. Для того, чтобы разработать этот интерфейс, нам придётся самим встать на место UX-дизайнера и подумать, каким образом приложение будет пытаться заинтересовать пользователя. Два сценария довольно очевидны:

- показать ближайшие кофейни и виды предлагаемого кофе в них («service discovery»-сценарий) — для пользователей-новичков, или просто людей без определённых предпочтений;
- показать ближайшие кофейни, где можно заказать конкретный вид кофе — для пользователей, которым нужен конкретный напиток.

Тогда наш новый интерфейс будет выглядеть примерно вот так:

```
POST /v1/offers/search
{
    // опционально
    "recipes": ["lungo", "americano"],
    "position": <географические координаты>,
    "sort_by": [
        { "field": "distance" }
    ],
    "limit": 10
}
→
{
    "results": [
        "coffee_machine",
        "place",
        "distance",
        "offer"
    ],
    "cursor"
}
```

Здесь:

- `offer` — некоторое «предложение»: на каких условиях можно заказать запрошенные виды кофе, если они были указаны, либо какое-то маркетинговое предложение — цены на самые популярные / интересные напитки, если пользователь не указал конкретные рецепты для поиска;

- `place` — место (кафе, автомат, ресторан), где находится машина; мы не вводили эту сущность ранее, но, очевидно, пользователю потребуются какие-то более понятные ориентиры, нежели географические координаты, чтобы найти нужную кофемашину.

NB. Мы могли бы не добавлять новый эндпойнт, а обогатить существующий `/coffee-machines`. Однако такое решение выглядит менее семантично: не стоит в рамках одного интерфейса смешивать способ перечисления объектов по порядку и по релевантности запросу, поскольку эти два вида ранжирования обладают существенно разными свойствами и сценариями использования. К тому же, обогащение поиска «предложениями» скорее выводит эту функциональность из неймспейса «кофемашины»: для пользователя всё-таки первичен факт получения предложения приготовить напиток на конкретных условиях, и кофемашина — лишь одно из них, не самое важное.

NB. На самом деле, наличие идентификатора кофе-машины в интерфейсах само по себе нарушает принцип изоляции уровней абстракции. Эта функциональность должна быть организована более сложно: кофейни должны распределять поступающие заказы по свободным кофемашинам, и только тип кофемашины (если кофейня оперирует несколькими одновременно) является значимой частью предложения. Мы сознательно допускаем это упрощение (пользователь сам выбирает кофемашину), чтобы не перегружать наш учебный пример.

Вернёмся к коду, который напишет разработчик. Теперь он будет выглядеть примерно так:

```
// Ищем предложения,  
// соответствующие запросу пользователя  
let offers = api.offerSearch(parameters);  
// Показываем пользователю  
app.display(offers);
```

Хелперы

Методы, подобные только что изобретённому нами `offers/search`, принято называть *хеллерами*. Цель их существования — обобщить понятные сценарии использования API и облегчить их. Под «облегчить» мы имеем в виду не только сократить многословность («бойлерплейт»), но и помочь разработчику избежать частых проблем и ошибок.

Рассмотрим, например, вопрос стоимости заказа. Наша функция поиска возвращает какие-то «предложения» с ценой. Но ведь цена может меняться: в «счастливый час» кофе может стоить меньше. Разработчик может ошибиться в имплементации этой функциональности трижды:

- кэшировать на клиентском устройстве результаты поиска слишком долго (в результате цена всегда будет неактуальна),
- либо, наоборот, слишком часто вызывать операцию поиска только лишь для того, чтобы актуализировать цены, создавая лишнюю нагрузку на сеть и наш сервер;
- создать заказ, не проверив актуальность цены (т.е. фактически обмануть пользователя, списав не ту стоимость, которая была показана).

Для решения третьей проблемы мы могли бы потребовать передать в функцию создания заказа его стоимость, и возвращать ошибку в случае несовпадения суммы с актуальной на текущий момент. (Более того, конечно же в любом API, работающем с деньгами, это нужно делать *обязательно*.) Но это не поможет с первым вопросом: гораздо более удобно с точки зрения UX не отображать ошибку в момент нажатия кнопки «разместить заказ», а всегда показывать пользователю актуальную цену.

Для решения этой проблемы мы можем поступить следующим образом: снабдить каждое предложение идентификатором, который необходимо указывать при создании заказа.

```
{  
  "results": [  
    {  
      "coffee_machine",  
      "place",  
      "distance",  
      "offer": {  
        "id",  
        "price",  
        "currency_code",  
        // Указываем дату и время,  
        // до наступления которых  
        // предложение является актуальным  
        "valid_until"  
      }  
    },  
    {"cursor"}  
  ]  
}
```

Поступая так, мы не только помогаем разработчику понять, когда ему надо обновить цены, но и решаем UX-задачу: как показать пользователю, что «счастливый час» скоро закончится. Идентификатор предложения может при этом быть `stateful` (фактически, аналогом сессии пользователя) или `stateless` (если мы точно знаем, до какого времени действительна цена, мы можем просто закодировать это время в идентификаторе).

Альтернативно, кстати, можно было бы разделить функциональность поиска по заданным параметрам и получения предложений, т.е. добавить эндпойнт, только актуализирующий цены в конкретных кофейнях.

Обработка ошибок

Сделаем ещё один небольшой шаг в сторону улучшения жизни разработчика. А каким образом будет выглядеть ошибка «неверная цена»?

```
POST /v1/orders  
{ "offer_id", ...}  
→ 409 Conflict  
{  
  "message": "Неверная цена"  
}
```

С формальной точки зрения такой ошибки достаточно: пользователю будет показано сообщение «неверная цена», и он должен будет повторить заказ. Конечно, это будет очень плохое решение с точки зрения UX (пользователь ведь не совершил никаких ошибок, да и толку ему от этого сообщения никакого).

Главное правило интерфейсов ошибок в API таково: из содержимого ошибки клиент должен в первую очередь понять, что ему делать с этой ошибкой. Содержимое ошибки должно отвечать на следующие вопросы:

1. На чьей стороне ошибка — сервера или клиента?

В HTTP API для индикации источника проблемы традиционно используются коды ответа: 4xx проблема клиента, 5xx проблема сервера (за исключением «статуса неопределённости» 404).

2. Если проблема на стороне сервера — то имеет ли смысл повторить запрос, и, если да, то когда?

3. Если проблема на стороне клиента — является ли она устранимой или нет?

Проблема с неправильной ценой является устранимой: клиент может получить новое предложение цены и создать заказ с ним. Однако если ошибка возникает из-за неправильно написанного клиентского кода — устранить её не представляется возможным, и не нужно заставлять пользователя повторно нажимать «создать заказ»: этот запрос не завершится успехом никогда.

Здесь и далее неустранимые проблемы мы индицируем кодом 400 Bad Request, а устранимые — кодом 409 Conflict.

4. Если проблема устранимая, то какого рода? Очевидно, клиент не сможет устранить проблему, о которой не знает, для каждой такой ошибки должен быть написан код (в нашем случае — перезапроса цены), т.е. должен существовать какой-то описанный набор таких ошибок.

5. Если один и тот же род ошибок возникает вследствие некорректной передачи какого-то одного или нескольких разных параметров — то какие конкретно параметры были переданы неверно?

6. Наконец, если какие-то параметры операции имеют недопустимые значения, то какие значения допустимы?

В нашем случае несовпадения цены ответ должен выглядеть так:

```
409 Conflict
{
    // Род ошибки
    "reason": "offer_invalid",
    "localized_message":
        "Что-то пошло не так.  
Попробуйте перезагрузить приложение."
    "details": {
        // Что конкретно неправильно?
        // Какие из проверок
        // валидности предложения
        // отработали с ошибкой?
        "checks_failed": [
            "offer_lifetime"
        ]
    }
}
```

Получив такую ошибку, клиент должен проверить её род (что-то с предложением), проверить конкретную причину ошибки (срок жизни оффера истёк) и отправить повторный запрос цены. При этом если бы `checks_failed` показал другую причину ошибки — например, указанный `offer_id` не принадлежит данному пользователю — действия клиента были бы иными (отправить пользователя повторно авторизоваться, а затем перезапросить цену). Если же обработка такого рода ошибок в коде не предусмотрена — следует показать пользователю сообщение `localized_message` и вернуться к обработке ошибок по умолчанию.

Важно также отметить, что неустранимые ошибки в моменте для клиента бесполезны (не зная причины ошибки клиент не может ничего разумного предложить пользователю), но это не значит, что у них не должно быть расширенной информации: их всё равно будет просматривать разработчик, когда будет исправлять эту проблему в коде (подробнее об этом в следующей главе).

Декомпозиция интерфейсов. Правило «7±2»

Исходя из нашего собственного опыта использования разных API, мы можем, не колеблясь, сказать, что самая большая ошибка проектирования сущностей в API (и, соответственно, головная боль разработчиков) — чрезмерная перегруженность интерфейсов полями, методами, событиями, параметрами и прочими атрибутами сущностей.

При этом существует «золотое правило», применимое не только к API, но ко множеству других областей проектирования: человек комфортно удерживает в краткосрочной памяти 7 ± 2 различных объекта. Манипулировать большим числом сущностей человеку уже сложно. Это правило также известно как «закон Миллера»¹.

Бороться с этим законом можно только одним способом: декомпозицией. На каждом уровне работы с вашим API нужно стремиться логически группировать сущности под одним именем там, где это возможно и таким образом, чтобы разработчику никогда не приходилось оперировать более чем 10 сущностями одновременно.

Рассмотрим простой пример: что должна возвращать функция поиска подходящей кофемашины. Для обеспечения хорошего UX приложения необходимо передать довольно значительные объёмы информации.

```
{
  "results": [
    // Данные кофемашины
    "coffee_machine_id", "coffee_machine_type",
    "coffee_machine_brand",
    // Данные кафе
    "place_name": "The Chamomile",
    "place_location_latitude",
    "place_location_longitude",
    "place_open_now", "working_hours",
    // Как добраться
    "walking_distance", "walking_time",
    // Как найти нужное место
    "location_tip",
    // Предложения
    "offers": [
      // Данные рецепта
      "recipe", "recipe_name",
      "recipe_description",
      // Параметры заказа
      "volume",
      // Данные предложения
      "offer_id", "offer_valid_until",
      "price": "19.00",
      "localized_price": [
        "Just $19 for a large coffee cup",
        "currency_code", "estimated_waiting_time"
      ],
      ...
    ],
    ...
  ]
}
```

Подход, увы, совершенно стандартный, его можно встретить практически в любом API. Как мы видим, количество полей сущностей вышло далеко за рекомендованные 7, и даже 9. При этом набор полей идёт плоским списком вперемешку, часто с одинаковыми префиксами.

В такой ситуации мы должны выделить в структуре информационные домены: какие поля логически относятся к одной предметной области. В данном случае мы можем выделить как минимум следующие виды данных:

- данные о заведении, в котором находится кофемашины;
- данные о самой кофемашине;
- данные о пути до точки;
- данные о рецепте;
- опции приготовления заказа;
- данные о предложении;
- данные о цене.

Попробуем сгруппировать:

```
{
  "results": [
    // Данные о заведении
    "place": { "name", "location" },
    // Данные о кофемашине
    "coffee-machine": { "id", "brand", "type" },
    // Как добраться
    "route": {
      "distance", "duration", "location_tip"
    },
    // Предложения напитков
    "offers": [
      // Рецепт
      "recipe": {
        { "id", "name", "description" },
        // Опции заказа
        "options": { "volume" },
        // Метаданные предложения
        "offer": { "id", "valid_until" },
        // Цена
        "pricing": {
          "currency_code", "price",
          "localized_price"
        },
        "estimated_waiting_time"
      }, ...
    ], ...
  }
}
```

Такой API читать и воспринимать гораздо удобнее, нежели сплошную простыню различных атрибутов. Более того, возможно, стоит на будущее сразу дополнительно сгруппировать, например, `place` и `route` в одну структуру `location`, или `offer` и `pricing` в одну более общую структуру.

Важно, что читабельность достигается не просто снижением количества сущностей на одном уровне. Декомпозиция должна производиться таким образом, чтобы разработчик при чтении интерфейса сразу понимал: так, вот здесь находится описание заведения, оно мне пока неинтересно и углубляться в эту ветку я пока не буду. Если перемешать данные, которые нужны в моменте одновременно для выполнения действия по разным композитам — это только ухудшит читабельность, а не улучшит.

Дополнительно правильная декомпозиция поможет нам в решении задачи расширения и развития API, о чём мы поговорим в разделе III.

Примечания

¹ Рабочая память. Оценка емкости рабочей памяти

<https://ru.wikipedia.org/wiki/%Do%Ao%Do%Bo%Do%B1%Do%BE%D1%87%Do%Bo%D1%8F%Do%BF%Do%Bo%Do%BC%D1%8F%D1%82%D1%8C%D1%86%Do%BA%Do%Bo%Do%BC%D1%8A%Do%BE%D1%81%D1%82%Do%BC%D1%88%CC%86%Do%BF%Do%Bo%Do%BC%D1%8F%D1%82%Do%BC%D1%88%CC%86>

Глава 13. Описание конечных интерфейсов

Определив все сущности, их ответственность и отношения друг с другом, мы переходим непосредственно к разработке API: нам осталось прописать номенклатуру всех объектов, полей, методов и функций в деталях. В этой главе мы дадим сугубо практические советы, как сделать API удобным и понятным.

Важнейшая задача разработчика API — добиться того, чтобы код, написанный поверх API другими разработчиками, легко читался и поддерживался. Помните, что закон больших чисел работает против вас: если какую-то концепцию или сигнатуру вызова можно понять неправильно, значит, её неизбежно будет понимать неправильно всё большее число партнеров по мере роста популярности API.

NB: примеры, приведённые в этой главе, прежде всего иллюстрируют проблемы консистентности и читабельности, возникающие при разработке API. Мы не ставим здесь цели дать рекомендации по разработке REST API (такого рода советы будут даны в соответствующем разделе) или стандартных библиотек языков программирования — важен не конкретный синтаксис, а общая идея.

Важное уточнение номер один:

1. Правила не должны применяться бездумно

Правило — это просто кратко сформулированное обобщение опыта. Они не действуют безусловно и не означают, что можно не думать головой. У каждого правила есть какая-то рациональная причина его существования. Если в вашей ситуации нет причин следовать правилу — значит, следовать ему не нужно.

Это соображение применимо ко всем принципам ниже. Если из-за следования правилам у вас получается неудобный, громоздкий, неочевидный API — это повод пересмотреть правила (или API).

Важно понимать, что вы вольны вводить свои собственные конвенции. Например, в некоторых фреймворках сознательно отказываются от парных методов `set_entity` / `get_entity` в пользу одного метода `entity` с опциональным параметром. Важно только проявить последовательность в её применении — если такая конвенция вводится, то абсолютно все методы API должны иметь подобную полиморфную сигнатуру, или по крайней мере должен существовать принцип именования, отличающий такие комбинированные методы от обычных вызовов.

2. Явное лучше неявного

Из названия любой сущности должно быть очевидно, что она делает, и к каким побочным эффектам может привести её использование.

Плохо:

```
// Отменяет заказ  
order.canceled = true;
```

Неочевидно, что поле состояния можно перезаписывать, и что это действие отменяет заказ.

Хорошо:

```
// Отменяет заказ  
order.cancel();
```

Плохо:

```
// Возвращает агрегированную  
// статистику заказов за всё время  
orders.getStats()
```

Даже если операция немодифицирующая, но вычислительно дорогая — следует об этом явно индицировать, особенно если вычислительные ресурсы тарифицируются для пользователя; тем более не стоит подбирать значения по умолчанию так, чтобы вызов операции без параметров максимально расходовал ресурсы.

Хорошо:

```
// Вычисляет и возвращает агрегированную  
// статистику заказов за указанный период  
orders.calculateAggregatedStats({  
    begin_date: <начало периода>  
    end_date: <конец_периода>  
});
```

Стремитесь к тому, чтобы из сигнатуры функции было абсолютно ясно, что она делает, что принимает на вход и что возвращает. Вообще, при прочтении кода, работающего с вашим API, должно быть сразу понятно, что, собственно, он делает — без подглядывания в документацию.

Два важных следствия:

1.1. Если операция модифицирующая, это должно быть очевидно из сигнатуры. В частности, модифицирующая операция не может называться `getSomething` или использоваться с HTTP-глаголом GET.

1.2. Если в номенклатуре вашего API есть как синхронные операции, так и асинхронные, то (а)синхронность должна быть очевидна из сигнатур, либо должна существовать конвенция именования, позволяющая отличать синхронные операции от асинхронных.

3. Указывайте использованные стандарты

К сожалению, человечество не в состоянии договориться о таких простейших вещах, как «с какого дня начинается неделя». Поэтому *всегда* указывайте, по какому конкретно стандарту вы отдаёте те или иные величины. Исключения возможны только там, где вы на 100% уверены, что в мире существует только один стандарт для этой сущности, и всё население земного шара о нём в курсе.

Плохо: "date": "11/12/2020" — существует огромное количество стандартов записи дат, плюс из этой записи невозможно даже понять, что здесь число, а что месяц.

Хорошо: "iso_date": "2020-11-12".

Плохо: "duration": 5000 — пять тысяч чего?

Хорошо:

"duration_ms": 5000
либо
"duration": "5000ms" либо "iso_duration": "PT5S" либо

```
"duration": {  
    "unit": "ms",  
    "value": 5000  
}
```

Отдельное следствие из этого правила — денежные величины *всегда* должны сопровождаться указанием кода валюты.

Также следует отметить, что в некоторых областях ситуация со стандартами настолько плоха, что, как ни сделай, — кто-то останется недовольным. Классический пример такого рода — порядок географических координат («широта-долгота» против «долгота-широта»). Здесь, увы, есть только один работающий метод борьбы с фрустрацией — Блокнот душевного покоя, который будет описан [в одноимённой главе](#).

4. Сущности должны именоваться конкретно

Избегайте одиночных слов-«амёб» без определённой семантики, таких как get, apply, make.

Плохо: user.get() — неочевидно, что конкретно будет возвращено.

Хорошо: user.get_id().

5. Не экономьте буквы

В XXI веке давно уже нет нужды называть переменные покороче.

Плохо: order.getTime() — неясно, о каком времени идёт речь: время создания заказа, время готовности заказа, время ожидания заказа?...

Хорошо: order.getEstimatedDeliveryTime().

Плохо:

```
// возвращает положение
// первого вхождения в строку str1
// любого символа из строки str2
strpbrk (str1, str2)
```

Возможно, автору этого API казалось, что аббревиатура `pbrk` что-то значит для читателя, но он явно ошибся. К тому же, невозможно сходу понять, какая из строк `str1`, `str2` является набором символов для поиска.

Хорошо:

```
str_search_for_characters(
    str,
    lookup_character_set
)
```

— однако необходимость существования такого метода вообще вызывает сомнения, достаточно было бы иметь удобную функцию поиска подстроки с нужными параметрами. Аналогично сокращение `string` до `str` выглядит совершенно бессмысленным, но, увы, является устоявшимся для большого количества предметных областей.

NB: иногда названия полей сокращают или вовсе опускают (например, возвращают массив разнородных объектов вместо набора именованных полей) в погоне за уменьшением количества трафика. В абсолютном большинстве случаев это бессмысленно, поскольку текстовые данные при передаче обычно дополнительно сжимают на уровне протокола.

6. Тип поля должен быть ясен из его названия

Если поле называется `recipe` — мы ожидаем, что его значением является сущность типа `Recipe`. Если поле называется `recipe_id` — мы ожидаем, что его значением является идентификатор, который мы сможем найти в составе сущности `Recipe`.

То же касается и базовых типов. Сущности-массивы должны именоваться во множественном числе или собирательными выражениями — `objects`, `children`; если это невозможно (термин неисчисляем), следует добавить префикс или постфикс, не оставляющий сомнений.

Плохо: GET /news — неясно, будет ли получена какая-то конкретная новость или массив новостей.

Хорошо: GET /news-list.

Аналогично, если ожидается булево значение, то это должно быть очевидно из названия, т.е. именование должно описывать некоторое качественное состояние, например, is_ready, open_now.

Плохо: "task.status": true — неочевидно, что статус бинарен, к тому же такой API будет нерасширяемым.

Хорошо: "task.is_finished": true.

Отдельно следует оговорить, что на разных платформах эти правила следует дополнить по-своему с учётом специфики first-class citizen-типов. Например, в JSON не существует объектов типа Date, и даты приходится передавать в виде числа или строки; разумно такие даты индицировать с помощью, например, постфикса _at (created_at, occurred_at и т.д.) или _date.

Если наименование сущности само по себе является каким-либо термином, способным смутить разработчика, лучше добавить лишний префикс или постфикс во избежание непонимания.

Плохо:

```
// Возвращает список
// встроенных функций кофемашины
GET /coffee-machines/{id}/functions
```

Слово "functions" многозначное: оно может означать и встроенные функции, и написанный код, и состояние (функционирует или не функционирует).

Хорошо:

```
GET /v1/coffee-machines/{id}+
/builtin-functions-list
```

7. Подобные сущности должны называться подобно и вести себя подобным образом

Плохо: `begin_transition / stop_transition`

— `begin` и `stop` — непарные термины; разработчик будет вынужден рыться в документации.

Хорошо: `begin_transition / end_transition` либо `start_transition / stop_transition`.

Плохо:

```
// Находит первую позицию строки `needle`  
// внутри строки `haystack`  
strpos(haystack, needle)  
// Находит и заменяет  
// все вхождения строки `needle`  
// внутри строки `haystack`  
// на строку `replace`  
str_replace(needle, replace, haystack)
```

Здесь нарушены сразу несколько правил:

- написание неконсистентно в части знака подчёркивания;
- близкие по смыслу методы имеют разный порядок аргументов `needle/haystack`;
- первый из методов находит только первое вхождение строки `needle`, а другой — все вхождения, и об этом поведении никак нельзя узнать из сигнатуры функций.

Упражнение «как сделать эти интерфейсы хорошо» предоставим читателю.

8. Избегайте двойных отрицаний

Плохо: `"dont_call_me": false`

— люди в целом плохо считывают двойные отрицания. Это провоцирует ошибки.

Лучше: "prohibit_calling": true или "avoid_calling": true

— читается лучше, хотя обольщаться всё равно не следует. Насколько это возможно откажитесь от семантических двойных отрицаний, даже если вы придумали «негативное» слово без явной приставки «не».

Стоит также отметить, что в использовании законов де Моргана¹ ошибиться ещё проще, чем в двойных отрицаниях. Предположим, что у вас есть два флага:

```
GET /coffee-machines/{id}/stocks
→
{
  "has_beans": true,
  "has_cup": true
}
```

Условие «кофе можно приготовить» будет выглядеть как has_beans && has_cup — есть и зерно, и стакан. Однако, если по какой-то причине в ответе будут отрицания тех же флагов:

```
{
  "no_beans": false,
  "no_cup": false
}
```

— то разработчику потребуется вычислить флаг !no_beans && !no_cup, что эквивалентно !(no_beans || no_cup), а вот в этом переходе ошибиться очень легко, и избегание двойных отрицаний помогает слабо. Здесь, к сожалению, есть только общий совет «избегайте ситуаций, когда разработчику нужно вычислять такие флаги».

9. Избегайте неявного приведения типов

Этот совет парадоксально противоположен предыдущему. Часто при разработке API возникает ситуация, когда добавляется новое необязательное поле с непустым значением по умолчанию. Например:

```
let orderParams = {  
    contactless_delivery: false  
};  
let order = api.createOrder(  
    orderParams  
);
```

Новая опция `contactless_delivery` является необязательной, однако её значение по умолчанию — `true`. Возникает вопрос, каким образом разработчик должен отличить явное *нежелание* пользоваться опцией (`false`) от незнания о её существовании (поле не задано). Приходится писать что-то типа такого:

```
let value = orderParams.contactless_delivery;  
if (Type(value) == 'Boolean' && value == false) {  
    ...  
}
```

Эта практика ведёт к усложнению кода, который пишут разработчики, и в этом коде легко допустить ошибку, которая по сути меняет значение поля на противоположное. То же произойдёт, если для индикации отсутствия значения поля использовать специальное значение типа `null` или `-1`.

Если протоколом не предусмотрена нативная поддержка таких кейсов (т.е. разработчик не может допустить ошибку, спутав отсутствие поля с пустым значением), универсальное правило — все новые необязательные булевые флаги должны иметь значение по умолчанию `false`.

Хорошо

```
let orderParams = {  
    force_contact_delivery: true  
};  
let order = api.createOrder(  
    orderParams  
);
```

Если же требуется ввести небулево поле, отсутствие которого трактуется специальным образом, то следует ввести пару полей.

Плохо:

```
// Создаёт пользователя
POST /v1/users
{
    ...
}
// Пользователи создаются по умолчанию
// с указанием лимита трат в месяц
{
    "spending_monthly_limit_usd": "100",
    ...
}
// Для отмены лимита требуется
// указать значение null
PUT /v1/users/{id}
{
    "spending_monthly_limit_usd": null,
    ...
}
```

Хорошо

```
POST /v1/users
{
    // true – у пользователя снят
    // лимит трат в месяц
    // false – лимит не снят
    // (значение по умолчанию)
    "abolish_spending_limit": false,
    // Необязательное поле, имеет смысл
    // только если предыдущий флаг
    // имеет значение false
    "spending_monthly_limit_usd": "100",
    ...
}
```

NB: противоречие с предыдущим советом состоит в том, что мы специально ввели отрицающий флаг («нет лимита»), который по правилу двойных отрицаний пришлось переименовать в `abolish_spending_limit`. Хотя это и хорошее название для отрицательного флага, семантика его довольно неочевидна, разработчикам придётся как минимум покопаться в документации. Таков путь.

10. Декларируйте технические ограничения явно

У любого поля в вашем API есть ограничения на допустимые значения: максимальная длина текста, объём прикладываемых документов в мегабайтах, разрешённые диапазоны цифровых значений. Часто разработчики API пренебрегают указанием этих лимитов — либо потому, что считают их очевидными, либо потому, что попросту не знают их сами. Это, разумеется, один большой антипаттерн: незнание пределов использования системы автоматически означает, что код партнёров может в любой момент перестать работать по не зависящим от них причинам.

Поэтому, во-первых, указывайте границы допустимых значений для всех без исключения полей в API, и, во-вторых, если эти границы нарушены, генерируйте машиночитаемую ошибку с описанием, какое ограничение на какое поле было нарушено.

То же соображение применимо и к квотам: партнёры должны иметь доступ к информации о том, какую долю доступных ресурсов они выбрали, и ошибки в случае превышения квоты должны быть информативными.

11. Любые запросы должны быть лимитированы

Ограничения должны быть не только на размеры полей, но и на размеры списков или агрегируемых интервалов.

Плохо: `getOrders()` — что, если пользователь совершил миллион заказов?

Хорошо: `getOrders({ limit, parameters })` — должно существовать ограничение сверху на размер обрабатываемых и возвращаемых данных и, соответственно, возможность уточнить запрос, если партнёру всё-таки требуется большее количество данных, чем разрешено обрабатывать в одном запросе.

12. Описывайте политику перезапросов

Одна из самых больших проблем с точки зрения производительности, с которой сталкивается почти любой разработчик API, и внутренних, и публичных — это отказ в обслуживании вследствие лавины перезапросов: временные проблемы на бэкенде API (например, повышение времени ответа) могут привести к полной неработоспособности сервера, если клиенты начнут очень быстро повторять запрос, не получив или не дождавшись ответа, сгенерировав, таким образом, кратно большую нагрузку в короткий срок.

Лучшая практика в такой ситуации — это требовать, чтобы клиенты перезапрашивали эндпоинты API с увеличивающимся интервалом (скажем, первый перезапрос происходит через одну секунду, второй — через две, третий через четыре, и так далее, но не больше одной минуты). Конечно, в случае публичного API такое требование никто не обязан соблюдать, но и хуже от его наличия вам точно не станет: хотя бы часть партнёров прочитает документацию и последует вашим рекомендациям.

Кроме того, вы можете разработать референсную реализацию политики перезапросов в ваших публичных SDK и следить за правильностью имплементации open-source модулей к вашему API.

13. Считайте трафик

В современном мире такой ресурс, как объём переданного трафика, считать уже почти не принято — считается, что Интернет всюду практически безлимитен. Однако он всё-таки не абсолютно безлимитен: всегда можно спроектировать систему так, что объём трафика окажется некомфортным даже и для современных сетей.

Три основные причины раздувания объёма трафика достаточно очевидны:

- клиент слишком часто запрашивает данные и/или слишком мало их кэширует;
- не предусмотрен постраничный перебор данных;
- не предусмотрены ограничения на размер значений полей и/или передаются большие бинарные данные (графика, аудио, видео и т.д.).

Все эти проблемы должны решаться через введение ограничений на размеры полей и правильную декомпозицию эндпойнтов. Если в рамках одной сущности необходимо предоставлять как «лёгкие» (скажем, название и описание рецепта), так и «тяжёлые» данные (скажем, промо-фотография напитка, которая легко может по размеру превышать текстовые поля в сотни раз), лучше разделить эндпойнты и отдавать только ссылку для доступа к «тяжёлым» данным (в нашем случае, ссылку на изображение) — это, как минимум, позволит задавать различные политики кэширования для разных данных.

Неплохим упражнением здесь будет промоделировать типовой жизненный цикл основной функциональности приложения партнёра (например, выполнение одного заказа) и подсчитать общее количество запросов и объём трафика на один цикл. Причиной слишком большого числа запросов / объёма трафика может оказаться ошибка проектирования подсистемы уведомлений об изменениях состояния. Подробнее этот вопрос мы рассмотрим в главе «Двунаправленные потоки данных» раздела «Паттерны API».

14. Отсутствие результата — тоже результат

Если сервер корректно обработал вопрос и никакой внештатной ситуации не возникло — следовательно, это не ошибка. К сожалению, весьма распространён антипаттерн, когда отсутствие результата считается ошибкой.

Плохо

```
POST /v1/coffee-machines/search
{
  "query": "lungo",
  "location": <положение пользователя>
}
→ 404 Not Found
{
  "localized_message":
    "Рядом с вами не делают лунго"
}
```

Статусы 4xx означают, что клиент допустил ошибку; однако в данном случае никакой ошибки сделано не было ни пользователем, ни разработчиком: клиент же не может знать заранее, готовят здесь лунго или нет.

Хорошо:

```
POST /v1/coffee-machines/search
{
  "query": "lungo",
  "location": <положение пользователя>
}
→ 200 OK
{
  "results": []
}
```

Это правило вообще можно упростить до следующего: если результатом операции является массив данных, то пустота этого массива — не ошибка, а штатный ответ. (Если, конечно, он допустим по смыслу; пустой массив координат, например, является ошибкой.)

NB: этот паттерн следует применять и в обратную сторону. Если в запросе можно указать массив сущностей, то следует отличать пустой массив от отсутствия параметра. Рассмотрим следующий пример:

```
// Находит все рецепты кофе
// без молока
POST /v1/recipes/search
{
  "filter": {
    "no_milk": true
  }
}
→ 200 OK
{
  "results": [
    {
      "recipe": "espresso"
    },
    {
      "recipe": "lungo",
    }
  ]
}
```

```
// Находит все предложения
// указанных рецептов
POST /v1/offers/search
{
  "location",
  "recipes": [
    "espresso",
    "lungo"
  ]
}
```

Представим теперь, что вызов первого метода вернул пустой массив результатов, т.е. ни одного рецепта кофе, удовлетворяющего условиям, не было найдено. Хорошо, если разработчик партнёра предусмотрит эту ситуацию и не будет делать запрос поиска предложений — но мы не можем быть стопроцентно в этом уверены. Если обработка пустого массива рецептов не предусмотрена, то приложение партнёра выполнит вот такой запрос:

```
POST /v1/offers/search
{
  "location",
  "recipes": []
}
```

Часто можно столкнуться с ситуацией, когда эндпойнт просто проигнорирует наличие пустого массива `recipes` и вернёт предложения так, как будто никакого фильтра по рецепту передано не было. В нашем примере это будет означать, что приложение просто проигнорирует требование пользователя показать только напитки без молока, что мы никак не можем счесть приемлемым поведением. Поэтому ответом на такой запрос с пустым массивом в качестве параметра должна быть либо ошибка, либо пустой же массив предложений.

15. Валидируйте ввод

Какой из вариантов действий выбрать в предыдущем примере — исключение или пустой ответ — напрямую зависит от того, что записано в вашем контракте. Если спецификация прямо предписывает, что массив `recipes` должен быть непустым, то необходимо сгенерировать исключение (иначе вы фактически нарушаете собственную спецификацию).

Это верно не только в случае непустых массивов, но и любых других зафиксированных в контракте ограничений. «Тихое» исправление недопустимых значений почти никогда не имеет никакого практического смысла:

Плохо:

```
POST /v1/offers/search
{
  "location": {
    "longitude": 20,
    "latitude": 100
  }
}
→ 200 OK
{
  // Предложения для точки
  // [0, 90]
  "offers"
}
```

Мы видим, что разработчик по какой-то причине передал некорректное значение широты (100 градусов). Да, мы можем его «исправить», т.е. редуцировать до ближайшего допустимого значения (90 градусов), но кому от этого стало лучше? Разработчик никогда не узнает о допущенной ошибке, а конечному пользователю предложения кофе на Северном полюсе, скорее всего, нерелевантны.

Хорошо:

```
POST /v1/coffee-machines/search
{
  "location": {
    "longitude": 20,
    "latitude": 100
  }
}
→ 400 Bad Request
{
  // описание ошибки
}
```

Желательно не только обращать внимание партнёров на ошибки, но и проактивно предупреждать их о поведении, возможно похожем на ошибку:

```

POST /v1/coffee-machines/search
{
  "location": {
    "latitude": 0,
    "longitude": 0
  }
}
→
{
  "results": [],
  "warnings": [
    {
      "type": "suspicious_coordinates",
      "message": "Location [0, 0] is probably a mistake"
    },
    {
      "type": "unknown_field",
      "message": "unknown field: `force_convact_delivery`. Did you mean `force_contact_delivery`?"
    }
  ]
}

```

Однако следует отметить, что далеко не во все интерфейсы можно удобно уложить дополнительно возврат предупреждений. В такой ситуации можно ввести дополнительный режим отладки или строгий режим, в котором уровень предупреждений эскалируется:

```

POST /v1/coffee-machines/search?
  strict_mode=true
{
  "location": {
    "latitude": 0,
    "longitude": 0
  }
}
→ 404 Bad Request
{
  "errors": [
    {
      "type": "suspicious_coordinates",
      "message": "Location [0, 0] is probably a mistake"
    }
  ],
  ...
}

```

Если всё-таки координаты [0, 0] не ошибка, то можно дополнительно разрешить задавать игнорируемые ошибки для конкретной операции:

```

POST /v1/coffee-machines/search?
  strict_mode=true
  &disable_errors=suspicious_coordinates

```

16. Значения по умолчанию должны быть осмысленны

Значения по умолчанию — один из самых ваших сильных инструментов, позволяющих избежать многословности при работе с API. Однако эти умолчания должны помогать разработчикам, а не маскировать их ошибки.

Плохо:

```
POST /v1/coffee-machines/search
{
  "recipes": [ "lungo" ]
  // Положение пользователя не задано
}
→
{
  "results": [
    // Результаты для какой-то
    // локации по умолчанию
  ]
}
```

Формально, подобное умолчание допустимо — почему бы не иметь концепции «географических координат по умолчанию». Однако в реальности результатом подобных политик «тихого» исправления ошибок становятся абсурдные ситуации типа «null island» — самой посещаемой точки в мире². Чем популярнее API, тем больше шансов, что партнеры просто не обратят внимания на такие пограничные ситуации.

Хорошо:

```
POST /v1/coffee-machines/search
{
  "recipes": [ "lungo" ]
  // Положение пользователя не задано
}
→ 400 Bad Request
{
  // описание ошибки
}
```

17. Ошибки должны быть информативными

Недостаточно просто валидировать ввод — необходимо ещё и уметь правильно описать, в чём состоит проблема. В ходе работы над интеграцией партнёры неизбежно будут допускать детские ошибки. Чем понятнее тексты сообщений, возвращаемых вашим API, тем меньше времени разработчик потратит на отладку, и тем приятнее работать с таким API.

Плохо:

```
POST /v1/coffee-machines/search
{
  "recipes": ["lngo"],
  "position": {
    "latitude": 110,
    "longitude": 55
  }
}
→ 400 Bad Request
{}
```

— да, конечно, допущенные ошибки (опечатка в "lngo" и неправильные координаты) очевидны. Но раз наш сервер всё равно их проверяет, почему не вернуть описание ошибок в читаемом виде?

Хорошо:

```
{
  "reason": "wrong_parameter_value",
  "localized_message":
    "Что-то пошло не так.  

     Обратитесь к разработчику приложения.",
  "details": {
    "checks_failed": [
      {
        "field": "recipe",
        "error_type": "wrong_value",
        "message":
          "Value 'lngo' unknown.  

           Did you mean 'lungo'?"
      },
      {
        "field": "position.latitude",
        "error_type": "constraintViolation",
        "constraints": {
          "min": -90,
          "max": 90
        },
        "message":
          "'position.latitude' value  

           must fall within  

            the [-90, 90] interval"
      }
    ]
  }
}
```

Также хорошей практикой является указание всех допущенных ошибок, а не только первой найденной.

18. Всегда показывайте неразрешимые ошибки прежде разрешимых

Рассмотрим пример с заказом кофе

```
POST /v1/orders
{
  // запрошенный рецепт
  "recipe": "lngo",
  // идентификатор предложения
  "offer"
}
→ 409 Conflict
{
  // ошибка: время действия
  // предложения истекло
  "reason": "offer_expired"
}
```

```
// Повторный запрос
// с новым `offer`
POST /v1/orders
{
  "recipe": "lngo",
  "offer"
}
→ 400 Bad Request
{
  // Ошибка: неизвестный рецепт
  "reason": "recipe_unknown"
}
```

Какой был смысл получать новый `offer`, если заказ всё равно не может быть создан? Для пользователя это будет выглядеть как бессмысленные действия (или бессмысленное ожидание), которые всё равно завершатся ошибкой, что бы он ни делал. Да, соблюдение порядка ошибок не изменит результат — заказ всё ещё нельзя сделать — но, во-первых, пользователь потратит меньше времени (а также сделает меньше запросов к бэкенду и внесёт меньший вклад в фон ошибок) и, во-вторых, диагностика проблемы будет гораздо проще читаться.

19. Начинайте исправление ошибок с более глобальных

Если ошибки исправимы (т.е. пользователь может совершить какие-то действия и всё же добиться желаемого), следует в первую очередь сообщать о тех, которые потребуют более глобального изменения состояния.

Плохо:

```
POST /v1/orders
{
  "items": [
    {
      "item_id": "123",
      "price": "0.10"
    }
  ]
}
→
409 Conflict
{
  // Ошибка: пока пользователь
  // совершил заказ, цена
  // товара изменилась
  "reason": "price_changed",
  "details": [
    {
      "item_id": "123",
      "actual_price": "0.20"
    }
  ]
}
```

```
// Повторный запрос
// с актуальной ценой
POST /v1/orders
{
  "items": [
    {
      "item_id": "123",
      "price": "0.20"
    }
  ]
}
→
409 Conflict
{
  // Ошибка: у пользователя слишком
  // много одновременных заказов,
  // создание новых заказов запрещено
  "reason": "order_limit_exceeded",
  "localized_message":
    "Лимит заказов превышен"
}
```

Какой был смысл показывать пользователю диалог об изменившейся цене, если и с правильной ценой заказ он сделать всё равно не сможет? Пока один из его предыдущих заказов завершится и можно будет сделать следующий заказ, цену, наличие и другие параметры заказа всё равно придётся корректировать ещё раз.

20. Проанализируйте потенциальные взаимные блокировки

В сложных системах не редки ситуации, когда исправление одной ошибки приводит к возникновению другой и наоборот.

```
// Создаём заказ с платной доставкой
POST /v1/orders
{
    "items": 3,
    "item_price": "3000.00",
    "currency_code": "MNT",
    "delivery_fee": "1000.00",
    "total": "10000.00"
}
→ 409 Conflict
// Ошибка: доставка становится бесплатной
// при стоимости заказа от 9000 тугриков
{
    "reason": "delivery_is_free"
}
```

```
// Создаём заказ с бесплатной доставкой
POST /v1/orders
{
    "items": 3,
    "item_price": "3000.00",
    "currency_code": "MNT",
    "delivery_fee": "0.00",
    "total": "9000.00"
}
→ 409 Conflict
// Ошибка: минимальная сумма заказа
// 10000 тугриков
{
    "reason": "below_minimal_sum",
    "currency_code": "MNT",
    "minimal_sum": "10000.00"
}
```

Легко заметить, что в этом примере нет способа разрешить ошибку в один шаг — эту ситуацию требуется предусмотреть отдельно, и либо изменить параметры расчёта (минимальная сумма заказа не учитывает скидки), либо ввести специальную ошибку для такого кейса.

21. Указывайте время жизни ресурсов и политики кэширования

В современных системах клиент, как правило, обладает собственным состоянием и почти всегда кэширует результаты запросов — неважно, долговременно ли или в течение сессии: у каждого объекта всегда есть какое-то время автономной жизни. Поэтому желательно вносить ясность; таким образом рекомендуется кэшировать результат должно быть понятно, если не из сигнатур операций, то хотя бы из документации.

Следует уточнить, что кэш мы понимаем в расширенном смысле, а именно: какое варьирование параметров операции (не только времени обращения, но и прочих переменных) следует считать достаточно близким к предыдущему запросу, чтобы можно было использовать результат из кэша?

Плохо:

```
// Возвращает цену лунго в кафе,  
// ближайшем к указанной точке  
GET /v1/price?recipe=lungo↵  
  &longitude={longitude}↵  
  &latitude={latitude}↵  
→ { "currency_code", "price" }
```

Возникает два вопроса:

- в течение какого времени эта цена действительна?
- на каком расстоянии от указанной точки цена всё ещё действительна?

Хорошо: Для указания времени жизни кэша можно пользоваться стандартными средствами протокола, например, заголовком Cache-Control. В ситуации, когда кэш существует не только во временном измерении (как, например, в нашем примере добавляется пространственное измерение), вам придётся разработать свой формат описания параметров кэширования.

```
GET /v1/price?recipe=lungo↵  
  &longitude={longitude}↵  
  &latitude={latitude}↵  
→ {  
  "offer": {  
    "id",  
    "currency_code",  
    "price",  
    "conditions": {  
      // До какого времени  
      // валидно предложение  
      "valid_until",  
      // Где валидно предложение:  
      // * город  
      // * географический объект  
      // * ...  
      "valid_within"  
    }  
  }  
}
```

NB: часто можно встретить подход, когда для неизменяемых данных выставляется очень длинный срок жизни кэша — год, а то и больше. С практической точки зрения это не имеет большого смысла (вряд ли можно всерьёз ожидать серьёзного снижения нагрузки на сервер по сравнению, скажем, с кэшированием на месяц), а вот цена ошибки существенно возрастает: если по какой-то причине будут закэшированы неверные данные (например, ошибка 404), эта проблема будет преследовать вас следующий год, а то и больше. Мы склонны рекомендовать выбирать разумные сроки кэширования в зависимости от того, насколько серьёзным окажется для бизнеса кэширование неверного значения.

22. Сохраняйте точность дробных чисел

Там, где это позволено протоколом, дробные числа с фиксированной запятой — такие, как денежные суммы, например — должны передаваться в виде специально предназначенных для этого объектов, например, `Decimal` или аналогичных.

Если в протоколе нет `Decimal`-типов (в частности, в JSON нет чисел с фиксированной запятой), следует либо привести к целому (путём домножения на указанный множитель), либо использовать строковый тип.

Если конвертация в формат с плавающей запятой заведомо приводит к потере точности (например, если мы переведём 20 минут в часы в виде десятичной дроби), то следует либо предпочесть формат без потери точности (т.е. предпочтеть формат `00:20` формату `0.333333...`), либо предоставить SDK работы с такими данными, либо (в крайнем случае) описать в документации принципы округления.

23. Все операции должны быть идемпотентны

Напомним, идемпотентность — это следующее свойство: повторный вызов той же операции с теми же параметрами не изменяет результат. Поскольку мы обсуждаем в первую очередь клиент-серверное взаимодействие, узким местом в котором является ненадежность сетевой составляющей, повтор запроса при обрыве соединения — не исключительная ситуация, а норма жизни.

Там, где идемпотентность не может быть обеспечена естественным образом, необходимо добавить явный параметр — ключ идемпотентности или ревизию.

Плохо:

```
// Создаёт заказ
POST /orders
```

Повтор запроса создаст два заказа!

Хорошо:

```
// Создаёт заказ
POST /v1/orders
X-Idempotency-Token: <случайная строка>
```

Клиент на своей стороне запоминает X-Idempotency-Token, и, в случае автоматического повторного перезапроса, обязан его сохранить. Сервер на своей стороне проверяет токен и, если заказ с таким токеном уже существует для этого клиента, не даёт создать заказ повторно.

Альтернатива:

```
// Создаёт черновик заказа
POST /v1/orders/drafts
→
{ "draft_id" }
```

```
// Подтверждает черновик заказа
PUT /v1/orders/drafts/{draft_id}/confirmation
{ "confirmed": true }
```

Создание черновика заказа — необязывающая операция, которая не приводит ни к каким последствиям, поэтому допустимо создавать черновики без токена идемпотентности. Операция подтверждения заказа — уже естественным образом идемпотентна, для неё draft_id играет роль ключа идемпотентности.

Другая альтернатива — использование техник оптимистичного управления параллелизмом, о которых мы поговорим в главе «[Стратегии синхронизации](#)».

Также стоит упомянуть, что добавление токенов идемпотентности к эндпоинтам, которые и так изначально идемпотентны, имеет определённый смысл, так как токен помогает различить две ситуации:

- клиент не получил ответ из-за сетевых проблем и пытается повторить запрос;
- клиент ошибся, пытаясь применить конфликтующие изменения.

Рассмотрим следующий пример: представим, что у нас есть ресурс с общим доступом, контролируемым посредством номера ревизии, и клиент пытается его обновить.

```
POST /resource/updates
{
  "resource_revision": 123
  "updates"
}
```

Сервер извлекает актуальный номер ревизии и обнаруживает, что он равен 124. Как ответить правильно? Можно просто вернуть 409 Conflict, но тогда клиент будет вынужден попытаться выяснить причину конфликта и как-то решить его, потенциально запутав пользователя. К тому же, фрагментировать алгоритмы разрешения конфликтов, разрешая каждому клиенту реализовать какой-то свой — плохая идея.

Сервер мог бы попытаться сравнить значения поля updates, предполагая, что одинаковые значения означают перезапрос, но это предположение будет опасно неверным (например, если ресурс представляет собой счётчик, то последовательные запросы с идентичным телом нормальны).

Добавление токена идемпотентности (явного в виде случайной строки или неявного в виде черновиков) решает эту проблему:

```
POST /resource/updates
X-Idempotency-Token: <токен>
{
  "resource_revision": 123
  "updates"
}
→ 201 Created
```

— сервер обнаружил, что ревизия 123 была создана с тем же токеном идемпотентности, а значит клиент просто повторяет запрос.

Или:

```
POST /resource/updates
X-Idempotency-Token: <токен>
{
  "resource_revision": 123
  "updates"
}
→ 409 Conflict
```

— сервер обнаружил, что ревизия 123 была создана с другим токеном, значит имеет место быть конфликт общего доступа к ресурсу.

Более того, добавление токена идемпотентности не только решает эту проблему, но и позволяет в будущем сделать продвинутые оптимизации. Если сервер обнаруживает конфликт общего доступа, он может попытаться решить его, «перебазировав» обновление, как это делают современные системы контроля версий, и вернуть 200 OK вместо 409 Conflict. Эта логика существенно улучшает пользовательский опыт и при этом полностью обратно совместима и предотвращает фрагментацию кода разрешения конфликтов.

Но имейте в виду: клиенты часто ошибаются при имплементации логики токенов идемпотентности. Две проблемы проявляются постоянно:

- нельзя полагаться на то, что клиенты генерируют честные случайные токены — они могут иметь одинаковый seed рандомизатора или просто использовать слабый алгоритм или источник энтропии; при проверке токенов нужны слабые ограничения: уникальность токена должна проверяться не глобально, а только применительно к конкретному пользователю и конкретной операции;
- клиентские разработчики могут неправильно понимать концепцию — или генерировать новый токен на каждый перезапрос (что на самом деле неопасно, в худшем случае деградирует UX), или, напротив, использовать один токен для разнородных запросов (а вот это опасно и может привести к катастрофически последствиям; ещё одна причина имплементировать совет из предыдущего пункта!); поэтому рекомендуется написать хорошую документацию и/или клиентскую библиотеку для перезапросов.

24. Не изобретайте безопасность

Если бы автору этой книги давали доллар каждый раз, когда ему приходилось бы имплементировать кем-то придуманный дополнительный протокол безопасности — он бы давно уже был на заслуженной пенсии. Любовь разработчиков API к подписыванию параметров запросов или сложным схемам обмена паролей на токены столь же несомненна, сколько и бессмысленна.

Во-первых, почти всегда процедуры, обеспечивающие безопасность той или иной операции, *уже разработаны*. Нет никакой нужды придумывать их заново, просто имплементируйте какой-то из существующих протоколов. Никакие самописные алгоритмы проверки сигнатур запросов не обеспечат вам того же уровня защиты от атаки Manipulator-in-the-middle (*MitM*)³, как соединение по протоколу TLS с взаимной проверкой сигнатур сертификатов⁴.

Во-вторых, чрезвычайно самонадеянно (и опасно) считать, что вы разбираетесь в вопросах безопасности. Новые вектора атаки появляются каждый день, и быть в курсе всех актуальных проблем — это само по себе работа на полный рабочий день. Если же вы полный рабочий день занимаетесь чем-то другим, спроектированная вами система защиты наверняка будет содержать уязвимости, о которых вы просто никогда не слышали — например, ваш алгоритм проверки паролей может быть подвержен атаке по времени⁵, а веб-сервер — атаке с разделением запросов⁶.

Фонд OWASP каждый год составляет список самых распространённых уязвимостей в API⁷, который мы настоятельно рекомендуем изучить.

Отдельно уточним: любые API должны предоставляться строго по протоколу TLS версии не ниже 1.2 (лучше 1.3).

25. Помогайте партнёрам не изобретать безопасность

Не менее важно не только обеспечивать безопасность API как такового, но и предоставить партнёрам такие интерфейсы, которые минимизируют возможные проблемы с безопасностью на их стороне.

Плохо:

```
// Позволяет партнёру задать
// описание для своего напитка
PUT /v1/partner-api/{partner-id}↵
/recipes/lungo/info
"<script>alert(document.cookie)</script>"
```

```
// возвращает описание
GET /v1/partner-api/{partner-id}↵
/recipes/lungo/info
→
"<script>alert(document.cookie)</script>"
```

Подобный интерфейс является прямым способом соорудить хранимую XSS, которым потенциально может воспользоваться злоумышленник. Да, это ответственность самого партнёра — не допускать сохранения подобного ввода и/или экранировать его при выводе. Но большие цифры по-прежнему работают против вас: всегда найдутся начинающие разработчики, которые не знают об этом виде уязвимости или не подумали о нём. В худшем случае существование таких хранимых XSS может затронуть не только конкретного партнёра, но и вообще всех пользователей API.

В таких ситуациях мы рекомендуем, во-первых, экранировать вводимые через API данные, если они выглядят потенциально эксплуатируемыми (предназначены для показа в UI и/или возвращаются по прямой ссылке), и, во-вторых, ограничивать радиус взрыва так, чтобы через уязвимости в коде одного партнёра нельзя было затронуть других партнёров. В случае, если функциональность небезопасного ввода всё же нужна, необходимо предупреждать о рисках максимально явно.

Лучше (но не идеально):

```
// Позволяет партнёру задать
// потенциально небезопасное
// описание для своего напитка
PUT /v1/partner-api/{partner-id}↵
/recipes/lungo/info
X-Dangerously-Disable-Sanitizing: true
"<script>alert(document.cookie)</script>"
```

```
// возвращает потенциально
// небезопасное описание
GET /v1/partner-api/{partner-id}↵
/recipes/lungo/info
X-Dangerously-Allow-Raw-Value: true
→
"<script>alert(document.cookie)</script>"
```

В частности, если вы позволяете посредством API выполнять какие-то текстовые скрипты, всегда предпочитайте безопасный ввод небезопасному.

Плохо:

```
POST /v1/run/sql
{
    // Передаёт готовый запрос целиком
    "query": "INSERT INTO data (name)↵
              VALUES ('Robert');↵
              DROP TABLE students;-- )"
```

Лучше:

```
POST /v1/run/sql
{
    // Передаёт шаблон запроса
    "query": "INSERT INTO data (name)↵
              VALUES (?)",
    // и параметры для подстановки
    "values": [
        "Robert");
        DROP TABLE students;--"
    ]
}
```

Во втором случае вы сможете централизованно экранировать небезопасный ввод и избежать тем самым SQL-инъекции. Напомним повторно, что делать это необходимо с помощью state-of-the-art инструментов, а не самописных регулярных выражений.

26. Используйте глобально уникальные идентификаторы

Хорошим тоном при разработке API будет использование для идентификаторов сущностей глобально уникальных строк, либо семантических (например, "lungo" для видов напитков), либо случайных (например UUID-4⁸). Это может чрезвычайно пригодиться, если вдруг придётся объединять данные из нескольких источников под одним идентификатором.

Мы вообще склонны порекомендовать использование идентификаторов в urn-подобном формате, т.е. urn:order:<uuid> (или просто order:<uuid>), это сильно помогает с отладкой legacy-систем, где по историческим причинам есть несколько разных идентификаторов для одной и той же сущности, в таком случае неймспейсы в urn помогут быстро понять, что это за идентификатор и нет ли здесь ошибки использования.

Отдельное важное следствие: **не используйте инкрементальные номера как внешние идентификаторы**. Помимо вышесказанного, это плохо ещё и тем, что ваши конкуренты легко смогут подсчитать, сколько у вас в системе каких сущностей и тем самым вычислить, например, точное количество заказов за каждый день наблюдений.

27. Предусмотрите ограничения доступа

С ростом популярности API вам неизбежно придётся внедрять технические средства защиты от недобросовестного использования — такие, как показ капчи, расстановка приманок-honeypot-ов, возврат ошибок вида «слишком много запросов», постановка прокси-защиты от DDoS перед эндпойнтами и так далее. Всё это невозможно сделать, если вы не предусмотрели такой возможности изначально, а именно — не ввели соответствующей номенклатуры ошибок и предупреждений.

Вы не обязаны с самого начала такие ошибки действительно генерировать — но вы можете предусмотреть их на будущее. Например, вы можете описать ошибку 429 Too Many Requests или перенаправление на показ капчи, но не имплементировать возврат таких ответов, пока не возникнет в этом необходимость.

Отдельно необходимо уточнить, что в тех случаях, когда через API можно совершать платежи, ввод дополнительных факторов аутентификации пользователя (через TOTP, SMS или технологии типа 3D-Secure) должен быть предусмотрен обязательно.

NB: из этого пункта вытекает достаточно очевидное правило, которое, тем не менее, часто нарушают разработчики API — **всегда разделяйте эндпойнты разных семейств API**. Если вы предоставляете и серверное API, и сервисы для конечных пользователей, и виджеты для встраивания в сторонние приложения — эти API должны обслуживаться с разных эндпойнтов для того, чтобы вы могли вводить разные меры безопасности (скажем, API-ключи, требование логина и капчу, соответственно).

28. Не предоставляйте endpoint-ов массового получения чувствительных данных

Если через API возможно получение персональных данных, номер банковских карт, переписки пользователей и прочей информации, раскрытие которой нанесёт большой ущерб пользователям, партнёрам и/или вам — методов массового получения таких данных в API быть не должно, или, по крайней мере, на них должны быть ограничения на частоту запросов, размер страницы данных, а в идеале ещё и многофакторная аутентификация.

Часто разумной практикой является предоставление таких массовых выгрузок по запросу, т.е. фактически в обход API.

29. Локализация и интернационализация

Все эндпойнты должны принимать на вход языковые параметры (например, в виде заголовка Accept-Language), даже если на текущем этапе нужды в локализации нет.

Важно понимать, что язык пользователя и юрисдикция, в которой пользователь находится — разные вещи. Цикл работы вашего API всегда должен хранить локацию пользователя. Либо она задаётся явно (в запросе указываются географические координаты), либо неявно (первый запрос с географическими координатами инициировал создание сессии, в которой

сохранена локация) — но без локации корректная локализация невозможна. В большинстве случаев локацию допустимо редуцировать до кода страны.

Дело в том, что множество параметров, потенциально влияющих на работу API, зависят не от языка, а именно от расположения пользователя. В частности, правила форматирования чисел (разделители целой и дробной частей, разделители разрядов) и дат, первый день недели, раскладка клавиатуры, система единиц измерения (которая к тому же может оказаться не десятичной!) и так далее. В некоторых ситуациях необходимо хранить две локации: та, в которой пользователь находится, и та, которую пользователь сейчас просматривает. Например, если пользователь из США планирует туристическую поездку в Европу, то цены ему желательно показывать в местной валюте, но отформатированными согласно правилам американского письма.

Следует иметь в виду, что явной передачи локации может оказаться недостаточно, поскольку в мире существуют территориальные конфликты и спорные территории. Каким образом API должен себя вести при попадании координат пользователя на такие территории — вопрос, к сожалению, в первую очередь юридический. Автору этой книги приходилось как-то разрабатывать API, в котором пришлось вводить концепцию «территория государства А по мнению официальных органов государства Б».

Важно: различайте локализацию для конечного пользователя и локализацию для разработчика. В примерах выше сообщение `localized_message` адресовано пользователю — его должно показать приложение, если в коде обработки такой ошибки не предусмотрена. Это сообщение должно быть написано на указанном в запросе языке и отформатировано согласно правилам локации пользователя. А вот сообщение `details.checks_failed[].message` написано не для пользователя, а для разработчика, который будет разбираться с проблемой. Соответственно, написано и отформатировано оно должно быть понятным для разработчика образом — что, скорее всего, означает «на английском языке», т.к. английский де-факто является стандартом в мире разработки программного обеспечения.

Следует отметить, что индикация, какие сообщения следует показать пользователю, а какие написаны для разработчика, должна, разумеется, быть явной конвенцией вашего API. В примере для этого используется префикс `localized_`.

И ещё одна вещь: все строки должны быть в кодировке UTF-8 и никакой другой.

Примечания

¹ Законы де Моргана

https://ru.wikipedia.org/wiki/%Do%97%Do%Bo%Do%BA%Do%BE%Do%BD%D1%8B_%Do%B4%Do%B5_%Do%9C%Do%BE%D1%80%Do%B3%Do%Bo%Do%BD%Do%Bo

² Hrala, J. Welcome to Null Island, The Most 'Visited' Place on Earth That Doesn't Actually Exist

<https://www.sciencealert.com/welcome-to-null-island-the-most-visited-place-that-doesn-t-exist>

³ Manipulator-in-the-middle Attack

https://owasp.org/www-community/attacks/Manipulator-in-the-middle_attack

⁴ Mutual Authentication. mTLS

https://en.wikipedia.org/wiki/Mutual_authentication#mTLS

⁵ Timing Attack

https://en.wikipedia.org/wiki/Timing_attack

⁶ HTTP Request Splitting

<https://capec.mitre.org/data/definitions/105.html>

⁷ OWASP API Security Project

<https://owasp.org/www-project-api-security/>

⁸ UUID-4

[https://en.wikipedia.org/wiki/Universally_unique_identifier#Version_4_\(random\)](https://en.wikipedia.org/wiki/Universally_unique_identifier#Version_4_(random))

Глава 14. Приложение к разделу I. Модельный API

Суммируем текущее состояние нашего учебного API.

1. Поиск предложений

```
POST /v1/offers/search
{
    // опционально
    "recipes": ["lungo", "americano"],
    "position": <географические координаты>,
    "sort_by": [ { "field": "distance" } ],
    "limit": 10
}
→
{
    "results": [
        // Данные о заведении
        "place": { "name", "location" },
        // Данные о кофемашине
        "coffee_machine": { "id", "brand", "type" },
        // Как добраться
        "route": {
            "distance", "duration", "location_tip"
        },
        // Предложения напитков
        "offers": [
            // Рецепт
            "recipe": { "id", "name", "description" },
            // Данные относительно того,
            // как рецепт готовят
            // на конкретной кофемашине
            "options": { "volume" },
            // Метаданные предложения
            "offer": { "id", "valid_until" },
            // Цена
            "pricing": {
                "currency_code",
                "price",
                "localized_price"
            },
            "estimated_waiting_time"
        ],
        ...
    ],
    "cursor"
}
```

2. Работа с рецептами

```
// Возвращает список рецептов
// Параметр cursor необязателен
GET /v1/recipes?cursor=<курсор>
→
{ "recipes", "cursor" }
```

```
// Возвращает конкретный рецепт
// по его идентификатору
GET /v1/recipes/{id}
→
{
  "recipe_id",
  "name",
  "description"
}
```

3. Работа с заказами

```
// Размещает заказ
POST /v1/orders
X-Idempotency-Token: <токен>
{
  "coffee_machine_id",
  "currency_code",
  "price",
  "recipe": "lungo",
  // Опционально
  "offer_id",
  // Опционально
  "volume": "800ml"
}
→
{ "order_id" }
```

```
// Возвращает состояние заказа
GET /v1/orders/{id}
→
{ "order_id", "status" }
```

```
// Отменяет заказ
POST /v1/orders/{id}/cancel
```

4. Работа с программами

```
// Возвращает идентификатор программы,  
// соответствующей указанному рецепту  
// на указанной кофемашине  
POST /v1/program-matcher  
{ "recipe", "coffee_machine" }  
→  
{ "program_id" }
```

```
// Возвращает описание  
// программы по её идентификатору  
GET /v1/programs/{id}  
→  
{  
    "program_id",  
    "api_type",  
    "commands": [  
        {  
            "sequence_id",  
            "type": "set_cup",  
            "parameters"  
        },  
        ...  
    ]  
}
```

5. Исполнение программ

```
// Запускает исполнение программы  
// с указанным идентификатором  
// на указанной машине  
// с указанными параметрами  
POST /v1/programs/{id}/run  
X-Idempotency-Token: <токен>  
{  
    "order_id",  
    "coffee_machine_id",  
    "parameters": [  
        {  
            "name": "volume",  
            "value": "800ml"  
        }  
    ]  
}  
→  
{ "program_run_id" }
```

```
// Останавливает исполнение программы  
POST /v1/runs/{id}/cancel
```

6. Управление рантаймами

```
// Создаёт новый рантайм  
POST /v1/runtimes  
{  
    "coffee_machine_id",  
    "program_id",  
    "parameters"  
}  
→  
{ "runtime_id", "state" }
```

```
// Возвращает текущее состояние рантайма  
// по его id  
GET /v1/runtimes/{runtime_id}/state  
{  
    "status": "ready_waiting",  
    // Текущая исполняемая команда  
    // (необязательное)  
    "command_sequence_id",  
    "resolution": "success",  
    "variables"  
}
```

```
// Прекращает исполнение рантайма  
POST /v1/runtimes/{id}/terminate
```

РАЗДЕЛ II. ПАТТЕРНЫ ДИЗАЙНА API

Глава 15. О паттернах проектирования в контексте API

Концепция «паттернов» в области разработки программного обеспечения была введена Кентом Бэком и Уордом Каннингемом в 1987 году¹, и популяризирован «бандой четырёх» (Эрих Гамма, Ричард Хелм, Ральф Джонсон и Джон Влиссидес) в их книге «Приёмы объектно-ориентированного проектирования. Паттерны проектирования», изданной в 1994 году². Согласно общепринятыму определению, паттерны программирования — «повторяемая архитектурная конструкция, представляющая собой решение проблемы проектирования в рамках некоторого часто возникающего контекста».

Если мы говорим об API, особенно если конечным потребителем этих API является разработчик (интерфейсы фреймворков, операционных систем), классические паттерны проектирования вполне к ним применимы. И действительно, многие из описанных в предыдущем разделе примеров представляют собой применение того или иного паттерна.

Однако, если мы попытаемся обобщить этот подход на разработку API в целом, то увидим, что большинство типичных проблем дизайна API являются более высокоуровневыми и не сводятся к базовым паттернам разработки ПО. Скажем, проблемы кэширования ресурсов (и инвалидации кэша) или организация пагинации классиками не покрыты.

В рамках этого раздела мы попытаемся описать те задачи проектирования API, которые представляются нам наиболее важными. Мы не претендуем здесь на то, чтобы охватить *все* проблемы и тем более — все решения, и скорее фокусируемся на описании подходов к решению типовых задач с их достоинствами и недостатками. Мы понимаем, что читатель, знакомый с классическими трудами «банды четырёх», Гради Буча и Мартина Фаулера ожидает от раздела с названием «Паттерны API» большей системности и ширины охвата, и заранее просим у него прощения.

NB: первый паттерн, о котором необходимо упомянуть — это API-first подход к разработке ПО, который мы описали [в соответствующей главе](#).

Принципы решения типовых проблем проектирования API

Прежде, чем излагать сами паттерны, нам нужно понять, чем же разработка API отличается от разработки обычных приложений. Ниже мы сформулируем три важных принципа, на которые будем ссылаться в последующих главах.

1. Чем более распределена и многосоставна система, чем более общий канал связи используется для коммуникации — тем более вероятны ошибки в процессе взаимодействия. В частности, в наиболее интересном нам кейсе распределённых многослойных клиент-серверных систем возникновение исключения на клиенте (например, потеря контекста в результате перезапуска приложения), на сервере (конвейер выполнения запроса выбросил исключение на каком-то шаге), в канале связи (соединение полностью или частично потеряно) или любом промежуточном агенте (например, промежуточный веб-сервер не дождался ответа бэкенда и вернул ошибку гейтвея) — норма жизни, и все системы должны проектироваться таким образом, что **в случае возникновения исключения любого рода клиенты API должны быть способны восстановить своё состояние и продолжить корректно работать**.
2. Чем больше различных партнёров подключено к API, тем больше вероятность того, что какие-то из предусмотренных вами механизмов обеспечения корректности взаимодействия будет имплементирован неправильно. Иными словами, **вы должны ожидать не только физических ошибок, связанных с состоянием сети или перегруженностью сервера, но и логических, связанных с неправильным использованием API** (и, в частности, предотвращать возможный отказ в обслуживании одних партнёров из-за ошибок в коде других партнёров).
3. Любая из частей системы может вносить непредсказуемые задержки исполнения запросов, причём достаточно высокого — секунды, десятки секунд — порядка. Даже если вы полностью контролируете среду исполнения и сеть, задержку может вносить само клиентское приложение, которое может быть просто написано неоптимальным образом или же работать на слабом или перегруженном устройстве. Поэтому **при проектировании API нельзя полагаться на то, что критические действия выполняются быстро**. В частности:

- для операций, состоящих из нескольких шагов, необходимо предусматривать возможность при необходимости продолжить выполнение с текущего шага, а не с начала;
- для операций с разделяемыми ресурсами необходимо предусматривать механизмы блокировки ресурса.

Примечания

¹ Software Design Pattern. History

https://en.wikipedia.org/wiki/Software_design_pattern#History

² Gamma, E., Helm, R., Johnson, R., Vlissides, J. (1994) *Design Patterns. Elements of Reusable Object-Oriented Software*

[ISBN 9780321700698](#)

Глава 16. Аутентификация партнёров и авторизация вызовов API

Прежде, чем мы перейдём к обсуждению технических проблем и их решений, мы не можем не остановиться на важном вопросе авторизации вызовов API и аутентификации осуществляющих вызов клиентов. Исходя из всё того же принципа мультиплликатора («API умножает как возможности, так и проблемы») организация авторизации и аутентификации (АА) — одна из самых насущных проблем провайдера API, особенно публичного. Тем удивительнее тот факт, что в настоящий момент не существует стандартного подхода к ней — почти каждый крупный сервис разрабатывает какой-то свой интерфейс для решения этих задач, причём зачастую достаточно архаичный.

Если отвлечься от технических деталей имплементации (в отношении которых мы ещё раз настоятельно рекомендуем не изобретать велосипед и использовать стандартные подходы и протоколы безопасности), то, по большому счёту, есть два основных способа авторизовать выполнение некоторой операции через API:

- завести в системе специальный тип аккаунта «робот» и выполнять операции от имени робота;
- авторизоватьзывающую систему (бэкенд или клиентское приложение) как единое целое (обычно для аутентификации таких вызовов используются API-ключи, подписи или сертификаты).

Разница между двумя подходами заключается в гранулярности доступа:

- если клиент API выполняет запросы от имени пользователя системы, то его доступ к эндпоинту может быть ограничен каким-то конкретным набором данных, к которым имеет доступ пользователь;
- если же авторизуетсязывающая система, то обычно подразумевается, что она имеет полный доступ к эндпоинту, и может передавать любые параметры (т.е. имеет доступ к полному набору данных, предоставляемых через эндпоинт).

Первый подход, таким образом, является более гранулярным (робот может быть «виртуальным сотрудником» организации, то есть иметь доступ только к ограниченному набору данных) и вообще является естественным выбором для тех API, которые являются дополнением к существующему сервису для конечных пользователей (и, таким образом, могут использовать уже

существующие системы АА). Недостатками же этого подхода являются:

1. Необходимо организовать какой-то процесс безопасного получения токенов авторизации для пользователя-робота (например, через получение для него токенов реальным пользователем из веб-интерфейса), поскольку стандартная логин-парольная схема логина (тем более двухфакторная) слаба применима к клиенту API.
2. Необходимо сделать для пользователей-роботов исключения из почти всех систем безопасности:
 - роботы выполняют намного больше запросов, чем обычные люди, и могут делать это в параллель (в том числе с разных IP-адресов, расположенных в разных data-центрах);
 - роботы не принимают куки и не могут решить капчу;
 - робота нельзя профилактически разлогинить и/или инвалидировать его токен (это чревато простоем бизнеса партнёра), поэтому для роботов часто приходится изобретать токены с большим временем жизни и/или процедуру «подновления» токена.
3. Наконец, вы столкнётесь с очень большими проблемами, если вам всё-таки понадобится дать роботу возможность выполнять операцию от имени другого пользователя (поскольку такую возможность придётся тогда либо выдать и обычным пользователям, либо каким-то образом скрыть её и разрешить только роботам).

Если же API не предоставляется как сервис для конечных пользователей, второй подход с авторизацией клиентов через API-ключи более прост в имплементации. Здесь можно добиться гранулярности уровня эндпойнта (т.е. партнёр может выставить для ключа набор эндпойнтов, которые можно с ним вызывать), но более гранулярные системы (когда ключу выставляются ещё и ограничения на уровне бизнес-сущностей) уже намного сложнее в разработке и применяются редко.

Обе схемы, в общем-то, можно свести друг к другу (если разрешить роботным пользователям выполнять операции от имени любых других пользователей, мы фактически получим авторизацию по ключу; если создать по API-ключу какой-то ограниченный сегмент данных в рамках которого выполняются запросы, то фактически мы получим систему аккаунтов пользователей), и иногда можно встретить гибридные схемы (когда запрос авторизуется и API-ключом, и токеном пользователя).

Глава 17. Стратегии синхронизации

Перейдём теперь к техническим проблемам, стоящим перед разработчикам API, и начнём с последней из описанных во вводной главе — необходимости синхронизировать состояния. Представим, что конечный пользователь размещает заказ на приготовление кофе через наш API. Пока этот запрос путешествует от клиента в кофейню и обратно, многое может произойти. Например, рассмотрим следующую последовательность событий:

1. Клиент отправляет запрос на создание нового заказа.
2. Из-за сетевых проблем запрос идёт до сервера очень долго, а клиент получает таймаут:
 - клиент, таким образом, не знает, был ли выполнен запрос или нет.
3. Клиент запрашивает текущее состояние системы и получает пустой ответ, поскольку таймаут случился раньше, чем запрос на создание заказа дошёл до сервера:

```
let pendingOrders = await
    api.getOngoingOrders(); // → []
```

4. Сервер, наконец, получает запрос на создание заказа и исполняет его.
5. Клиент, не зная об этом, создаёт заказ повторно.

Поскольку действия чтения списка актуальных заказов и создания нового заказа разнесены во времени, мы не можем гарантировать, что между этими запросами состояние системы не изменилось. Если же мы хотим такую гарантию дать, нам нужно обеспечить какую-то из стратегий синхронизации¹. Если в случае, скажем, API операционных систем или клиентских фреймворков мы можем воспользоваться предоставляемыми платформой примитивами, то в кейсе распределённых сетевых API такой примитив нам придётся разработать самостоятельно.

Существуют два основных подхода к решению этой проблемы — пессимистичный (программная реализация блокировок) и оптимистичный (версионирование ресурсов).

NB: вообще, лучший способ избежать проблемы — не иметь её вовсе. Если ваш API идемпотентен, то никакой повторной обработки запроса не будет происходить. Однако не все операции в реальном мире идемпотентны в принципе: например, создание нового заказа такой операцией не является. Мы можем добавлять механики, предотвращающие *автоматические* перезапросы (такие как, например, генерируемый клиентом токен идемпотентности), но не можем запретить пользователю просто взять и повторно создать точно такой же заказ.

Программные блокировки

Первый подход — очевидным образом перенести стандартные примитивы синхронизации на уровень API. Например, вот так:

```
let lock;
try {
    // Захватываем право
    // на эксклюзивное исполнение
    // операции создания заказа
    lock = await api.acquireLock(ORDER_CREATION);
    // Получаем текущий список
    // заказов, известных системе
    let pendingOrders = await
        api.getPendingOrders();
    // Если нашего заказа ещё нет,
    // создаём его
    if (pendingOrders.length == 0) {
        let order = await api.createOrder(...)
    }
} catch (e) {
    // Обработка ошибок
} finally {
    // Разблокировка
    await lock.release();
}
```

Достаточно очевидно, что подобного рода подход крайне редко реализуется в распределённых сетевых API, из-за комплекса связанных проблем:

1. Ожидание получения блокировки вносит во взаимодействие дополнительные плохо предсказуемые и, в худшем случае, весьма длительные задержки.

2. Сама по себе блокировка — это ещё одна сущность, для работы с которой нужно иметь отдельную весьма производительную подсистему, поскольку для работы блокировок требуется ещё и обеспечить сильную консистентность² в API: метод `getPendingOrders` должен вернуть актуальное состояние системы, иначе повторный заказ всё равно будет создан.
3. Поскольку клиентская часть разрабатывается сторонними партнёрами, мы не можем гарантировать, что написанный ими код корректно работает с блокировками; неизбежно в системе появятся «висящие» блокировки, а, значит, придётся предоставлять партнёрам инструменты для отслеживания и отладки возникающих проблем.
4. Необходимо разработать достаточную гранулярность блокировок, чтобы партнёры не могли влиять на работоспособность друг друга. Хорошо, если мы можем ограничить блокировку, скажем, конкретным конечным пользователем в конкретной системе партнёра; но если этого сделать не получается (например, если система авторизации общая и все партнёры имеют доступ к одному и тому же профилю пользователя), то необходимо разрабатывать ещё более комплексные системы, которые будут исправлять потенциальные ошибки в коде партнёров — например, вводить квоты на блокировки.

Оптимистичное управление параллелизмом

Более щадящий с точки зрения сложности имплементации вариант — это реализовать оптимистичное управление параллелизмом³ и потребовать от клиента передавать признак того, что он располагает актуальным состоянием разделяемого ресурса.

```
// Получаем состояние
let orderState =
  await api.getOrderState();
// Частью состояния является
// версия ресурса
let version =
  orderState.latestVersion;
// Заказ можно создать,
// только если версия состояния
// не изменилась с момента чтения
try {
  let task = await api
    .createOrder(version, ...);
} catch (e) {
  // Если версия неверна, т.е. состояние
  // было параллельно изменено
  // другим клиентом, произойдёт ошибка
  if (Type(e) == INCORRECT_VERSION) {
    // Которую нужно как-то обработать...
  }
}
```

NB: внимательный читатель может возразить нам, что необходимость имплементировать стратегии синхронизации и строгую консистентность никуда не пропала, т.к. где-то в системе должен существовать компонент, осуществляющий блокирующее чтение версии с её последующим изменением. Это не совсем так: стратегии синхронизации и строгая консистентность *пропали из публичного API*. Расстояние между клиентом, устанавливающим блокировку, и сервером, её обрабатывающим, стало намного меньше, и всё взаимодействие теперь происходит в контролируемой среде (это вообще может быть одна подсистема, если мы используем ACID-совместимую базу данных⁴ или вовсе держим состояние ресурса в оперативной памяти).

Вместо версий можно использовать дату последней модификации ресурса (что в целом гораздо менее надёжно ввиду неидеальной синхронизации часов в разных узлах системы; не забывайте, как минимум, сохранять дату с максимально доступной точностью!) либо идентификаторы сущности (ETag).

Достоинством оптимистичного управления параллелизмом является, таким образом, возможность «спрятать» сложную в имплементации и масштабировании часть «под капотом». Недостаток же состоит в том, что ошибки версионирования теперь являются штатным поведением, и клиентам придётся написать правильную работу с ними, иначе их приложение может вообще оказаться неработоспособным — пользователь будет вечно пытаться создать заказ с неактуальной версией.

NB. Выбор ресурса, версию которого мы требуем передать для получения доступа, очень важен. Если в нашем примере мы заведём глобальную версию всей системы, которая изменяется при поступлении любого заказа, то, очевидно, у пользователя будут околонулевые шансы успешно разместить заказ.

Примечания

¹ Synchronization (Computer Science)

[https://en.wikipedia.org/wiki/Synchronization_\(computer_science\)](https://en.wikipedia.org/wiki/Synchronization_(computer_science))

² Strong consistency

https://en.wikipedia.org/wiki/Strong_consistency

³ Optimistic Concurrency Control

https://en.wikipedia.org/wiki/Optimistic_concurrency_control

⁴ ACID

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

Глава 18. Слабая консистентность

Описанный в предыдущей главе подход фактически представляет собой размен производительности API на «нормальный» (т.е. ожидаемый) фон ошибок при работе с ним путём изоляции компонента, отвечающего за строгую консистентность и управление параллелизмом внутри системы. Тем не менее, его пропускная способность всё равно ограничена, и увеличить её мы можем единственным образом — убрав строгую консистентность из внешнего API, что даст возможность осуществлять чтение состояния системы из реплик:

```
// Получаем состояние,
// возможно, из реплики
let orderState =
  await api.getOrderState();
let version =
  orderState.latestVersion;
try {
  // Обработчик запроса на
  // создание заказа прочитает
  // актуальную версию
  // из мастер-данных
  let task = await api
    .createOrder(version, ...);
} catch (e) {
}
...
```

Т.к. заказы создаются намного реже, нежели читаются, мы можем существенно повысить производительность системы, если откажемся от гарантии возврата всегда самого актуального состояния ресурса из операции на чтение. Версионирование же поможет нам избежать проблем: создать заказ, не получив актуальной версии, невозможно. Фактически мы пришли к модели событийной консистентности¹ (т.н. «согласованность в конечном счёте»): клиент сможет выполнить свой запрос *когда-нибудь*, когда получит, наконец, актуальные данные. В самом деле, согласованность в конечном счёте — скорее норма жизни для современных микросервисных архитектур, в которой может оказаться очень сложно как раз добиться обратного, т.е. строгой консистентности.

NB: на всякий случай уточним, что выбирать подходящий подход вы можете только в случае разработки новых API. Если вы уже предоставляете эндпойнт, реализующий какую-то модель консистентности, вы не можете понизить её уровень (в частности, сменить строгую консистентность на слабую), даже если вы никогда не документировали текущее поведение явно (мы обсудим это требование детальнее в главе «[О ватерлинии айсберга](#)» раздела «Обратная совместимость»).

Однако, выбор слабой консистентности вместо сильной влечёт за собой и другие проблемы. Да, мы можем потребовать от партнёров дождаться получения последнего актуального состояния ресурса перед внесением изменений. Но очень неочевидно (и в самом деле неудобно) требовать от партнёров быть готовыми к тому, что они должны дождаться появления в том числе и тех изменений, которые сами же внесли.

```
// Создаёт заказ
let api = await api
    .createOrder(...)
// Возвращает список заказов
let pendingOrders = await api.
    getOngoingOrders(); // → []
// список пуст
```

Если мы не гарантируем сильную консистентность, то второй вызов может запросить вернуть пустой результат, ведь при чтении из реплики новый заказ мог просто до неё ещё не дойти.

Важный паттерн, который поможет в этой ситуации — это имплементация модели «read-your-writes»², а именно гарантии, что клиент всегда «видит» те изменения, которые сам же и внёс. Поднять уровень слабой консистентности до read-your-writes можно, если предложить клиенту самому передать токен, описывающий его последние изменения.

```
let order = await api
    .createOrder(...);
let pendingOrders = await api.
    getOngoingOrders({
        ...
        // Передаём идентификатор
        // последней операции
        // совершённой клиентом
        last_known_order_id: order.id
    })
```

В качестве такого токена может выступать, например:

- идентификатор или идентификаторы последних модифицирующих операций, выполненных клиентом;
- последняя известная клиенту версия ресурса (дата изменения, ETag).

Получив такой токен, сервер должен проверить, что ответ (список текущих операций, который он возвращает) соответствует токену, т.е. консистентность «в конечном счёте» сошлась. Если же она не сошлась (клиент передал дату модификации / версию / идентификатор последнего заказа новее, чем известна в данном узле сети), то сервер может реализовать одну из трёх стратегий (или их произвольную комбинацию):

- запросить данные из нижележащего БД или другого хранилища повторно;
- вернуть клиенту ошибку, индицирующую необходимость повторить запрос через некоторое время;
- обратиться к основной реплике БД, если таковая имеется, либо иным образом инициировать запрос мастер-данных из хранилища.

Достоинством этого подхода является удобство разработки клиента (по сравнению с полным отсутствием гарантий): ценой хранения токена версии разработчик клиента избавляется от возможной неконсистентности получаемых из API данных. Недостатков же здесь два:

- вам всё ещё нужно выбрать между масштабируемостью системы и постоянным фоном ошибок;
 - если при несовпадении версий клиента и сервера вы обращаетесь к мастер-реплике или перезапрашиваете данные, то увеличиваете нагрузку на хранилище сложно прогнозируемым образом;
 - если же вы генерируете ошибку для клиента, то в вашей системе всегда будет достаточно заметный фон таких ошибок, и, к тому же, партнёрам придётся написать клиентский код для их обработки;
- этот подход вероятностный и спасает только в части ситуаций — о чём мы расскажем в следующей главе.

Учитывая, что клиентское приложение может быть перезапущено или просто потерять токен, наиболее правильное (хотя не всегда приемлемое с точки зрения нагрузки) поведение сервера при отсутствии токена в запросе — форсировать возврат актуальных мастер-данных.

Риски перехода к событийной консистентности

Прежде всего, давайте зафиксируем один важный тезис: все обсуждаемые в настоящем разделе техники решения архитектурных проблем — вероятностные. Отказ от строгой консистентности означает, что даже при идеальной работе компонентов системы клиентские ошибки все равно будут возникать. Может показаться, что этот фон ошибок можно просто проигнорировать, но это весьма рискованно.

Представим, что в нашей системе из-за событийной консистентности клиенты с какой-то вероятностью не могут сделать заказ с первой попытки. Например, пользователь добавляет в приложении новый метод оплаты, но при создании заказа попадает на реплику, которая ещё не получила данные о новом способе оплаты. Так как пользователи довольно часто совершают эти две операции (добавление банковской карты и заказ) подряд, фон ошибок будет довольно значительным — пусть для примера 1% — но нас это пока не беспокоит: в худшем случае клиент выполнит автоматический перезапрос.

Предположим теперь, однако, что в новой версии приложения была допущена ошибка, и 0.1% пользователей не могут выполнить заказ вовсе по причине того, что клиент отсылает неправильный идентификатор метода оплаты. В отсутствие 1% ошибок консистентности данных эта проблема была бы выявлена очень быстро; но на фоне имеющихся ошибок найти её весьма непросто: для этого требуется настроить мониторинги так, чтобы они точно исключали ошибки, вызванные нестрогой консистентностью данных, а это может быть весьма непросто, а то и вообще невозможно. Автор этой книги сталкивался с такими ситуациями в своей работе: ошибку, затрагивающую небольшой процент пользователей, можно не замечать месяцами.

Таким образом, задача проактивного снижения фона ошибок критически важна. Мы можем постараться сделать так, чтобы при типичном профиле использования системы ошибок было меньше.

NB: оговорка про «типичный профиль» здесь не просто так: API предполагает вариативность сценариев его применения, и вполне может оказаться так, что кейсы использования API делятся на несколько сильно отличающихся с точки зрения толерантности к ошибкам групп (классический пример — это клиентские API, где завершения операций ждёт реальный пользователь,

против серверных API, где время исполнения само по себе менее важно, но может оказаться важным, например, массовый параллелизм операций). Если такое происходит — это сильный сигнал для того, чтобы выделить API для различных типовых сценариев в отдельные продукты в семействе API, о чём мы поговорим в главе «[Линейка сервисов API](#)» раздела «API как продукт».

Проиллюстрируем этот принцип на нашем примере с заказом кофе. Предположим, что мы реализуем следующую схему:

- оптимистичное управление синхронизацией (скажем, через идентификатор последнего заказа);
- «read-your-writes»-политика чтения списка заказов (вновь через отправку последнего идентификатора заказа в качестве токена);
- если токен не передан, клиент всегда получает актуальное состояние.

Тогда получить ошибку создания заказа можно только в одном из двух случаев:

- клиент неверно обращается с данными (не сохраняет идентификатор последнего заказа или ключ идемпотентности при перезапросах);
- клиент создаёт заказы одновременно с двух разных экземпляров приложения, которые не разделяют между собой состояние.

В первом случае речь идёт об ошибке имплементации приложения партнёра; второй случай означает, что пользователь намеренно пытается проверить систему на прочность, что вряд ли можно рассматривать как частотный кейс (либо, например, у пользователя сел телефон и он очень быстро продолжает работу с приложением с планшета — согласитесь, маловероятное развитие событий.)

Всё вышесказанное означает, что возникновение ошибки — исключительная ситуация, которая может действительно требовать расследования на предмет ошибки в коде.

Теперь посмотрим, что произойдёт, если мы откажемся от третьего требования, т.е. возврата мастер-данных клиенту, не передающему токен. У нас появится третья ситуация, когда клиент получит ошибку, а именно:

- клиентское приложение потеряло часть данных (токен синхронизации), и пробует повторить последний запрос.

NB: важно, что перезапрос может случить и по совершенно не техническим причинам: конечному пользователю может просто надоест ждать, он вручную перезапустит приложение и вручную создаст повторный заказ.

Математически вероятность получения ошибки выражается довольно просто: она равна отношению периода времени, требуемого для получения актуального состояния к типичному периоду времени, за который пользователь перезапускает приложение и повторяет заказ. (Следует, правда, отметить, что клиентское приложение может быть реализовано так, что даст вам ещё меньше времени, если оно пытается повторить несозданный заказ автоматически при запуске). Если первое зависит от технических характеристик системы (в частности, лага синхронизации, т.е. задержки репликации между мастером и копиями на чтение). А вот второе зависит от того, какого рода клиент выполняет операцию.

Если мы говорим о приложения для конечного пользователя, то типично время перезапуска измеряется для них в секундах, что в норме не должно превышать суммарного лага синхронизации — таким образом, клиентские ошибки будут возникать только в случае проблем с репликацией данных / ненадежной сети / перегрузки сервера.

Однако если мы говорим не о клиентских, а о серверных приложениях, здесь ситуация совершенно иная: если сервер решает повторить запрос (например, потому, что процесс был убит супервизором), он сделает это условно моментально — задержка может составлять миллисекунды. И в этом случае фон ошибок создания заказа будет достаточно значительным.

Таким образом, возвращать по умолчанию событийно-консистентные данные вы можете, если готовы мириться с фоном ошибок или если вы можете обеспечить задержку получения актуального состояния много меньшую, чем время перезапуска приложения на целевой платформе.

Примечания

¹ Consistency Model. Eventual Consistency

https://en.wikipedia.org/wiki/Consistency_model#Eventual_consistency

² Consistency Model. Read-Your-Writes Consistency

https://en.wikipedia.org/wiki/Consistency_model#Read-your-writes_consistency

Глава 19. Асинхронность и управление временем

Продолжим рассматривать предыдущий пример. Пусть на старте приложение получает *какое-то* состояние системы, возможно, не самое актуальное. От чего ещё зависит вероятность коллизий и как мы можем её снизить?

Напомним, что вероятность эта равна отношению периода времени, требуемого для получения актуального состояния к типичному периоду времени, за который пользователь перезапускает приложение и повторяет заказ. Повлиять на знаменатель этой дроби мы практически не можем (если только не будем преднамеренно вносить задержку инициализации API, что мы всё же считаем крайней мерой). Обратимся теперь к числителю.

Наш сценарий использования, напомним, выглядит так:

```
let pendingOrders = await api.  
    getOngoingOrders();  
if (pendingOrder.length == 0) {  
    let order = await api  
        .createOrder(...);  
}  
// Здесь происходит крэш приложения,  
// и те же операции выполняются  
// повторно  
let pendingOrders = await api.  
    getOngoingOrders(); // → []  
if (pendingOrder.length == 0) {  
    let order = await api  
        .createOrder(...);  
}
```

Таким образом, мы стремимся минимизировать следующий временной интервал: сетевая задержка передачи команды `createOrder` + время выполнения `createOrder` + время пропагации изменений до реплик. Первое мы вновь не контролируем (но, по счастью, мы можем надеяться на то, что сетевые задержки в пределах сессии величина плюс-минус постоянная, и, таким образом, последующий вызов `getOngoingOrders` будет задержан примерно на ту же величину); третья, скорее всего, будет обеспечиваться инфраструктурой нашего бэкенда. Поговорим теперь о втором времени.

Мы видим, что, если создание заказа само по себе происходит очень долго (здесь «очень долго» = «сопоставимо со временем запуска приложения»), то все наши усилия практически бесполезны. Пользователь может устать ждать исполнения вызова `createOrder`, выгрузить приложение и послать второй (и более) `createOrder`. В наших интересах сделать так, чтобы этого не происходило.

Но каким образом мы реально можем улучшить это время? Ведь создание заказа *действительно* может быть длительным — нам нужно выполнить множество проверок и дождаться ответа платёжного шлюза, подтверждения приёма заказа кофейней и т.д.

Здесь нам на помощь приходят асинхронные вызовы. Если наша цель — уменьшить число коллизий, то нам нет никакой нужды дожидаться, когда заказ будет *действительно* создан; наша цель — максимально быстро распространить по репликам знание о том, что заказ *принят к созданию*. Мы можем поступить следующим образом: создавать не заказ, а задание на создание заказа, и возвращать его идентификатор.

```
let pendingOrders = await api.  
    getOngoingOrders();  
if (pendingOrder.length == 0) {  
    // Вместо создания заказа  
    // размещаем задание на создание  
    let task = await api  
        .putOrderCreationTask(...);  
}  
// Здесь происходит крэш приложения,  
// и те же операции выполняются  
// повторно  
let pendingOrders = await api.  
    getOngoingOrders();  
    // → { tasks: [task] }
```

Здесь мы предполагаем, что создание задания требует минимальных проверок и не ожидает исполнения каких-то длительных операций, а потому происходит много быстрее. Кроме того, саму эту операцию — создание асинхронного задания — мы можем поручитьциальному сервису абстрактных заданий в составе бэкенда. Между тем, имея функциональность создания заданий и получения списка текущих заданий, мы значительно уменьшаем «серые зоны» состояния неопределённости, когда клиент не может узнать текущее состояние сервера точно.

Таким образом, мы естественным образом приходим к паттерну организации асинхронного API через очереди заданий. Мы используем здесь термин «асинхронность» логически — подразумевая отсутствие взаимных логических блокировок: посылающая сторона получает ответ на свой запрос сразу, не дожидаясь окончания исполнения запрошенной функциональности, и может продолжать взаимодействие с API, пока операция выполняется. При этом технически в современных системах блокировки клиента (и сервера) почти всегда не происходит и при обращении к синхронным эндпоинтам — однако логически продолжать работать с API, не дождавшись ответа на синхронный запрос, может быть чревато коллизиями подобно описанным выше.

Асинхронный подход может применяться не только для устранения коллизий и неопределённости, но и для решения других прикладных задач:

- организация ссылок на результаты операции и их кэширование (предполагается, что, если клиенту необходимо снова прочитать результат операции или же поделиться им с другим агентом, он может использовать для этого идентификатор задания);
- обеспечение идемпотентности операций (для этого необходимо ввести подтверждение задания, и мы фактически получим схему с черновиками операции, описанную в главе «[Описание конечных интерфейсов](#)»);
- нативное же обеспечение устойчивости к временному всплеску нагрузки на сервис — новые задачи встают в очередь (возможно, приоритизированную), фактически имплементируя «маркерное ведро»¹;
- организация взаимодействия в тех случаях, когда время исполнения операции превышает разумные значения (в случае сетевых API — типичное время срабатывания сетевых таймаутов, т.е. десятки секунд) либо является непредсказуемым.

Кроме того, асинхронное взаимодействие удобнее с точки зрения развития API в будущем: устройство системы, обрабатывающей такие запросы, может меняться в сторону усложнения и удлинения конвейера исполнения задачи, в то время как синхронным функциям придётся укладываться в разумные временные рамки, чтобы оставаться синхронными — что, конечно, ограничивает возможности рефакторинга внутренних механик.

NB: иногда можно встретить решение, при котором эндпойнт имеет двойной интерфейс и может вернуть как результат, так и ссылку на исполнение задания. Хотя для вас как разработчика API он может выглядеть логично (смогли «быстро» выполнить запрос, например, получить результат из кэша — вернули ответ; не смогли — вернули ссылку на задание), для пользователей API это решение крайне неудобно, поскольку заставляет поддерживать две ветки кода одновременно. Также встречается парадигма предоставления на выбор разработчику два набора эндпойнтов, синхронный и асинхронный, но по факту это просто перекладывание ответственности на партнёра.

Популярность данного паттерна также обусловлена тем, что многие современные микросервисные архитектуры «под капотом» также взаимодействуют асинхронно — либо через потоки событий, либо через промежуточный pub/sub сервис, реализующий паттерн «издатель-подписчик». Имплементация аналогичной асинхронности во внешнем API является самым простым способом обойти возникающие проблемы (читай, те же непредсказуемые и возможно очень большие задержки выполнения операций). Доходит до того, что в некоторых API абсолютно все операции делаются асинхронными (включая чтение данных), даже если никакой необходимости в этом нет.

Мы, однако, не можем не отметить, что, несмотря на свою привлекательность, повсеместная асинхронность влечёт за собой ряд достаточно неприятных проблем.

1. Если используется единый сервис очередей на все эндпойнты, то он становится единой точкой отказа. Если события не успевают публиковаться и/или обрабатываться — возникает задержка исполнения всех эндпойнтов. Если же, напротив, для каждого функционального домена организуется свой сервис очередей, то это приводит к кратному усложнению внутренней архитектуры и увеличению расходов на мониторинг и исправление проблем.
2. Написание кода для партнёра становится гораздо сложнее. Дело даже не в физическом объёме кода (в конце концов, создание общего компонента взаимодействия с очередью заданий — не такая уж и сложная задача), а в том, что теперь в отношении каждого вызова разработчик должен поставить себе вопрос: что произойдёт, если его обработка займёт длительное время. Если в случае с синхронными эндпойнтами мы по

умолчанию полагаем, что они отрабатывают за какое-то разумное время, меньшее, чем типичный таймаут запросов (например, в клиентских приложения можно просто показать пользователю спиннер), то в случае асинхронных эндпоинтов такой гарантии у нас не просто нет — она не может быть дана.

3. Использование очередей заданий может повлечь за собой свои собственные проблемы, не связанные с собственно обработкой запроса:

- задание может быть «потеряно», т.е. никогда не быть обработанным;
- события смены статусов могут приходить в неверном порядке и/или повторяться, что может повлиять на публичные интерфейсы;
- под идентификатором задания могут быть по ошибке размещены неправильные данные (соответствующие другому заданию) или же данные могут быть повреждены.

Эти ситуации могут оказаться совершенно неожиданными для разработчиков и приводить к крайне сложным в воспроизведении ошибкам в приложениях.

4. Как следствие вышесказанного, возникает вопрос осмыслинности SLA такого сервиса. Через асинхронные задачи легко можно поднять аптайм API до 100% — просто некоторые запросы будут выполнены через пару недель, когда команда поддержки, наконец, найдёт причину задержки. Но такие гарантии пользователям вашего API, разумеется, совершенно не нужны: их пользователи обычно хотят выполнить задачу *сейчас* или хотя бы за разумное время, а не через две недели.

Поэтому, при всей привлекательности идеи, мы всё же склонны рекомендовать ограничиться асинхронными интерфейсами только там, где они действительно критически важны (как в примере выше, где они снижают вероятность коллизий), и при этом иметь отдельные очереди для каждого кейса. Идеальное решение с очередями — то, которое вписано в бизнес-логику и вообще не выглядит очередью. Например, ничто не мешает нам объявить состояние «задание на создание заказа принято и ожидает исполнения» просто отдельным статусом заказа, а его идентификатор сделать идентификатором будущего заказа:

```
let pendingOrders = await api.  
    getOngoingOrders();  
if (pendingOrder.length == 0) {  
    // Не называем это «заданием» –  
    // просто создаём заказ  
    let order = await api  
        .createOrder(...);  
}  
// Здесь происходит крэш приложения,  
// и те же операции выполняются  
// повторно  
let pendingOrders = await api.  
    getOngoingOrders();  
/* → { orders: [  
    order_id: <идентификатор задания>,  
    status: "new"  
] } */
```

NB: отметим также, что в формате асинхронного взаимодействия можно передавать не только бинарный статус (выполнено задание или нет), но и прогресс выполнения в процентах, если это возможно.

Примечания

¹ Token bucket

https://en.wikipedia.org/wiki/Token_bucket

Глава 20. Списки и организация доступа к ним

В предыдущей главе мы пришли вот к такому интерфейсу, позволяющему минимизировать коллизии при создании заказов:

```
let pendingOrders = await api
  .getOngoingOrders();
→
{ orders: [{
  order_id: <идентификатор задания>,
  status: "new"
}, ...]}
```

Внимательный читатель может подметить, что этот интерфейс нарушает нашу же рекомендацию, данную в главе «[Описание конечных интерфейсов](#)»: количество возвращаемых данных в любом ответе должно быть ограничено, но в нашем интерфейсе отсутствуют какие-либо лимиты. Эта проблема существовала и в предыдущих версиях этого эндпойнта, но отказ от синхронного создания заказа её усугубил: операция создания задания должна работать максимально быстро, и, следовательно, почти все проверки лимитов мы должны проводить асинхронно — а значит, клиент потенциально может создать очень много заданий, что может многократно увеличить размер ответа функции `getOngoingOrders`.

NB: конечно, не иметь *вообще никакого* ограничения на создание заданий — не самое мудрое решение; какие-то легковесные проверки лимитов должны быть в API. Тем не менее, в рамках этой главы мы фокусируемся именно на проблеме размера ответа сервера.

Исправить эту проблему достаточно просто — можно ввести лимит записей и параметры фильтрации и сортировки, например так:

```
api.getOngoingOrders({
  // необязательное, но имеющее
  // значение по умолчанию
  "limit": 100,
  "parameters": {
    "order_by": [
      {
        "field": "created_iso_time",
        "direction": "desc"
      }
    ]
  }
})
```

Однако введение лимита ставит другой вопрос: если всё же количество записей, которые нужно выбрать, превышает лимит, каким образом клиент должен получить к ним доступ?

Стандартный подход к этой проблеме — введение параметра `offset` или номера страницы данных:

```
api.getOngoingOrders({
    // необязательное, но имеющее
    // значение по умолчанию
    "limit": 100,
    // По умолчанию - 0
    "offset": 100
    "parameters"
});
```

Однако, как нетрудно заметить, в нашем случае этот подход приведёт к новым проблемам. Пусть для простоты в системе от имени пользователя выполняется три заказа:

```
[{
    "id": 3,
    "created_iso_time": "2022-12-22T15:35",
    "status": "new"
}, {
    "id": 2,
    "created_iso_time": "2022-12-22T15:34",
    "status": "new"
}, {
    "id": 1,
    "created_iso_time": "2022-12-22T15:33",
    "status": "new"
}]
```

Приложение партнёра запросило первую страницу списка заказов:

```
api.getOrders({
  "limit": 2,
  "parameters": {
    "order_by": [
      {"field": "created_iso_time",
       "direction": "desc"
     }
   ]
})
→
{
  "orders": [
    {"id": 3, ...},
    {"id": 2, ...}
  ]
}
```

Теперь приложение запрашивает вторую страницу "limit": 2, "offset": 2 и ожидает получить заказ "id": 1. Предположим, однако, что за время, прошедшее с момента первого запроса, в системе появился новый заказ с "id": 4.

```
[{
  "id": 4,
  "created_iso_time": "2022-12-22T15:36",
  "status": "new"
}, {
  "id": 3,
  "created_iso_time": "2022-12-22T15:35",
  "status": "new"
}, {
  "id": 2,
  "created_iso_time": "2022-12-22T15:34",
  "status": "ready"
}, {
  "id": 1,
  "created_iso_time": "2022-12-22T15:33",
  "status": "new"
}]
```

Тогда, запросив вторую страницу заказов, вместо одного заказа "id": 1, приложение партнёра получит повторно заказ "id": 2:

```
api.getOrders({
  "limit": 2,
  "offset": 2
  "parameters"
})
→
{
  "orders": [
    {
      "id": 2, ...
    },
    {
      "id": 1, ...
    }
  ]
}
```

Такие перестановки крайне неудобны и для пользовательских интерфейсов — если, допустим, предположить, что заказы запрашивает бухгалтер партнёра, чтобы рассчитать выплаты, то и он легко может просто не заметить, что какой-то заказ посчитан дважды. Однако в случае *программной* интеграции ситуация становится намного сложнее: разработчику приложения нужно написать достаточно неочевидный код (сохраняющий состояние уже полученных страниц данных), чтобы провести такой перебор корректно.

Отметим теперь, что ситуацию легко можно сделать гораздо более запутанной. Например, если мы добавим сортировку не только по дате создания, но и по статусу заказа:

```
api.getOrders({
  "limit": 2,
  "parameters": {
    "order_by": [
      {
        "field": "status",
        "direction": "desc"
      },
      {
        "field": "created_iso_time",
        "direction": "desc"
      }
    ]
  }
})
→
{
  "orders": [
    {
      "id": 3,
      "status": "new"
    },
    {
      "id": 2,
      "status": "new"
    }
  ]
}
```

Предположим, что в интервале между запросами первой и второй страницы заказ "id": 1 изменил свой статус, и, соответственно, свое положение в списке, став самым первым. Тогда, запросив вторую страницу, приложение партнёра получит (повторно) только заказ с "id": 2, а заказ "id": 1 попросту вообще пропустит, и вновь не будет располагать вообще никаким способом узнать об этом пропуске.

Повторимся, такой подход плохо работает для визуальных интерфейсов, но в программных интерфейсах он практически гарантированно приведёт к ошибкам. **API должно предоставлять способы перебора больших списков, которые гарантируют клиенту получение полного и целостного набора данных.**

Если не вдаваться в детали имплементации, то можно выделить три основных паттерна организации такого перебора — в зависимости от того, как сами по себе организованы данные.

Иммутабельные списки

Проще всего организовать доступ, конечно, если список в принципе не может измениться, т.е. все данные в нём фиксированы. Тогда даже схема с limit/offset прекрасно работает и не требует дополнительных ухищрений. К сожалению, в реальных предметных областях встречается редко.

Пополняемые списки, иммутабельные данные

Более распространённый случай — когда не меняются данные в списке, но появляются новые элементы. Чаще всего речь идёт об очередях событий — например, новых сообщений или уведомлений. Представим, что в нашем кофейном API есть эндпойнт для партнёра для получения истории предложений:

```
GET /v1/partners/{id}/offers/history?  
?limit=<лимит>  
→  
{  
  "offer_history": [ {  
    // Идентификатор элемента  
    // списка  
    "id",  
    // Идентификатор пользователя,  
    // получившего оффер  
    "user_id",  
    // Время и дата поиска  
    "occurred_at",  
    // Установленные пользователем  
    // параметры поиска предложений  
    "search_parameters",  
    // Офферы, которые пользователь  
    // увидел  
    "offers"  
  } ]  
}
```

Данные в списке по своей природе неизменны — они отражают уже случившийся факт: пользователь искал предложения, и увидел вот такой их список. Но новые элементы списка постоянно возникают, причём вполне могут возникать большими сериями, если пользователь сделал несколько поисков подряд.

Партнёр может использовать эти данные, например, для реализации двух сценариев:

1. Анализ поведения пользователей в реальном времени (скажем, партнёр может отправить пользователю пуш-уведомление с предложением скидки тем пользователям, которые искали).
2. Построение статистического отчёта (скажем, подсчёт конверсии по часам).

Для этих сценариев нам необходимо предоставить партнёру две операции со списками:

1. Для первой задачи, получение в реальном времени всех новых элементов с момента последнего запроса.
2. Для второй задачи, перебор списка, т.е. получение всех запросов за указанный временной интервал.

Оба сценария покрываются limit/offset-схемой, но требуют значительных усилий при написании кода, так как партнёру в обоих случаях нужно как-то ориентироваться, на сколько элементов очередь событий сдвинулась с момента последнего запроса. Отдельно отметим, что использование limit/offset-подхода приводит к невозможности кэширования ответов — повторные запросы с той же парой limit/offset могут возвращать совершенно разные результаты.

Решить эту проблему мы можем, если будем ориентироваться не на позицию элемента в списке (которая может меняться), а на какие-то другие признаки. Нам важно здесь следующее условие: по этому признаку мы можем однозначно определить, какие элементы списка «более новые» по отношению к нему (т.е. имеют меньшие индексы), а какие «более старые».

Если хранилище данных, в котором находятся элементы списка, позволяет использовать монотонно растущие идентификаторы (что на практике означает два условия: (1) база данных поддерживает автоинкрементные колонки, (2) вставка данных осуществляется блокирующим образом), то идентификатор элемента в списке является максимально удобным способом организовать перебор:

```
// Получить записи новее,  
// чем запись с указанным id  
GET /v1/partners/{id}/offers/history?  
newer_than=<item_id>&limit=<limit>  
// Получить записи более старые,  
// чем запись с указанным id  
GET /v1/partners/{id}/offers/history?  
older_than=<item_id>&limit=<limit>
```

Первый формат запроса позволяет решить задачу (1), т.е. получить все элементы списка, появившиеся позднее последнего известного; второй формат — задачу (2), т.е. перебрать нужно количество записей в истории запросов. Важно, что первый запрос при этом ещё и кэшируемый.

NB: отметим, что в главе «[Описание конечных интерфейсов](#)» мы давали рекомендацию не предоставлять доступ во внешнем API к инкрементальным id. Однако, схема этого и не требует: внешние идентификаторы могут быть произвольными (не обязательно монотонными) — достаточно, чтобы они однозначно конвертировались во внутренние монотонные идентификаторы.

Другим способом организации такого перебора может быть дата создания записи, но этот способ чуть сложнее в имплементации:

- дата создания двух записей может полностью совпадать, особенно если записи могут массово генерироваться программно; в худшем случае может получиться так, что в один момент времени было создано больше записей, чем максимальный лимит их извлечения, и тогда часть записей вообще нельзя будет перебрать;
- если хранилище данных поддерживает распределённую запись, то может оказаться, что более новая запись имеет чуть меньшую дату создания, нежели предыдущая известная (поскольку часы на разных виртуальных машинах могут идти чуть по-разному, и добиться хотя бы микросекундной точности крайне сложно¹, т.е. нарушится требование монотонности по признаку даты; если использование такого хранилища не имеет альтернативы, необходимо выбрать одно из двух зол:
 - внести рукотворные задержки, т.е. возвращать в API только элементы, созданные более чем N секунд назад — так, чтобы N было заведомо больше неравномерности хода часов (эта техника может использоваться и в тех случаях, когда список формируется асинхронно) — однако надо иметь в виду, что это решение вероятностное и всегда есть шанс отдачи неверных данных в случае проблем с синхронизацией на бэкенде;
 - описать нестабильность порядка новых элементов списка в документации и переложить решение этой проблемы на партнёров.

Часто подобные интерфейсы перебора данных (путём указания граничного значения) обобщают через введение понятия *курсор*:

```
// Инициализируем поиск
POST /v1/partners/{id}/offers/history
  /search
{
  "order_by": [
    {
      "field": "created",
      "direction": "desc"
    }
  ]
}
→
{
  "cursor": "TmluZSBQcm1uY2VzIGluIEFtYmVy"
}
```

```
// Получение порции данных
GET /v1/partners/{id}/offers/history
  ?cursor=TmluZSBQcm1uY2VzIGluIEFtYmVy
  &limit=100
→
{
  "items": [...],
  // Указатель на следующую
  // страницу данных
  "cursor": "R3VucyBvZiBBdmFsb24"
}
```

Курсором в данной ситуации может представлять собой просто идентификатор последней записи, а может содержать зашифрованное представление всех параметров поиска. Одним из преимуществ использования абстрактного курсора вместо конкретных монотонных полей является возможность сменить нижележащую технологию (например, перейти от использования последнего известного идентификатора к использованию даты последней известной записи) без слома обратной совместимости. (Поэтому курсоры часто представляют собой «непрозрачные» строки: предоставление читаемых курсоров будет означать, что вы теперь обязаны поддерживать формат курсора, даже если никогда его не документировали. Лучше возвращать курсоры зашифрованными или хотя бы в таком виде, который не вызывал бы желания его раскодировать и поэкспериментировать с параметрами.)

В подходе с курсорами вы сможете без нарушения обратной совместимости добавлять новые фильтры и виды сортировки — при условии, конечно, что вы сможете организовать хранение данных таким образом, чтобы перебор с курсором работал однозначно.

```

// Инициализируем поиск
POST /v1/partners/{id}/offers/history
  search
{
  // Добавим фильтр по виду кофе
  "filter": {
    "recipe": "americano"
  },
  // добавим новую сортировку
  // по удалённости от указанной
  // географической точки
  "order_by": [
    {
      "mode": "distance",
      "location": [-86.2, 39.8]
    }
  ]
}
→
{
  "items": [...],
  "cursor":
    "Q29mZmVlIGFuZCDBb250ZW1wbGF0aW9u"
}

```

Небольшое примечание: признаком окончания перебора часто выступает отсутствие курсора на последней странице с данными; мы бы рекомендовали так не делать (т.е. всё же возвращать курсор, указывающий на пустой список), поскольку это позволит добавить функциональность динамической вставки данных в конец списка.

NB: в некоторых источниках перебор через идентификаторы / даты создания / курсор, напротив, не рекомендуется по следующей причине: пользователю невозможно показать список страниц и дать возможность выбрать произвольную. Здесь следует отметить, что:

- подобный кейс — список страниц и выбор страниц — существует только для пользовательских интерфейсов; представить себе API, в котором действительно требуется доступ к случайным страницам данных мы можем с очень большим трудом;
- если же мы всё-таки говорим об API приложения, которое содержит элемент управления с постраничной навигацией, то наиболее правильный подход — подготавливать данные для этого элемента управления на стороне сервера, в т.ч. генерировать ссылки на страницы;
- подход с курсором не означает, что `limit/offset` использовать нельзя — ничто не мешает сделать двойной интерфейс, который будет отвечать и на запросы вида `GET /items?cursor=...`, и на запросы вида `GET /items?offset=... &limit=...`;

- наконец, если возникает необходимость предоставлять доступ к произвольной странице в пользовательском интерфейсе, то следует задать себе вопрос, какая проблема тем самым решается; вероятнее всего с помощью этой функциональности пользователь что-то ищет: определенный элемент списка или может быть позицию, на которой он закончил работу со списком в прошлый раз; возможно, для этих задач следует предоставить более удобные элементы управления, нежели перебор страниц.

Общий сценарий

Увы, далеко не всегда данные организованы таким образом, чтобы из них можно было составить иммутабельные списки. Например, в указанном выше примере поиска текущих заказов мы никак не можем представить постраничный список заказов, находящихся сейчас в статусе «исполняется» — просто потому, что заказы переходят в другие статусы и в реальном времени пропадают из списка. Для таких сложных случаев нам нужно в первую очередь ориентироваться на *сценарии использования* данных.

Бывает так, что задачу можно *свести* к иммутабельному списку, если по запросу создавать какой-то слепок запрошенных данных. Во многих случаях работа с таким срезом данных по состоянию на определённую дату более удобна и для партнёров, поскольку снимает необходимость учитывать текущие изменения. Часто такой подход работает с «холодными» хранилищами, которые по запросу выгружают какой-то подмассив данных в «горячее» хранилище.

```
POST /v1/orders/archive/retrieve
{
  "created_iso_date": {
    "from": "1980-01-01",
    "to": "1990-01-01"
  }
}
{
  "task_id": <идентификатор
             задания на выгрузку данных>
}
```

Недостаток такого подхода понятен — он требует дополнительных (и зачастую немалых) затрат на создание и хранение слепка, а потому требует и отдельной тарификации. Кроме того, проблема-то сама по себе никуда не делась: мы перенесли её из публичного API на уровень реализации нашего бэкенда, но нам всё ещё нужно каким-то образом перебрать массив данных и сформировать консистентный слепок.

Обратный подход к организации такого перебора — это принципиально не предоставлять больше одной страницы данных. Т.е. партнёр может запросить только «последние» в каком-то смысле записи. Такой подход обычно применяется в одном из трёх случаев:

- если эндпойнт представляет собой поисковый алгоритм, который выбирает наиболее релевантные данные — как мы все отлично знаем, вторая страница поисковой выдачи уже никому не нужна;
- если эндпойнт нужен для того, чтобы *изменить* данные — например, сервис партнёра достаёт все заказы в статусе "new" и переводит в статус «принято к исполнению»; тогда пагинация на самом деле и не нужна, поскольку каждым своим действием партнёр удаляет часть элементов из списка;
 - частный случай такого изменения — просто пометить полученные данные прочитанными;
- наконец, если через эндпойнт предоставляются только «горячие» необработанные данные, а к обработанным данным доступ предоставляемся уже через стандартные интерфейсы.

Если ни один из описанных вариантов не подходит по тем или иным причинам, единственный способ организации доступа — это изменение предметной области. Если мы не можем консистентно упорядочить элементы списка, нам нужно найти какой-то другой срез тех же данных, который мы можем упорядочить. Например, в нашем случае доступа к новым заказам мы можем упорядочить *список событий* создания нового заказа:

```

// Получить все события создания
// заказа, более старые,
// чем запись с указанным id
GET /v1/orders/created-history4
?older_than=<item_id>&limit=<limit>
→
{
  "orders_created_events": [
    {
      "id": <идентификатор события>,
      "occurred_at",
      // Идентификатор заказа
      "order_id"
    }, ...
  }
}

```

События иммутабельны, и их список только пополняется, следовательно, организовать перебор этого списка вполне возможно. Да, событие — это не то же самое, что и сам заказ: к моменту прочтения партнёром события, заказ уже давно может изменить статус. Но, тем не менее, мы предоставили возможность перебрать *все* новые заказы, пусть и не самым оптимальным образом.

NB: в вышеприведённых фрагментах кода мы опустили метаданные ответа — такие как общее число элементов в списке, флаг типа `has_more_items` для индикации необходимости продолжить перебор и т.д. Хотя эти метаданные необязательны (клиент узнает размер списка, когда переберёт его полностью), их наличие повышает удобство работы с API для разработчиков, и мы рекомендуем их добавлять.

Примечания

⁴ Ranganathan, K. A Matter of Time: Evolving Clock Sync for Distributed Databases
<https://www.yugabyte.com/blog/evolving-clock-sync-for-distributed-databases/>

Глава 21. Двунаправленные потоки данных. Push и poll-модели

В предыдущей главе мы рассмотрели следующий кейс: партнёр получает информацию о новых событиях, произошедших в системе, периодически опрашивая эндпойнт, поддерживающий отдачу упорядоченных списков.

```
GET /v1/orders/created-history<
?older_than=<item_id>&limit=<limit>
→
{
  "orders_created_events": [ {
    "id",
    "occurred_at",
    "order_id"
  }, ... ]
}
```

Подобный паттерн (известный как «поллинг»¹) — наиболее часто встречающийся способ организации двунаправленной связи в API, когда партнёру требуется не только отправлять какие-то данные на сервер, но и получать оповещения от сервера об изменении какого-то состояния.

При всей простоте, поллинг всегда заставляет искать компромисс между отзывчивостью, производительностью и пропускной способностью системы:

- чем длиннее интервал между последовательными запросами, тем больше будет задержка между изменением состояния на сервере и получением информации об этом на клиенте, и тем потенциально большим будет объём данных, которые необходимо будет передать за одну итерацию;
- с другой стороны, чем этот интервал короче, чем большее количество запросов будет совершаться зря, т.к. никаких изменений в системе за прошедшее время не произошло.

Иными словами, поллинг всегда создаёт какой-то фоновый трафик в системе, но никогда не гарантирует максимальной отзывчивости. Иногда эту проблему решают с помощью «долгого поллинга» (long polling) ⁽²⁾ — т.е. целенаправленно замедляют отдачу сервером ответа на длительное (секунды, десятки секунд) время до тех пор, пока на сервере не появится сообщение для

передачи — однако мы не рекомендуем использовать этот подход в современных системах из-за связанных технических проблем (в частности, в условиях ненадёжной сети у клиента нет способа понять, что соединение на самом деле потеряно, и нужно отправить новый запрос, а не ожидать ответа на текущий).

Если оказывается, что обычного поллинга для решения пользовательских задач недостаточно, то можно перейти к обратной модели (*push*): сервер сам сообщает клиенту, что в системе произошли изменения.

Хотя и проблема, и способы её решения выглядят похоже, в настоящий момент применяются совершенно разные технологии для доставки сообщений от бэкенда к бэкенду и от бэкенда к клиентскому устройству.

Доставка сообщений на клиентское устройство

Поскольку разнообразные мобильные платформы и «умные устройства» (Internet of Things, IoT) сейчас составляют значительную долю всех клиентских устройств, на технологии взаимного обмена данных между сервером и конечным пользователем накладываются значительные ограничения с точки зрения экономии заряда батареи (и отчасти трафика). Многие производители платформ и устройств следят за потребляемыми приложением ресурсами, и могут отправлять приложение в фон или вовсе закрывать открытые соединения. В такой ситуации частый поллинг стоит применять только в активных фазах работы приложения (т.е. когда пользователь непосредственно взаимодействует с UI) либо если приложение работает в контролируемой среде (например, используется сотрудниками компании-партнера непосредственно в работе, и может быть добавлено в системные исключения).

Альтернатив поллингу на данный момент можно предложить три:

1. Дуплексные соединения

Самый очевидный вариант — использование технологий, позволяющих передавать по одному соединению сообщения в обе стороны. Наиболее известной из таких технологий является *WebSockets*³. Иногда для организации полнодуплексного соединения применяется *Server Push*, предусмотренный протоколом *HTTP/2*⁴, однако надо отметить, что формально спецификация не

предусматривает такого использования. Также существует протокол WebRTC⁵, но он, в основном, используется для обмена медиа-данными между клиентами, редко для клиент-серверного взаимодействия.

Несмотря на то, что идея в целом выглядит достаточно простой и привлекательной, в реальности её использование довольно ограничено. Поддержки инициирования *сервером* отправки сообщения обратно на клиент практически нет в популярном серверном ПО и фреймворках (gRPC поддерживает потоки сообщений с сервера⁶, но их всё равно должен инициировать клиент; использование потоков для пересылки сообщений по мере их возникновения — то же самое использование HTTP/2 Server Push в обход спецификации, что, фактически, работает как тот же самый long polling, только чуть более современный), и существующие стандарты спецификаций API также не поддерживают такой обмен данными: WebSockets является низкоуровневым протоколом, и формат взаимодействия придётся разработать самостоятельно.

Дуплексные соединения по-прежнему страдают от ненадёжной сети и требуют дополнительных ухищрений для того, чтобы отличить сетевую проблему от отсутствия новых сообщений. Всё это приводит к тому, что данная технология используется в основном веб-приложениями.

2. Раздельный канал обратного вызова

Вместо дуплексных соединений можно использовать два раздельных канала — один для отправки сообщений на сервер, другой для получения сообщений с сервера. Наиболее популярной технологией такого рода является MQTT⁷. Хотя эта технология считается максимально эффективной в силу использования низкоуровневых протоколов, её достоинства порождают и её недостатки:

- технология в первую очередь предназначена для имплементации паттерна pub/sub и ценна наличием соответствующего серверного ПО (MQTT Broker); применить её для других задач, особенно для двунаправленного обмена данными, может быть сложно;
- низкоуровневый протокол диктует необходимость разработки собственного формата данных.

Существует также веб-стандарт отправки серверных сообщений Server-Sent Events⁸ (SSE). Однако по сравнению с WebSocket он менее функциональный (только текстовые данные, односторонний поток сообщений) и поэтому используется редко.

3. Сторонние сервисы отправки push-уведомлений

Одна из неприятных особенностей технологии типа long polling / WebSocket / SSE / MQTT — необходимость поддерживать открытое соединение между клиентом и сервером, что для мобильных приложений может быть проблемой с точки зрения производительности и энергопотребления. Один из вариантов решения этой проблемы — делегирование отправки уведомлений стороннему сервису (самым популярным выбором на сегодня является Firebase Cloud Messaging от Google), который в свою очередь доставит уведомление через встроенные механизмы платформы. Использование встроенных в платформу сервисов получения уведомлений снимает с разработчика головную боль по написанию кода, поддерживающего открытое соединение, и снижает риски неполучения сообщения. Недостатками third-party серверов сообщений является необходимость платить за них и ограничения на размер сообщения.

Кроме того, отправка push-уведомлений на устройство конечного пользователя страдает от одной большой проблемы: процент успешной доставки уведомлений никогда не равен 100; потери сообщений могут достигать десятков процентов. С учётом ограничений на размер контента, скорее правильно говорить не о push-модели, а о комбинированной: приложение продолжает периодически опрашивать сервер, а пуши являются триггером для внеочередного опроса. (На самом деле, это соображение в той или иной мере применимо к любой технологии доставки сообщений на клиент. Низкоуровневые протоколы предоставляют больше возможностей управлять гарантиями доставки, но, с учётом ситуации с принудительным закрытием соединений системой, иметь в качестве страховки низкочастотный поллинг в приложении почти никогда не бывает лишним.)

Использование push-технологий в публичном API

Следствием описанной выше фрагментации клиентских технологий является фактическая невозможность использовать любую из них кроме обычного поллинга в публичном API. Требование к партнёрам реализовать получение сообщений через WebSocket / MQTT / SSE каналы значительно повышает порог входа в API, т.к. работа с низкоуровневыми протоколами, к тому же плохо покрытыми существующими IDL и кодогенерацией, требует значительных ресурсов и чревата ошибками имплементации. Если же вы решите предоставлять готовый SDK к такому API, то вам придётся самостоятельно разработать его под каждую целевую платформу (что, повторимся, само по себе трудоёмко). Учитывая, что HTTP-поллинг кратно проще в реализации, а его недостатки проявляются только там, где *действительно* нужно экономить трафик и вычислительные ресурсы, мы склонны рекомендовать предоставлять альтернативные каналы получения сообщений только *в дополнение* к поллингу, но никак не вместо него.

Хорошим решением для публичного API могли бы стать системные пуши, но здесь возникает другая проблема: разработчики приложений не склонны давать сторонним сервисам право на отсылку push-уведомлений, и на то есть большой список причин, начиная от расходов на отправку и заканчивая проблемами безопасности.

Фактически самый удобный способ организовать доставку сообщений от бэкенда публичного API пользователю партнёрского сервиса — это доставить сообщение с бэкенда на бэкенд, чтобы сервис партнёра сам транслировал сообщение на клиенты через push-уведомления или любую другую технологию, которую партнёр выбрал для разработки своего приложения.

Доставка сообщений **backend-to-backend**

В отличие от клиентских приложений, серверные API практически безальтернативно используют единственный подход для организации двустороннего взаимодействия [помимо поллинга, который работает на сервере точно так же, как и на клиенте, и имеет те же достоинства и недостатки] — отдельный канал связи для обратных вызовов. В случае публичных API практически безальтернативно такой технологией является использование URL обратного вызова (т.н. «webhook»).

Хотя long polling, WebSocket, MQTT и HTTP/2 Push тоже вполне применимы для backend-2-backend взаимодействия, мы сходу затрудняемся назвать примеры популярных API, которые использовали бы эти технологии. Главными причинами такого положения дел нам видятся:

- меньшая критичность к проблемам производительности (у сервера фактически нет ограничений по расходу трафика, и поддержание открытых соединений тоже не является проблемой);
- большая требовательность к гарантиям доставки;
- широкий выбор готовых компонентов для разработки webhook-ов (поскольку, фактически, это просто обычный веб-сервер);
- возможность описать такое взаимодействие спецификацией и использовать кодогенерацию.

При интеграции через webhook, партнёр указывает URL своего собственного сервера обработки сообщений, и сервер API вызывает этот эндпойнт для оповещения о произошедшем событии.

Предположим, что в нашем кофейном примере партнёр располагает некоторым бэкендом, готовым принимать оповещения о новых заказах, поступивших в его кофейни, и нам нужно договориться о формате взаимодействия. Решение этой задачи декомпозируется на несколько шагов:

1. Договорённость о контракте

В зависимости от важности партнёра для вашего бизнеса здесь возможны разные варианты:

- производитель API может реализовать возможность вызова webhook-а в формате, предложенном партнёром;
- наоборот, партнёр должен разработать эндпойнт в стандартном формате, предлагаемом производителем API;
- любой промежуточный вариант.

Важно, что в любом случае должен существовать формальный контракт (очень желательно — в виде спецификации) на форматы запросов и ответов эндпойнта-webhook-а и возникающие ошибки.

2. Договорённость о способах авторизации и аутентификации

Так как webhook-и представляют собой обратный канал взаимодействия, для него придётся разработать отдельный способ авторизации — это партнёр должен проверить, что запрос исходит от нашего бэкенда, а не наоборот. Мы повторяем здесь настоятельную рекомендацию не изобретать безопасность и использовать существующие стандартные механизмы, например, mTLS⁹, хотя в реальном мире с большой долей вероятности придётся использовать архаичные техники типа фиксации IP-адреса вызывающего сервера.

3. API для задания адреса webhook-а

Так как callback-эндпойнт разрабатывается партнёром, его URL нам априори неизвестен. Должен существовать интерфейс (возможно, в виде кабинета партнёра) для задания URL webhook-а (и публичных ключей авторизации).

Важно. К операции задания адреса callback-а нужно подходить с максимально возможной серьёзностью (очень желательно требовать второй фактор авторизации для подтверждения этой операции), поскольку, получив доступ к такой функциональности, злоумышленник может совершить множество весьма неприятных атак:

- если указать в качестве приёмника сторонний URL, можно получить доступ к потоку всех заказов партнёра и при этом вызвать перебои в его работе;
- такая уязвимость может также эксплуатироваться с целью организации DoS-атаки на сторонние сервисы;

- если указать в качестве webhook-а URL интранет-сервисов компании-провайдера API, можно осуществить SSRF-атаку¹⁰ на инфраструктуру самой компании.

Типичные проблемы интеграции через webhook

Двунаправленные интеграции (и клиентские, и серверные — хотя последние в большей степени) несут в себе очень неприятные риски для провайдера API. Если в общем случае качество работы API зависит в первую очередь от самого разработчика API, то в случае обратных вызовов всё в точности наоборот: качество работы интеграции напрямую зависит от того, как код webhook-эндпойнта написан партнёром. Мы можем столкнуться здесь с самыми различными видами проблем в партнёрском коде:

- webhook может возвращать false-positive ответы, когда сообщение не было обработано, но сервер партнёра тем не менее ошибочно вернул код успеха;
- и наоборот, возможны false-negative ответы, когда сообщение было обработано, но эндпойнт почему-то вернул ошибку (или просто ответил в неправильном формате);
- webhook может обрабатывать входящие запросы очень долго — возможно, настолько долго, что сервер сообщений просто не будет успевать их отправить;
- могут быть допущены ошибки в реализации идемпотентности, и повторная обработка одного и того же сообщения партнёром может приводить к ошибкам или некорректности данных в системе партнёра;
- размер тела сообщение может превысить лимит, выставленный на веб-сервере партнёра;
- авторизация на стороне партнёра может не проверяться или проверяться некорректно, и злоумышленник легко может отправить какие-то выгодные ему запросы, представившись сервером API;
- наконец, эндпойнт может быть просто недоступен по множеству различных причин, от проблем в data-центре, где расположены сервера партнёра, до банальной человеческой ошибки при смене URL webhook-а.

Очевидно, вы никак не можете гарантировать, что партнёр не совершил какую-то из перечисленных ошибок. Но вы можете попытаться минимизировать возможный ущерб:

1. Состояние системы должно быть восстановимо. Даже если партнёр неправильно обработал сообщения, всегда должна быть возможность реабилитироваться и получить список последних событий и/или полное состояние системы, чтобы исправить случившиеся ошибки.
2. Помогите партнёру написать правильный код, зафиксировав в документации неочевидные моменты, с которыми могут быть незнакомы неопытные разработчики:
 - ключи идемпотентности каждой операции;
 - гарантии доставки (exactly once, at least once; см. описание гарантий доставки¹¹ на примере технологии Apache Kafka);
 - будет ли сервер генерировать параллельные запросы к webhook-у и, если да, каково максимальное количество одновременных запросов;
 - гарантирует ли сервер строгий порядок сообщений (запросы всегда доставляются в порядке от самого старого к самому новому)
 - размеры полей и сообщений в байтах;
 - политика перезапросов при получении ошибки.
3. Должна быть реализована система мониторинга состояния партнёрских эндпойнтов:
 - при появлении большого числа ошибок (таймаутов) должно срабатывать оповещение (в т.ч. оповещение партнёра о проблеме), возможно, с несколькими уровнями эскалации;
 - если в очереди скапливается большое количество необработанных событий, должен существовать механизм деградации (ограничения количества запросов в адрес партнёра — возможно в виде срезания спроса, т.е. частичного отказа в обслуживании конечных пользователей) и полного аварийного отключения партнёра.

Очереди сообщений

Для внутренних API технология webhook-ов (то есть наличия программной возможности задавать URL обратного вызова) либо вовсе не нужна, либо решается с помощью протоколов Service Discovery¹², поскольку сервисы в составе одного бэкенда как правило равноправны — если сервис А может вызывать сервис Б, то и сервис Б может вызывать сервис А.

Однако все проблемы Webhook-ов, описанные нами выше, для таких обратных вызовов всё ещё актуальны. Вызов внутреннего сервиса всё ещё может окончиться `false negative`-ошибкой, внутренние клиенты могут не ожидать нарушения порядка пересылки сообщений и так далее.

Для решения этих проблем, а также для большей горизонтальной масштабируемости технологий обратного вызова, были созданы сервисы очередей сообщений¹³ и, в частности, различные серверные реализации паттерна pub/sub¹⁴. В настоящий момент pub/sub-архитектуры пользуются большой популярностью среди разработчиков, вплоть до перевода любого межсервисного взаимодействия на очереди событий.

NB: отметим, что ничего бесплатного в мире не бывает, и за эти гарантии доставки и горизонтальную масштабируемость необходимо платить:

- межсерверное взаимодействие становится событийно-консистентным со всеми вытекающими отсюда проблемами;
- хорошая горизонтальная масштабируемость и дешевизна использования очередей достигается при использовании политик at least once/at most once и отсутствии гарантии строгого порядка событий;
- в очереди могут скапливаться необработанные сообщения, внося нарастающие задержки, и решение этой проблемы на стороне подписчика может оказаться отнюдь не тривиальным.

Отметим также, что в публичных API зачастую используются обе технологии в связке — бэкенд API отправляет задание на вызов webhook-а в виде публикации события, которое специально предназначенный для этого внутренний сервис будет пытаться обработать путём вызова webhook-а.

Теоретически можно представить себе и такую интеграцию, в которой разработчик API даёт партнёрам непосредственно прямой доступ к внутренней очереди сообщений, однако примеры таких API нам неизвестны.

Примечания

¹ Polling (Computer Science)

[https://en.wikipedia.org/wiki/Polling_\(computer_science\)](https://en.wikipedia.org/wiki/Polling_(computer_science))

² Long Polling

https://en.wikipedia.org/wiki/Push_technology#Long_polling

³ WebSockets

<https://websockets.spec.whatwg.org/>

⁴ Hypertext Transfer Protocol Version 2 (HTTP/2). Server Push

<https://datatracker.ietf.org/doc/html/rfc7540#section-8.2>

⁵ WebRTC

<https://www.w3.org/TR/webrtc/>

⁶ gRPC. Server streaming RPC

<https://grpc.io/docs/what-is-grpc/core-concepts/#server-streaming-rpc>

⁷ MQTT

<https://docs.oasis-open.org/mqtt/mqtt/v5.0/mqtt-v5.0.html>

⁸ HTML Living Standard. Server-Sent Events

<https://html.spec.whatwg.org/multipage/server-sent-events.html>

⁹ Mutual Authentication. mTLS

https://en.wikipedia.org/wiki/Mutual_authentication#mTLS

¹⁰ SSRF

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

¹¹ Apache Kafka. Kafka Design. Message Delivery Guarantees

<https://docs.confluent.io/kafka/design/delivery-semantics.html>

¹² Web Services Discovery

https://en.wikipedia.org/wiki/Web_Services_Discovery

¹³ Message Queue

https://en.wikipedia.org/wiki/Message_queue

¹⁴ Publish / Subscribe Pattern

https://en.wikipedia.org/wiki/Publish%2E%80%93subscribe_pattern

Глава 22. Мультиплексирование сообщений. Асинхронная обработка событий

Одно из неприятных ограничений почти всех перечисленных в предыдущей главе технологий — это относительно невысокий размер сообщения. Наиболее проблематичная ситуация с push-уведомлениями: Google Firebase Messaging на момент написания настоящей главы разрешал сообщения не более 4000 байт. Но и в серверной разработке ограничения заметны: например, Amazon SQS лимитирует размер сообщения 256 килобайтами. При разработке webhook-ов вы рискуете быстро упереться в размеры тел сообщений, выставленных на веб-серверах партнёров (например, в nginx по умолчанию разрешены тела запросов не более одного мегабайта). Это приводит нас к необходимости сделать два технических выбора:

- содержит ли тело сообщения все данные необходимые для его обработки, или только уведомляет о факте изменения состояния;
- если второе, то содержит ли один вызов извещение об одном изменении, или может уведомлять сразу о нескольких таких событиях.

Рассмотрим на примере нашего кофейного API:

```
// Вариант 1: тело сообщения
// содержит все данные о заказе
POST /partner/webhook
Host: partners.host
{
  "event_id",
  "occurred_at",
  "order": {
    "id",
    "status",
    "recipe_id",
    "volume",
    // Все прочие детали заказа
    ...
  }
}
```

```

// Вариант 2: тело сообщения
// содержит только информацию
// о самом событии
POST /partner/webhook
Host: partners.host
{
    "event_id",
    // Тип сообщения: нотификация
    // о появлении нового заказа
    "event_type": "new_order",
    "occurred_at",
    // Все поля данных, необходимые
    // для обращения за полным
    // состоянием. В нашем случае –
    // идентификатор заказа
    "order_id"
}
// При обработке сообщения,
// возможно, отложенной,
// партнёр должен обратиться
// к нашему API
GET /v1/orders/{id}
→
{ /* все детали заказа */ }

```

```

// Вариант 3: мы уведомляем
// партнёра, что его реакции
// ожидают три новых заказа
POST /partner/webhook
Host: partners.host
{
    // Здесь может быть версия
    // состояния системы или курсор
    "occurred_at",
    "pending_order_count":
        <число новых заказов>
}
// В ответ партнёр должен вызвать
// эндпойнт получения списка заказов
GET /v1/orders/pending
→
{
    "orders",
    "cursor"
}

```

Выбор подходящей модели зависит от предметной области (в частности, допустимых размерах тел сообщений) и того, каким образом партнёр будет обрабатывать сообщение. В нашем конкретном случае, когда партнёр должен каждый новый заказ обработать отдельно, при этом на один заказ не может приходить больше одного-двух уведомлений, естественным выбором является вариант 1 (если тело запроса не содержит никаких непредсказуемо больших данных) или 2. Третий подход будет естественным выбором, если:

- API генерирует большое число сообщений об изменениях состояния на одну логическую сущность;
- партнёров интересуют только наиболее свежие изменения;
- или обработка событий требует последовательного исполнения и не подразумевает параллельности.

NB: третий (и отчасти второй) варианты естественным образом приводят нас к схеме, характерной для клиентских устройств: push-уведомление само по себе не почти содержит полезной информации и только является сигналом для внеочередного поллинга.

Применение техник с отправкой только ограниченного набора данных помимо усложнения схемы взаимодействия и увеличения количества запросов имеет ещё один важный недостаток. Если в варианте 1 (сообщение содержит в себе все релевантные данные) мы можем рассчитывать на то, что возврат кода успеха подписчиком эквивалентен успешной обработке сообщения партнёром (что, вообще говоря, тоже не гарантировано, т.к. партнёр может использовать асинхронные схемы), то для вариантов 2 и 3 это заведомо не так: для обработки сообщений партнёр должен выполнить дополнительные действия, начиная с получения нужных данных о заказе. В этом случае нам необходимо иметь раздельные статусы — сообщение доставлено и сообщение обработано; в идеале, второе должно вытекать из логики работы API, т.е. сигналом о том, что сообщение обработано, является какое-то действие, совершающееся партнёром. В нашем кофейном примере это может быть перевод заказа партнёром из статуса "new" (заказ создан пользователем) в статус "accepted" или "rejected" (кофейня партнёра приняла или отклонила заказ). Тогда полный цикл обработки уведомления будет выглядеть так:

```
// Уведомляем партнёра о том,
// что его реакции
// ожидают три новых заказа
POST /partner/webhook
Host: partners.host
{
  "occurred_at",
  "pending_order_count":
    <число новых заказов>
}
```

```
// В ответ партнёр вызывает
// эндпойнт получения списка заказов
GET /v1/orders/pending
→
{
  "orders",
  "cursor"
}
```

```
// После того, как заказы обработаны,
// партнёр уведомляет нас об
// изменениях статуса
POST /v1/orders/bulk-status-change
{
  "status_changes": [
    {
      "order_id",
      "new_status": "accepted",
      // Иная релевантная информация,
      // например, время готовности
      ...
    },
    {
      "order_id",
      "new_status": "rejected",
      "reason"
    },
    ...
  }
}
```

Если такого нативного способа оповестить об успешной обработке события схема работы нашего API не предполагает, мы можем ввести эндпойнт который явно помечает сообщения прочитанными. Этот шаг, вообще говоря, необязательный (мы можем просто договориться о том, что это ответственность партнёра обрабатывать события и мы не ждём от него никаких подтверждений), но это лишает нас полезного инструмента мониторинга — что происходит на стороне партнёра, успевает ли он обрабатывать события — что в свою очередь затрудняет разработку упомянутых в предыдущей главе механизмов деградации и аварийного отключения интеграции.

Глава 23. Атомарность массовых изменений

Вернёмся теперь от webhook-ов обратно к разработке API прямого вызова. Дизайн эндпойнта `orders/bulk-status-change`, описанный в предыдущей главе, ставит перед нами интересный вопрос: что делать, если наш бэкенд часть изменений смог обработать, а часть — нет?

Пусть партнёр уведомляет нас об изменении статусов двух заказов:

```
POST /v1/orders/bulk-status-change
{
  "status_changes": [
    {
      "order_id": "1",
      "new_status": "accepted",
      // Иная релевантная информация,
      // например, время готовности
    },
    {
      "order_id": "2",
      "new_status": "rejected",
      "reason"
    }
  ]
}
→ 500 Internal Server Error
```

Возникает вопрос, каким образом должен быть организован данный «зонтичный» эндпойнт, который фактически представляет собой прокси для выполнения списка вложенных подзапросов, если при изменении статуса одного из двух заказов возникла ошибка. Мы можем предложить по крайней мере четыре варианта:

- А. Гарантировать идемпотентность и атомарность: если хотя бы один из подзапросов не был выполнен, все остальные изменения также не применяются.
- В. Гарантировать идемпотентность, но не атомарность: если какие-то подзапросы не были успешны, то повтор запроса (с тем же ключом идемпотентности) не выполняет никаких действий и оставляет систему точно в том же состоянии (т.е. неуспешные запросы никогда не выполняются, даже если этому ничего не препятствует, пока не будет отправлен запрос с новым ключом идемпотентности).
- С. Не гарантировать ни идемпотентность, ни атомарность: применять подзапросы полностью независимо.

- D. Не гарантировать атомарность и вообще запретить перезапросы через требование указания актуальной ревизии ресурса (см. главу «[Стратегии синхронизации](#)»).

Из общих соображений кажется, что первый вариант для публичных API подходит лучше сего: если вы можете гарантировать атомарность (что не всегда легко с т.з. производительности), сделайте это. В первой редакции настоящей книги мы рекомендовали безусловно придерживаться этого правила.

Однако, если мы взглянем на ситуацию со стороны партнёра, она окажется далеко не такой простой. Предположим, что на стороне партнёра реализована следующая функциональность:

1. Через webhook на бэкенд поступают уведомления о новых заказах.
2. Партнёр опрашивает свои кофейни, готовы ли они принять заказ.
3. Периодически, скажем, раз в 10 секунд, партнёр собирает все изменения статуса (т.е. все новые ответы кофеен) и вызывает наш эндпойнт `bulk-status-change`, передавая список изменений.

Предположим, что на шаге (3) партнёр получил от сервера API ошибку. Что разработчик должен в этой ситуации сделать? Вероятнее всего, в коде партнёра будет реализован один из трёх вариантов:

1. Безусловный повтор запроса:

```

// Получаем все текущие заказы
let pendingOrders = await api
  .getPendingOrders();
// Партнёр проверяет статус
// каждого из них в своей
// системе и готовит
// необходимые изменения
let changes =
  await prepareStatusChanges(
    pendingOrders
  );

let result;
let tryNo = 0;
let timeout = DEFAULT_RETRY_TIMEOUT;
while (result && tryNo++ < MAX_RETRIES) {
  try {
    // Отправляем массив изменений статусов
    result = await api.bulkStatusChange(
      changes,
      // Указываем известную ревизию
      pendingOrders.revision
    );
  } catch (e) {
    // если получена ошибка, повторяем
    // операцию отправки
    logger.error(e);
    await wait(timeout);
    timeout = min(timeout*2, MAX_TIMEOUT);
  }
}

```

NB: в примере выше мы приводим «правильную» политику перезапросов (с экспоненциально растущим периодом ожидания и лимитом на количество попыток), как мы ранее рекомендовали в главе «[Описание конечных интерфейсов](#)». Следует, однако, иметь в виду, что в реальном коде партнёров с большой долей вероятности ничего подобного реализовано не будет. В дальнейших примерах эту громоздкую конструкцию мы также будем опускать, чтобы упростить чтение кода.

2. Повтор только неудавшихся подзапросов:

```

let pendingOrders = await api
    .getPendingOrders();
let changes =
    await prepareStatusChanges(
        pendingOrders
    );

let result;
while (changes.length) {
    let failedChanges = [];
    try {
        result = await api.bulkStatusChange(
            changes,
            pendingOrders.revision
        );
    } catch (e) {
        let i = 0;
        // Предполагаем, что поле e.changes
        // содержит разбивку подзапросов
        // по статусу исполнения
        for (; i < e.changes.length; i++) {
            if (e.changes[i].status == 'failed') {
                failedChanges.push(e.changes[i]);
            }
        }
        // Формируем новый запрос, состоящий
        // только из неуспешных подзапросов
        changes = failedChanges;
    }
}

```

3. Рестарт всей операции, т.е. в нашем случае — перезапрос всех новых заказов и формирование нового запроса на изменение:

```

do {
    let pendingOrders = await api
        .getPendingOrders();
    let changes =
        await prepareStatusChanges(
            pendingOrders
        );
    // Отсылаем изменения,
    // если они есть
    if (changes.length) {
        await api.bulkStatusChange(
            changes,
            pendingOrders.revision
        );
    }
} while (pendingOrders.length);

```

Если мы проанализируем комбинации возможных реализаций клиента и сервера, то увидим, что подходы (B) и (D) не работают с решением (1), поскольку клиент будет пытаться повторять заведомо неисполнимый запрос, пока не исчерпает лимит попыток.

Теперь добавим к постановке задачи ещё одно важное условие: предположим, что иногда ошибка подзапроса не может быть устранена его повторением — например, партнёр пытается подтвердить заказ, который был отменён пользователем. Если в составе массового вызова есть такой подзапрос, то атомарный сервер, реализованный по схеме (A), моментально партнёра «накажет»: сколько бы он запрос ни повторял, *валидные подзапросы не будут выполнены, если есть хоть один невалидный*. В то время как неатомарный сервер, по крайней мере, продолжит подтверждать валидные запросы.

Это приводит нас к парадоксальному умозаключению: гарантировать, что партнёрский код будет *как-то* работать и давать партнёру время разобраться с ошибочными запросами, можно только реализовав максимально нестрогий неидемпотентный неатомарный подход к операции массовых изменений. Однако и этот вывод мы считаем ошибочным, и вот почему: описанный нами «зоопарк» возможных имплементаций клиента и сервера очень хорошо демонстрирует *нежелательность* эндпойнтов массовых изменений как таковых. Такие эндпойнты требуют реализации дополнительного уровня логики и на клиенте, и на сервере, причём логики весьма неочевидной. Функциональность неатомарных массовых изменений очень быстро приведёт нас к крайне неприятным ситуациям:

```

// Партнёр делает рефанд
// и отменяет заказ
POST /v1/bulk-status-change
{
  "changes": [ {
    "operation": "refund",
    "order_id"
  }, {
    "operation": "cancel",
    "order_id"
  } ]
}
→
// Пока длилась операция,
// пользователь успел дойти
// до кофейни и забрать заказ
{
  "changes": [ {
    // Рефанд проведён успешно...
    "status": "success"
  }, {
    // ...а отмена заказа нет
    "status": "fail",
    "reason": "already_served"
  } ]
}

```

Если операции в списке как-то зависят одна от другой (как в примере выше — партнёру нужно *и* сделать рефанд, *и* отменить заказ, выполнение только одной из этих операций бессмысленно) либо важен порядок исполнения операций, неатомарные эндпойнты будут постоянно приводить к проблемам. И даже если вам кажется, что в вашей предметной области таких проблем нет, в какой-то момент может оказаться, что вы чего-то не учли.

Поэтому наши рекомендации по организации эндпойнтов массовых изменений таковы:

1. Если вы можете обойтись без таких эндпойнтов — обойдитесь. В server-to-server интеграциях экономия копеечная, в современных сетях с поддержкой протокола QUIC¹ и мультиплексирования запросов тоже весьма сомнительная.
2. Если такой эндпойнт всё же нужен, лучше реализовать его атомарно и предоставить SDK, которые помогут партнёрам не допускать типичные ошибки.
3. Если реализовать атомарный эндпойнт невозможно, тщательно продумайте дизайн API, чтобы не допустить ошибок, подобных описанным выше.

4. Вне зависимости от выбранного подхода, ответы сервера должны включать разбивку по подзапросам. В случае атомарных эндпойнтов это означает включение в ответ списка ошибок, из-за которых исполнение запроса не удалось, в идеале — со всеми потенциальными ошибками (т.е. с результатами проверок каждого подзапроса на валидность). Для неатомарных эндпойнтов необходимо возвращать список со статусами каждого подзапроса и всеми возникшими ошибками.

Один из подходов, позволяющих минимизировать возможные проблемы — разработать смешанный эндпойнт, в котором потенциально зависящие друг от друга операции группированы, например, вот так:

```
POST /v1/bulk-status-change
{
  "changes": [
    {
      "order_id": <первый заказ>
      // Операции по одному
      // заказу группируются
      // в одну структуру
      // и выполняются атомарно
      "operations": [
        "refund",
        "cancel"
      ],
      {
        // Группы операции по разным
        // заказам могут выполняться
        // параллельно и неатомарно
        "order_id": <второй заказ>
      }
    }
  ]
}
```

На всякий случай уточним, что вложенные операции должны быть сами по себе идемпотентны. Если же это не так, то следует каким-то детерминированным образом сгенерировать внутренние ключи идемпотентности на каждую вложенную операцию в отдельности (в простейшем случае — считать токен идемпотентности внутренних запросов равным токену идемпотентности внешнего запроса, если это допустимо в рамках предметной области; иначе придётся использовать составные токены — в нашем случае, например, в виде `<order_id>:<external_token>`).

Примечания

¹ QUIC

<https://datatracker.ietf.org/doc/html/rfc9000>

Глава 24. Частичные обновления

Описанный в предыдущей главе пример со списком операций, который может быть выполнен частично, естественным образом подводит нас к следующей проблеме дизайна API. Что, если изменение не является атомарной идемпотентной операцией (как изменение статуса заказа), а представляет собой низкоуровневую перезапись нескольких полей объекта? Рассмотрим следующий пример.

```
// Создаёт заказ из двух напитков
POST /v1/orders/
X-Idempotency-Token: <токен>
{
  "delivery_address",
  "items": [
    {
      "recipe": "lungo"
    },
    {
      "recipe": "latte",
      "milk_type": "oat"
    }
  ]
}
→
{ "order_id" }
```

```
// Частично перезаписывает заказ,
// обновляет объём второго напитка
PATCH /v1/orders/{id}
{
  "items": [
    // `null` показывает, что
    // параметры первого напитка
    // менять не надо
    null,
    // список изменений свойств
    // второго напитка
    {"volume": "800ml"}
  ]
}
→
{ /* изменения приняты */ }
```

Эта сигнатура плоха сама по себе, поскольку её читабельность сомнительна. Что обозначает пустой первый элемент массива — это удаление элемента или указание на отсутствие изменений? Что произойдёт с полями, которые не указаны в операции обновления (`delivery_address`, `milk_type`) — они будут сброшены в значения по умолчанию или останутся неизменными?

Самое неприятное здесь — какой бы вариант вы ни выбрали, это только начало проблем. Допустим, мы договорились, что конструкция `{"items": [null, {...}]}` означает, что с первым элементом массива ничего не происходит, он не меняется. А как тогда всё-таки его удалить? Придумать одно «зануляемое» значение специально для удаления? Аналогично, если значения неуказанных полей остаются без изменений — как сбросить их в значения по умолчанию?

Частичные изменения состояния ресурсов — одна из самых частых задач, которые решает разработчик API, и, увы, одна из самых сложных. Попытки обойтись малой кровью и упростить имплементацию зачастую приводят к очень большим проблемам в будущем.

Простое решение состоит в том, чтобы всегда перезаписывать объект целиком, т.е. требовать передачи полного объекта, полностью заменять им текущее состояние и возвращать в ответ на операцию новое состояние целиком. Однако это простое решение часто не принимается по нескольким причинам:

- повышенные размеры запросов и, как следствие, расход трафика;
- необходимость вычислять, какие конкретно поля изменились — в частности для того, чтобы правильно сгенерировать сигналы (события) для подписчиков на изменения;
- невозможность совместного доступа к объекту, когда два клиента независимо редактируют его свойства, поскольку клиенты всегда посыпают полное состояние объекта, известное им, и переписывают изменения друг друга, поскольку о них не знают.

Во избежание перечисленных проблем разработчики, как правило, реализуют некоторое **наивное решение**:

- клиент передаёт только те поля, которые изменились;
- для сброса значения поля в значение по умолчанию или пропуска/удаления элементов массивов используются специально оговоренные значения.

Если обратиться к примеру выше, наивный подход выглядит примерно так:

```

// Частично перезаписывает заказ:
//   * сбрасывает адрес доставки
//   в значение по умолчанию
//   * не изменяет первый напиток
//   * удаляет второй напиток
PATCH /v1/orders/{id}
{
    // Специальное значение №1:
    // обнулить поле
    "delivery_address": null
    "items": [
        // Специальное значение №2:
        // не выполнять никаких
        // операций с объектом
        {},
        // Специальное значение №3:
        // удалить объект
        false
    ]
}

```

Предполагается, что:

- повышенного расхода трафика можно избежать, передавая только изменившиеся поля и заменяя пропускаемые элементы специальными значениями ({} в нашем случае);
- события изменения значения поля также будут генерироваться только по тем полям и объектам, которые переданы в запросе;
- если два клиента делают одновременный запрос, но изменяют различные поля, конфликта доступа не происходит, и оба изменения применяются.

Все эти соображения, однако, на поверку оказываются мнимыми:

- причины увеличенного расхода трафика (слишком частый поллинг, отсутствие пагинации и/или ограничений на размеры полей) мы разбирали в главе «[Описание конечных интерфейсов](#)», и передача лишних полей к ним не относится (а если и относится, то это повод декомпозировать эндпойнт);
- концепция передачи только изменившихся полей по факту перекладывает ответственность определения, какие поля изменились, на клиент:
 - это не только не снижает сложность имплементации этого кода, но и чревато его фрагментацией на несколько независимых клиентских реализаций;

- существование клиентского алгоритма построения diff-ов не отменяет обязанность сервера уметь делать то же самое — поскольку клиентские разработчики могли ошибиться или просто поленились правильно вычислить изменившиеся поля;
- наконец, подобная наивная концепция организации совместного доступа работает ровно до того момента, пока изменения транзитивны, т.е. результат не зависит от порядка выполнения операций (в нашем примере это уже не так — операции удаления первого элемента и редактирования первого элемента нетранзитивны);
 - кроме того, часто в рамках той же концепции экономят и на исходящем трафике, возвращая пустой ответ сервера для модифицирующих операций; таким образом, два клиента, редактирующих одну и ту же сущность, не видят изменения друг друга, что ещё больше повышает вероятность получить совершенно неожиданные результаты.

Это решение можно улучшить путём ввода явных управляющих конструкций вместо «магических значений» и введением мета-опций операции (скажем, фильтра по именам полей, как это принято в gRPC поверх Protobuf¹), например, так:

```
// Частично перезаписывает заказ:
//   * сбрасывает адрес доставки
//   в значение по умолчанию
//   * не изменяет первый напиток
//   * удаляет второй напиток
PATCH /v1/orders/{id}«
// мета-фильтр: какие поля
// переопределются
?field_mask=delivery_address,items
{
  // Специальное значение №1:
  // обнулить поле
  "delivery_address": {
    // Префикс `__` нужен, чтобы
    // избежать коллизий
    // с реальными именами полей
    "__operation": "reset"
  },
  "items": [
    // Специальное значение №2:
    // не выполнять никаких
    // операций с объектом
    { "__operation": "skip" },
    // Специальное значение №3:
    // удалить объект
    { "__operation": "delete" }
  ]
}
```

Такой подход выглядит более надёжным, но в реальности мало что меняет в постановке проблемы:

- «магические значения» заменены «магическими» префиксами;
- фрагментация алгоритмов и нетранзитивность операций сохраняется.

При этом формат перестаёт быть простым и интуитивно понятным, что с нашей точки зрения делает такое улучшение спорным.

Более консистентное решение: разделить эндпойнт на несколько идемпотентных суб-эндпойнтов, имеющих независимые идентификаторы и/или адреса (чего обычно достаточно для обеспечения транзитивности независимых операций). Этот подход также хорошо согласуется с принципом декомпозиции, который мы рассматривали в предыдущем главе [«Разграничение областей ответственности»](#).

```
// Создаёт заказ из двух напитков
POST /v1/orders/
{
  "parameters": {
    "delivery_address"
  },
  "items": [
    {
      "recipe": "lungo"
    },
    {
      "recipe": "latte",
      "milk_type": "oats"
    }
  ]
}
→
{
  "order_id",
  "created_at",
  "parameters": {
    "delivery_address"
  },
  "items": [
    { "item_id", "status" },
    { "item_id", "status" }
  ]
}
```

```
// Изменяет параметры,
// относящиеся ко всему заказу
PUT /v1/orders/{id}/parameters
{ "delivery_address" }
→
{ "delivery_address" }
```

```
// Частично перезаписывает заказ
// обновляет объём одного напитка
PUT /v1/orders/{id}/items/{item_id}
{
    // Все поля передаются, даже если
    // изменилось только какое-то одно
    "recipe", "volume", "milk_type"
}
→
{ "recipe", "volume", "milk_type" }
```

```
// Удаляет один из напитков в заказе
DELETE /v1/orders/{id}/items/{item_id}
```

Теперь для удаления `volume` достаточно *не* передавать его в `PUT items/{item_id}`. Кроме того, обратите внимание, что операции удаления одного напитка и модификации другого теперь стали транзитивными.

Этот подход также позволяет отделить неизменяемые и вычисляемые поля (`created_at` и `status`) от изменяемых, не создавая двусмысленных ситуаций (что произойдёт, если клиент попытается изменить `created_at`?).

Применения этого паттерна, как правило, достаточно для большинства API, манипулирующих сложносоставленными сущностями, однако и недостатки у него тоже есть: высокие требования к качеству проектирования декомпозиции (иначе велик шанс, что стройное API развалится при дальнейшем расширении функциональности) и необходимость совершать множество запросов для изменения всей сущности целиком (из чего вытекает необходимость создания функциональности для внесения массовых изменений, нежелательность которой мы обсуждали в предыдущей главе).

NB: при декомпозиции эндпойнтов велик соблазн провести границу так, чтобы разделить изменяемые и неизменяемые данные. Тогда последние можно объявить кэшируемыми условно вечно и вообще не думать над проблемами пагинации и формата обновления. На бумаге план выглядит отлично, однако с ростом API неизменяемые данные частенько перестают быть таковыми, и тогда потребуется выпускать новые интерфейсы работы с данными. Мы скорее рекомендуем объявлять данные иммутабельными в одном из двух случаев:

либо (1) они действительно не могут стать изменяемыми без слома обратной совместимости, либо (2) ссылка на ресурс (например, на изображение) поступает через API же, и вы обладаете возможностью сделать эти ссылки персистентными (т.е. при необходимости обновить изображение будете генерировать новую ссылку, а не перезаписывать контент по старой ссылке).

Разрешение конфликтов совместного редактирования

Идея организации изменения состояния ресурса через независимые атомарные идемпотентные операции выглядит достаточно привлекательно и с точки зрения разрешения конфликтов доступа. Так как составляющие ресурса перезаписываются целиком, результатом записи будет именно то, что пользователь видел своими глазами на экране своего устройства, даже если при этом он смотрел на неактуальную версию. Однако этот подход очень мало помогает нам, если мы действительно обеспечить максимально гранулярное изменение данных, как, например, это сделано в онлайн-сервисах совместной работы с документами или системах контроля версий (поскольку для этого нам придётся сделать столь же гранулярные эндпоинты, т.е. буквально адресовать каждый символ документа по отдельности).

Для «настоящего» совместного редактирования необходимо будет разработать отдельный формат описания изменений, который позволит:

- иметь максимально возможную гранулярность (т.е. одна операция соответствует одному действию клиента);
- реализовать политику разрешения конфликтов.

В нашем случае мы можем пойти, например, вот таким путём:

```
POST /v1/order/changes
X-Idempotency-Token: <токен>
{
    // Какую ревизию ресурса
    // видел пользователь, когда
    // выполнял изменения
    "known_revision",
    "changes": [
        {
            "type": "set",
            "field": "delivery_address",
            "value": <новое значение>
        },
        {
            "type": "unset_item_field",
            "item_id": "1234567890",
            "field": "volume"
        }
    ],
    ...
}
```

Этот подход существенно сложнее в имплементации, но является единственным возможным вариантом реализации совместного редактирования, поскольку он явно отражает, что в действительности делал пользователь с представлением объекта. Имея данные в таком формате возможно организовать и оффлайн-редактирование, когда пользовательские изменения накапливаются и сервер впоследствии автоматически разрешает возможные конфликты, основываясь на истории ревизий.

NB: один из подходов к этой задаче — разработка такой номенклатуры операций над данными (например, conflict-free replicated data type (CRDT)²), в которой любые действия транзитивны (т.е. конечное состояние системы не зависит от того, в каком порядке они были применены). Мы, однако, склонны считать такой подход применимым только к весьма ограниченным предметным областям — поскольку в реальной жизни нетранзитивные действия находятся почти всегда. Если один пользователь ввёл в документ новый текст, а другой пользователь удалил документ — никакого разумного (т.е. удовлетворительного с точки зрения обоих акторов) способа автоматического разрешения конфликта здесь нет, необходимо явно спросить пользователей, что бы они хотели сделать с возникшим конфликтом.

Примечания

¹ Protocol Buffers. Field Masks in Update Operations
<https://protobuf.dev/reference/protobuf/google.protobuf/#field-masks-updates>

² Conflict-Free Replicated Data Type
https://en.wikipedia.org/wiki/Conflict-free_replicated_data_type

Глава 25. Деградация и предсказуемость

В предыдущих главах мы много говорили о том, что фон ошибок — не только неизбежное зло в любой достаточно большой системе, но и, зачастую, осознанное решение, которое позволяет сделать систему более масштабируемой и предсказуемой.

Зададим себе, однако, вопрос: а что значит «более предсказуемая» система? Для нас как для вендора API это достаточно просто: процент ошибок (в разбивке по типам) достаточно стабилен, и им можно пользоваться как индикатором возникающих технических проблем (если он растёт) и как KPI для технических улучшений и рефакторингов (если он падает).

Но вот для разработчиков-партнёров понятие «предсказуемость поведения API» имеет совершенно другой смысл: насколько хорошо и полно они в своём коде могут покрыть различные сценарии использования API и потенциальные проблемы — или, иными словами, насколько явно из документации и номенклатуры методов и ошибок API становится ясно, какие типовые ошибки могут возникнуть и как с ними работать.

Чем, например, оптимистичное управление параллелизмом (см. главу «Стратегии синхронизации») лучше блокировок с точки зрения партнёра? Тем, что, получив ошибку несовпадения ревизий, разработчик понимает, какой код он должен написать: обновить состояние и попробовать ещё раз (в простейшем случае — показав новое состояние пользователю и предложив ему решить, что делать дальше). Если же разработчик пытается захватить lock и не может сделать этого в течение разумного времени, то... а что он может полезного сделать? Попробовать ещё раз — но результат ведь, скорее всего, не изменится. Показать пользователю... что? Бесконечный спиннер? Попросить пользователя принять какое решение — сдаться или ещё подождать?

При проектировании поведения вашего API исключительно важно представить себя на месте разработчика и попытаться понять, какой код он должен написать для разрешения возникающих ситуаций (включая сетевые таймауты и/или частичную недоступность вашего бэкенда). В этой книге приведено множество частных советов, как поступать в той или иной ситуации, но они, конечно, покрывают только типовые сценарии. О нетиповых вам придётся подумать самостоятельно.

Несколько общих советов, которые могут вам пригодиться:

- если вы можете включить в саму ошибку рекомендации, как с ней бороться — сделайте это не раздумывая (имейте в виду, что таких рекомендаций должно быть две — для пользователя, который увидит ошибку в приложении, и для разработчика, который будет разбирать логи);
- если ошибки в каком-то эндпойнте некритичны для основной функциональности интеграции, очень желательно описать этот факт в документации (потому что разработчик может просто не догадаться обернуть соответствующий вызов в `try-catch`), а лучше — привести примеры, каким значением/поведением по умолчанию следует воспользоваться в случае получения ошибки;
- не забывайте, что, какую бы стройную и всеобъемлющую систему ошибок вы ни выстроили, почти в любой среде разработчик может получить ошибку транспортного уровня или таймаут выполнения, а, значит, оказаться в ситуации, когда восстанавливать состояние надо, а «подсказки» от бэкенда недоступны; должна существовать какая-то достаточно очевидная последовательность действий «по умолчанию» для восстановления работы интеграции из любой точки;
- наконец, при введении новых ошибок не забывайте о старых клиентах, которые про эти новые типы ошибок не знают; «реакция по умолчанию» на неизвестные ошибки должна в том числе покрывать и эти новые сценарии.

В идеальном мире для «правильной деградации» клиентов желательно иметь мета-API, позволяющее определить статус доступности для эндпойнтов основного API — тогда партнёры смогут, например, автоматически включать fallback-и, если какая-то функциональность не работает. (В реальном мире, увы, если на уровне сервиса наблюдаются масштабные проблемы, то обычно и API статуса доступности оказывается ими затронуто.)

РАЗДЕЛ III. ОБРАТНАЯ СОВМЕСТИМОСТЬ

Глава 26. Постановка проблемы обратной совместимости

Как обычно, дадим смысловое определение «обратной совместимости», прежде чем начинать изложение.

Обратная совместимость — это свойство всей системы API быть стабильной во времени. Это значит следующее: **код, написанный разработчиками с использованием вашего API, продолжает работать функционально корректно в течение длительного времени**. К этому определению есть два больших вопроса, и два уточнения к ним.

1. Что значит «функционально корректно»?

Это значит, что код продолжает выполнять свою функцию — решать какую-то задачу пользователя. Это не означает, что он продолжает работать одинаково: например, если вы предоставляете UI-библиотеку, то изменение функционально несущественных деталей дизайна, типа глубины теней или формы штриха границы, обратную совместимость не нарушит. А вот, например, изменение размеров визуальных компонентов, скорее всего, приведёт к тому, что какие-то пользовательские макеты развалятся.

2. Что значит «длительное время»?

С нашей точки зрения длительность поддержания обратной совместимости следует увязывать с длительностью жизненных циклов приложений в соответствующей предметной области. Хороший ориентир в большинстве случаев — это LTS-периоды платформ. Так как приложение всё равно будет переписано в связи с окончанием поддержки платформы, нормально предложить также и переход на новую версию API. В основных предметных областях (десктопные и мобильные операционные системы) этот срок исчисляется несколькими годами.

Почему обратную совместимость необходимо поддерживать (в том числе предпринимать необходимые меры ещё на этапе проектирования API) — понятно из определения. Прекращение работы приложения (полное или частичное) по вине поставщика API — крайне неприятное событие, а то и катастрофа, для любого разработчика, особенно если он платит за этот API

деньги.

Но развернём теперь проблему в другую сторону: а почему вообще возникает проблема с поддержанием обратной совместимости? Почему мы можем хотеть её нарушить? Ответ на этот вопрос, при кажущейся простоте, намного сложнее, чем на предыдущий.

Мы могли бы сказать, что *обратную совместимость приходится нарушать для расширения функциональности API*. Но это лукавство: новая функциональность на то и новая, что она не может затронуть код приложений, который её не использует. Да, конечно, есть ряд сопутствующих проблем, приводящих к стремлению переписать *наш* код, код самого API, с выпуском новой мажорной версии:

- код банально морально устарел, внесение в него изменений, пусть даже в виде расширения функциональности, нецелесообразно;
- новая функциональность не была предусмотрена в старом интерфейсе: мы хотели бы наделить уже существующие сущности новыми свойствами, но не можем;
- наконец, за прошедшее после изначального релиза время мы узнали о предметной области и практике применения нашего API гораздо больше, и сделали бы многие вещи иначе.

Эти аргументы можно обобщить как «разработчики API не хотят работать со старым кодом», не сильно покривив душой. Но и это объяснение неполно: даже если вы не собираетесь переписывать код API при добавлении новой функциональности, или вы вовсе её и не собирались добавлять, выпускать новые версии API — мажорные и минорные — всё равно придётся.

NB: в рамках этой главы мы не разделяем минорные версии и патчи: под словами «минорная версия» имеется в виду любой обратно совместимый релиз API.

Напомним, что API — это мост, средство соединения разных программируемых контекстов. И как бы нам ни хотелось зафиксировать конструкцию моста, наши возможности ограничены: мост-то мы можем зафиксировать — да вот края ущелья, как и само ущелье, не можем. В этом корень проблемы: мы не можем оставить *свой* код без изменений, поэтому нам придётся рано или поздно потребовать, чтобы клиенты изменили *свой*.

Помимо наших собственных поползновений в сторону изменения архитектуры API, три других тектонических процесса происходят одновременно: размывание клиентов, предметной области и нижележащей платформы.

Фрагментация клиентских приложений

В тот момент, когда вы выпустили первую версию API, и первые клиенты начали использовать её — ситуация идеальна. Есть только одна версия, и все клиенты работают с ней. А вот дальше возможны два варианта развития событий:

1. Если платформа поддерживает on-demand получение кода, как старый-добрый Веб, и вы не поленились это получение кода реализовать (в виде платформенного SDK, например, JS API), то развитие API более или менее находится под вашим контролем. Поддержание обратной совместимости сводится к поддержанию обратной совместимости *клиентской библиотеки*, а вот в части сервера и клиент-серверного взаимодействия вы свободны.

Это не означает, что вы не можете нарушить обратную совместимость — всё ещё можно напортачить с заголовками кэширования SDK или банально допустить баг в коде. Кроме того, даже on-demand системы всё равно не обновляются мгновенно — автор сталкивался с ситуацией, когда пользователи намеренно держали вкладку браузера открытой *неделями*, чтобы не обновляться на новые версии. Тем не менее, вам почти не придётся поддерживать более двух (последней и предпоследней) минорных версий клиентского SDK. Более того, вы можете попытаться в какой-то момент переписать предыдущую мажорную версию библиотеки, имплементировав её на основе API новой версии.

2. Если поддержка on-demand кода платформой не поддерживается или запрещена условиями, как это произошло с современными мобильными платформами, то ситуация становится гораздо сложнее. По сути, каждый клиент — это «слепок» кода, который работает с вашим API, зафиксированный в том состоянии, в котором он был на момент компиляции. Обновление клиентских приложений по времени растянуто гораздо дольше, нежели Web-приложений; самое неприятное здесь состоит в том, что некоторые клиенты *не обновятся вообще никогда* — по одной из трёх причин:

- разработчики просто не выпускают новую версию приложения, его развитие заморожено;
- пользователь не хочет обновляться (в том числе потому, что, по мнению пользователя, разработчики приложения его «испортили» в новых версиях);
- пользователь не может обновиться вообще, потому что его устройство больше не поддерживается.

В современных реалиях все три категории в сумме легко могут составлять десятки процентов аудитории. Это означает, что прекращение поддержки любой версии API является весьма заметным событием — особенно если приложения разработчика поддерживают более широкий спектр версий платформы, нежели ваш API.

Вы можете не выпускать вообще никаких SDK, предоставляя только серверный API в виде, например, HTTP эндпойнтов. Вам может показаться, что таким образом, пусть ваш API и стал менее конкурентоспособным на рынке из-за отсутствия SDK, вы облегчили себе задачу поддержания обратной совместимости. На самом деле это совершенно не так: раз вы не предоставляете свой SDK — или разработчики возьмут неофициальный SDK (если кто-то его сделает), или просто каждый из них напишет по фреймворку. Стратегия «ваш фреймворк — ваша ответственность», к счастью или к сожалению, работает плохо: если на вашем API пишут некачественные приложения — значит, ваш API сам некачественный. Уж точно по мнению разработчиков, а может и по мнению пользователей, если работа API внутри приложения пользователю видна.

Конечно, если ваш API достаточно stateless и не требует клиентских SDK (или же можно обойтись просто автогенерацией SDK из спецификации), эти проблемы будут гораздо менее заметны, но избежать их полностью можно только одним способом — никогда не выпуская новых версий API. Во всех остальных случаях вы будете иметь дело с какой-то гребёнкой распределения количества пользователей по версиям API и версиям SDK.

Эволюция предметной области

Другая сторона ущелья — та самая нижележащая функциональность, к которой вы предоставляете API. Она, разумеется, тоже не статична и развивается в какую-то сторону:

- появляется новая функциональность;
- старая функциональность перестаёт поддерживаться;
- меняются интерфейсы.

Как правило, API изначально покрывает только какую-то часть существующей предметной области. В случае нашего [примера с API кофемашин](#) разумно ожидать, что будут появляться новые модели с новым API, которые нам придётся включать в свою платформу, и гарантировать возможность сохранения того же интерфейса абстракции — весьма непросто. Даже если просто добавлять поддержку новых видов нижележащих устройств, не добавляя ничего во внешний интерфейс — это всё равно изменения в коде, которые могут в итоге привести к несовместимости, пусть и ненамеренно.

Стоит также отметить, что далеко не все поставщики API относятся к поддержанию обратной совместимости, да и вообще к качеству своего ПО, так же серьёзно, как и (надеемся) вы. Стоит быть готовым к тому, что заниматься поддержанием вашего API в рабочем состоянии, то есть написанием и поддержкой фасадов к меняющемуся ландшафту предметной области, придётся именно вам, и зачастую довольно внезапно.

Дрифт платформ

Наконец, есть и третья сторона вопроса — «ущелье», через которое вы перекинули свой мост в виде API. Код, который напишут разработчики, исполняется в некоторой среде, которую вы не можете контролировать, и она тоже эволюционирует. Появляются новые версии операционной системы, браузеров, протоколов, языка SDK. Разрабатываются новые стандарты и принимаются новые соглашения, некоторые из которых сами по себе обратно несовместимы, и поделать с этим ничего нельзя.

Как и в случае со старыми версиями приложений, старые версии платформ также приводят к фрагментации, поскольку разработчикам (в том числе и разработчикам API) объективно тяжело поддерживать старые платформы, а пользователям столь же объективно тяжело обновляться, так как обновление операционной системы зачастую невозможно без замены самого устройства на более новое.

Самое неприятное во всём этом то, что к изменениям в API подталкивает не только поступательный прогресс в виде новых платформ и протоколов, но и банальная мода и вкусовщина. Буквально несколько лет назад были в моде объёмные реалистичные иконки, от которых все отказались в пользу плоских и абстрактных — и большинству разработчиков визуальных компонентов пришлось, вслед за модой, переделывать свои библиотеки, выпуская новые наборы иконок или заменяя старые. Аналогично прямо сейчас повсеместно внедряется поддержка «ночных» тем интерфейсов, что требует изменений в большом количестве API.

Обратная совместимость на уровне спецификаций

В случае применения API-first подхода, понятие «обратная совместимость» обретает дополнительное измерение, поскольку теперь в системе появляется ещё спецификация и кодогенерация по ней. Становится возможным нарушить обратную совместимость, не нарушая спецификации (например, изменив строгую консистентность на слабую) — и, напротив, несовместимо изменить спецификацию, не изменив протокол и, таким образом, никак не затронув существующие интеграции (например, изменив `additionalProperties` с `false` на `true` в OpenAPI).

Вообще вопрос того, являются ли две версии спецификации обратно совместимыми — относится скорее к серой зоне, поскольку в самих стандартах спецификаций такое понятие не определено. Из общих соображений, утверждение «изменение спецификации является обратно-совместимым» тождественно утверждению «любой клиентский код, написанный или сгенерированный по этой спецификации, продолжит работать функционально корректно после релиза сервера, соответствующего обновлённой версии спецификации», однако в практическом смысле следовать этому определению достаточно тяжело. Изучить поведение всех мыслимых генераторов кода по спецификациям крайне трудоёмко (в частности, очень сложно предсказать, переживёт ли код, сгенерированный по спецификации с `additionalProperties: false` появление дополнительных полей в ответе).

Таким образом, использование IDL для описания API при всех плюсах этого подхода приводит к ещё одной существенной проблеме дрифта технологий: версии IDL и, что важнее, основанного на нём программного обеспечения, тоже постоянно обновляются, и далеко не всегда предсказуемым образом. Если же разработчик API придерживается подхода «code-first», т.е. генерирует спецификацию из актуального кода API, то появление обратно-несовместимых изменений в цепочке код сервера — спецификация — кодогенерированный SDK — клиентское приложение можно считать делом времени.

NB: мы здесь склонны советовать придерживаться разумного подхода, а именно — не использовать потенциально проблемные с точки зрения обратной совместимости возможности (включая упомянутый `additionalProperties: false`) и при оценке совместимости изменений исходить из соображения, что сгенерированный по спецификации код ведёт

себя так же, как и написанный вручную. В случае же неразрешимых сомнений вам не остаётся ничего другого, кроме как перебрать все имеющиеся кодогенераторы и проверить работоспособность их выдачи.

Политика обратной совместимости

Итого, если суммировать:

- вследствие итерационного развития приложений, платформ и предметной области вы будете вынуждены выпускать новые версии вашего API; в разных предметных областях скорость развития разная, но почти никогда не нулевая;
- вкупе это приведёт к фрагментации используемой версии API по приложениям и платформам;
- вам придётся принимать решения, критически влияющие на надёжность вашего API в глазах потребителей.

Опишем кратко эти решения и ключевые принципы их принятия.

1. Как часто выпускать мажорные версии API.

Это в основном *продуктовый* вопрос. Новая мажорная версия API выпускается, когда накоплена критическая масса функциональности, которую невозможно или слишком дорого поддерживать в рамках предыдущей мажорной версии. В стабильной ситуации такая необходимость возникает, как правило, раз в несколько лет. На динамично развивающихся рынках новые мажорные версии можно выпускать чаще, здесь ограничителем являются только ваши возможности выделить достаточно ресурсов для поддержания зоопарка версий. Однако следует заметить, что выпуск новой мажорной версии раньше, чем была стабилизирована предыдущая (а на это, как правило, требуется от нескольких месяцев до года), выглядит для разработчиков очень плохим сигналом, означающим риск *постоянно* сидеть на сырой платформе.

2. Какое количество мажорных версий поддерживать одновременно.

Теоретически, все когда-либо выпущенные. *Практически* следует смотреть на долю потребителей, реально продолжающих пользоваться версией, и выработать какие-то правила, когда прекращать поддержку старой версии.

3. Какое количество минорных версий (в рамках одной мажорной) поддерживать одновременно.

Для минорных версий возможны два варианта:

- если вы предоставляете только серверный API и компилируемые SDK, вы можете в принципе не поддерживать никакие минорные версии API, помимо актуальной (см. ниже); однако, на определённом этапе развития популярных API становится хорошим тоном поддерживать по крайней мере две последние версии;
- если вы предоставляете code-on-demand SDK, то вот здесь хорошим тоном является поддержка предыдущих минорных версий SDK в работающем состоянии на срок, достаточный для того, чтобы разработчики могли протестировать своё приложение с новой версией и внести какие-то правки по необходимости. Так как полностью переписывать приложения при этом не надо, разумно ориентироваться на длину релизных циклов в вашей индустрии, обычно это несколько месяцев в худшем случае.

NB: отдельным случаем являются «чистые» библиотеки (в том числе устанавливаемые пакетными менеджерами), которые не опираются на какие-то нижележащие API-сервисы. Как правило, разработчики таких библиотек предоставляют всю историю сборок, так что потребители могут использовать любую версию по своему усмотрению — что иногда оказывается необходимым из-за «ада зависимостей» между множеством таких библиотек.

Одновременный доступ к нескольким минорным версиям API

В современной промышленной разработке, особенно если мы говорим о внутренних API, новая версия, как правило, полностью заменяет предыдущую. Если в новой версии обнаруживаются критические ошибки, она может быть откачена (путём релиза предыдущей версии), но одновременно две сборки не существуют. В случае публичных API такой подход становится тем более опасным, чем больше партнёров используют API.

В самом деле, с ростом количества потребителей подход «откатить проблемную версию API в случае массовых жалоб» становится всё более деструктивным. Для партнёров, вообще говоря, оптимальным вариантом является жёсткая фиксация той версии API, для которой функциональность приложения была протестирована (и чтобы поставщик API при этом как-то незаметно исправлял возможные проблемы с информационной безопасностью и приводил своё ПО в соответствие с вновь возникающими законами).

NB. Из тех же соображений следует, что для популярных API также становится всё более желательным предоставление возможности подключать бета-, а может быть и альфа-версии для того, чтобы у партнёров была возможность заранее понять, какие проблемы ожидают их с релизом новой версии API.

Несомненный и очень важный плюс semver состоит в том, что она предоставляет возможность подключать версии с нужной гранулярностью:

- указание первой цифры (мажорной версии) позволяет гарантировать получение обратно совместимой версии API;
- указание двух цифр (минорной и мажорной версий) позволяет получить не просто обратно совместимую версию, но и необходимую функциональность, которая была добавлена уже после начального релиза;
- наконец, указание всех трёх цифр (мажорной, минорной и патча) позволяет жёстко зафиксировать конкретный релиз API, со всеми возможными особенностями (и ошибками) в его работе, что — теоретически — означает, что работоспособность интеграции не будет нарушена, пока эта версия физически доступна.

Понятно, что бесконечное сохранение минорных версий в большинстве случаев невозможно (в т.ч. из-за накапливающихся проблем с безопасностью и соответствием законодательству), однако предоставление такого доступа в течение разумного времени для больших API является гигиенической нормой.

NB. Часто в защиту политики только одной доступной версии API можно услышать аргумент о том, что кодом SDK или сервера API проблема обратной совместимости не исчерпывается, т.к. он может опираться на какие-то неверсионируемые сервисы (например, на какие-то данные в БД, которые должны разделяться между всеми версиями API) или другие API, придерживающиеся менее строгих политик. Это соображение, на самом деле, является лишь дополнительным аргументом в пользу изоляции таких зависимостей (см. главу «[Блокнот душевного покоя](#)»), поскольку это означает только лишь то, что изменения в этих подсистемах могут привести к неработоспособности API сами по себе.

Глава 27. О ватерлинии айсберга

Прежде, чем начинать разговор о принципах проектирования расширяемого API, следует обсудить гигиенический минимум. Огромное количество проблем не случилось бы, если бы разработчики API чуть ответственнее подходили к обозначению зоны своей ответственности.

1. Предоставляйте минимальный объём функциональности

В любой момент времени ваш API подобен айсбергу: у него есть видимая (документированная) часть и невидимая — недокументированная. В хорошем API эти две части соотносятся друг с другом примерно как надводная и подводная часть настоящего айсберга, 1 к 10. Почему так? Из двух очевидных соображений.

- Компьютеры существуют, чтобы сложные вещи делать просто, не наоборот. Код, который напишут разработчики поверх вашего API, должен в простых и лаконичных выражениях описывать решение сложной проблемы. Поэтому «внутри» ваш код, скорее всего, будет опираться на мощную номенклатуру непубличной функциональности.
- Изъятие функциональности из API невозможно без серьёзных потерь. Если вы пообещали предоставлять какую-то функциональность — вам теперь придётся предоставлять её «вечно» (до окончания поддержки этой мажорной версии API). Объявление функциональности неподдерживаемой — очень сложный и чреватый потенциальными конфликтами с потребителем процесс.

Правило № 1 самое простое: если какую-то функциональность можно не выставлять наружу — значит, выставлять её не надо. Можно сформулировать и так: каждая сущность, каждое поле, каждый метод в публичном API — это *продуктовое* решение. Должны существовать веские *продуктовые* причины, по которым та или иная сущность документирована.

2. Избегайте серых зон и недосказанности

Ваши обязательства по поддержанию функциональности должны быть оговорены настолько чётко, насколько это возможно. Особенно это касается тех сред и платформ, где нет способа нативно ограничить доступ к недокументированной функциональности. К сожалению, разработчики часто считают, что, если они «нашли» какую-то непубличную особенность, то они могут ей пользоваться — а производитель API, соответственно, обязан её поддерживать. Поэтому политика компании относительно таких «находок» должна быть явно сформулирована. Тогда в случае несанкционированного использования скрытой функциональности вы по крайней мере сможете сослаться на документацию и быть формально правы в глазах комьюнити.

Однако достаточно часто разработчики API сами легитимизируют такие серые зоны, например:

- отдают недокументированные поля в ответах эндпойнтов;
- используют непубличную функциональность в примерах кода — в документации, в ответ на обращения пользователей, в выступлениях на конференциях и т.д.

Нельзя принять обязательства наполовину. Или вы гарантируете работу этого кода всегда, или не подавайте никаких намёков на то, что такая функциональность существует.

3. Фиксируйте неявные договорённости

Посмотрите внимательно на код, который предлагаете написать разработчикам: нет ли в нём каких-то условностей, которые считаются очевидными, но при этом нигде не зафиксированы?

Пример 1. Рассмотрим SDK работы с заказами.

```
// Создаёт заказ
let order = api.createOrder();
// Получает статус заказа
let status = api.getStatus(order.id);
```

Предположим, что в какой-то момент при масштабировании вашего сервиса вы пришли к событийной консистентности (см. [соответствующую главу](#)). К чему это приведёт? К тому, что код выше перестанет работать. Разработчик создал заказ, пытается получить его статус — и получает ошибку. Очень тяжело предсказать, какую реакцию на эту ошибку предусмотрят разработчики — вероятнее всего, никакую.

Вы можете сказать: «Позвольте, но мы нигде и не обещали строгую консистентность!» — и это будет, конечно, неправдой. Вы можете так сказать если, и только если, вы действительно в документации метода `createOrder` явно описали нестрогую консистентность, а все ваши примеры использования SDK написаны как-то так:

```
let order = api.createOrder();
let status;
while (true) {
  try {
    status = api.getStatus(order.id);
  } catch (e) {
    if (e.statusCode != 404 || timeoutExceeded()) {
      break;
    }
  }
}
if (status) {
  ...
}
```

Мы полагаем, что можно не уточнять, что писать код, подобный вышеприведённому, ни в коем случае нельзя. Уж если вы действительно предоставляете нестрогое консистентный API, то либо операция `createOrder` в SDK должна быть асинхронной и возвращать результат только по готовности всех реплик, либо политика перезапросов должна быть скрыта внутри операции `getStatus`.

Если же нестрогая консистентность не была описана с самого начала — вы не можете внести такие изменения в API. Это эффективный слом обратной совместимости, который к тому же приведёт к огромным проблемам ваших потребителей, поскольку проблема будет воспроизводиться случайным образом.

Пример 2. Представьте себе следующий код:

```
let resolve;
let promise = new Promise(
  function (innerResolve) {
    resolve = innerResolve;
  }
);
resolve();
```

Этот код полагается на то, что callback-функция, переданная в `new Promise` будет выполнена *синхронно*, и переменная `resolve` будет инициализирована к моменту вызова `resolve()`. Однако это конвенция абсолютно ниоткуда не следует: ничто в сигнатуре конструктора `new Promise` не указывает на синхронный вызов callback-a.

Разработчики языка, конечно, могут позволить себе такие фокусы. Однако вы как разработчик API — не можете. Вы должны как минимум задокументировать это поведение и подобрать сигнатуры так, чтобы оно было очевидно; но вообще хорошим советом будет избегать таких конвенций, поскольку они банально неочевидны при прочтении кода, использующего ваш API. Ну и конечно же ни при каких обстоятельствах вы не можете изменить это поведение с синхронного на асинхронное.

Пример 3. Представьте, что вы предоставляете API для анимаций, в котором есть две независимые функции:

```
// Анимирует ширину некоторого объекта
// от первого значения до второго
// за указанное время
object.animateWidth('100px', '500px', '1s');
// Наблюдает за изменением размеров объекта
object.observe('widthchange', observerFunction);
```

Возникает вопрос: с какой частотой и в каких точках будет вызываться `observerFunction`? Допустим, в первой версии SDK вы эмулировали анимацию пошагово с частотой 10 кадров в секунду — тогда `observerFunction` будет вызвана 10 раз и получит значения '140px', '180px' и т.д. вплоть до '500px'. Но затем в новой версии API вы решили воспользоваться системными функциями для обеих операций — и теперь вы попросту не знаете, когда и с какой частотой будет вызвана `observerFunction`.

Даже просто изменение частоты вызовов вполне может сделать чей-то код неработающим — например, если обработчик выполняет на каждом шаге тяжелые вычисления, и разработчик не предусмотрел никакого ограничения частоты выполнения, полагаясь на то, что ваш SDK вызывает его обработчик всего лишь 10 раз в секунду. А вот если, например, `observerFunction` перестанет вызываться с финальным значением '500px' вследствие каких-то особенностей системных алгоритмов — чей-то код вы сломаете абсолютно точно.

В данном случае следует задокументировать конкретный контракт — как и когда вызывается `callback` — и придерживаться его даже при смене нижележащей технологии.

Пример 4. Представьте, что потребитель совершає заказ, которые проходит через вполне определённую цепочку преобразований:

```
GET /v1/orders/{id}/events/history
→
{
  "event_history": [
    {
      "iso_datetime": "2020-12-29T00:35:00+03:00",
      "new_status": "created"
    },
    {
      "iso_datetime": "2020-12-29T00:35:10+03:00",
      "new_status": "payment_approved"
    },
    {
      "iso_datetime": "2020-12-29T00:35:20+03:00",
      "new_status": "preparing_started"
    },
    {
      "iso_datetime": "2020-12-29T00:35:30+03:00",
      "new_status": "ready"
    }
  ]
}
```

Допустим, в какой-то момент вы решили надёжным клиентам с хорошей историей заказов предоставлять кофе «в кредит», не дожидаясь подтверждения платежа. Т.е. заказ перейдёт в статус "preparing_started", а может и "ready", вообще без события "payment_approved". Вам может показаться, что это изменение является обратно-совместимым — в самом деле, вы же и не обещали никакого конкретного порядка событий. Но это, конечно, не так.

Предположим, что у разработчика (вероятно, бизнес-партнёра вашей компании) написан какой-то код, выполняющий какую-то полезную бизнес функцию поверх этих событий — например, строит аналитику по затратам и доходам. Вполне логично ожидать, что этот код будет оперировать какой-то машиной состояний, которая будет переходить в то или иное состояние в зависимости от получения или неполучения события. Аналитический код наверняка сломается вследствие изменения порядка событий. В лучшем случае разработчик увидит какие-то исключения и будет вынужден разбираться с причиной; в худшем случае партнёр будет оперировать неправильной статистикой неопределенное время, пока не найдёт в ней ошибку.

Правильным решением было бы, во-первых, изначально задокументировать порядок событий и допустимые состояния; во-вторых, продолжать генерировать событие "payment_approved" перед "preparing_started" (если вы приняли решение выполнять такой заказ — значит, по сути, подтвердили платёж) и добавить расширенную информацию о платеже.

Этот пример подводит нас к ещё к одному правилу.

4. Продуктовая логика тоже должна быть обратно совместимой

Такие критичные вещи, как график переходов между статусами, порядок событий и возможные причины тех или иных изменений — должны быть документированы. Далеко не все детали бизнес-логики можно выразить в форме контрактов на эндпоинты, а некоторые вещи нельзя выразить вовсе.

Представьте, что в один прекрасный день вы заводите специальный номер телефона, по которому клиент может позвонить в колл-центр и отменить заказ. Вы даже можете сделать это *технически* обратно-совместимым образом, добавив новых необязательных полей в сущность «заказ». Но конечный потребитель может просто знать нужный номер телефона, и позвонить по нему, даже если приложение его не показало. При этом код бизнес-аналитика партнёра всё так же может сломаться или начать показывать погоду на Марсе, т.к. он был написан когда-то, ничего не зная о возможности отменить заказ, сделанный в приложении партнёра, каким-то иным образом, не через самого партнёра же.

Технически корректным решением в данной ситуации могло бы быть добавление параметра «разрешено отменять через колл-центр» в функцию создания заказа — и, соответственно, запрет операторам колл-центра отменять заказы, если флаг не был указан при их создании. Но это в свою очередь плохое решение *с точки зрения продукта*. «Хорошее» решение здесь только одно — изначально предусмотреть возможность внешних отмен в API; если же вы её не предвидели — остаётся воспользоваться «блокнотом душевного спокойствия», речь о котором пойдёт в [последней главе настоящего раздела](#).

Глава 28. Расширение через абстрагирование

В предыдущих разделах мы старались приводить теоретические правила и иллюстрировать их на практических примерах. Однако понимание принципов проектирования API, устойчивого к изменениям, как ничто другое требует прежде всего практики. Знание о том, куда стоит «постелить соломку» — оно во многом «сын ошибок трудных». Нельзя предусмотреть всего — но можно выработать необходимый уровень технической интуиции.

Поэтому в этом разделе мы поступим следующим образом: возьмём наш [модельный API](#) из предыдущего раздела, и проверим его на устойчивость в каждой возможной точке — проведём некоторый «вариационный анализ» наших интерфейсов. Ещё более конкретно — к каждой сущности мы подойдём с вопросом «что, если?» — что, если нам потребуется предоставить партнёрам возможность написать свою независимую реализацию этого фрагмента логики.

NB. В рассматриваемых нами примерах мы будем выстраивать интерфейсы так, чтобы связывание разных сущностей происходило динамически в реальном времени; на практике такие интеграции будут делаться на стороне сервера путём написания *ad hoc* кода и формирования конкретных договорённостей с конкретным клиентом, однако мы для целей обучения специально будем идти более сложным и абстрактным путём. Динамическое связывание в реальном времени применимо скорее к сложным программным конструктам типа API операционных систем или встраиваемых библиотек; приводить обучающие примеры на основе систем подобной сложности было бы, однако, чересчур затруднительно.

Начнём с базового интерфейса. Предположим, что мы пока что вообще не раскрывали никакой функциональности помимо поиска предложений и заказа, т.е. мы предоставляем API из двух методов — `POST /offers/search` и `POST /orders`.

Сделаем следующий логический шаг и предположим, что партнёры захотят динамически подключать к нашей платформе свои собственные кофемашины с каким-то новым API. Для этого нам будет необходимо договориться о формате обратного вызова, таким образом мы будем вызывать API партнёра, и предоставить два новых эндпойнта для:

- регистрации в системе новых типов API;
- загрузки списка кофемашин партнёра с указанием типа API.

Например, можно предоставить второе семейство API (специально для партнёров), содержащее вот такие методы.

```
// 1. Зарегистрировать новый тип API
PUT /v1/api-types/{api_type}
{
    "order_execution_endpoint": {
        // Callback function description
    }
}
```

```
// 2. Предоставить список кофемашин с разбивкой
// по типу API
PUT /v1/partners/{partnerId}/coffee-machines
{
    "coffee_machines": [
        {
            "api_type",
            "location",
            "supported_recipes"
        }, ...
    ]
}
```

Таким образом механика следующая:

- партнёр описывает свои виды API, кофемашины и поддерживаемые рецепты;
- при получении заказа, который необходимо выполнить на конкретной кофемашине, наш сервер обратится к функции обратного вызова, передав ей данные о заказе в оговорённом формате.

Теперь партнёры могут динамически подключать свои кофемашины и обрабатывать заказы. Займёмся теперь, однако, вот каким упражнением:

- перечислим все неявные предположения, которые мы допустили;
- перечислим все неявные механизмы связывания, которые необходимы для функционирования платформы.

Может показаться, что в нашем API нет ни того, ни другого, ведь он очень прост и по сути просто сводится к вызову какого-то HTTP-метода — но это неправда.

1. Предполагается, что каждая кофемашина поддерживает все возможные опции заказа (например, допустимый объём напитка).
2. Нет необходимости показывать пользователю какую-то дополнительную информацию о том, что заказ готовится на новых типах кофемашин.
3. Цена напитка не зависит ни от партнёра, ни от типа кофемашины.

Эти пункты мы выписали с одной целью: нам нужно понять, каким конкретно образом мы будем переводить неявные договорённости в явные, если нам это потребуется. Например, если разные кофемашины предоставляют разный объём функциональности — допустим, в каких-то кофейнях объём кофе фиксирован — что должно измениться в нашем API?

Универсальный паттерн внесения подобных изменений таков: мы должны рассмотреть существующий интерфейс как частный случай некоторого более общего, в котором значения некоторых параметров приняты известными по умолчанию, а потому опущены. Таким образом, внесение изменений всегда происходит в три шага.

1. Явная фиксация программного контракта *в том объёме, в котором она действует на текущий момент*.
2. Расширение функциональности: добавление нового метода, который позволяет обойти ограничение, зафиксированное в п. 1.
3. Объявление существующих вызовов (из п. 1) "хелперами" к новому формату (из п. 2), в которых значение новых опций считается равным значению по умолчанию.

На нашем примере с изменением списка доступных опций заказа мы должны поступить следующим образом.

1. Документируем текущее состояние. Все кофемашины, подключаемые по API, обязаны поддерживать три опции: посыпку корицей, изменение объёма и бесконтактную выдачу.
2. Добавляем новый метод `with-options`:

```
PUT /v1/partners/{partner_id}/coffee-machines-with-options
{
  "coffee_machines": [
    {
      "id",
      "api_type",
      "location",
      "supported_recipes",
      "supported_options": [
        {"type": "volume_change"}
      ],
    },
    ...
  }
}
```

3. Объявляем, что вызов `PUT /coffee-machines`, как он представлен сейчас в протоколе, эквивалентен вызову `PUT /coffee-machines-with-options`, если в последний передать три опции — посыпку корицей, изменение объёма и бесконтактную выдачу, — и, таким образом, является частным случаем — хелпером к более общему вызову.

Часто вместо добавления нового метода можно добавить просто необязательный параметр к существующему интерфейсу — в нашем случае, можно добавить необязательный параметр `options` к вызову `PUT /coffee-machines`.

NB. Когда мы говорим о фиксации договоренностей, действующих в настоящий момент — речь идёт о *внутренних* договорённостях. Мы должны были потребовать от партнёров поддерживать указанный список опций, когда обговаривали формат взаимодействия. Если же мы этого не сделали изначально, а потом решили зафиксировать договорённости в ходе расширения функциональности внешнего API — это очень серьёзная заявка на нарушение обратной совместимости, и так делать ни в коем случае не надо, см. главу «[О ватерлинии айсберга](#)».

Границы применимости

Хотя это упражнение выглядит весьма простым и универсальным, его использование возможно только при наличии хорошо продуманной архитектуры сущностей и, что ещё более важно, понятного вектора дальнейшего развития API. Представим, что через какое-то время к поддерживаемым опциям добавились новые — ну, скажем, добавление сиропа и второго шота эспрессо. Список опций расширить мы можем — а вот изменить соглашение по умолчанию уже нет. Через некоторое время это приведёт к тому, что «дефолтный» интерфейс `PUT /coffee-machines` окажется никому не нужен, поскольку «дефолтный» список из трёх опций окажется не только редко востребованным, но и просто абсурдным — почему эти три, чем они лучше всех остальных? По сути значения по умолчанию и номенклатура старых методов начнут отражать исторические этапы развития нашего API, а это совершенно не то, чего мы хотели бы от номенклатуры хелперов и значений по умолчанию.

Увы, здесь мы сталкиваемся с плохо разрешимым противоречием: мы хотим, с одной стороны, чтобы разработчик писал лаконичный код, следовательно, должны предоставлять хорошие хелперные методы и значения по умолчанию. С другой, знать наперёд какими будут самые частотные наборы опций через несколько лет развития API — очень сложно.

NB. Замаскировать эту проблему можно так: в какой-то момент собрать все эти «странные» в одном месте и переопределить все значения по умолчанию скопом под одним параметром. Условно говоря, вызов одного метода, например, `POST /use-defaults {"version": "v2"}` переопределяет все значения по умолчанию на более разумные. Это упростит порог входа и уменьшит количество вопросов, но документация от этого станет выглядеть только хуже.

В реальной жизни как-то нивелировать проблему помогает лишь слабая связность объектов, речь о которой пойдёт в следующей главе.

Глава 29. Сильная связность и сопутствующие проблемы

Для демонстрации проблем сильной связности перейдём теперь к *действительно интересным* вещам. Продолжим наш «вариационный анализ»: что, если партнёры хотят не просто готовить кофе по стандартным рецептам, но и предлагать свои авторские напитки? Вопрос этот с подвохом: в том виде, как мы описали партнёрский API в предыдущей главе, факт существования партнёрской сети никак не отражён в нашем API с точки зрения продукта, предлагаемого пользователю, а потому представляет собой довольно простой кейс. Если же мы пытаемся предоставить не какую-то дополнительную возможность, а модифицировать саму базовую функциональность API, то мы быстро столкнёмся с проблемами совсем другого порядка.

Итак, добавим ещё один эндпойнт — для регистрации собственного рецепта партнёра.

```
// Добавляет новый рецепт
POST /v1/recipes
{
    "id",
    "product_properties": {
        "name",
        "description",
        "default_volume"
        // Прочие параметры, описывающие
        // напиток для пользователя
        ...
    }
}
```

На первый взгляд, вполне разумный и простой интерфейс, который явно декомпозируется согласно уровням абстракции. Попробуем теперь представить, что произойдёт в будущем — как дальнейшее развитие функциональности повлияет на этот интерфейс.

Первая проблема очевидна тем, кто внимательно читал главу «[Описание конечных интерфейсов](#)»: продуктовые данные должны быть локализованы. Это приведёт нас к первому изменению:

```
"product_properties": {  
    // "l10n" – стандартное сокращение  
    // для "localization"  
    "l10n": [{  
        "language_code": "en",  
        "country_code": "US",  
        "name",  
        "description"  
    }, /* другие языки и страны */ ... ]  
}
```

И здесь возникает первый большой вопрос — а что делать с `default_volume`? С одной стороны, это объективная величина, выраженная в стандартизованных единицах измерения, и она используется для запуска программы на исполнение. С другой стороны, для таких стран, как США, мы будем обязаны указать объём не в виде «300 мл», а в виде «10 унций». Мы можем предложить одно из двух решений:

- либо партнёр указывает только числовой объём, а числовые представления мы сделаем сами;
- либо партнёр указывает и объём, и все его локализованные представления.

Первый вариант плох тем, что партнёр с помощью нашего API может как раз захотеть разработать сервис для какой-то новой страны или языка — и не сможет, пока локализация для этого региона не будет поддержана в самом API. Второй вариант плох тем, что сработает только для заранее заданных объёмов — заказать кофе произвольного объёма нельзя. И вот практически первым же действием мы сами загоняем себя в тупик.

Проблемами с локализацией, однако, недостатки дизайна этого API не заканчиваются. Следует задать себе вопрос — а *зачем* вообще здесь нужны `name` и `description`? Ведь это по сути просто строки, не имеющие никакой определённой семантики. На первый взгляд — чтобы возвращать их обратно из метода `/v1/search`, но ведь это тоже не ответ: а *зачем* эти строки возвращаются из `search`?

Корректный ответ — потому что существует некоторое представление, UI для выбора типа напитка. По-видимому, `name` и `description` — это просто два описания напитка, короткое (для показа в общем прейскуранте) и длинное (для показа расширенной информации о продукте). Получается, что мы устанавливаем требования на API исходя из вполне конкретного дизайна. Но что, если партнёр сам делает UI для своего приложения? Мало того, что ему

могут быть не нужны два описания, так мы по сути ещё и вводим его в заблуждение: `name` — это не «какое-то» название, оно предполагает некоторые ограничения. Во-первых, у него есть некоторая рекомендованная длина, оптимальная для конкретного UI; во-вторых, оно должно консистентно выглядеть в одном списке с другими напитками. В самом деле, будет очень странно смотреться, если среди «Капучино», «Лунго» и «Латте» вдруг появится «Бодрящая свежесть» или «Наш самый качественный кофе».

Эта проблема разворачивается и в другую сторону — UI (наш или партнёра) обязательно будет развиваться, в нём будут появляться новые элементы (картинка для кофе, его пищевая ценность, информация об аллергенах и так далее). `product_properties` со временем превратится в свалку из большого количества необязательных полей, и выяснить, задание каких из них приведёт к каким эффектам в каком приложении можно будет только методом проб и ошибок.

Проблемы, с которыми мы столкнулись — это проблемы *сильной связности*. Каждый раз, предлагая интерфейс, подобный вышеприведённому, мы фактически описываем имплементацию одной сущности (рецепта) через имплементации других (визуального макета, правил локализации). Этот подход противоречит самому принципу проектирования API «сверху вниз», поскольку **низкоуровневые сущности не должны определять высокоуровневые**.

Правило контекстов

Как бы парадоксально это ни звучало, обратное утверждение тоже верно: высокоуровневые сущности тоже не должны определять низкоуровневые. Это попросту не их ответственность. Выход из этого логического лабиринта таков: высокоуровневые сущности должны *определять контекст*, который другие объекты будут интерпретировать. Чтобы спроектировать добавление нового рецепта нам нужно не формат данных подобрать — нам нужно понять, какие (возможно, неявные, т.е. не представленные в виде API) контексты существуют в нашей предметной области.

Как уже понятно, существует контекст локализации. Есть какой-то набор языков и регионов, которые мы поддерживаем в нашем API, и есть требования — что конкретно необходимо предоставить партнёру, чтобы API заработал на новом языке в новом регионе. Конкретно в случае объёма кофе где-то в недрах нашего API (во внутренней реализации или в составе SDK) есть функция форматирования строк для отображения объёма напитка:

```
l10n.volume.format = function(  
    value, language_code, country_code  
) { ... }  
/*  
  l10n.formatVolume(  
    '300ml', 'en', 'UK'  
  ) → '300 ml'  
  l10n.formatVolume(  
    '300ml', 'en', 'US'  
  ) → '10 fl oz'  
*/
```

Чтобы наш API корректно заработал с новым языком или регионом, партнёр должен или задать эту функцию через партнёрский API, или указать, какую из существующих локализаций необходимо использовать. Для этого мы абстрагируем-и-расширяем API, в соответствии с описанной в предыдущей главе процедурой, и добавляем новый эндпойнт — настройки форматирования:

```
// Добавляем общее правило форматирования
// для русского языка
PUT /formatters/volume/ru
{
    "template": "{volume} мл"
}
// Добавляем частное правило форматирования
// для русского языка в регионе «США»
PUT /formatters/volume/ru/US
{
    // В США требуется сначала пересчитать
    // объём, потом добавить постфикс
    "value_transform": {
        "action": "divide",
        "divisor": 30
    },
    "template": "{volume} ун."
}
```

Таким образом, реализация вышеупомянутой функции `l10n.volume.format` сможет извлечь правила для локали `ru/US` и отформатировать представление объёма согласно им.

NB: мы, разумеется, в курсе, что таким простым форматом локализации единиц измерения в реальной жизни обойтись невозможно, и необходимо либо положиться на существующие библиотеки, либо разработать сложный формат описания (учитывающий, например, падежи слов и необходимую точность округления), либо принимать правила форматирования в императивном виде (т.е. в виде кода функции). Пример выше приведён исключительно в учебных целях.

Вернёмся теперь к проблеме `name` и `description`. Для того, чтобы снизить связность в этом аспекте, нужно прежде всего формализовать (возможно, для нас самих, необязательно во внешнем API) понятие «макета». Мы требуем `name` и `description` не просто так в вакууме, а чтобы представить их во вполне конкретном UI. Этому конкретному UI можно дать идентификатор или значимое имя.

```

GET /v1/layouts/{layout_id}
{
  "id",
  // Макетов вполне возможно
  // будет много разных,
  // поэтому имеет смысл сразу заложить
  // расширяемость
  "kind": "recipe_search",
  // Описываем каждое свойство рецепта,
  // которое должно быть задано для
  // корректной работы макета
  "properties": [
    {
      // Раз уж мы договорились, что `name`
      // на самом деле нужен как заголовок
      // в списке результатов поиска -
      // разумнее его так и назвать
      // `search_title`
      "field": "search_title",
      "view": {
        // Машиночитаемое описание того,
        // как будет показано поле
        "min_length": "5em",
        "max_length": "20em",
        "overflow": "ellipsis"
      }
    },
    ...],
    // Какие поля обязательны
    "required": [
      "search_title",
      "search_description"
    ]
}

```

Таким образом, партнёр сможет сам решить, какой вариант ему предпочтителен. Можно задать необходимые поля для стандартного макета:

```

PUT /v1/recipes/{id}«
  /properties/l10n/{lang}
{
  "search_title", "search_description"
}

```

Либо создать свой макет и задавать нужные для него поля. В конце концов, партнёр может отрисовывать UI самостоятельно и вообще не пользоваться этой техникой, не задавая ни макеты, ни поля.

Наш интерфейс добавления рецепта получит в итоге вот такой вид:

```
POST /v1/recipes
{
  "id"
}
→
{
  "id"
}
```

Этот вывод может показаться совершенно контринтуитивным, однако отсутствие полей у сущности «рецепт» говорит нам только о том, что сама по себе она не несёт никакой семантики и служит просто способом указания контекста привязки других сущностей. В реальном мире следовало бы, пожалуй, собрать эндпойнт-строитель, который может создавать сразу все нужные контексты одним запросом:

```
POST /v1/recipe-builder
{
  "id",
  // Задаём свойства рецепта
  "product_properties": {
    "default_volume",
    "110n"
  },
  // Создаём необходимые макеты
  "layouts": [
    {
      "id", "kind", "properties"
    }
  ],
  // Добавляем нужные форматтеры
  "formatters": {
    "volume": [
      {
        "language_code",
        "template"
      },
      {
        "language_code",
        "country_code",
        "template"
      }
    ]
  },
  // Прочие действия, которые необходимо
  // выполнить для корректного заведения
  // нового рецепта в системе
  ...
}
```

Заметим, что передача идентификатора вновь создаваемой сущности клиентом — не лучший паттерн. Но раз уж мы с самого начала решили, что идентификаторы рецептов — не просто случайные наборы символов, а значимые строки, то нам теперь придётся с этим как-то жить. Очевидно, в такой ситуации мы рискуем многочисленными коллизиями между

названиями рецептов разных партнёров, поэтому операцию, на самом деле, следует модифицировать: либо для партнёрских рецептов всегда пользоваться парой идентификаторов (партнёра и рецепта), либо ввести составные идентификаторы, как мы ранее рекомендовали в главе «[Описание конечных интерфейсов](#)».

```
POST /v1/recipes/custom
{
    // Первая часть идентификатора:
    // например, в виде идентификатора клиента
    "namespace": "my-coffee-company",
    // Вторая часть идентификатора
    "id_component": "lungo-customato"
}
→
{
    "id":
        "my-coffee-company:lungo-customato"
}
```

Заметим, что в таком формате мы сразу закладываем важное допущение: различные партнёры могут иметь как полностью изолированные неймспейсы, так и разделять их. Более того, мы можем ввести специальные неймспейсы типа "common", которые позволят публиковать новые рецепты для всех. (Это, кстати говоря, хорошо ещё и тем, что такой API мы сможем использовать для организации нашей собственной панели управления контентом.)

NB: внимательный читатель может подметить, что этот приём уже был продемонстрирован в нашем учебном API гораздо раньше в главе «[Разделение уровней абстракции](#)» на примере сущностей «программа» и «запуск программы». В самом деле, мы могли бы обойтись без программ и без эндпоинта `program-matcher` и пойти вот таким путём:

```
GET /v1/recipes/{id}/run-data/{api_type}
→
{ /* описание способа запуска
   указанного рецепта на
   машинах с поддержкой
   указанного типа API */ }
```

Тогда разработчикам пришлось бы сделать примерно следующее для запуска приготовления кофе:

- выяснить тип API конкретной кофемашины;

- получить описание способа запуска программы выполнения рецепта на машине с API такого типа;
- в зависимости от типа API выполнить специфические команды запуска.

Очевидно, что такой интерфейс совершенно недопустим — просто потому, что в подавляющем большинстве случаев разработчикам совершенно неинтересно, какого рода API поддерживает та или иная кофемашина. Для того чтобы не допустить такого плохого интерфейса, мы ввели новую сущность «программа», которая по факту представляет собой не более чем просто идентификатор контекста, как и сущность «рецепт».

Аналогичным образом устроена и сущность `program_run_id`, идентификатор запуска программы. Он также по сути не имеет почти никакого интерфейса и состоит только из идентификатора запуска.

Глава 30. Слабая связность

В предыдущей главе мы продемонстрировали, как разрыв сильной связности приводит к декомпозиции сущностей и схлопыванию публичных интерфейсов до минимума. Вернёмся теперь к вопросу, который мы вскользь затронули в главе «[Расширение через абстрагирование](#)»: каким образом нам нужно параметризовать приготовление заказа, если оно исполняется через сторонний API? Иными словами, что такое этот самый `order_execution_endpoint`, передавать который мы потребовали при регистрации нового типа API?

```
PUT /v1/api-types/{api_type}
{
    "order_execution_endpoint": {
        // ???
    }
}
```

Исходя из общей логики мы можем предположить, что любой API так или иначе будет выполнять три функции: запускать программы с указанными параметрами, возвращать текущий статус запуска и завершать (отменять) заказ. Самый очевидный подход к реализации такого API — просто потребовать от партнёра имплементировать вызов этих трёх функций удалённо, например следующим образом:

```
PUT /v1/api-types/{api_type}
{
    "order_execution_endpoint": {
        "program_run_endpoint": {
            /* Какое-то описание
               удалённого вызова */
            "type": "grpc",
            "endpoint": <URL>,
            "parameters"
        },
        "program_get_state_endpoint",
        "program_cancel_endpoint"
    }
}
```

NB: во многом таким образом мы переносим сложность разработки API в плоскость разработки форматов данных (каким образом мы будем передавать параметры запуска в `program_run_endpoint`, и в каком формате должен отвечать `program_get_state_endpoint`, но в рамках этой главы мы сфокусируемся на других вопросах.)

Хотя это API и кажется абсолютно универсальным, на его примере можно легко показать, каким образом изначально простые и понятные API превращаются в сложные и запутанные. У этого дизайна есть две основные проблемы:

1. Он хорошо описывает уже реализованные нами интеграции (т.е. в эту схему легко добавить поддержку известных нам типов API), но не привносит никакой гибкости в подход: по сути мы описали только известные нам способы интеграции, не попытавшись взглянуть на более общую картину.
2. Этот дизайн изначально основан на следующем принципе: любое приготовление заказа можно описать этими тремя императивными командами.

Пункт 2 очень легко опровергнуть, что автоматически вскроет проблемы пункта 1. Предположим для начала, что в ходе развития функциональности мы решили дать пользователю возможность изменять свой заказ уже после того, как он создан — например, попросить выдать заказ бесконтактно. Это автоматически влечёт за собой добавление нового эндпойнта, ну скажем, `program_modify_endpoint`, и новых сложностей в формате обмена данными (нам нужно пересылать данные о появлении требования о бесконтактной доставке и его удовлетворении). Что важно, и то, и другое (и эндпойнт, и новые поля данных) из соображений обратной совместимости будут необязательными.

Теперь попытаемся придумать какой-нибудь пример реального мира, который не описывается нашими тремя императивами. Это довольно легко: допустим, мы подключим через наш API не кофейню, а вендинговый автомат. Это, с одной стороны, означает, что эндпойнт `modify` и вся его связь для этого типа API бесполезны — требование бесконтактной выдачи попросту ничего не значит для автомата. С другой, автомат, в отличие от оперируемой людьми кофейни, требует программного способа *подтверждения выдачи* напитка: пользователь делает заказ, находясь где-то в другом месте, потом доходит до автомата и

нажимает в приложении кнопку «выдать заказ». Мы могли бы, конечно, потребовать, чтобы пользователь создавал заказ автомату, стоя прямо перед ним, но это, в свою очередь, противоречит нашей изначальной концепции, в которой пользователь выбирает и заказывает напиток, исходя из доступных опций, а потом идёт в указанную точку, чтобы его забрать.

Программная выдача напитка потребует добавления ещё одного эндпойнта, ну скажем, `program_takeout_endpoint`. И вот мы уже запутались в лесу из пяти эндпойнтов:

- для работы вендинговых автоматов нужно реализовать эндпойнт `program_takeout_endpoint`, но не нужно реализовывать `program_modify_endpoint`;
- для работы обычных кофеен нужно реализовать эндпойнт `program_modify_endpoint`, но не нужно реализовывать `program_takeout_endpoint`.

При этом в документации интерфейса мы опишем и тот, и другой эндпойнт. Как несложно заметить, интерфейс `takeout` весьма специфичен. Если запрос бесконтактной доставки мы как-то скрыли за общим `modify`, то на вот такие операции типа подтверждения выдачи нам каждый раз придётся заводить новый метод с уникальным названием. Несложно представить себе, как через несколько итераций интерфейс превратится в свалку из визуально похожих методов, притом формально необязательных — но для подключения своего API нужно будет прочитать документацию каждого и разобраться в том, нужен ли он в конкретной ситуации или нет.

NB: в этом примере мы предполагаем, что наличие эндпойнта `program_takeout_endpoint` является триггером для приложения, которое должно показать кнопку «выдать заказ». Было бы лучше добавить что-то типа поля `supported_flow` в параметры вызова `PUT /api-types/`, чтобы этот флаг задавался явно, а не определялся из неочевидной конвенции. Однако в проблематике замусоривания интерфейсов optionalными методами это ничего не меняет, так что мы опустили эту тонкость ради лаконичности примеров.

Мы не знаем, правда ли в реальном мире API кофемашин возникнет проблема, подобная описанной. Но мы можем сказать со всей уверенностью, что *всегда*, когда речь идёт об интеграции «железного» уровня, происходят именно те процессы, которые мы описали: меняется нижележащая технология, и вроде бы понятный и ясный API превращается в свалку из legacy-методов, половина из которых не несёт в себе никакого практического смысла в рамках конкретной интеграции. Если мы добавим к проблеме ещё и технический прогресс — представим, например, что со временем все кофейни станут автоматическими — то мы быстро придём к ситуации, когда половина методов *вообще не нужна*, как метод запроса бесконтактной выдачи напитка.

Заметим также, что мы невольно начали нарушать принцип изоляции уровней абстракции. На уровне API вендингового автомата вообще не существует понятия «бесконтактная выдача», это по сути продуктовый термин.

Каким же образом мы можем решить эту проблему? Одним из двух способов: или досконально изучить предметную область и тренды её развития на несколько лет вперёд, или перейти от сильной связности к слабой. Как выглядит идеальное решение с точки зрения обеих взаимодействующих сторон? Как-то так:

- вышестоящий API программ не знает, как устроен уровень исполнения его команд; он формулирует задание так, как понимает на своём уровне: сварить такой-то кофе такого-то объёма, передать пожелания пользователя партнёру, выдать заказ;
- нижележащий API исполнения программы не заботится о том, какие ещё вокруг бывают API того же уровня; он трактует только ту часть задания, которая имеет для него смысл.

Если мы посмотрим на принципы, описанные в предыдущей главе, то обнаружим, что этот принцип мы уже формулировали: нам необходимо задать *информационный контекст* на каждом из уровней абстракции, и разработать механизм его трансляции. Более того, в общем виде он был сформулирован ещё в разделе «Потоки данных» главы [«Разделение уровней абстракции»](#).

В нашем конкретном примере нам нужно имплементировать следующие механизмы:

- запуск программы создаёт контекст её исполнения, содержащий все существенные параметры;

- существует поток обмена информацией об изменении состояния: исполнитель может читать контекст, узнавать о всех его модификациях и сообщать обратно о изменениях своего состояния.

Организовать и то, и другое можно разными способами (см. [соответствующую главу](#) раздела «Паттерны дизайна API»); по сути мы всегда имеем два контекста и поток событий между ними. В случае SDK эту идею можно было бы выразить через генерацию событий:

```
/* Имплементация партнёром интерфейса
запуска программы на его кофемашинах */
registerProgramRunHandler(
  apiType, (program) => {
    // Инициализация исполнения заказа
    // на стороне партнера
    let execution = initExecution(...);
    // Подписка на изменения состояния
    // родительского контекста
    program.context.on(
      'takeout_requested', () => {
        // Если запрошена выдача заказа,
        // инициировать нужные операции
        await execution.prepareTakeout();
        // Как только напиток готов к выдаче,
        // оповестить об этом
        execution.context.emit('takeout_ready');
      }
    );
    program.context.on(
      'order_canceled', () => {
        await execution.cancel();
        execution.context.emit('canceled');
      }
    );
    return execution.context;
  );
}
```

NB: в случае HTTP API соответствующий пример будет выглядеть более громоздко, поскольку потребует создания отдельных эндпоинтов для обмена сообщениями типа GET /program-run/events и GET /partner/{id}/execution/events — это упражнение мы оставляем читателю.

Внимательный читатель может возразить нам, что фактически, если мы посмотрим на номенклатуру возникающих сущностей, мы ничего не изменили в постановке задачи, и даже усложнили её:

- вместо вызова метода takeout мы теперь генерируем пару событий takeout_requested / takeout_ready;

- вместо длинного списка методов, которые необходимо реализовать для интеграции API партнёра, появляются длинные списки полей разных контекстов и событий, которые они генерирует;
- проблема устаревания технологии не меняется, вместо устаревших методов мы теперь имеем устаревшие поля и события.

Это замечание совершенно верно. Изменение формата API само по себе не решает проблем, связанных с эволюцией функциональности и нижележащей технологии. Формат API решает другую проблему: как оставить при этом партнерский код читаемым и поддерживаемым. Почему в примере с интеграцией через методы код становится нечитаемым? Потому что обе стороны *вынуждены* имплементировать функциональность, которая в их контексте бессмысленна. Код интеграции вендинговых автоматов *должен* ответить «принято» на запрос бесконтактной выдачи — и таким образом, со временем все имплементации будут состоять из множества методов, просто безусловно возвращающих `true` (или `false`).

Разница между жёстким связыванием и слабым в данном случае состоит в том, что механизм полей и событий *не является обязывающим*. Вспомним, чего мы добивались:

- верхнеуровневый контекст не знает, как устроен низкоуровневый API — и он действительно не знает; он описывает те изменения, которые происходят *в нём самом* и реагирует только на те события, которые имеют смысл *для него самого*;
- низкоуровневый контекст не знает ничего об альтернативных реализациях — он обрабатывает только те события, которые имеют смысл на его уровне, и оповещает только о тех событиях, которые могут происходить в его конкретной реализации.

В пределе может вообще оказаться так, что обе стороны вообще ничего не знают друг о друге и никак не взаимодействуют — не исключаем, что на каком-то этапе развития технологии именно так и произойдёт.

NB: в реальном мире этого может и не произойти — мы, вероятно, всё-таки хотим, чтобы приложение обладало знанием о том, был ли запрос на выдачу напитка успешно выполнен или нет, что означает подписку на событие `takeout_ready` и проверку соответствующего флага в состоянии контекста исполнения. Тем не менее, сама по себе *возможность не знать детали*

имплементации очень важна, поскольку она позволяет сделать код приложения гораздо проще — если это знание неважно для пользователя, конечно.

Ещё одним важным свойством слабой связности является то, что она позволяет сущности иметь несколько родительских контекстов. В обычных предметных областях такая ситуация выглядела бы ошибкой дизайна API, но в сложных системах, где присутствуют одновременно несколько агентов, влияющих на состояние системы, такая ситуация не является редкостью. В частности, вы почти наверняка столкнётесь с такого рода проблемами при разработке пользовательского UI. Более подробно о подобных двойных иерархиях мы расскажем в разделе «SDK и UI-библиотеки» настоящей книги.

Инверсия ответственности

Как несложно понять из вышесказанного, двусторонняя слабая связь означает существенное усложнение имплементации обоих уровней, что во многих ситуациях может оказаться излишним. Часто двустороннюю слабую связь можно без потери качества заменить на одностороннюю, а именно — разрешить нижележащей сущности вместо генерации событий напрямую вызывать методы из интерфейса более высокого уровня. Наш пример изменится примерно вот так:

```
/* Имплементация партнёром интерфейса
запуска программы на его кофемашинах */
registerProgramRunHandler(
  apiType, (program) => {
    // Инициализация исполнения заказа
    // на стороне партнера
    let execution = initExecution(...);
    // Подписка на изменения состояния
    // родительского контекста
    program.context.on(
      'takeout_requested', () => {
        // Если запрошена выдача заказа,
        // инициировать нужные операции
        await execution.prepareTakeout();
        /* Когда заказ готов к выдаче,
        сигнализируем об этом вызовом
        метода родительского контекста,
        а не генерацией события */
        // execution.context
        // .emit('takeout_ready')
        program.context.set('takeout_ready');
        // Или даже более строго
        // program.setTakeoutReady();
      }
    );
    // Так как мы модифицируем родительский
    // контекст вместо генерации событий,
    // нам не нужно что-либо возвращать
  }
);
```

Вновь такое решение выглядит континтуитивным, ведь мы снова вернулись к сильной связи двух уровней через жёстко определённые методы. Однако здесь есть важный момент: мы городим весь этот огород потому, что ожидаем появления альтернативных реализаций *нижележащего* уровня абстракции. Ситуации, когда появляются альтернативные реализации *вышележащего* уровня абстракции, конечно, возможны, но крайне редки. Обычно дерево альтернативных реализаций растёт от корня к листьям.

Другой аргумент в пользу такого подхода заключается в том, что, хотя серьёзные изменения концепции возможны на любом из уровней абстракции, их вес принципиально разный:

- если меняется технический уровень, это не должно существенно влиять на продукт, а значит — на написанный партнёрами код;
- если меняется сам продукт, ну например мы начинаем продавать билеты на самолёт вместо приготовления кофе на заказ, сохранять обратную совместимость на промежуточных уровнях API бесполезно. Мы вполне можем продавать билеты на самолёт тем же самым API программ и контекстов, да только написанный партнёрами код всё равно надо будет полностью переписывать с нуля.

В конечном итоге это приводит к тому, что API вышележащих сущностей меняется медленнее и более последовательно по сравнению с API нижележащих уровней, а значит подобного рода «обратная» жёсткая связь зачастую вполне допустима и даже желательна исходя из соотношения «цена-качество».

NB: во многих современных системах используется подход с общим разделяемым состоянием приложения. Пожалуй, самый популярный пример такой системы — Redux. В парадигме Redux вышеприведённый код выглядел бы так:

```
program.context.on(
  'takeout_requested',
  () => {
    await execution.prepareTakeout();
    // Вместо генерации событий
    // или вызова методов родительского
    // контекста, сущность `execution`
    // обращается к глобальному
    // или квази-глобальному методу
    // `dispatch`, который изменяет
    // глобальное состояние
    dispatch(takeoutReady());
  }
);
```

Надо отметить, что такой подход *в принципе* не противоречит описанным идеям снижения связности компонентов, но нарушает другой — изоляцию уровней абстракции, а поэтому плохо подходит для написания сложных API, в которых не гарантирована жёсткая иерархия компонентов. При этом использовать глобальный (или квази-глобальный) менеджер состояния в таких системах

вполне возможно, но требуется имплементировать более сложную пропагацию сообщений по иерархии, а именно: подчинённый объект всегда вызывает методы только ближайшего вышестоящего объекта, а уже тот решает, как и каким образом этот вызов передать выше по иерархии.

```
program.context.on(  
  'takeout_requested',  
  () => {  
    await execution.prepareTakeout();  
    // Вместо вызова глобального `dispatch`,  
    // сущность `execution` вызывает  
    // функциональность `dispatch`  
    // на своём родительском контексте  
    program.context.dispatch(takeoutReady());  
  }  
);
```

```
// Имплементация program.context.dispatch  
ProgramContext.dispatch = (action) => {  
  // program.context обращается к своему  
  // вышестоящему объекту, или к глобальному  
  // состоянию, если такого объекта нет  
  globalContext.dispatch(  
    // При этом сама суть действия  
    // может и должна быть переформулирована  
    // в терминах соответствующего уровня  
    // абстракции  
    this.generateAction(action)  
  )  
}
```

Делегируй!

Из описанных выше принципов следует ещё один чрезвычайно важный вывод: выполнение реальной работы, то есть реализация каких-то конкретных действий (приготовление кофе, в нашем случае) должна быть делегирована низшим уровням иерархии абстракций. Если верхние уровни абстракции попробуют предписать конкретные алгоритмы исполнения, то, как мы увидели в примере с `order_execution_endpoint`, мы быстро придём к ситуации противоречивой номенклатуры методов и протоколов взаимодействия, большая часть которых в рамках конкретного «железа» не имеет смысла.

Напротив, применяя парадигму конкретизации контекста на каждом новом уровне абстракции мы рано или поздно спустимся вниз по кроличьей норе достаточно глубоко, чтобы конкретизировать было уже нечего: контекст однозначно соотносится с функциональностью, доступной для программного управления. И вот на этом уровне мы должны отказаться от дальнейшей детализации и непосредственно реализовать нужные алгоритмы. Важно отметить, что глубина абстрагирования будет различной для различных нижележащих платформ.

NB. В рамках главы «[Разделение уровней абстракции](#)» мы именно этот принцип и проиллюстрировали: в рамках API кофемашин первого типа нет нужды продолжать растить дерево абстракций, можно ограничиться запуском программ; в рамках API второго типа требуется дополнительный промежуточный контекст в виде рантаймов.

Глава 31. Интерфейсы как универсальный паттерн

Попробуем кратко суммировать написанное в трёх предыдущих главах.

1. Расширение функциональности API производится через абстрагирование: необходимо так переосмыслить номенклатуру сущностей, чтобы существующие методы стали частным (желательно — самым частотным) упрощённым случаем реализации.
2. Вышестоящие сущности должны при этом оставаться информационными контекстами для нижестоящих, т.е. не предписывать конкретное поведение, а только сообщать о своём состоянии и предоставлять функциональность для его изменения (прямую через соответствующие методы либо косвенную через получение определённых событий).
3. Конкретная функциональность, т.е. работа непосредственно с «железом», нижележащим API платформы, должна быть делегирована сущностям самого низкого уровня.

NB. В этих правилах нет ничего особенно нового: в них легко опознаются принципы архитектуры SOLID¹² — что неудивительно, поскольку SOLID концентрируется на контрактно-ориентированном подходе к разработке, а API по определению и есть контракт. Мы лишь добавляем в эти принципы понятие уровней абстракции и информационных контекстов.

Остается, однако, неотвеченным вопрос о том, как изначально выстроить номенклатуру сущностей таким образом, чтобы расширение API не превращало её в мешанину из различных неконсистентных методов разных эпох. Впрочем, ответ на него довольно очевиден: чтобы при абстрагировании не возникало неловких ситуаций, подобно рассмотренному нами примеру с полями рецептов, все сущности необходимо *изначально* рассматривать как частную реализацию некоторого более общего интерфейса, даже если никаких альтернативных реализаций в настоящий момент не предвидится.

Например, разрабатывая API эндпойнта POST /search мы должны были задать себе вопрос: а «результат поиска» — это абстракция над каким интерфейсом? Для этого нам нужно аккуратно декомпозировать эту сущность, чтобы понять, каким своим срезом она выступает во взаимодействии с какими объектами.

Тогда мы придём к пониманию, что результат поиска — это, на самом деле, композиция двух интерфейсов:

- при создании заказа из всего результата поиска необходимы поля, описывающие собственно заказ; это может быть структура вида:

```
{coffee_machine_id, recipe_id, volume, currency_code, price},
```

либо мы можем закодировать все эти данные в одном `offer_id`;

- при отображении результата поиска в приложении нам важны другие поля — `name`, `description`, а также отформатированная и локализованная цена.

Таким образом, наш интерфейс (назовём его `ISearchResult`) — это композиция двух других интерфейсов: `IOrderParameters` (сущности, позволяющей сделать заказ) и `ISearchItemViewParameters` (некоторого абстрактного представления результатов поиска в UI). Подобное разделение должно автоматически подводить нас к ряду вопросов.

1. Каким образом мы будем связывать одно с другим? Очевидно, что эти два суб-интерфейса зависимы: например, отформатированная человекочитаемая цена должна совпадать с машиночитаемой. Это естественным образом подводит нас к концепции абстрагирования форматирования, описанной в главе «[Сильная связность и сопутствующие проблемы](#)».
2. А что такое, в свою очередь, «абстрактное представление результатов поиска в UI»? Есть ли у нас какие-то другие виды поисков, не является ли `ISearchItemViewParameters` сам наследником какого-либо другого интерфейса или композицией других интерфейсов?

Замена конкретных реализаций интерфейсами позволяет не только точнее ответить на многие вопросы, которые должны были у вас возникнуть в ходе проектирования API, но и наметить множество возможных векторов развития API, что поможет избежать проблем с неконсистентностью API в ходе дальнейшей эволюции программного продукта.

Примечания

¹ SOLID

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

² Martin, R. C. *Design Principles and Design Patterns*

http://staff.cs.utu.fi/~jounsm/does_o6/material/DesignPrinciplesAndPatterns.pdf

Глава 32. Блокнот душевного покоя

Помимо вышеперечисленных абстрактных принципов хотелось бы также привести набор вполне конкретных рекомендаций по внесению изменений в существующий API с поддержанием обратной совместимости.

1. Помните о подводной части айсберга

То, что вы не давали формальных гарантий и обязательств, совершенно не означает, что эти неформальные гарантии и обязательства можно нарушать. Зачастую даже исправление ошибок в API может привести к неработоспособности чьего-то кода. Можно привести следующий пример из реальной жизни, с которым столкнулся автор этой книги:

- существовал некоторый API размещения кнопок в визуальном контейнере; по контракту оно принимало позицию размещаемой кнопки (отступы от углов контейнера) в качестве обязательного параметра;
- в реализации была допущена ошибка: если позицию не передать, то исключения не происходило — добавленные таким образом кнопки размещались в левом верхнем углу контейнера одна за другой;
- в день, когда ошибка была исправлена, в техническую поддержку пришло множество обращений от разработчиков, чей код перестал работать; как оказалось, клиенты использовали эту ошибку для того, чтобы последовательно размещать кнопки в левом верхнем углу контейнера.

Если исправления ошибок затрагивают реальных потребителей — вам ничего не остаётся кроме как продолжать эмулировать ошибочное поведение до следующего мажорного релиза. При разработке больших API с широким кругом потребителей такие ситуации встречаются сплошь и рядом — например, разработчики API операционных систем буквально вынуждены портировать старые баги в новые версии ОС.

2. Тестируйте формальные интерфейсы

Любое программное обеспечение должно тестироваться, и API не исключение. Однако здесь есть свои тонкости: поскольку API предоставляет формальные интерфейсы, тестируться должны именно они. Это приводит к ошибкам нескольких видов.

1. Часто требования вида «функция `getEntity` возвращает значение, установленное вызовом функции `setEntity`» кажутся и разработчикам, и QA-инженерам самоочевидными и не проверяются. Между тем допустить ошибку в их реализации очень даже возможно, мы встречались с такими случаями на практике.
2. Принцип абстрагирования интерфейсов тоже необходимо проверять. В теории вы можете быть и рассматриваете каждую сущность как конкретную имплементацию абстрактного интерфейса — но на практике может оказаться, что вы чего-то не учли и ваш абстрактный интерфейс на деле невозможен. Для целей тестирования очень желательно иметь пустую условную, но отличную от базовой реализацию каждого интерфейса.

3. Изолируйте зависимости

В случае, если API является гейтвейем, предоставляющим доступ к какому-то нижележащему API или агрегирующему несколько различных API за одним фасадом, велик соблазн предоставить оригинальный интерфейс *as is*, не внося в него изменений и не усложняя себя жизнью разработкой слабо связанного взаимодействия. Например, разрабатывая интерфейс для запуска программ, описанный в главе «[Разделение уровней абстракции](#)», мы могли бы взять за основу интерфейс кофемашин первого типа и предоставить его в виде API, проксируя запросы и ответы как есть. Делать так ни в коем случае нельзя по нескольким причинам:

- как правило, у вас нет никаких гарантий, что партнёр будет поддерживать свой API в обратно-совместимом или хотя бы концептуально похожем виде;
- любые проблемы партнёра будут автоматически отражаться на ваших клиентах.

Напротив, хорошей практикой будет изолировать использование API третьей стороны, т.е. разработать программную обвязку, которая позволит:

- сохранять обратную совместимость за счёт правильно подобранных точек расширения;
- нивелировать проблемы партнёра техническими средствами:
 - ограничивать нагрузку на API партнёра в случае непредвиденного всплеска нагрузки на ваш API;
 - реализовывать политики перезапросов и иных способов восстановления после ошибок;
 - кэшировать какие-то критичные данные и состояния, чтобы иметь возможность предоставлять какую-то (хотя бы частичную) функциональность, даже если API партнёра недоступен полностью;
 - наконец, настроить автоматическое переключение на другого партнёра или альтернативное API.

4. Реализуйте функциональность своего API поверх публичных интерфейсов

Часто можно увидеть антипаттерн: разработчики API используют внутренние непубличные реализации тех или иных методов взамен существующих в их API публичных. Это происходит по двум причинам:

- часто публичный API является лишь дополнением к более специализированному внутреннему ПО компании, и наработки, представленные в публичном API, не портируются обратно в непубличную часть проекта, или же разработчики публичного API попросту не знают о существовании аналогичных непубличных функций;
- в ходе развития API некоторые интерфейсы абстрагируются, но имплементация уже существующих интерфейсов при этом по разным причинам не затрагивается; например, можно представить себе, что при реализации интерфейса `PUT /formatters`, описанного в главе «[Сильная связность и сопутствующие проблемы](#)», разработчики сделали отдельную, более общую, версию функции форматирования объёма для пользовательских языков в API, но не переписали существующую функцию форматирования для известных языков поверх неё.

Помимо очевидных частных проблем, вытекающих из такого подхода (неконсистентность поведения разных функций в API, не найденные при тестировании ошибки), здесь есть и одна глобальная: легко может оказаться, что вашим API попросту невозможно будет пользоваться, если сделать хоть один «шаг в сторону» — попытка воспользоваться любой нестандартной функциональностью может привести к проблемам производительности, многочисленным ошибкам, нестабильной работе и так далее.

NB. Идеальным примером строгого избегания данного антипаттерна следует признать разработку компиляторов — в этой сфере принято компилировать новую версию компилятора при помощи его же предыдущей версии.

5. Заведите блокнот

Несмотря на все приёмы и принципы, изложенные в настоящем разделе, с большой вероятностью вы *ничего* не сможете сделать с накапливающейся неконсистентностью вашего API. Да, можно замедлить скорость накопления, предусмотреть какие-то проблемы заранее, заложить запасы устойчивости — но предугадать *всё* решительно невозможно. На этом этапе многие разработчики склонны принимать скоропалительные решения — т.е. выпускать новые минорные версии API с явным или неявным нарушением обратной совместимости в целях исправления ошибок дизайна.

Так делать мы крайне не рекомендуем — поскольку, напомним, API является помимо прочего и мультиликатором ваших ошибок. Что мы рекомендуем — так это завести блокнот душевного покоя, где вы будете записывать выученные уроки, которые потом нужно будет не забыть применить на практике при выпуске новой мажорной версии API.

РАЗДЕЛ IV. HTTP API И АРХИТЕКТУРНЫЕ ПРИНЦИПЫ REST

Глава 33. О концепции HTTP API. Парадигмы разработки клиент-серверного взаимодействия

Вопросы организации HTTP API — к большому сожалению, одни из самых «холиварных». Будучи одной из самых популярных и притом весьма непростых в понимании технологий (ввиду большого по объёму и фрагментированного на отдельные RFC стандарта), спецификация HTTP обречена быть плохо понятой и превратно истолкованной миллионами разработчиков и многими тысячами учебных пособий. Поэтому, прежде чем переходить непосредственно к полезной части настоящего раздела, мы обязаны дать уточнения, о чём же всё-таки пойдёт речь.

Начнём с небольшой исторической справки. Выполнение запросов пользователя на удалённом сервере — одна из базовых задач в программировании ещё со времён мейнфреймов, естественным образом получившая новый импульс развития с появлением ARPANET. Хотя первые высокоуровневые протоколы сетевого взаимодействия работали в терминах отправки по сети сообщений (см., например, протокол DEL предложенный в одном из самых первых RFC — RFC-5 от 1969 года¹), довольно быстро теоретики пришли к мысли, что было бы гораздо удобнее, если бы обращение к удалённому узлу сети и использование удалённых ресурсов *с точки зрения сигнатуры вызова* ничем не отличались от обращения к локальной памяти и локальным ресурсам. Эту концепцию строго сформулировал под названием Remote Procedure Call (RPC) в 1981 году сотрудник знаменитой лаборатории Xerox в Пало-Альто Брюс Нельсон² и он же был соавтором первой практической имплементации предложенной парадигмы — Sun RPC³⁴, существующей и сегодня под названием ONC RPC.

Первые широко распространённые RPC-протоколы — такие как упомянутый Sun RPC, Java RMI⁵, CORBA⁶ — чётко следовали парадигме. Технология позволила сделать именно то, о чём писал Нельсон — не различать локальное и удалённое исполнение кода. Вся «магия» скрыта внутри обвязки, которая генерирует имплементацию работы с удалённым сервером, а с форматом передачи данных разработчик и вовсе не сталкивается.

Удобство использования технологии, однако, оказалось её же Ахиллесовой пятой:

- требование работы с удалёнными вызовами как с локальными приводит к высокой сложности протокола в силу необходимости поддерживать разнообразные возможности высокоуровневых языков программирования;
- RPC первого поколения диктуют выбор языка и платформы и для клиента, и для сервера:
 - Sun RPC не работал под Windows;
 - Java RMI требовал виртуальную машину Java для работы;
 - некоторые протоколы (CORBA, в частности) декларировали возможность разработки адаптеров для любого языка программирования, однако фактическая имплементация поддержки произвольных языков оказалась весьма сложной;
- проксирование запросов и шардирование данных осложнено необходимостью вычитывать и разбирать тело запроса, что может быть ресурсоёмко;
- возможность адресовать объекты в памяти удалённого сервера так же, как и локальные накладывает также огромные ограничения на масштабирование такой системы;
 - любопытно, что ни один заметный RPC-протокол управление разделяемой памятью не реализовал, но сама *возможность* в протоколы (в частности, Sun RPC) была заложена.

Одновременно с идеологическим кризисом RPC-подходов первого поколения, который стал особенно заметен с появлением массовых клиент-серверных приложений, где производительность намного важнее удобства разработчиков, происходит и другой процесс — стандартизации сетевых протоколов. Ещё в начале 90-х наблюдалось огромное разнообразие форматов взаимодействия, но постепенно сетевой стек в двух точках практических полностью унифицировался. Одна из них — это Internet protocol suite, состоящий из базового протокола IP и надстройки в виде TCP или UDP над ним. На сегодняшний день альтернативы TCP/IP используются в чрезвычайно ограниченном спектре задач, и средний разработчик практически не сталкивается ни с каким другим сетевым стеком.

Однако у TCP/IP с прикладной точки зрения есть существенный недостаток — он оперирует поверх системы IP-адресов, которые плохо подходят для организации распределённых систем:

- во-первых, люди не запоминают IP-адреса и предпочитают оперировать «говорящими» именами;
- во-вторых, IP-адрес является технической сущностью, связанной с узлом сети, а разработчики хотели бы иметь возможность добавлять и изменять узлы, не нарушая работы своих приложений.

Удобной (и опять же имеющей почти стопроцентное проникновение) абстракцией над IP-адресами оказалась система доменных имён, позволяющий назначить узлам сети человекочитаемые синонимы. Появление доменных имён потребовало разработки клиент-серверных протоколов более высокого, чем TCP/IP, уровня, и для передачи текстовых (гипертекстовых) данных таким протоколом стал HTTP 0.9⁷, разработанный Тимом Бёрнерсом-Ли опубликованный в 1991 году. Помимо поддержки обращения к узлам сети по именам, HTTP также предоставил ещё одну очень удобную абстракцию, а именно назначение собственных адресов эндпоинтам, работающим на одном сетевом узле.

Протокол был очень прост и всего лишь описывал способ получить документ, открыв TCP/IP соединение с сервером и передав строку вида GET адрес_документа. Позднее протокол был дополнен стандартом URL, позволяющим детализировать адрес документа, и далее протокол начал развиваться стремительно: появились новые глаголы помимо GET, статусы ответов, заголовки, типы данных и так далее.

HTTP появился изначально для передачи размеченного гипертекста, что для программных интерфейсов подходит слабо. Однако HTML со временем эволюционировал в более строгий и машиночитаемый XML, который быстро стал одним из общепринятых форматов описания вызовов API. (Забегая вперёд, скажем, что в начале 2000-х XML был быстро вытеснен ещё более простым и интероперабельным JSON.)

Поскольку, с одной стороны, HTTP был простым и понятным протоколом, позволяющим осуществлять произвольные запросы к удаленным серверам по их доменным именам, и, с другой стороны, быстро оброс почти бесконечным количеством разнообразных расширений над базовой функциональностью, он стал второй точкой, к которой сходятся сетевые технологии: практически все запросы к API внутри TCP/IP-сетей осуществляются по протоколу HTTP (и даже если используется альтернативный протокол, запросы в нём всё равно зачастую оформлены в виде HTTP-пакетов просто ради удобства). HTTP,

однако, идеологически совершенно противоположен RPC, поскольку не предполагает ни нативной обвязки для функций удалённого вызова, ни тем более разделяемого доступа к памяти. Зато HTTP предложил несколько очень удобных концепций для наращивания производительности серверов, такие как управление кэшированием из коробки и концепцию прозрачных прокси.

В итоге в середине 1990-х годов происходит постепенный отказ от RPC-фреймворков первого поколения в пользу нового подхода, который позднее Рой Филдинг в своей диссертации 2000 года обобщит под названием «Representational State Transfer» или «REST» (о чём мы поговорим чуть позже в соответствующей главе). В новой парадигме отношения между данными и операциями над ними переворачиваются с ног на голову:

- клиент не вызывает процедуры на сервере с передачей параметров — клиент указывает серверу абстрактный адрес (локатор) фрагмента данных (*ресурса*), к которому он хочет применить операцию;
 - сам список операций лимитирован и стандартизирован, семантика их чётко определена в стандарте;
- клиент и сервер независимы и *принципиально* не имеют никакого разделяемого состояния; все необходимые для исполнения операции параметры должны быть переданы явно;
 - между клиентом и сервером может находиться множество промежуточных узлов сети (прокси и гейтвеев), что не влияет на протокол;
 - если все важные параметры операции (в частности, идентификаторы ресурсов) включены в URL, данные легко можно шардировать без серьёзных затрат;
- сервер размечает параметры кэширования передаваемых данных (представления ресурсов), клиент (и промежуточные прокси) имеют право кэшировать данные в соответствии с разметкой.

NB: отказ от архитектур, где клиент и сервер жёстко связаны, в пользу ресурсоориентированных stateless подходов собственно породил понятие дизайна клиент-серверного API, поскольку потребовалось зафиксировать *контракт* между клиентом и сервером. В парадигме ранних RPC-фреймворков говорить о дизайне API бессмысленно, поскольку написанный разработчиком код и есть API взаимодействия, а с нижележащими протоколами разработчику и вовсе не было нужды знакомиться.

Хотя новый подход оказался весьма удачным с точки зрения разработки высокопроизводительных сервисов, проблемы неудобства работы с декларативными API из императивных языков никуда не делось. К тому же изначально простой стандарт быстро превратился в монстра Франкенштейна, сшитого из десятков разных подстандартов. Не думаем, что сильно ошибёмся, если скажем, что стандарт HTTP целиком со всеми многочисленными дополнительными RFC не знает полностью ни один человек.

Начиная с конца 2010-х годов мы наблюдаем расцвет RPC-технологий нового поколения — или, вернее было бы сказать, комбинированных технологий, которые одновременно и удобны в применении в императивных языках программирования (поставляются с обвязкой, позволяющей эффективно использовать кодогенерацию), и интероперабельны (работают поверх строго стандартизованных протоколов, которые не зависят от конкретного языка программирования), и масштабируемы (абстрагируют понятие ресурса и не предоставляют прямого доступа к памяти сервера).

Фактически, на сегодня *идеологически* разница между современным API, следующим классическому архитектурному стилю REST, и современным RPC заключается лишь в разметке кэшируемых данных и принципах адресации: в первом случае единицей доступа является ресурс (а параметры операции передаются дополнительно), а во втором — имя операции (а адреса ресурсов, над которыми она выполняется, передаются дополнительно).

Мы рассмотрим конкретные технологии разработки таких API в следующей главе, но отметим здесь важный момент: почти все они (за исключением разве что MQTT) работают поверх протокола HTTP. Так что большинство современных RPC-протоколов *одновременно* является HTTP API.

Тем не менее, *обычно* словосочетание «HTTP API» используется не просто в значении «любой API, использующий протокол HTTP»; говоря «HTTP API» мы *скорее* подразумеваем, что он используется не как дополнительный третий протокол транспортного уровня, а именно как протокол уровня приложения, то есть составляющие протокола (такие как: URL, заголовки, HTTP-глаголы, статусы ответа, политики кэширования и т.д.) используются в соответствии с их семантикой, определённой в стандартах. *Обычно* также подразумевается, что в HTTP API использует какой-то из текстовых форматов передачи данных (JSON, XML) для описания вызовов.

В рамках настоящего раздела мы поговорим о дизайне сетевых API, обладающих следующими характеристиками:

- протоколом взаимодействия является HTTP версий 1.1 и выше;
- форматом данных является JSON (за исключением эндпойнтов, специально предназначенных для передачи данных, как правило, файлов, в других форматах);
- в качестве идентификаторов ресурсов используется URL в соответствии со стандартом;
- семантика вызовов HTTP-эндпойнтов соответствует спецификации;
- никакие из веб-стандартов нигде не нарушаются специально.

Такое API мы будем для краткости называть просто «HTTP API» или «JSON-over-HTTP API». Мы понимаем, что такое использование терминологически не полностью корректно, но писать каждый раз «JSON-over-HTTP эндпойнты, утилизирующие семантику, описанную в стандартах HTTP и URL» или «JSON-over-HTTP API, соответствующий архитектурным ограничениям REST» не представляется возможным. Что касается термина «REST API», то как мы покажем дальше, у него нет консистентного определения, поэтому его использования мы также стараемся избегать.

Примечания

¹ RFC-5. DEL

<https://datatracker.ietf.org/doc/html/rfc5>

² Nelson, B. J. (1981) Remote procedure call

<https://www.semanticscholar.org/paper/Remote-procedure-call-Nelson/c860de40a88090055948b72d04dd79b02195eo6b>

³ Birrell, A. D., Nelson, B. J. (1984) Implementing remote procedure calls

<https://dl.acm.org/doi/10.1145/2080.357392>

⁴ RPC: Remote Procedure Call Protocol Specification

<https://datatracker.ietf.org/doc/html/rfc1050>

⁵ Remote Method Invocation (RMI)

<https://www.oracle.com/java/technologies/javase/remote-method-invocation-home.html>

⁶ CORBA

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

⁷ The Original HTTP as defined in 1991

<https://www.w3.org/Protocols/HTTP/AsImplemented.html>

Глава 34. Преимущества и недостатки HTTP API в сравнении с альтернативными технологиями

Как мы обсудили в предыдущей главе, в настоящий момент выбор технологии для разработки клиент-серверных API сводится к выбору либо ресурсоориентированного подхода (то, что принято называть «REST API», а мы, напомним, будем использовать термин «HTTP API»), либо одного из современных RPC-протоколов. Как мы отмечали, *концептуально* разница не очень значительна; однако *технически* разные фреймворки используют протокол HTTP совершенно по-разному:

Во-первых, разные фреймворки опираются на разные форматы передаваемых данных:

- HTTP API и некоторые RPC (JSON-RPC¹, GraphQL²) полагаются в основном на формат JSON³ (опционально дополненный передачей бинарных файлов);
- gRPC⁴, а также Thrift⁵, Avro⁶ и другие специализированные RPC-протоколы полагаются на бинарные форматы (такие как Protocol Buffers⁷, FlatBuffers⁸ и собственный формат Apache Avro);
- наконец, некоторые RPC-протоколы (SOAP⁹, XML-RPC¹⁰) используют для передачи данных формат XML¹¹ (что многими разработчиками сегодня воспринимается скорее как устаревшая практика).

Во-вторых, существующие реализации различаются подходом к утилизации протокола HTTP:

- либо клиент-серверное взаимодействие опирается на описанные в стандарте HTTP возможности — этот подход характеризует HTTP API;
- либо HTTP утилизируется как транспорт, и поверх него выстроен дополнительный уровень абстракции (т.е. возможности HTTP, такие как номенклатура ошибок или заголовков, сознательно редуцируются до минимального уровня, а вся мета-информация переносится на уровень вышестоящего протокола) — этот подход характерен для RPC протоколов.

У читателя может возникнуть резонный вопрос — а почему вообще существует такая дилемма: одни API полагаются на стандартную семантику HTTP, другие полностью от неё отказываются в пользу новоизобретённых стандартов, а третьи существуют где-то посередине. Например, если мы посмотрим на формат ответа в JSON-RPC¹², то мы обнаружим, что он легко мог бы быть заменён на стандартные средства протокола HTTP. Вместо

```
HTTP/1.1 200 OK

{
  "jsonrpc": "2.0",
  "id",
  "error": {
    "code": -32600,
    "message": "Invalid request"
  }
}
```

сервер мог бы ответить просто 400 Bad Request (с передачей идентификатора запроса, ну скажем, в заголовке X-JSONRPC2-RequestId). Тем не менее, разработчики протокола посчитали нужным разработать свой собственный формат.

Такая ситуация (не только конкретно с JSON-RPC, а почти со всеми высокоуровневыми протоколами поверх HTTP) сложилась по множеству причин, включая разнообразные исторические (например, невозможность использовать многие возможности HTTP из ранних реализаций XMLHttpRequest в браузерах). Однако, новые варианты RPC-протоколов, использующих абсолютный минимум возможностей HTTP, продолжают появляться и сегодня.

Мы можем попытаться выделить по крайней мере три группы причин (помимо идеологических, описанных в предыдущей главе), приводящих к такому размежеванию.

1. Машиночитаемость метаданных

Обратим внимание на принципиальное различие между использованием протоколов уровня приложения (как в нашем примере с JSON-RPC) и чистого HTTP. В примере выше ошибку 400 Bad Request может прочитать практически любой сетевой агент, если мы используем чистый HTTP; если же мы используем собственный формат JSON-RPC, ошибка прозрачной не является — во-первых, потому что понять её может только агент с поддержкой JSON-RPC, а, во-вторых, что более важно, в JSON-RPC статус запроса *не является метаинформацией*. Протокол HTTP позволяет прочитать такие детали, как метод и URL запроса, статус операции, заголовки запроса и ответа, *не читая тело запроса или ответа целиком*. Для большинства протоколов более высокого уровня, включая JSON-RPC, это не так: даже если агент и обладает поддержкой протокола, ему необходимо прочитать и разобрать тело ответа.

Каким образом эта самая возможность читать метаданные может быть полезна? Современный стек взаимодействия между клиентом и сервером является многослойным. Мы можем выделить множество агентов разного уровня, которые, так или иначе, обрабатывают сетевые запросы и ответы:

- разработчик пишет код поверх какого-то фреймворка, который отправляет запросы;
- фреймворк базируется на API языка программирования, компилятор или интерпретатор которого, в свою очередь, полагается на API операционной системы;
- запрос доходит до сервера, возможно, через промежуточные HTTP-прокси;
- сервер, в свою очередь, тоже представляет собой несколько слоёв абстракции в виде фреймворка, языка программирования и ОС;
- перед конечным обработчиком запроса, как правило, находится веб-сервер, проксирующий запрос, а зачастую и не один;
- в современных облачных архитектурах HTTP-запрос, прежде чем дойти до конечного обработчика, пройдёт через несколько абстракций в виде прокси и гейтвеев.

Главное преимущество, которое предоставляет следование букве стандарта HTTP — возможность положиться на то, что промежуточные агенты, от клиентских фреймворков до API-гейтвеев, умеют читать метаданные запроса и выполнять какие-то действия с их использованием — настраивать политику перезапросов и таймауты, логировать, кэшировать, шардировать, проксировать и так далее — без необходимости писать какой-то дополнительный код. Если попытаться сформулировать главный принцип разработки HTTP API, то мы получим примерно следующее: **лучше бы ты разрабатывал API так, чтобы промежуточные агенты могли читать и интерпретировать метаданные запроса и ответа.**

Главным недостатком HTTP API является то, что промежуточные агенты, от клиентских фреймворков до API-гейтвеев, умеют читать метаданные запроса и выполнять какие-то действия с их использованием — настраивать политику перезапросов и таймауты, логировать, кэшировать, шардировать, проксировать и так далее — даже если вы их об этом не просили. Более того, так как стандарты HTTP являются сложными, концепция REST — непонятной, а разработчики программного обеспечения — неидеальными, то промежуточные агенты (и разработчики партнёра!) могут трактовать метаданные запроса *неправильно*. Особенно это касается каких-то экзотических и сложных в имплементации стандартов. Как правило, одной из причин разработки новых RPC-фреймворков декларируется стремление обеспечить простоту и консистентность работы с протоколом, чтобы таким образом уменьшить поле для потенциальных ошибок в реализации интеграции с API.

2. Качество решений

Возможность читать и интерпретировать запросы приводит к широкой фрагментации доступных инструментов для работы с HTTP API. На рынке доступно (зачастую бесплатно) множество самых разных инструментов, таких как:

- различные прокси и API-гейтвеи (nginx, Envoy);
- различные форматы описания спецификаций (в первую очередь, OpenAPI) и связанные инструменты для работы со спецификациями (Redoc, Swagger UI) и кодогенерации;
- ПО для разработчиков, позволяющее удобным образом разрабатывать и отлаживать клиенты API (Postman, Insomnia);

- и так далее.

Конечно, большинство этих инструментов применимы и для работы с API, реализующими альтернативные парадигмы. Однако именно способность промежуточных агентов считывать метаданные HTTP запросов и одинаково их интерпретировать позволяет легко строить сложные конвейеры типа экспортировать access-логи nginx в Prometheus и из коробки получить удобные мониторинги статус-кодов ответов в Grafana.

Обратной стороной этой гибкости является качество самих решений и количество времени, необходимого на их интеграцию, особенно если ваш стек чем-то отличается от стандартного. В то же время основной импульс развития альтернативных технологий исходит от какой-то одной крупной IT-компании (Facebook, Google, Apache Software Foundation), которые предоставляют полный набор инструментов для работы со своим протоколом. Такой фреймворк может быть менее функциональным, но почти наверняка является более гомогенным и качественным в плане удобства разработки, поддержки пользователей и количества известных ошибок.

Соображения выше распространяются не только на программное обеспечение, но и на его создателей. Представление разработчиков о HTTP API, увы, также фрагментировано. Практически любой программист как-то умеет работать с HTTP API, но редко при этом досконально знает стандарт или хотя бы консультируется с ним при написании кода. Это ведёт к тому, что добиться качественной и консistentной реализации логики работы с HTTP API может быть сложнее, нежели при использовании альтернативных технологий — причём это соображение справедливо как для партнёров-интеграторов, так и для самого провайдера API.

3. Вопросы производительности

В пользу многих современных альтернатив HTTP API — таких как GraphQL, gRPC, Apache Thrift — часто приводят аргумент о низкой производительности JSON-over-HTTP API по сравнению с рассматриваемой технологией; конкретнее, называются следующие проблемы:

1. Избыточность формата:

- в JSON необходимо всякий раз передавать имена всех полей, даже если передаётся массив из большого количества одинаковых объектов;
 - большое количество технических символов — кавычек, скобок, запятых — и необходимость экранировать служебные символы в строках.
2. HTTP API в ответ на запрос к ресурсу возвращает представление ресурса целиком, хотя клиенту могут быть интересны только отдельные поля.
 3. Низкая производительность операций сериализации и десериализации данных.
 4. Передача бинарных данных требует дополнительного кодирования, например, через Base64.
 5. Низкая производительность самого протокола HTTP (в частности, невозможность мультиплексирования нескольких запросов и ответов по одному соединению).

Будем здесь честны: большинство существующих имплементаций HTTP API действительно страдают от указанных проблем. Тем не менее, мы берём на себя смелость заявить, что все эти проблемы большей частью надуманы, и их решению не уделяют большого внимания потому, что указанные накладные расходы не являются сколько-нибудь заметными для большинства вендоров API. В частности:

1. Если мы говорим об избыточности формата, то необходимо сделать важную оговорку: всё вышесказанное верно, если мы не применяем сжатие. Сравнения показывают¹³, что использование gzip практически нивелирует разницу в размере JSON документов относительно альтернативных бинарных форматов (а есть ещё и специально предназначенные для текстовых данных архиваторы, например, brotli¹⁴).
2. Вообще говоря, если такая нужда появляется, то и в рамках HTTP API вполне можно регулировать список возвращаемых полей ответа, это вполне соответствует духу и букве стандарта. Однако, мы должны заметить, что экономия трафика на возврате частичных состояний (которую мы рассматривали подробно в главе «Частичные обновления») очень редко бывает оправдана.

3. Если использовать стандартные десериализаторы JSON, разница по сравнению с бинарными форматами может оказаться действительно очень большой. Если, однако, эти накладные расходы являются проблемой, стоит обратиться к альтернативным десериализаторам — в частности, `simdjson`¹⁵. Благодаря оптимизированному низкоуровневому коду `simdjson` показывает отличную производительность, которой может не хватить только совсем уж экзотическим API.

- Комбинация `gzip/brotli + simdjson` во многом делает бессмысленным использование в клиент-серверной коммуникации оптимизированных вариантов JSON — таких как, например, `BSON`¹⁶.

4. Вообще говоря, парадигма HTTP API подразумевает, что для бинарных данных (такие как изображения или видеофайлы) предлагаются отдельные эндпоинты. Передача бинарных данных в теле JSON-ответа необходима только в случаях, когда отдельный запрос за ними представляет собой проблему с точки зрения производительности. Такой проблемы фактически не существует в `server-2-server` взаимодействии и в протоколе HTTP 2.0 и выше.

5. Протокол HTTP 1.1 действительно неидеален с точки зрения мультиплексирования запросов. Однако альтернативные парадигмы организации API для решения этой проблемы опираются... на HTTP версии 2.0. Разумеется, и HTTP API можно построить поверх версии 2.0.

На всякий случай ещё раз уточним: JSON-over-HTTP API *действительно* проигрывает с точки зрения производительности современным бинарным протоколам. Мы, однако, берём на себя смелость утверждать, что производительность хорошо спроектированного и оптимизированного HTTP API достаточна для абсолютного большинства предметных областей, и реальный выигрыш от перехода на альтернативные протоколы окажется незначителен.

Преимущества и недостатки формата JSON

Как нетрудно заметить, большинство претензий, предъявляемых к концепции HTTP API, относятся вовсе не к HTTP, а к использованию формата JSON. В самом деле, ничто не мешает разработать API, которое будет использовать любой бинарный формат вместо JSON (включая те же Protocol Buffers), и тогда разница между Protobuf-over-HTTP API и gRPC сведётся только к (не)использованию подробных URL, статус-кодов и заголовков запросов и ответов (и вытекающей отсюда (не)возможности использовать то или иное стандартное программное обеспечение «из коробки»).

Однако, во многих случаях (включая настоящую книгу) разработчики предпочитают текстовый JSON бинарным Protobuf (Flatbuffers, Thrift, Avro и т.д.) по очень простой причине: JSON очень легко и удобно читать. Во-первых, он текстовый и не требует дополнительной расшифровки; во-вторых, имена полей включены в сам файл. Если сообщение в формате protobuf невозможно прочитать без .proto-файла, то по JSON-документу почти всегда можно попытаться понять, что за данные в нём описаны. В совокупности с тем, что при разработке HTTP API мы также стараемся следовать стандартной семантике всей остальной обвязки, в итоге мы получаем API, запросы и ответы к которому (по крайней мере в теории) удобно читаются и интуитивно понятны.

Помимо человекочитаемости у JSON есть ещё одно важное преимущество: он максимально формален. В нём нет никаких конструкций, которые могут быть по-разному истолкованы в разных архитектурах (с точностью до ограничений на длины чисел и строк), и при этом он удобно ложится в нативные структуры данных (индексные и ассоциативные массивы) почти любого языка программирования. С этой точки зрения у нас фактически не было никакого другого выбора, какой ещё формат данных мы могли бы использовать при написании примеров кода для этой книги.

Выбор технологии разработки клиент-серверного API

Как мы видим из вышесказанного, HTTP API и альтернативные RPC-протоколы занимают разные ниши:

- для публичных API предоставление JSON-over-HTTP эндпоинтов является выбором по умолчанию, поскольку эта технология:

- понятна максимально широкому кругу программистов;
- позволяет разрабатывать клиентские приложения практически на любой платформе;
- для узкоспециализированных API логично использовать узкоспециализированные фреймворки (например, Apache Avro для работы с Apache Hadoop).

Практика предоставления публичных API в формате, скажем, gRPC, постепенно набирает популярность, но пока ещё незначительна на общем фоне. Таким образом, проблема выбора технологии возникает только для непубличных API общего назначения. На сегодня этот выбор выглядит так:

- HTTP («REST») API;
- gRPC;
- GraphQL;
- множество технологий поменьше, на которых мы не будем останавливаться подробно.

gRPC является классической технологией второго поколения, сочетающей все перечисленные выше достоинства:

- полагается на новейшие возможности протокола HTTP/2 и эффективный формат обмена данными Protobuf (последнее необязательно, но практически абсолютное большинство gRPC API полагается именно на него);
- разрабатывается компанией Google и поставляется с широким выбором разнообразных инструментов;
- предлагает contract-first подход, в котором разработка API начинается с написания спецификации;
- благодаря кодогенерации, позволяет удобно работать с протоколом на императивных языках программирования.

К недостаткам gRPC следует отнести:

- сложность декодирования сообщений и отладки коммуникации;
- слабую поддержку браузеров;
- меньшую распространённость (и отсюда более высокий порог входления для разработчиков);
- потенциальную зависимость от Google.

В остальном, gRPC вне всяких сомнений один из наиболее современных и производительных протоколов.

GraphQL представляет собой любопытный подход, который объединяет концепцию «ресурсов» в HTML (т.е. фокусируется на подробном описании форматов доступных данных и отношений между доменами), при этом предоставляя чрезвычайно богатый язык запросов для извлечения нужных наборов полей. Основная область его применения — это насыщенные разнородными данными предметные области (как, в общем-то, и следует из названия, GraphQL — скорее механизм распределённых запросов к абстрактному хранилищу данных, нежели парадигма разработки API). Предоставление *внешних* GraphQL API на сегодня скорее экзотика, поскольку с ростом количества данных и запросов GraphQL-сервисом становится очень сложно управлять¹⁷.

NB: теоретически, API может обладать двойным интерфейсом, например, и JSON-over-HTTP, и gRPC. Так как в основе всех современных фреймворков лежит формальное описание форматов данных и доступных операций, и эти форматы можно при желании транслировать друг в друга, такой мульти-API технически возможен. Практически же примеры таких API нам неизвестны — по-видимому, накладные расходы на поддержание двойного интерфейса не покрывают потенциальную выгоду от большего удобства для разработчиков.

Примечания

¹ JSON-RPC

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

² GraphQL

<https://graphql.org/>

³ JSON

<https://www.ecma-international.org/publications-and-standards/standards/ecma-404/>

⁴ gRPC

<https://grpc.io>

⁵ Apache Thrift

<https://thrift.apache.org/>

⁶ Apache Avro

<https://avro.apache.org/docs/>

⁷ Protocol Buffers

<https://protobuf.dev/>

⁸ FlatBuffers

<https://flatbuffers.dev/>

⁹ SOAP

<https://www.w3.org/TR/soap12/>

¹⁰ XML-RPC

<http://xmlrpc.com/>

¹¹ Extensible Markup Language (XML)

<https://www.w3.org/TR/xml/>

¹² JSON-RPC 2.0 Specification. Response object

https://www.jsonrpc.org/specification#response_object

¹³ Comparing sizes of protobuf vs json

<https://nilsmagnus.github.io/post/proto-json-sizes/>

¹⁴ Brotli Compressed Data Format

<https://datatracker.ietf.org/doc/html/rfc7932>

¹⁵ simdjson : Parsing gigabytes of JSON per second

<https://github.com/simdjson/simdjson>

¹⁶ BSON

<https://bsonspec.org/>

¹⁷ Mehta, S., Barodiya, K. Lessons learned from running GraphQL at scale

<https://blog.dreamengineering.com/lessons-learned-from-running-graphql-at-scale-2ad60b3cefeb>

Глава 35. Мифология REST

Прежде, чем перейти непосредственно к паттернам проектирования HTTP API, мы должны сделать ещё одно терминологическое отступление. Очень часто HTTP API, соответствующие данному нами в главе «[О концепции HTTP API](#)» определению, называют «REST API» или «RESTful API». В настоящем разделе мы эти термины не используем, поскольку оба этих термина неформальные и не несут конкретного смысла.

Что такое «REST»? Как мы упоминали ранее, в 2000 году один из авторов спецификаций HTTP и URI Рой Филдинг защитил докторскую диссертацию на тему «Архитектурные стили и дизайн архитектуры сетевого программного обеспечения», пятая глава которой была озаглавлена как «Representational State Transfer (REST)»¹.

Как нетрудно убедиться, прочитав эту главу, она представляет собой абстрактный обзор распределённой сетевой архитектуры, вообще не привязанной ни к HTTP, ни к URL. Более того, она вовсе не посвящена правилам дизайна API — в этой главе Филдинг методично *перечисляет ограничения*, с которыми приходится сталкиваться разработчику распределённого сетевого программного обеспечения. Вот они:

- клиент и сервер не знают внутреннего устройства друг друга (клиент-серверная архитектура);
- сессия хранится на клиенте (*stateless*-дизайн);
- данные должны размечаться как кэшируемые или некэшируемые;
- интерфейсы взаимодействия между компонентами должны быть стандартизированы;
- сетевые системы являются многослойными, т.е. сервер может быть только прокси к другим серверам;
- функциональность клиента может быть расширена через поставку кода с сервера.

На этом определение REST заканчивается. Далее Филдинг конкретизирует аспекты имплементации систем в указанных ограничениях, но все они точно так же являются совершенно абстрактными. Буквально: «ключевая информационная абстракция в REST — ресурс; любая информация, которой можно дать наименование, может быть ресурсом».

Ключевой вывод, который следует из определения REST по Филдингу-2000, вообще говоря, таков: *любое сетевое ПО в мире соответствует принципам REST*, за очень-очень редкими исключениями.

В самом деле:

- очень сложно представить себе систему, в которой не было бы хоть какой-нибудь стандартизации взаимодействия между компонентами, иначе её просто невозможно будет разрабатывать — в частности, как мы уже отмечали, почти всё сетевое взаимодействие в мире использует стек TCP/IP;
- раз есть интерфейс взаимодействия, значит, под него всегда можно мимикрировать, а значит, требование независимости имплементации клиента и сервера всегда выполнимо;
- раз можно сделать альтернативную имплементацию сервера — значит, можно сделать и многослойную архитектуру, поставив дополнительный прокси между клиентом и сервером;
- поскольку клиент представляет собой вычислительную машину, он всегда хранит хоть какое-то состояние и кэширует хоть какие-то данные;
- наконец, *code-on-demand* вообще лукавое требование, поскольку в архитектуре фон Неймана² всегда можно объявить данные, полученные по сети, «инструкциями» на некотором формальном языке, а код клиента — их интерпретатором.

Да, конечно, вышеприведённое рассуждение является софизмом, доведением до абсурда. Самое забавное в этом упражнении состоит в том, что мы можем довести его до абсурда и в другую сторону, объявив ограничения REST неисполнимыми. Например, очевидно, что требование *code-on-demand* противоречит требованию независимости клиента и сервера — клиент должен уметь интерпретировать код с сервера, написанный на вполне конкретном языке. Что касается правила на букву S («stateless»), то систем, в которых сервер *вообще не хранит никакого контекста клиента* в мире вообще практически нет, поскольку почти ничего полезного для клиента в такой системе сделать нельзя. (Чего, кстати, Филдинг прямым текстом требует: «коммуникация ... не может получать никаких преимуществ от того, что на сервере хранится какой-то контекст».)

Наконец, сам Филдинг внёс дополнительную энтропию в вопрос, выпустив в 2008 году разъяснение³, что же он имел в виду. В частности, в этой статье утверждается, что:

- разработка REST API должна фокусироваться на описании медиатипов, представляющих ресурсы; при этом клиент вообще ничего про эти медиатипы знать не должен;
- в REST API не должно быть фиксированных имён ресурсов и операций над ними, клиент должен извлекать эту информацию из ответов сервера.

REST по Филдингу-2008 подразумевает, что клиент, получив каким-то образом ссылку на точку входа в REST API, далее должен быть в состоянии полностью выстроить взаимодействие с API, не обладая вообще никаким априорным знанием о нём, и уж тем более не должен содержать никакого специально написанного кода для работы с этим API. Это требование — гораздо более сильное, нежели принципы, описанные в диссертации 2000 года. В частности, из идеи REST-2008 вытекает отсутствие фиксированных шаблонов URL для выполнения операций над ресурсами — предполагается, что такие URL присутствуют в виде гиперссылок в представлениях ресурсов (эта концепция известна также под названием HATEOAS⁴). Диссертация же 2000 года никаких строгих определений «гипермедиа», которые препятствовали бы идее конструирования ссылок на основе априорных знаний об API (например, по спецификации), не содержит.

NB: оставляя за скобками тот факт, что Филдинг весьма вольно истолковал свою собственную диссертацию, просто отметим, что ни одна существующая система в мире не удовлетворяет описанию REST по Филдингу-2008.

Нам неизвестно, почему из всех обзоров абстрактной сетевой архитектуры именно концепция Филдинга обрела столь широкую популярность; очевидно другое: теория Филдинга, преломившись в умах миллионов программистов (включая самого Филдинга), превратилась в целую инженерную субкультуру. Путём редукции абстракций REST применительно конкретно к протоколу HTTP и стандарту URL родилась химера «RESTful API», конкретного смысла которой никто не знает⁵.

Хотим ли мы тем самым сказать, что REST является бессмысленной концепцией? Отнюдь нет. Мы только хотели показать, что она допускает чересчур широкую интерпретацию, в чём одновременно кроется и её сила, и её слабость.

С одной стороны, благодаря многообразию интерпретаций, разработчики API выстроили какое-то размытое, но всё-таки полезное представление о «правильной» архитектуре HTTP API. С другой стороны, за отсутствием чётких определений тема REST API превратилась в один из самых больших источников холиваров среди программистов — притом холиваров совершенно бессмысленных, поскольку популярное представление о REST не имеет вообще никакого отношения ни к тому REST, который описан в диссертации Филдинга (и тем более к тому REST, который Филдинг описал в своём манифесте 2008 года).

Термин «архитектурный стиль REST» и производный от него «REST API» в последующих главах мы использовать не будем, поскольку, как видно из написанного выше, в этом нет никакой нужды — на все описанные Филдингом ограничения мы многократно ссылались по ходу предыдущих глав, поскольку, повторимся, распределённое сетевое API попросту невозможно разработать, не руководствуясь ими. Однако HTTP API (подразумевая под этим JSON-over-HTTP эндпойнты, утилизирующие семантику, описанную в стандартах HTTP и URL), как мы его будем определять в последующих главах, фактически соответствует усреднённому представлению о «REST/RESTful API», как его можно найти в многочисленных учебных пособиях.

Примечания

¹ Fielding, R. (2000) *Architectural Styles and the Design of Network-based Software Architectures*. Representational State Transfer (REST)

https://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm

² Von Neumann Architecture

https://en.wikipedia.org/wiki/Von_Neumann_architecture

³ Fielding, R. T. REST APIs must be hypertext-driven

<https://roy.gbiv.com/untangled/2008/rest-apis-must-be-hypertext-driven>

⁴ HATEOAS

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

⁵ Gupta, L. What is REST

<https://restfulapi.net/>

Глава 36. Составляющие HTTP запросов и их семантика

Важное подготовительное упражнение, которое мы должны сделать — это дать описание формата HTTP-запросов и ответов и прояснить базовые понятия. Многое из написанного ниже может показаться читателю самоочевидным, но, увы, специфика протокола такова, что даже базовые сведения о нём, без которых мы не сможем двигаться дальше, разбросаны по обширной и фрагментированной документации, и даже опытные разработчики могут не знать тех или иных нюансов. Ниже мы попытаемся дать структурированный обзор протокола в том объёме, который необходим нам для проектирования HTTP API.

В описании семантики и формата протокола мы будем руководствоваться свежевышедшим RFC 9110¹, который заменил аж девять предыдущих спецификаций, описывавших разные аспекты технологии (при этом большое количество различной дополнительной функциональности всё ещё покрывается отдельными стандартами. В частности, принципы HTTP-кэширования описаны в отдельном RFC 9111², а широко используемый в API метод PATCH так и не вошёл в основной RFC и регулируется RFC 5789³).

HTTP-запрос представляет собой (1) применение определённого глагола к URL с (2) указанием версии протокола, (3) передачей дополнительной метаинформации в заголовках и, возможно, (4) каких-то данных в теле запроса:

```
POST /v1/orders HTTP/1.1
Host: our-api-host.tld
Content-Type: application/json

{
  "coffee_machine_id": 123,
  "currency_code": "MNT",
  "price": "10.23",
  "recipe": "lungo",
  "offer_id": 321,
  "volume": "800ml"
}
```

Ответом на HTTP-запрос будет являться конструкция, состоящая из (1) версии протокола, (2) статус-кода ответа, (3) сообщения, (4) заголовков и, возможно, (5) тела ответа:

```
HTTP/1.1 201 Created
Location: /v1/orders/123
Content-Type: application/json

{
    "id": 123
}
```

NB: в HTTP/2 (и будущем HTTP/3) вместо единого текстового формата используются отдельные бинарные фреймы для передачи заголовков и данных⁴. Этот факт не влияет на излагаемые архитектурные принципы, но во избежание двусмысленности мы будем давать примеры в формате HTTP/1.1.

1. URL

URL — единица адресации в HTTP API (некоторые евангелисты технологии даже используют термин «пространство URL» как синоним для Мировой паутины). Предполагается, что HTTP API должен использовать систему адресов столь же гранулярную, как и предметная область; иными словами, у любых сущностей, которыми мы можем манипулировать независимо, должен быть свой URL.

Формат URL регулируется отдельным стандартом⁵, который развивает независимое сообщество Web Hypertext Application Technology Working Group (WHATWG). Считается, что концепция URL (вместе с понятием универсального имени ресурса, URN) составляет более общую сущность URI (универсальный идентификатор ресурса). (Разница между URL и URN заключается в том, что URL позволяет *найти* некоторый ресурс в рамках некоторого протокола доступа, в то время как URN — «внутреннее» имя объекта, которое само по себе никак не помогает получить к нему доступ.)

URL принято раскладывать на составляющие, каждая из которых опциональна. В стандарте перечислены разнообразные исторические наследия (например, передача логинов и паролей в URL или использование не-UTF кодировки), которые мы опустим. В рамках дизайна HTTP API нам интересны следующие компоненты:

- схема (scheme) — протокол обращения (в нашем случае всегда `https:`);
- хост (host; домен или IP-адрес) — самая крупная единица адресации;
 - домен может включать в себя поддомены;

- порт (port);
- путь (path) — часть URL между именем хоста (с портом) и символами ?, # или концом строки
 - путь принято разбивать по символу / и работать с каждой частью как отдельным токеном — но стандарт, вообще говоря, не предписывает никакой семантики такому разбиению;
 - пути с символом / и без символа / в конце (скажем /root/leaf и /root/leaf/) с точки зрения стандарта являются разными (и URL, отличающиеся только наличием/отсутствием слэша, считаются разными URL), хотя практически нам неизвестны аргументы в пользу того, чтобы не считать такие пути эквивалентными;
 - пути могут содержать секции . и/или .., которые предлагается трактовать по аналогии с такими же символами в путях на файловой системе (и, соответственно, считать URL /root/leaf, /root/./leaf, /root/branch/../leaf эквивалентными);
- запрос (query) — часть URL после знака ? до знака # или конца строки;
 - query принято раскладывать на пары ключ=значение, разделённые символом &; следует вновь иметь в виду, что стандарт не предписывает query строго соответствовать этому формату и не определяет никакой семантики;
 - также стандарт не предписывает никакой нормализации — два URL, которые различаются только порядком ключей в query, по стандарту являются разными URL;
- фрагмент (fragment; также якорь, anchor) — часть URL после знака #;
 - фрагмент традиционно рассматривается как адресация внутри запрошенного документа, поэтому многими агентами опускается при выполнении запроса;
 - два URL, отличающихся только значением фрагмента, могут считаться одинаковыми — а могут не считаться, зависит от контекста.

В HTTP-запросах, как правило (но не обязательно) схема, хост и порт опускаются (и считаются совпадающими с параметрами соединения). (Это соглашение, кстати, Филдинг считает самой большой проблемой дизайна протокола.)

Традиционно считается, что части пути описывают строгую иерархию подчинения ресурсов (например, URL конкретной кофе-машины в нашем API мог бы выглядеть как `/places/{id}/coffee-machines/{id}`, поскольку кофе-машина принадлежит строго одной кофейне), а через запрос выражаются нестрогие иерархии и параметры операций (например, URL поиска предложений мог бы выглядеть как `/search?location=<точка на карте>`).

Также стандарт содержит правила сериализации, нормализации и сравнения URL, которые в целом полезно знать разработчику HTTP API.

2. Заголовки

Заголовки — это *метаинформация*, привязанная к запросу или ответу. Она может описывать какие-то свойства передаваемых данных (например, `Content-Length`), дополнительные сведения о клиенте или сервере (`User-Agent`, `Date`), или просто содержать поля, не относящиеся непосредственно к смыслу запроса/ответа (например, `Authorization`).

Важное свойство заголовков — это возможность считывать их до того, как получено тело сообщения. Таким образом, заголовки могут, во-первых, сами по себе влиять на обработку запроса или ответа, и ими можно относительно легко манипулировать при проксировании — и многие сетевые агенты действительно это делают, добавляя или модифицируя заголовки по своему усмотрению (в частности, современные веб-браузеры добавляют к запросам целую коллекцию заголовков: `User-Agent`, `Origin`, `Accept-Language`, `Connection`, `Referer`, `Sec-Fetch-*` и так далее, а современное ПО веб-серверов, в свою очередь, автоматически добавляет или модифицирует такие заголовки как `X-Powered-By`, `Date`, `Content-Length`, `Content-Encoding`, `X-Forwarded-For`).

Подобное вольное обращение с заголовками создаёт определённые проблемы, если ваш API предусматривает передачу дополнительных полей метаданных, поскольку придуманные вами имена полей могут случайно совпасть с какими-то из существующих стандартных имен (или ещё хуже — в будущем появится новое стандартное поле, совпадающее с вашим). Долгое время во избежание подобных коллизий использовался префикс `X-`; уже более 10 лет как эта практика объявлена устаревшей и не рекомендуется к использованию (см. подробный разбор вопроса в RFC 6648⁶), однако отказа от этого префикса по факту не произошло (и многие широко распространённые нестандартные заголовки, например, `X-Forwarded-For`, его всё ещё содержат). Таким образом,

использование префикса X- вероятность коллизий снижает, но не устраняет. Тот же RFC вполне разумно предлагает использовать вместо X- префикс в виде имени компании. (Мы со своей стороны склонны рекомендовать использовать оба префикса в формате X-*ApiName*-*FieldName*; префикс X- для читабельности [чтобы отличать специальные заголовки от стандартных], а префикс с именем компании или API — чтобы не произошло коллизий с каким-нибудь другим нестандартным префиксом.)

Помимо прочего заголовки используются как управляющие конструкции — это т.н. «content negotiation», т.е. договорённость клиента и сервера о формате ответа (через заголовки *Accept**) и условные запросы, позволяющие сэкономить трафик на возврате ответа целиком или частично (через заголовки *If-**-заголовки, такие как *If-Range*, *If-Modified-Since* и так далее).

Стандарт предписывает как минимум один обязательный заголовок — *Host*. Ещё несколько заголовков необязательны, но на практике почти всегда используются:

- *Accept*, *Accept-Encoding*, *Content-Type* и *Content-Encoding* для описания форматов данных запроса и ответа;
- *Date* для синхронизации часов клиента и сервера;
- *Content-Length* для описания размера передаваемых данных (некоторые прокси вообще не пропускают запросы без *Content-Length*).

В дальнейших примерах мы эти заголовки будем опускать для лучшей читабельности.

3. HTTP-глаголы

Важнейшая составляющая HTTP запроса — это глагол (метод), описывающий операцию, применяемую к ресурсу. RFC 9110 стандартизирует восемь глаголов — GET, POST, PUT, DELETE, HEAD, CONNECT, OPTIONS и TRACE — из которых нас как разработчиков API интересует первые четыре. CONNECT, OPTIONS и TRACE — технические методы, которые очень редко используются в HTTP API (за исключением OPTIONS, который необходимо реализовать, если необходим доступ к API из браузера). Теоретически, HEAD (метод получения *только метаданных*, то есть заголовков, ресурса) мог бы быть весьма полезен в HTTP API, но по неизвестным нам причинам практически в этом смысле не используется.

Помимо RFC 9110, множество других RFC предлагают использовать дополнительные HTTP-глаголы (такие, например, как COPY, LOCK, SEARCH — полный список можно найти в реестре⁷), однако из всего разнообразия предложенных стандартов лишь один имеет широкое хождение — метод PATCH. Причины такого положения дел довольно тривиальны — этих пяти методов (GET, POST, PUT, DELETE, PATCH) достаточно для почти любого HTTP API.

HTTP-глагол определяет два важных свойства HTTP-вызова:

- его семантику (что представляет собой операция);
- его побочные действия, а именно:
 - является ли запрос модифицирующим (и можно ли кэшировать ответ);
 - является ли запрос идемпотентным.

Глагол	Семантика	Безопасный (немодифицирующий)	Идемпотентный	Может иметь тело
GET	Возвращает представление ресурса	да	да	не рекомендуется
PUT	Заменяет (полностью перезаписывает) ресурс согласно данным, переданным в теле запроса	нет	да	да
DELETE	Удаляет ресурс	нет	да	не рекомендуется
POST	Обрабатывает запрос в соответствии со своим внутренним устройством	нет	нет	да
PATCH	Модифицирует (частично)	нет	нет	да

Глагол	Семантика	Безопасный (немодифицирующий)	Идемпотентный	Может иметь тело
	перезаписывает ресурс согласно данным, переданным в теле запроса			

NB: распространено мнение, что метод POST предназначен только для создания новых ресурсов. Это совершенно не так, создание ресурса только один из вариантов «обработки запроса согласно внутреннему устройству» эндпойнта.

Важное свойство модифицирующих идемпотентных глаголов — это то, что **URL запроса является его ключом идемпотентности**. PUT /url полностью перезаписывает ресурс, заданный своим URL (/url), и, таким образом, повторный запрос не изменяет ресурс. Аналогично, повторный вызов DELETE /url должен оставить систему в том же состоянии (ресурс /url удалён). Учитывая, что метод GET /url семантически должен вернуть представление целевого ресурса /url, то, если этот метод реализован, он должен возвращать консистентное предыдущим PUT / DELETE представление. Если ресурс был перезаписан через PUT /url, GET /url должен вернуть представление, соответствующее переданном в PUT /url телу (в случае JSON-over-HTTP API это, как правило, просто означает, что GET /url возвращает в точности тот же контент, чтобы передан в PUT /url, с точностью до значений полей по умолчанию). DELETE /url обязан удалить указанный ресурс — так, что GET /url должен вернуть 404 или 410.

Идемпотентность и симметричность методов GET / PUT / DELETE влечёт за собой нежелательность для GET и DELETE запросов иметь тело (поскольку этому телу невозможно приписать никакой осмысленной роли). Однако (по-видимому в связи с тем, что многие разработчики попросту не знают семантику этих методов) распространённое ПО веб-серверов обычно разрешает этим методам иметь тело запроса и транслирует его дальше к коду обработки эндпойнта (использование этой практики мы решительно не рекомендуем).

Достаточно очевидным образом ответы на модифицирующие запросы не кэшируются (хотя при определённых условиях закэшированный ответ метода POST может быть использован при последующем GET-запросе) и, таким образом, повторный POST / PUT / DELETE / PATCH запрос обязательно будет доставлен до конечного сервера (ни один промежуточный агент не имеет права ответить из кэша). В случае GET-запроса это, вообще говоря, неверно — гарантией может служить только наличие в ответе директив кэширования `post-store` или `no-cache`.

Один из самых частых антипаттернов разработки HTTP API — это использование HTTP-глаголов в нарушение их семантики:

- Размещение модифицирующих операций за GET:
 - промежуточные агенты могут ответить на такой запрос из кэша, если какая-то из директив кэширования отсутствует, либо, напротив, повторить запрос при получении сетевого таймаута;
 - некоторые агенты считают себя вправе переходить по таким ссылкам без явного волеизъявления пользователя или разработчика; например, социальные сети и мессенджеры выполняют такие вызовы для генерации оформления ссылки, если пользователь пытается ей поделиться.
- Размещение неидемпотентных операций за идемпотентными методами PUT / DELETE. Хотя промежуточные агенты редко автоматически повторяют модифицирующие запросы, тем не менее это легко может сделать используемый разработчиком клиента или сервера фреймворк. Обычно эта ошибка сочетается с наличием у DELETE-запроса тела (чтобы всё-таки отличать, что конкретно нужно перезаписать или удалить), что является само по себе проблемой, так как любой сетевой агент вправе это тело проигнорировать.
- Несоблюдение требования симметричности операций GET / PUT / DELETE:
 - например, после выполнения DELETE /url операция GET /url продолжает возвращать какие-то данные или PUT /url ориентируется не на URL, а на данные внутри тела запроса для определения сущности, над которой выполняется операция, и, таким образом, GET /url никак не может вернуть представление объекта, только что переданного в PUT /url.

4. Статус-коды

Статус-код — это машиночитаемое описание результата HTTP-запроса в виде трёхзначного числа. Все статус-коды делятся на пять больших групп:

- 1xx — информационные (фактически, какое-то хождение имеет разве что 100 Continue);
- 2xx — коды успеха операции;
- 3xx — коды перенаправлений (индицируют необходимость выполнения дополнительных действий, чтобы считать операцию успешной);
- 4xx — клиентские ошибки;
- 5xx — серверные ошибки.

NB: разделение на группы по первой цифре кода имеет очень важное практическое значение. В случае, если возвращаемый сервером код ошибки xuz неизвестен клиенту, согласно спецификации клиент обязан выполнить то действие, которое выполнил бы при получении ошибки x00.

В основе технологии статус-кодов лежит понятное желание сделать ошибки машиночитаемыми, так, чтобы все промежуточные агенты могли понять, что конкретно произошло с запросом. Номенклатура статус-кодов HTTP действительно подробно описывает почти любые проблемы, которые могут случиться с HTTP-запросом: недопустимые значения Accept-*-заголовков, отсутствующий Content-Length, неподдерживаемый HTTP-метод, слишком длинный URI и так далее.

К сожалению, для описаний ошибок, возникающих в бизнес-логике, номенклатура статус-кодов HTTP совершенно недостаточна и вынуждает использовать статус-коды в нарушение стандарта и/или обогащать ответ дополнительной информацией об ошибке. Проблемы имплементации системы ошибок в HTTP API мы обсудим подробнее в главе «Работа с ошибками в HTTP API».

NB: обратите внимание на проблему дизайна спецификации. По умолчанию все 4xx коды не кэшируются, за исключением: 404, 405, 410, 414. Мы не сомневаемся, что это было сделано из благих намерений, но подозреваем, что множество людей, знающих об этих тонкостях, примерно совпадает с множеством редакторов спецификации HTTP.

Важное замечание о кэшировании

Кэширование — исключительно важная часть любой современной микросервисной архитектуры, и велик соблазн управлять им на уровне протокола — благо, стандарт предоставляет весьма функциональную и продвинутую функциональность работы с кэшами. Однако автор этой книги должен предостеречь читателя: если вы планируете имплементировать такую логику, прочтайте стандарт очень внимательно. Неверное толкование тех или иных параметров кэширования может приводить к крайне неприятным ситуациям; например, в практике автора был случай, когда случайное удаление настроек для определённой географической области (эндпоинт начал возвращать 404) привело к неработоспособности сервиса на протяжении нескольких часов, поскольку разработчики протокола не учли, что статус 404 по умолчанию кэшируется, и клиенты просто не запрашивают новую версию настроек, пока не истечёт время жизни кэша.

Важное замечание о консистентности

Один и тот же параметр в разных ситуациях может находиться в разных частях запроса. Скажем, идентификатор партнёра, совершающего запрос, может быть передан:

- в имени поддомена `{partner_id}.domain.tld`;
- как часть пути `/v1/{partner_id}/orders`;
- как query-параметр `/v1/orders?partner_id=<partner_id>`;
- как заголовок

```
GET /v1/orders HTTP/1.1
X-ApiName-Partner-Id: <partner_id>
```

- как поле в теле запроса

```
POST /v1/orders/retrieve HTTP/1.1
{
  "partner_id": <partner_id>
}
```

Возможны и более экзотические варианты: размещение параметра в схеме запроса или заголовке Content-Type.

Однако при перемещении параметра между различными составляющими запроса мы столкнёмся с тремя неприятными явлениями:

- некоторые значения чувствительны к регистру (путь, query-параметры, имена полей в JSON), некоторые нет (домен, имена заголовков);
 - при этом со *значениями* заголовков и вовсе неразбериха: часть из них по стандарту обязательно нечувствительна к регистру (в частности, Content-Type), а часть, напротив, обязательно чувствительна (например, ETag);
- наборы допустимых символов и правила экранирования также различны для разных частей запроса
 - для path, например, стандарта экранирования символов /, ? и # не существует;
 - символы unicode могут использоваться в доменных именах (хотя эта функциональность не везде поддерживается) только через своеобразную технику кодирования под названием Punycode⁸;
- для разных частей запросов используется разный кейсинг:
 - kebab-case для домена, заголовков и пути;
 - snake_case для query-параметров;
 - snake_case или camelCase для тела запроса;

При этом использование и snake_case, и camelCase в доменном имени невозможно, так как знак подчёркивания в доменных именах недопустим, а заглавные буквы будут приведены к строчным.

Чисто теоретически возможно использование kebab-case во всех случаях, но в большинстве языков программирования имена переменных и полей объектов в kebab-case недопустимы, что приведёт к неудобству работы с таким API.

Короче говоря, ситуация с кейсингом настолько плоха и запутана, что консистентного и удобного решения попросту нет. В этой книге мы придерживаемся следующего правила: токены даются в том кейсинге, который является общепринятым для той секции запроса, в которой находится токен; если положение токена меняется, то меняется и кейсинг. (Мы далеки от того, чтобы рекомендовать этот подход всюду; наша общая рекомендация, скорее — не умножать энтропию и пытаться минимизировать такого рода коллизии.)

NB: вообще говоря, JSON исходно — это JavaScript Object Notation, а в языке JavaScript кейсинг по умолчанию — camelCase. Мы, тем не менее, позволим себе утверждать, что JSON давно перестал быть форматом данных, привязанным к JavaScript, и в настоящее время используется для организации взаимодействия агентов, реализованных на любых языках программирования. Использование snake_case, по крайней мере, позволяет легко перебрасывать параметр из query в тело и обратно, что, обычно, является наиболее частотным кейсом при разработке HTTP API. Впрочем, обратный вариант (использование camelCase в именах query-параметров) тоже допустим.

Примечания

¹ RFC 9110 HTTP Semantics

<https://www.rfc-editor.org/rfc/rfc9110.html>

² RFC 9111 HTTP Caching

<https://www.rfc-editor.org/rfc/rfc9111.html>

³ PATCH Method for HTTP

<https://www.rfc-editor.org/rfc/rfc5789.html>

⁴ Grigorik, I. (2013) *High Performance Browser Networking*. Chapter 12. HTTP/2

<https://hpbn.co/http2/>

⁵ URL Living Standard

<https://url.spec.whatwg.org/>

⁶ Deprecating the "X-" Prefix and Similar Constructs in Application Protocols

<https://www.rfc-editor.org/rfc/rfc6648>

⁷ Hypertext Transfer Protocol (HTTP) Method Registry

<https://www.iana.org/assignments/http-methods/http-methods.xhtml>

⁸ Punycode: A Bootstrap encoding of Unicode for Internationalized Domain Names in Applications (IDNA)

<https://www.rfc-editor.org/rfc/rfc3492.txt>

Глава 37. Организация HTTP API согласно принципам REST

Перейдём теперь к конкретике: что конкретно означает «следовать семантике протокола» и «разрабатывать приложение в соответствии с архитектурным стилем REST». Напомним, речь идёт о следующих принципах:

- операции должны быть stateless;
- данные должны размечаться как кэшируемые или некэшируемые;
- интерфейсы взаимодействия между компонентами должны быть стандартизированы;
- сетевые системы многослойны.

Эти принципы мы должны применить к протоколу HTTP, соблюдая дух и букву стандарта:

- URL операции должен идентифицировать ресурс, к которому применяется действие, и быть ключом кэширования для GET и ключом идемпотентности — для PUT и DELETE;
- HTTP-глаголы должны использоваться в соответствии с их семантикой;
- свойства операции (безопасность, кэшируемость, идемпотентность, а также симметрия GET / PUT / DELETE-методов), заголовки запросов и ответов, статус-коды ответов должны соответствовать спецификации.

NB: мы намеренно опускаем многие тонкости стандарта:

- ключ кэширования фактически является составным [включает в себя заголовки запроса], если в ответе содержится заголовок `Vary`;
- ключ идемпотентности также может быть составным, если в запросе содержится заголовок `Range`;
- политика кэширования в отсутствие явных заголовков кэширования определяется не только глаголом, но и статус-кодом и другими заголовками запроса и ответа, а также политиками платформы;
 - в целях сохранения размеров глав в рамках разумного касаться этих вопросов мы не будем, но стандарт всё-таки рекомендуем внимательно прочитать.

Рассмотрим построение HTTP API на конкретном примере. Представим себе, например, процедуру старта приложения. Как правило, на старте требуется, используя сохранённый токен аутентификации, получить профиль текущего пользователя и важную информацию о нём (в нашем случае — текущие заказы). Мы можем достаточно очевидным образом предложить для этого эндпойнт:

```
GET /v1/state HTTP/1.1
Authorization: Bearer <token>
→
HTTP/1.1 200 OK
{
  "profile", "orders"
}
```

Получив такой запрос, сервер проверит валидность токена, получит идентификатор пользователя `user_id`, обратится к базе данных и вернёт профиль пользователя и список его заказов.

Подобный простой монолитный API-сервис нарушает сразу несколько архитектурных принципов REST:

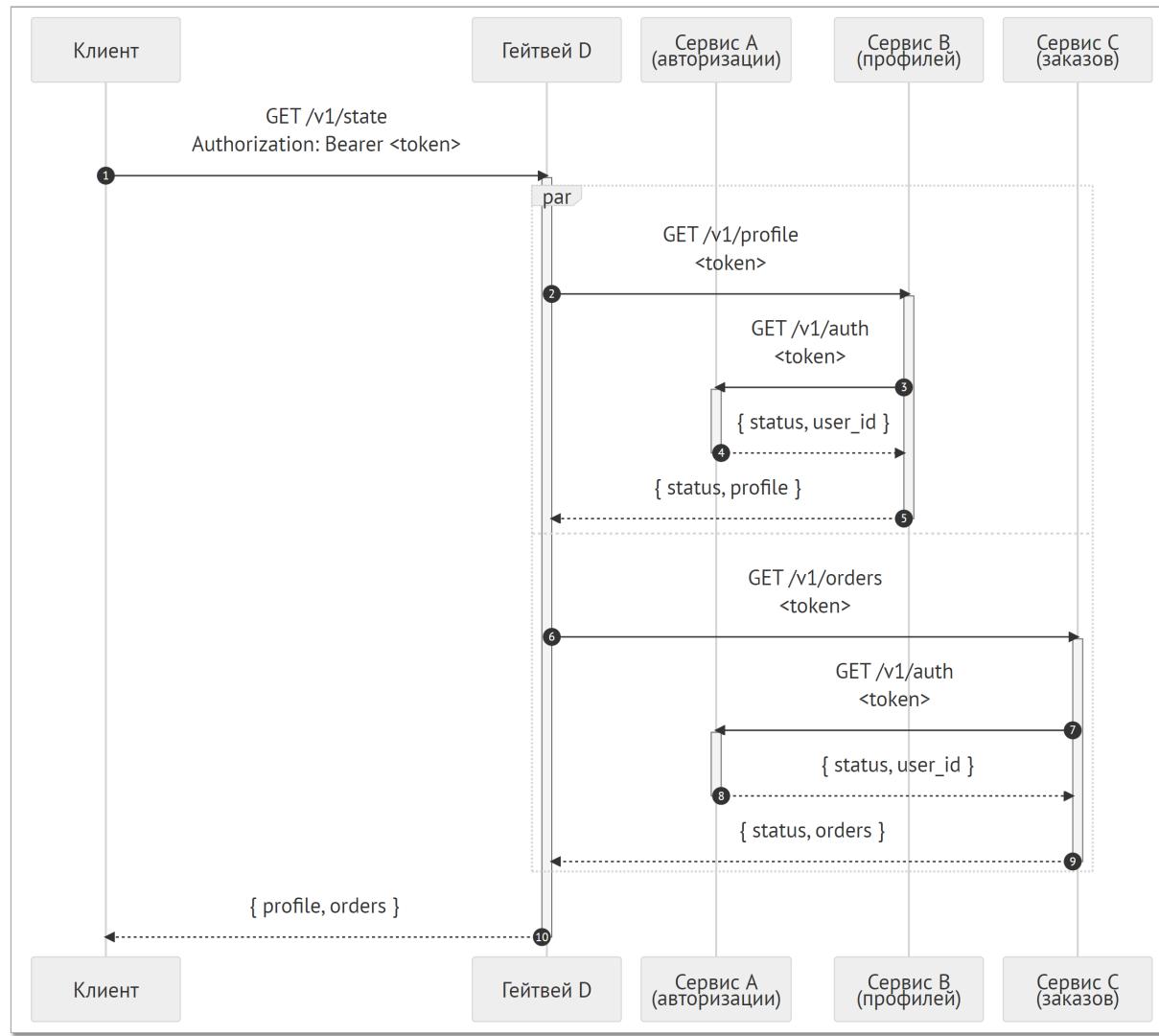
- нет очевидного способа кэшировать ответ на клиенте (данные о заказе часто меняются и их нет смысла сохранять);
- операция является `stateful`, т.к. сервер должен хранить токены в памяти, чтобы извлечь из них идентификатор клиента (к которому привязаны запрошенные данные);
- система однослойна (и таким образом вопрос об унифицированном интерфейсе бессмыслен).

Пока вопросы масштабирования бэкенда нас не волнуют, подобная схема прекрасно работает. Однако, с ростом количества пользователей и функциональности сервиса (а также количества программистов, над ним работающим), мы рано или поздно столкнёмся с тем, что подобная монолитная архитектура нам слишком дорого обходится. Допустим, мы приняли решение декомпозировать единый бэкенд на четыре микросервиса:

- сервис А, проверяющий авторизационные токены;
- сервис В, хранящий профили пользователей;
- сервис С, хранящий заказы пользователей;
- сервис-гейтвей D, который маршрутизирует запросы между другими микросервисами.

Таким образом, запрос будет проходить по следующему пути:

- гейтвей D получит запрос и отправит его в сервисы B и C;
- сервисы B и C обратятся к сервису A, проверят токен (переданный через проксирование заголовка `Authorization` или как явный параметр запроса), и вернут данные по запросу — профиль пользователя и список его заказов;
- сервис D скомбинирует ответы сервисов B и C и вернёт их клиенту.



Исходная схема организации микросервисов

Нетрудно заметить, что мы тем самым создаём излишнюю нагрузку на сервис A: теперь к нему обращается каждый из вложенных микросервисов; даже если мы откажемся от аутентификации пользователей в конечных сервисах, оставив её только в сервисе D, проблему это не решит, поскольку сервисы B и C самостоятельно выяснить идентификатор пользователя не могут. Очевидный

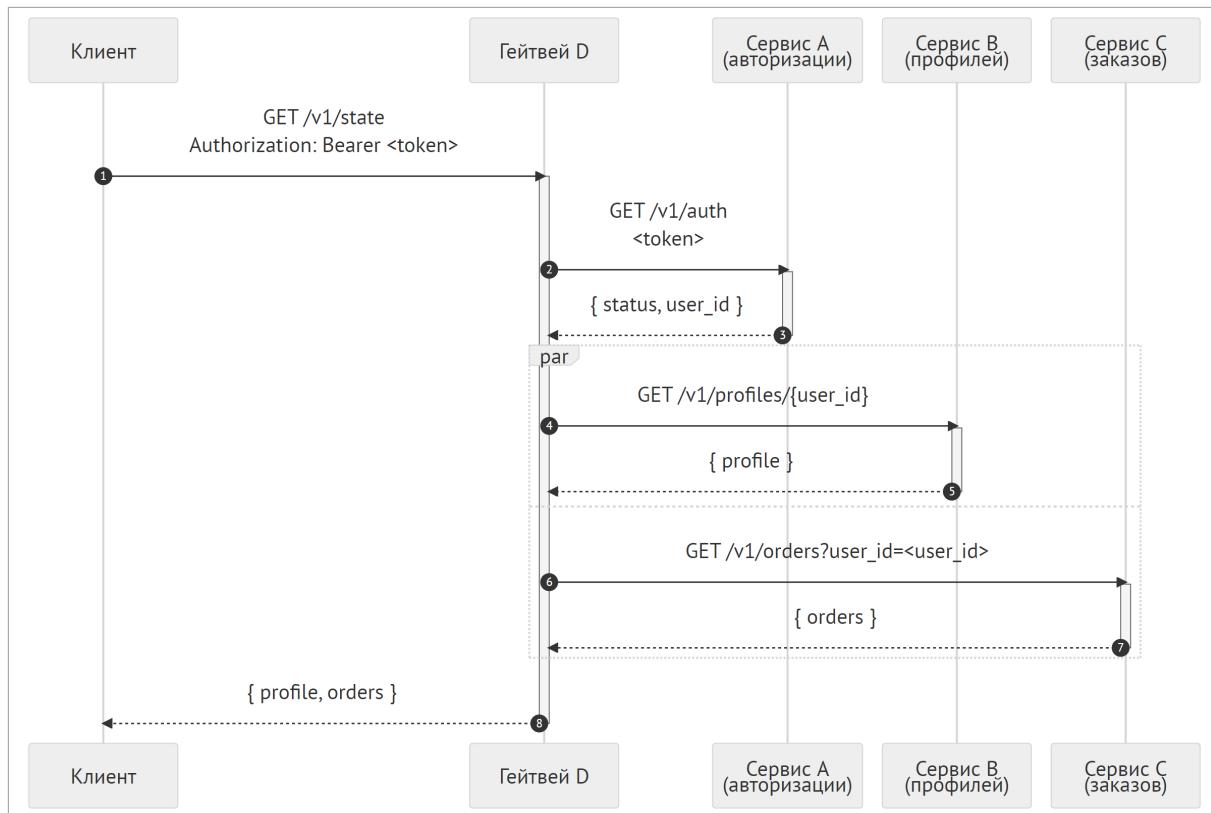
способ избавиться от лишних запросов — сделать так, чтобы однажды полученный user_id передавался остальным сервисам по цепочке:

- гейтвей D получает запрос и через сервис A меняет токен на user_id
- гейтвей D обращается к сервису B

`GET /v1/profiles/{user_id}`

и к сервису C

`GET /v1/orders?user_id=<user id>`



Шаг 1. Явные идентификаторы пользователей

NB: мы использовали нотацию `/v1/orders?user_id`, а не, допустим, `/v1/users/{user_id}/orders` по двум причинам:

- сервис текущих заказов хранит заказы, а не пользователей — логично если URL будет это отражать;

- если нам потребуется в будущем позволить нескольким пользователям делать общий заказ, нотация `/v1/orders?user_id` будет лучше отражать отношения между сущностями.

Более подробно о принципах формирования URL в HTTP API мы поговорим в следующей главе.

Теперь сервисы В и С получают запрос в таком виде, что им не требуется выполнение дополнительных действий (идентификации пользователя через сервис А) для получения результата. Тем самым мы переформулировали запрос так, что он *не требует от (микро)сервиса обращаться за данными за пределами его области ответственности*, добившись соответствия stateless-принципу.

Отметим, что вопрос о разнице между **stateless** и **stateful** подходами, вообще говоря, не имеет простого ответа. Микросервис В сам по себе хранит состояние клиента (профиль пользователя) и, таким образом, является stateful с точки зрения буквы диссертации Филдинга. Тем не менее, мы скорее интуитивно соглашаемся с тем, что хранить данные по профилю пользователя и только проверять валидность токена — это более правильный подход, чем хранить те же данные плюс кэш токенов, из которого можно извлечь идентификатор пользователя. Фактически, мы говорим здесь о *логическом* принципе разделения уровней абстракции, который мы подробно обсуждали в [соответствующей главе](#):

- **микросервисы разрабатываются так, чтобы иметь чётко очерченную зону ответственности и не хранить данные, относящиеся к другим уровням абстракции;**
- такие «внешние» данные являются лишь идентификаторами контекстов, и сам микросервис никак их не трактует;
- если всё же какие-то дополнительные операции с внешними данными требуется производить (например, проверять, авторизована ли запрашивающая сторона на выполнение операции), то следует **организовать операцию так, чтобы свести её к проверке целостности переданных данных**.

В нашем примере мы могли бы избавиться от лишних запросов к сервису А иначе — начав использовать stateless-токены, например, по стандарту JWT¹. Тогда сервисы В и С смогут сами раскодировать токен и извлечь идентификатор пользователя.

Пойдём теперь чуть дальше и подметим, что профиль пользователя меняется достаточно редко, и нет никакой нужды каждый раз получать его заново — мы могли бы организовать кэш профилей на стороне гейтвея D. Для этого нам нужно сформировать ключ кэша, которым фактически является идентификатор клиента. Мы можем пойти длинным путём:

- перед обращением в сервис В составить ключ и обратиться к кэшу;
- если данные имеются в кэше, ответить клиенту из кэша; иначе обратиться к сервису В и сохранить полученные данные в кэш.

А можем просто положиться на HTTP-кэширование, которое наверняка или реализовано в нашем фреймворке, или добавляется в качестве плагина за пять минут. Тогда гейтвей D обратится к ресурсу `/v1/profiles/{user_id}` в сервисе В, получит данные и заголовки с параметрами кэширования, и сохранит их локально.

Теперь рассмотрим сервис С. Результат его работы мы тоже могли бы кэшировать, однако состояние текущего заказа меняется гораздо чаще профиля пользователя, и возврат неверного состояния может приводить к крайне неприятным последствиям. Вспомним, однако, описанный нами в главе «[Стратегии синхронизации](#)» паттерн оптимистичного управления параллелизмом: для корректной работы сервиса нам нужна ревизия состояния ресурса, и ничто не мешает нам воспользоваться этой ревизией как ключом кэша. Пусть сервис С возвращает нам тэг, соответствующий текущему состоянию заказов пользователя:

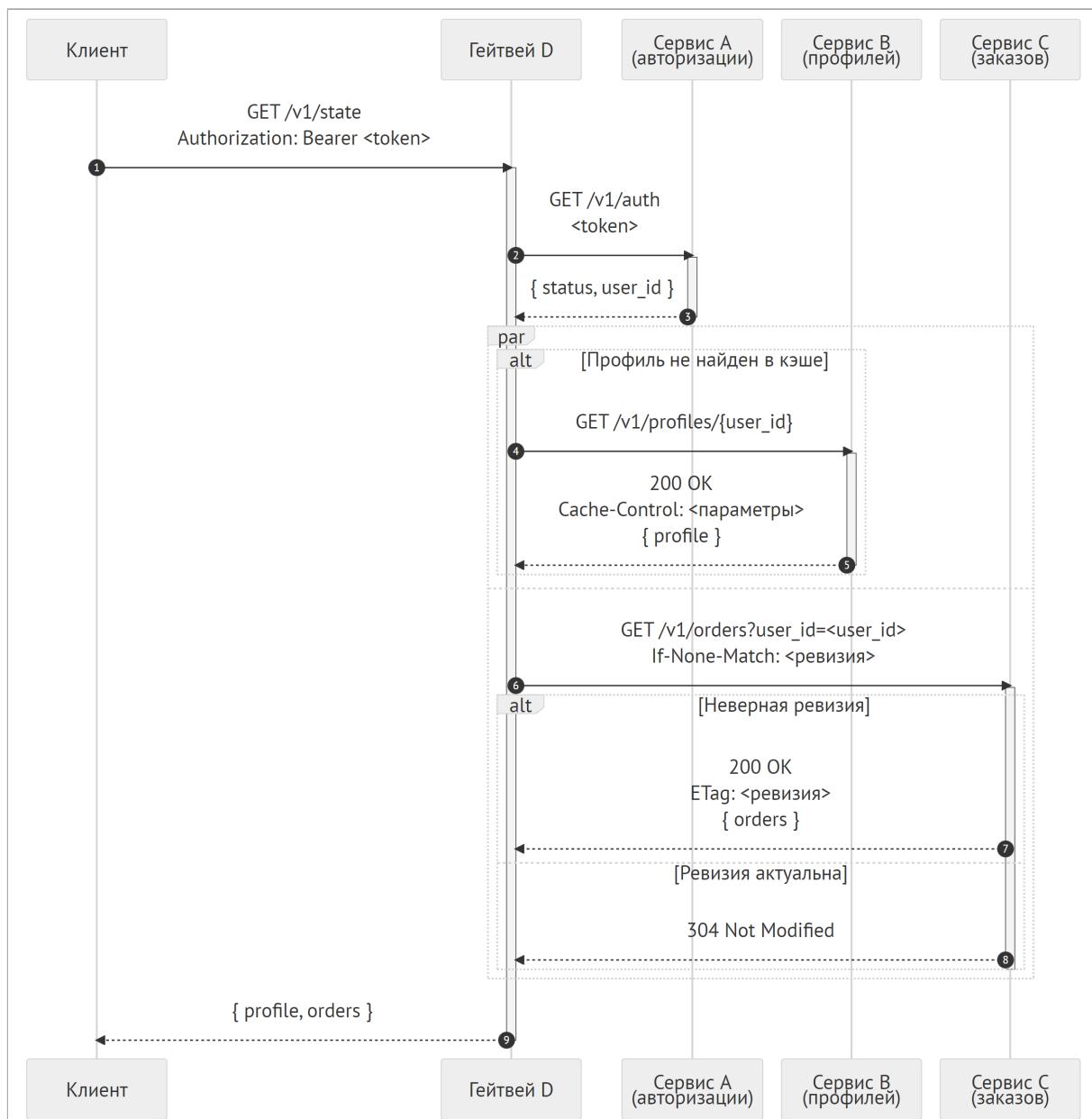
```
GET /v1/orders?user_id=<user_id> HTTP/1.1
→
HTTP/1.1 200 OK
ETag: <ревизия>
...
...
```

И тогда гейтвей D при выполнении запроса может:

1. Закэшировать результат выполнения GET `/v1/orders?user_id=<user_id>`, использовав URL как ключ кэша
2. При получении повторного запроса:
 - найти закэшированное состояние, если оно есть;
 - отправить запрос к сервису С вида

```
GET /v1/orders?user_id=<user_id> HTTP/1.1
If-None-Match: <ревизия>
```

- если сервис С отвечает статусом 304 Not Modified, вернуть данные из кэша;
- если сервис С отвечает новой версией данных, сохранить её в кэш и вернуть обновленный результат клиенту.



Шаг 2. Добавление серверного кэширования

Использовав такое решение [функциональность управления кэшом через ETag ресурсов], мы автоматически получаем ещё один приятный бонус: эти же данные пригодятся нам, если пользователь попытается создать новый заказ. Если мы используем оптимистичное управление параллелизмом, то клиент должен передать в запросе актуальную ревизию ресурса orders:

```
POST /v1/orders HTTP/1.1
If-Match: <ревизия>
```

Гейтвей D подставляет в запрос идентификатор пользователя и формирует запрос к сервису C:

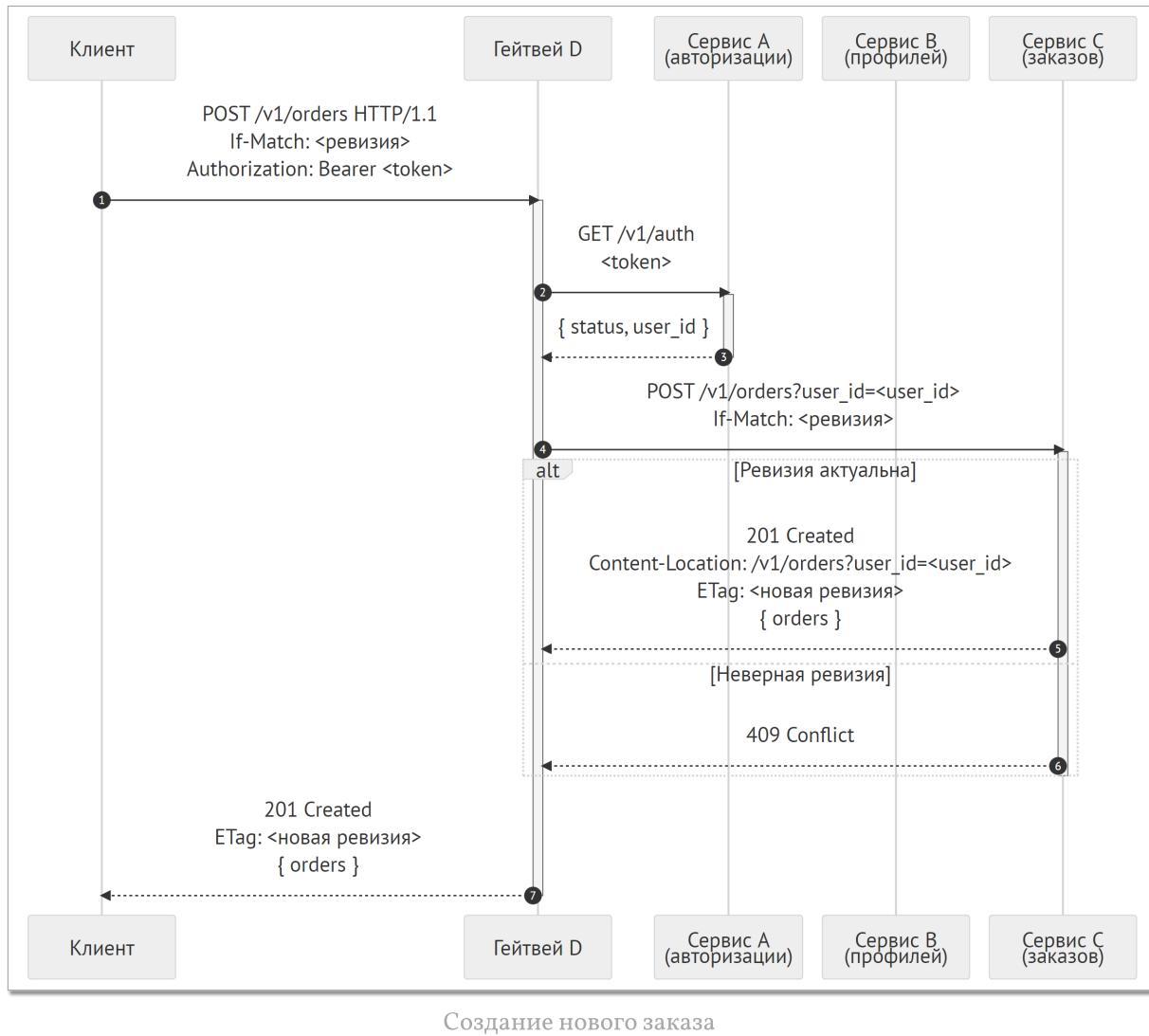
```
POST /v1/orders?user_id=<user_id> HTTP/1.1
If-Match: <ревизия>
```

Если ревизия правильная, гейтвей D может сразу же получить в ответе сервиса C обновлённый список заказов и его ревизию:

```
HTTP/1.1 201 Created
Content-Location: /v1/orders?user_id=<user_id>
ETag: <новая ревизия>

{ /* обновлённый список текущих заказов */ }
```

и обновить кэш в соответствии с новыми данными.



Важно: обратите внимание на то, что, после всех преобразований, мы получили систему, в которой мы можем *убрать гейтвей D и возложить его функции непосредственно на клиентский код*. В самом деле, ничто не мешает клиенту:

- хранить на своей стороне user_id (либо извлекать его из токена, если формат позволяет) и последний полученный ETag состояния списка заказов;
- вместо одного запроса GET /v1/state сделать два запроса (GET /v1/profiles/{user_id} и GET /v1/orders?user_id=<user_id>), благо протокол HTTP/2 поддерживает мультиплексирование запросов по одному соединению;
- поддерживать на своей стороне кэширование результатов обоих запросов с помощью стандартных библиотек и/или плагинов.

С точки зрения реализации сервисов В и С наличие или отсутствие гейтвея перед ними ни на что не влияет кроме механики авторизации запросов. Мы также можем добавить и второй гейтвей в цепочку, если, скажем, мы захотим разделить хранение заказов на «горячее» и «холодное» хранилища, или заставить какой-то из сервисов В или С работать в качестве гейтвея.

Если мы теперь обратимся к началу главы, мы обнаружим, что мы построили систему, полностью соответствующую требованиям REST:

- запросы к сервисам уже несут в себе все данные, которые необходимы для выполнения запроса;
- интерфейс взаимодействия настолько унифицирован, что мы можем передавать функции гейтвея клиенту или другому промежуточному агенту;
- политика кэширования каждого вида данных размечена.

Повторимся, что мы можем добиться того же самого, использовав RPC-протоколы или разработав свой формат описания статуса операции, параметров кэширования, версионирования ресурсов, приписывания и чтения метаданных и параметров операции. Но автор этой книги позволит себе, во-первых, высказать некоторые сомнения в качестве получившегося решения, и, во-вторых, отметить значительное количество кода, которое придётся написать для реализации всего вышеперечисленного.

Авторизация stateless-запросов

Рассмотрим подробнее подход, в котором авторизационного сервиса А фактически нет (точнее, он имплементируется как библиотека или локальный демон в составе сервисов В, С и D), и все необходимые данные зашифрованы в самом токене авторизации. Тогда каждый сервис должен выполнять следующие действия:

1. Получить запрос вида

```
GET /v1/profiles/{user_id}
Authorization: Bearer <token>
```

2. Расшифровать токен и получить вложенные данные, например, в следующем виде:

```
{  
    // Идентификатор пользователя-  
    // владельца токена  
    "user_id",  
    // Таймстемп создания токена  
    "iat"  
}
```

3. Проверить, что указанные в данных токена права доступа соответствуют параметрам операции — в данном случае сравнить user_id, переданный как query-параметр, и user_id, содержащийся в токене — и вынести решение о (не)допустимости операции.

Требование передавать user_id дважды и потом сравнивать две копии друг с другом может показаться нелогичным и избыточным. Однако это мнение ошибочно, и проистекает из широко распространённого (анти)паттерна, с описания которого мы начали главу, а именно — stateful-определение параметров операции:

```
GET /v1/profile  
Authorization: Bearer <token>
```

Такой эндпоинт фактически выполняет все три операции контроля доступа:

- *аутентифицирует* пользователя путём поиска токена в кэше токенов;
- *идентифицирует* пользователя путём извлечения связанного с токеном идентификатора;
- *авторизует* операцию, дополнив её параметры и *неявно* предполагая, что пользователь всегда имеет доступ к своим собственным данным.

Проблема с таким подходом заключается в том, что *разделить* эти операции не представляется возможным. Вспомним описанные нами в главе «[Аутентификация партнёров и авторизация вызовов API](#)» варианты авторизации вызовов API: в любой достаточно сложной системе нам придётся разрешать пользователю X выполнять действия от имени пользователя Y — например, если мы продаем функциональность заказа кофе как B2B API, и директор компании-партнёра желает лично или программно контролировать заказы, сделанные сотрудниками компании.

В случае «тройственного» эндпойнта проверки доступа мы можем только разработать новый эндпойнт с новым интерфейсом. В случае stateless-токенов мы можем поступить так:

1. Зашифровать в токене *список* пользователей, доступ к которым возможен через предъявление настоящего токена:

```
{  
    // Идентификаторы пользователей,  
    // доступ к профилям которых  
    // разрешён с настоящим токеном  
    "user_ids",  
    // Таймстемп создания токена  
    "iat"  
}
```

2. Изменить проверку авторизации (=внести изменения в код локального SDK или демона) так, чтобы она разрешала выполнение операции, если user_id в query-параметре содержится в списке user_ids токена.

Этот подход можно в дальнейшем усложнять: добавлять гранулярные разрешения выполнять конкретные операции, вводить уровни доступа, проверку прав в реальном времени через дополнительный вызов ACL-сервиса и так далее.

Важно, что кажущаяся избыточность перестала быть таковой: user_id в запросе теперь не дублируется в данных токена; эти идентификаторы имеют разный смысл: *над каким ресурсом* исполняется операция и *кто* исполняет операцию. Совпадение этих двух сущностей — пусть частотный, но всё же частный случай. Что, к сожалению, не отменяет его неочевидности и возможности легко забыть выполнить проверку в коде. Таков путь.

Примечания

¹ JSON Web Token (JWT)

<https://www.rfc-editor.org/rfc/rfc7519>

Глава 38. Разработка номенклатуры URL ресурсов. CRUD-операции

Как мы уже отмечали в предыдущих главах, стандарты HTTP и URL, а также принципы REST, не предписывают определённой семантики значимым компонентам URL (в частности, частям path и парам ключ-значение в query). **Правила организации URL в HTTP API существуют только для читабельности кода и удобства разработчика.** Что, впрочем, совершенно не означает, что они неважны: напротив, URL в HTTP API являются средством выразить уровни абстракции и области ответственности объектов. Правильный дизайн иерархии сущностей в API должен быть отражён в правильном дизайне номенклатуры URL.

NB: отсутствие строгих правил естественным образом привело к тому, что многие разработчики их просто придумали сами для себя. Некоторые наиболее распространённые стихийные практики, например, требование использовать в URL только существительные, в советах по разработке HTTP API в Интернете часто выдаются за стандарты или требования REST, которыми они не являются. Тем не менее, демонстративное игнорирование таких самопровозглашённых правил тоже не лучший подход для провайдера API, поскольку он увеличивает шансы быть неверно понятым.

Традиционно частям URL приписывается следующая семантика:

- части path (фрагменты пути между символами /) используются для организации вложенных сущностей вида /partner/{id}/coffee-machines/{id}; при этом путь часто может наращиваться, т.е. к конкретному пути продолжают приписываться новые суффиксы, указывающие на подчинённые ресурсы;
- query используется для организации нестрогой иерархии (отношений «многие ко многим», например /recipes/?partner=<partner_id>) либо как способ передать параметры операции (/search/?recipe=lungo).

Подобная конвенция достаточно хорошо подходит для того, чтобы отразить номенклатуру сущностей почти любого API, поэтому следовать ей вполне разумно (и, наоборот, демонстративное нарушение этого устоявшегося соглашения чревато тем, что разработчики вас просто неправильно поймут). Однако подобная некодифицированная и размытая концепция неизбежно вызывает множество разнотечений в конкретных моментах:

1. Где заканчиваются метаданные операции и начинаются «просто» данные и насколько допустимо дублировать поля и там, и там? Например, одна из распространённых практик в HTTP API — индицировать тип возвращаемых данных путём добавки «расширения» к URL, подобно именам файлов в файловых системах (т.е. при обращении к ресурсу `/v1/orders/{id}.xml` ответ будет получен в формате XML, а при обращении к `/v1/orders/{id}.json` — в формате JSON). С одной стороны, для определения формата данных предназначены `Accept*`-заголовки. С другой стороны, читабельность кода явно повышается с введением параметра «формат» прямо в URL.

2. Как следствие предыдущего пункта — каким образом в HTTP API правильно указывать версию самого API? Навскидку можно предложить как минимум три варианта, каждый из которых вполне соответствует букве стандарта:

- как path-параметр: `/v1/orders/{id}`;
- как query-параметр: `/orders/{id}?version=1`;
- как заголовок:

```
GET /orders/{id} HTTP/1.1
X-OurCoffeeAPI-Version: 1
```

Сюда можно приплусовать и более экзотические варианты, такие как указание схемы в кастомизированном медиатипе или протоколе запроса.

3. Каким образом организовывать эндпоинты, связывающие две сущности, между которыми нет явных отношений подчинения? Скажем, каким должен быть URL запуска приготовления лунго на конкретной кофемашине?

- `/coffee-machines/{id}/recipes/lungo/prepare`
- `/recipes/lungo/coffee-machines/{id}/prepare`
- `/coffee-machines/{id}/prepare?recipe=lungo`
- `/recipes/lungo/prepare?coffee_machine_id=<id>`
- `/prepare?coffee_machine_id=<id>&recipe=lungo`
- `/?action=prepare&coffee_machine_id=<id>&recipe=lungo`

Все эти варианты семантически вполне допустимы и в общем-то равноправны.

4. Насколько строго должна выдерживаться буквальная интерпретация конструкции ГЛАГОЛ /ресурс? Если мы принимаем правило «части URL обязаны быть существительными» (и ведь странно применять глагол к глаголу!), то в примерах выше должно быть не `prepare`, а `preparator` или `preparer` (а вариант `/action=prepare&coffee_machine_id=<id>&recipe=lungo` вовсе недопустим, так как нет объекта действия), что, честно говоря, лишь добавляет визуального шума в виде суффиксов «ator», но никак не способствует большей лаконичности и однозначности понимания.
5. Если сигнатура вызова по умолчанию модифицирующая или неидемпотентная, означает ли это, что операция *обязана* быть модифицирующей / идемпотентной? Двойственность смысловой нагрузки глаголов (семантика vs побочные действия) порождает неопределенность в вопросах организации API. Рассмотрим, например, ресурс `/v1/search`, осуществляющий поиск предложений кофе в нашем учебном API. С каким глаголом мы должны к нему обращаться?
 - С одной стороны, `GET /v1/search?query=<поисковый запрос>` позволяет явно декларировать, что никаких посторонних эффектов у этого запроса нет (никакие данные не перезаписываются) и результаты его можно кэшировать (при условии, что все значимые параметры передаются в URL).
 - С другой стороны, согласно семантике операции, `GET /v1/search` должен возвращать *представление ресурса search*. Но разве результаты поиска являются представлением ресурса-поисковика? Смысл операции «поиск» гораздо точнее описывается фразой «обработка запроса в соответствии с внутренней семантикой ресурса», т.е. соответствует методу `POST`. Кроме того, можем ли мы вообще говорить о кэшировании поисковых запросов? Страница результатов поиска формируется динамически из множества источников, и повторный запрос с той же поисковой фразой почти наверняка выдаст другой список результатов.

Иными словами, для любых операций, результат которых представляет собой результат работы какого-то алгоритма (например, список релевантных предложений по запросу) мы всегда будем сталкиваться с выбором, что важнее: семантика глагола или отсутствие побочных эффектов? Кэширование ответа или индикация того, что операция вычисляет результаты на лету?

NB: эта дилемма волнует не только нас, но и авторов стандарта, которые в конечном итоге предложили новый глагол `QUERY`¹, который по сути является немодифицирующим `POST`. Мы, однако, сомневаемся, что он получит широкое распространение — поскольку уже существующий `SEARCH`² оказался в этом качестве никому не нужен.

Простых ответов на вопросы выше у нас, к сожалению, нет. В рамках настоящей книги мы придерживаемся следующего подхода: сигнатура вызова в первую очередь должна быть лаконична и читабельна. Усложнение сигнатур в угоду абстрактным концепциям нежелательно. Применительно к указанным проблемам это означает, что:

1. Метаданные операции не должны менять смысл операции; если запрос доходит до конечного микросервиса вообще без заголовков, он всё ещё должен быть выполним, хотя какая-то вспомогательная функциональность может деградировать или отсутствовать.
2. Мы используем указание версии в path по одной простой причине: все остальные способы сделать это имеют смысл, если и только если при изменении мажорной версии протокола номенклатура URL останется прежней. Но, если номенклатура ресурсов может быть сохранена, то нет никакой нужды нарушать обратную совместимость.
3. Иерархия ресурсов выдерживается там, где она однозначна (т.е., если сущность низшего уровня абстракции однозначно подчинена сущности высшего уровня абстракции, то отношения между ними будут выражены в виде вложенных путей).
 - Если есть сомнения в том, что иерархия в ходе дальнейшего развития API останется неизменной, лучше завести новый верхнеуровневый префикс, а не вкладывать новые сущности в уже существующие.

4. Для выполнения «кросс-доменных» операций (т.е. при необходимости сослаться на объекты разных уровней абстракции в одном вызове) предпочтительнее завести специальный ресурс, выполняющий операцию (т.е. в примере с кофе-машинами и рецептами автор этой книги выбрал бы вариант `/prepare?coffee_machine_id=<id>&recipe=lungo`).
5. Семантика HTTP-вызова приоритетнее ложного предупреждения о небезопасности/неидемпотентности (в частности, если операция является безопасной, но ресурсозатратной, с нашей точки зрения вполне разумно использовать метод POST для индикации этого факта).

NB: отметим, что передача параметров в виде пути или query-параметра в URL влияет не только на читабельность. Вернёмся к примеру из предыдущей главы и представим, что гейтвей D реализован в виде stateless прокси с декларативной конфигурацией. Тогда получать от клиента запрос в виде:

- `GET /v1/state?user_id=<user_id>`
и преобразовывать в пару вложенных запросов
- `GET /v1/profiles?user_id=<user_id>`
- `GET /v1/orders?user_id=<user_id>`

гораздо удобнее, чем извлекать идентификатор из path и преобразовывать его в query-параметр. Первую операцию [замена одного path целиком на другой] достаточно просто описать декларативно, и в большинстве ПО для веб-серверов она поддерживается из коробки. Напротив, извлечение данных из разных компонентов и полная пересборка запроса — достаточно сложная функциональность, которая, скорее всего, потребует от гейтвея поддержки скриптового языка программирования и/или написания специального модуля для таких манипуляций. Аналогично, автоматическое построение мониторинговых панелей в популярных сервисах типа связки Prometheus+Grafana (да и в целом любой инструмент разбора логов) гораздо проще организовать по path, чем вычленять из данных запроса какой-то синтетический ключ группировки запросов.

Всё это приводит нас к соображению, что поддержание одинаковой структуры URL, в которой меняется только путь или домен, а параметры всегда находятся в query и именуются одинаково, приводит к ещё более унифицированному интерфейсу, хотя бы и в ущерб читабельности и семантичности URL. Во многих внутренних системах выбор в пользу удобства выглядит самоочевидным, хотя во внешних API мы бы такой подход не рекомендовали.

CRUD-операции

Одно из самых популярных приложений HTTP API — это реализация CRUD-интерфейсов. Акроним CRUD (Create, Read, Update, Delete) был популяризирован ещё в 1983 году Джеймсом Мартином, но с развитием HTTP API обрёл второе дыхание. Ключевая идея соответствия CRUD и HTTP заключается в том, что каждой из CRUD-операций соответствует один из глаголов HTTP:

- операции создания — создание ресурса через метод POST;
- операции чтения — возврат представления ресурса через метод GET;
- операции редактирования — перезапись ресурса через метод PUT или редактирование через PATCH;
- операции удаления — удаление ресурса через метод DELETE.

NB: фактически, подобное соответствие — это просто мнемоническое правило, позволяющее определить, какой глагол следует использовать к какой операции. Мы, однако, должны предостеречь читателя: глагол следует выбирать по его семантике согласно стандарту, а не по мнемоническим правилам. Может показаться, что, например, операцию удаления 3-го элемента списка нужно реализовать через DELETE:

- `DELETE /v1/list/{list_id}/?position=3`

но, как мы помним, делать так категорически нельзя: во-первых, такой вызов неидемпотентен; во-вторых, нарушает требование консистентности GET и DELETE.

С точки зрения удобства разработки концепция соответствия CRUD и HTTP выглядит очень удобной — каждому виду ресурсов соответствует свой URL, каждой операции — свой глагол. При пристальном рассмотрении, однако, оказывается, что это отношение — очень упрощённое представление о манипуляции ресурсами, и, что самое неприятное, плохо расширяемое.

1. Создание

Начнём с операции создания ресурса. Как мы помним из главы «[Стратегии синхронизации](#)», операция создания в любой сколько-нибудь ответственной предметной области обязана быть идемпотентной и, очень желательно, ещё и позволять управлять параллелизмом. В рамках парадигмы HTTP API идемпотентное создание можно организовать одним из трёх способов:

1. Через метод POST с передачей токена идемпотентности (им может выступать, в частности, ETag ресурса):

```
POST /v1/orders/?user_id=<user_id> HTTP/1.1
If-Match: <ревизия>
{ ... }
```

2. Через метод PUT, предполагая, что идентификатор заказа сгенерирован клиентом (ревизия при этом всё ещё может использоваться для управления параллелизмом, но токеном идемпотентности является сам URL):

```
PUT /v1/orders/{order_id} HTTP/1.1
If-Match: <ревизия>
{ ... }
```

3. Через схему создания черновика методом POST и его подтверждения методом PUT:

```
POST /v1/drafts HTTP/1.1
{
  ...
}
HTTP/1.1 201 Created
Location: /v1/drafts/{id}
```

```
PUT /v1/drafts/{id}/commit
If-Match: <ревизия>
{
  "status": "confirmed"
}
HTTP/1.1 200 OK
Location: /v1/orders/{id}
```

Метод (2) в современных системах используется редко, так как вынуждает доверять правильности генерации идентификатора заказа клиентом. Если же рассматривать варианты (1) и (3), то необходимо отметить, что семантике протокола вариант (3) соответствует лучше, так как POST-запросы по умолчанию считаются неидемпотентными, и их автоматический повтор в случае получения сетевого таймаута или ошибки сервера будет выглядеть для постороннего наблюдателя опасной операцией (которой запрос и правда может стать, если сервер изменит политику проверки заголовка If-Match на более мягкую). Повтор PUT-запроса (а мы предполагаем, что таймауты и серверные ошибки на «тяжёлой» операции создания заказа намного более вероятны, чем на «лёгкой» операции создания черновика) вполне может быть автоматизирован, и не будет создавать дубликаты заказа, даже если проверка ревизии будет отключена вообще.

2. Чтение

Идём дальше. Операция чтения на первый взгляд не вызывает сомнений:

- GET /v1/orders/{id}.

Стоит, однако, присмотреться внимательнее, и всё оказывается не так просто. Клиент как минимум должен обладать способом выяснить, какие заказы сейчас выполняются от его имени, что требует создания отдельного ресурса-поисковика:

- GET /v1/orders/?user_id=<user_id>.

Передача списков без ограничений по их длине — потенциально плохая идея, а значит необходимо ввести поддержку пагинации:

- GET /v1/orders/?user_id=<user_id>&cursor=<cursor>.

Если заказов много, наверняка пользователю понадобятся фильтры, скажем, по названию напитка:

- GET /v1/orders/?user_id=<user_id>&recipe=lungo.

Однако, если пользователь захочет видеть в одном списке и латте и лунго, этот интерфейс уже окажется ограниченно применимым, поскольку общепринятого стандарта передачи в URL более сложных структур, чем пары ключ-значение, не существует. Довольно скоро мы придём к тому, что, наряду с доступом по идентификатору заказа потребуется ещё и поисковый эндпойнт со сложной семантикой (которую гораздо удобнее было бы разместить за POST):

- POST /v1/orders/search { /* parameters */ }

Кроме того, если к заказу можно прикладывать какие-то медиа-данные (скажем, фотографии), то для доступа к ним придётся разработать отдельные URL:

- GET /v1/orders/{order_id}/attachments/{id}

3. Редактирование

Проблемы частичного обновления ресурсов мы подробно разбирали в [соответствующей главе](#) раздела «Паттерны дизайна API». Напомним, что полная перезапись ресурса методом PUT возможна, но быстро разбивается о необходимость работать с вычисляемыми и неизменяемыми полями, необходимость совместного редактирования и/или большой объём передаваемых данных. Работа через метод PATCH возможна, но, так как этот метод по умолчанию считается неидемпотентным (и часто нетранзитивным), для него справедливо всё то же соображение об опасности автоматических перезапросов. Достаточно быстро мы придём к одному из двух вариантов:

- либо PUT декомпозирован на множество составных PUT /v1/orders/{id}/address, PUT /v1/orders/{id}/volume и т.д. — по ресурсу для каждой частной операции;

- либо существует отдельный ресурс, принимающий список изменений, причём, вероятнее всего, через схему черновик-подтверждение в виде пары методов POST + PUT.

Если к сущности прилагаются медиаданные, для их редактирования также придётся разработать отдельные эндпойнты.

4. Удаление

С удалением ситуация проще всего: никакие данные в современных сервисах не удаляются моментально, а лишь архивируются или помечаются удалёнными. Таким образом, вместо DELETE /v1/orders/{id} необходимо разработать эндпойнт типа PUT /v1/orders/{id}/archive или PUT /v1/archive?order=<order_id>.

CRUD-операции в реальной жизни

Изложенные выше соображения не следует считать критикой концепции CRUD как таковой. Мы лишь указываем, что что в сложных предметных областях «срезание углов» и следование мнемоническим правилам редко работает. Мы начали с двух URL и четырёх-пяти методов манипуляции ими:

- /v1/orders/ для вызова с глаголом POST;
- /v1/orders/{id} для вызова с глаголами GET / PUT / DELETE / опционально PATCH.

Однако, если мы выдвинем требования наличия возможностей:

- контроля параллелизма при создании сущностей;
- совместного редактирования;
- архивирования;
- поиска с фильтрацией;

то мы быстро придём к номенклатуре из 8 URL и 9-10 методов:

- GET /v1/orders/?user_id=<user_id> для получения списка текущих заказов, возможно с какими-то простыми фильтрами;

- `/v1/orders/drafts/?user_id=<user_id>` для создания черновика заказа через метод POST и получения списка текущих черновиков и актуальной ревизии через метод GET;
- `PUT /v1/orders/drafts/{id}/commit` для подтверждения черновиков и создания заказов;
- `GET /v1/orders/{id}` для получения вновь созданного заказа;
- `POST /v1/orders/{id}/drafts` для создания черновика внесения списка изменений;
- `PUT /v1/orders/{id}/drafts/{id}/commit` для подтверждения черновика со списком изменений;
- `/v1/orders/search?user_id=<user_id>` для поиска заказов через метод GET (простые случаи) или POST (если необходимы сложные фильтры);
- `PUT /v1/orders/{id}/archive` для архивирования заказа.

плюс, вероятно, потребуются частные операции типа POST `/v1/orders/{id}/cancel` для проведения атомарных изменений. Именно это произойдёт в реальной жизни: идея CRUD как методологии описания типичных операций над ресурсом с помощью небольшого набора HTTP-глаголов быстро превратится в семейство эндпойнтов, каждый из которых покрывает какой-то аспект жизненного цикла сущности. Это всего лишь означает, что CRUD-мнемоника даёт только стартовый набор гипотез; любая конкретная предметная область требует вдумчивого подхода и дизайна подходящего API. Если же перед вами стоит задача разработать «универсальный» интерфейс, который подходит для работы с любыми сущностями, лучше сразу начинайте с номенклатуры в 10 методов типа описанной выше.

Примечания

¹ The HTTP QUERY Method

<https://www.ietf.org/archive/id/draft-ietf-httpbis-safe-method-w-body-o2.html>

² Web Distributed Authoring and Versioning (WebDAV) SEARCH

<https://www.rfc-editor.org/rfc/rfc5323>

Глава 39. Работа с ошибками в HTTP API

Рассмотренные в предыдущих главах примеры организации API согласно стандарту HTTP и принципам REST покрывают т.н. «*happy path*», т.е. стандартный процесс работы с API в отсутствие ошибок. Конечно, нам не менее интересен и обратный кейс — каким образом HTTP API следует работать с ошибками, и чем стандарт и архитектурные принципы могут нам в этом помочь. Пусть какой-то агент в системе (неважно, клиент или гейтвей) пытается создать новый заказ:

```
POST /v1/orders?user_id=<user_id> HTTP/1.1
Authorization: Bearer <token>
If-Match: <ревизия>

{ /* параметры заказа */ }
```

Какие потенциальные неприятности могут ожидать нас при выполнении этого запроса? Навскидку, это:

1. Запрос не может быть прочитан (недопустимые символы, нарушение синтаксиса).
2. Токен авторизации отсутствует.
3. Токен авторизации невалиден.
4. Токен валиден, но пользователь не обладает правами создавать новый заказ.
5. Пользователь удалён или деактивирован.
6. Идентификатор пользователя неверен (не существует).
7. Ревизия не передана.
8. Ревизия не совпадает с последней актуальной.
9. В теле запроса отсутствуют обязательные поля.
10. Какое-то из полей запроса имеет недопустимое значение.
11. Превышены лимиты на допустимое количество запросов.
12. Сервер перегружен и не может ответить в настоящий момент.
13. Неизвестная серверная ошибка (т.е. сервер сломан настолько, что диагностика ошибки невозможна).

Исходя из общих соображений, соблазнительной кажется идея назначить каждой из ошибок свой статус-код. Скажем, для ошибки (4) напрашивается код 403, а для ошибки (11) — 429. Не будем, однако, торопиться, и прежде зададим себе вопрос с какой целью мы хотим назначить тот или иной код ошибки.

В нашей системе в общем случае присутствуют три агента: пользователь приложения, само приложение (клиент) и сервер. Каждому из этих акторов необходимо понимать ответ на четыре вопроса относительно ошибки (причём для каждого из акторов ответ может быть разным):

1. Кто допустил ошибку (конечный пользователь, разработчик клиента, разработчик сервера или какой-то промежуточный агент, например, программист сетевого стека).
 - Не забудем учесть тот факт, что и конечный пользователь, и разработчик клиента могут допустить ошибку *намеренно*, например, пытаясь перебором подобрать пароль к чужому аккаунту.
2. Можно ли исправить ошибку, просто повторив запрос.
 - Если да, то через какое время.
3. Если повтором запроса ошибку исправить нельзя, то можно ли её исправить, переформулировав запрос.
4. Если ошибку вообще нельзя исправить, то что с этим делать.

На один из этих вопрос в рамках стандарта HTTP ответить достаточно легко: регулировать желаемое время повтора запроса можно через параметры кэширования ответа и заголовок `Retry-After`. Также HTTP частично помогает с первым вопросом: для определения, на чьей стороне произошла ошибка, используется первая цифра статус-кода (см. ниже).

Со всеми остальными вопросами, увы, ситуация сильно сложнее.

Клиентские ошибки

Статус-коды, начинающиеся с цифры 4, индицируют, что ошибка допущена пользователем или клиентом (или, по крайней мере, сервер так считает). *Обычно*, полученную 4xx повторять бессмысленно — если не предпринять дополнительных действий по изменению состояния сервиса, этот запрос не будет выполнен успешно никогда. Однако из этого правила есть исключения, самые важные из которых — 429 Too Many Requests и 404 Not Found. Последняя по стандарту имеет смысл «состояния неопределённости»: сервер имеет право использовать её, если не желает раскрывать причины ошибки. После получения ошибки 404, можно сделать повторный запрос, и он вполне может отработать успешно. Для индикации *персистентной* ошибки «ресурс не найден» используется отдельный статус 410 Gone.

Более интересный вопрос — а что всё-таки клиент может (или должен) сделать, получив такую ошибку. Как мы указывали в главе «[Разграничение областей ответственности](#)», если ошибка может быть исправлена программно, необходимо в машиночитаемом виде индицировать это клиенту; если ошибка не может быть исправлена, необходимо включить человекочитаемые сообщения для пользователя (даже просто «попробуйте начать сначала / перезагрузить приложение» лучше с точки зрения UX, чем «неизвестная ошибка») и для разработчика, который будет разбираться с проблемой.

С восстановимыми ошибками в HTTP, к сожалению, ситуация достаточно сложная. С одной стороны, протокол включает в себя множество специальных кодов, которые индицируют проблемы с использованием самого протокола — такие как 405 Method Not Allowed (данний глагол неприменим к указанному ресурсу), 406 Not Acceptable (сервер не может вернуть ответ согласно Accept-заголовкам запроса), 411 Length Required, 414 URI Too Long и так далее. Код клиента может обработать данные ошибки и даже, возможно, предпринять какие-то действия по их устранению (например, добавить заголовок Content-Length в запрос после получения ошибки 411), но все они очень плохо применимы к ошибкам в бизнес-логике. Например, мы можем вернуть 429 Too Many Requests при превышении лимитов запросов, но у нас нет никакого стандартного способа указать, *какой именно* лимит был превышен.

Частично проблему отсутствия стандартных подходов к возврату ошибок компенсируют использованием различных близких по смыслу статус-кодов для индикации разных состояний (либо и вовсе выбор произвольного кода ошибки и придания ему нового смысла в рамках конкретного API). В частности, сегодня де-факто стандартом является возврат кода 401 Unauthorized при отсутствии заголовков авторизации или невалидном токене (получение этого кода, таким образом, является сигналом для приложения предложить пользователю залогиниться в системе), что противоречит стандарту (который требует при возврате 401 обязательно указать заголовок WWW-Authenticate с описанием способа аутентификации пользователя; нам неизвестны реальные API, которые выполняют это требованием).

Однако таких кодов, которые могут отражать нюансы одной и той же проблемы, в стандарте очень мало. Фактически, мы приходим к тому, что множество различных ошибок в логике приложения приходится возвращать под очень небольшим набором статус-кодов:

- 400 Bad Request для всех ошибок валидации запроса (некоторые пуристы утверждают, что, вообще говоря, 400 соответствует нарушению формата запроса — невалидному JSON, например — а для логических ошибок следует использовать код 422 Unprocessable Content; в постановке задачи это мало что меняет);
- 403 Forbidden для любых проблем, связанных с авторизацией действий клиента;
- 404 Not Found в случае, если какие-то из указанных в запросе сущностей не найдены либо раскрытие причин ошибки нежелательно;
- 409 Conflict при нарушении целостности данных;
- 410 Gone если ресурс был удалён;
- 429 Too Many Requests при превышении лимитов.

Разработчики стандарта HTTP об этой проблеме вполне осведомлены, и отдельно отмечают, что для решения бизнес-сценариев необходимо передавать в метаданных либо теле ответа дополнительные данные для описания возникшей ситуации («the server SHOULD send a representation containing an explanation of the error situation, and whether it is a temporary or permanent condition»), что (как и введение новых специальных кодов ошибок) противоречит самой идеи унифицированного машиночитаемого формата ошибок. (Отметим, что отсутствие стандартов описания ошибок в бизнес-

логике — одна из основных причин, по которым мы считаем разработку REST API как его описал Филдинг в манифесте 2008 года невозможной; клиент должен обладать априорным знанием о том, как работать с метаинформацией об ошибке, иначе он сможет восстанавливать своё состояние после ошибки только перезагрузкой.)

NB: не так давно разработчики стандарта предложили собственную версию спецификации JSON-описания HTTP-ошибок — RFC 9457¹. Вы можете воспользоваться ей, но имейте в виду, что она покрывает только самый базовый сценарий:

- подтип ошибки не передаётся в мета-информации;
- нет разделения на сообщение для пользователя и сообщение для разработчика;
- конкретный машиночитаемый формат описания ошибок остаётся на усмотрение разработчика.

Дополнительно, у проблемы есть и третье измерение в виде серверного ПО мониторинга состояния системы, которое часто полагается на статус-коды ответов при построении графиков и уведомлений. Между тем, ошибки, скрывающиеся под одним статус кодом — например ввод неправильного пароля и истёкший срок жизни токена — могут быть очень разными по смыслу; повышенный фон первой ошибки может говорить о потенциальной попытке взлома путём перебора паролей, а второй — о потенциальных ошибках в новой версии приложения, которая может неверно кэшировать токены авторизации.

Всё это естественным образом подводит нас к следующему выводу: если мы хотим использовать ошибки для диагностики и (возможно) восстановления состояния клиента, нам необходимо добавить машиночитаемую метаинформацию о подвиде ошибки и, возможно, тело ошибки с указанием подробной информации о проблемах — например, как мы предлагали в главе «[Описание конечных интерфейсов](#)»:

```

POST /v1/coffee-machines/search HTTP/1.1

{
  "recipes": ["lngo"],
  "position": {"latitude": 110, "longitude": 55}
}
→
HTTP/1.1 400 Bad Request
X-OurCoffeeAPI-Error-Kind: wrong_parameter_value

{
  "reason": "wrong_parameter_value",
  "localized_message": "Что-то пошло не так.←
    Обратитесь к разработчику приложения.",
  "details": { "checks_failed": [
    { "field": "recipe",
      "error_type": "wrong_value",
      "message": "Value 'lngo' unknown.←
        Did you mean 'lungo'?" },
    { "field": "position.latitude",
      "error_type": "constraintViolation",
      "constraints": { "min": -90, "max": 90},
      "message": "'position.latitude' value←
        must fall within the [-90, 90] interval" }
  ]}}

```

Также напомним, что любые неизвестные 4xx-статус-коды клиент должен трактовать как ошибку 400 Bad Request, следовательно, формат (мета)данных ошибки 400 должен быть максимально общим.

Серверные ошибки

Ошибки 5xx индицируют, что клиент, со своей стороны, выполнил запрос правильно, и проблема заключается в сервере. Для клиента, по большому счёту, важно только то, имеет ли смысл повторять запрос и, если да, то через какое время. Если учесть, что в любых публично доступных API причины серверных ошибок, как правило, не раскрывают — в абсолютном большинстве кодов 500 Internal Server Error и 503 Service Unavailable достаточно для индикации серверных ошибок (второй код указывает, что отказ в обслуживании имеет разовый характер и есть смысл автоматически повторить запрос), или можно вовсе ограничиться одним из них с опциональным заголовком Retry-After.

Для внутренних систем, вообще говоря, такое рассуждение неверно. Для построения правильных мониторингов и системы оповещений необходимо, чтобы серверные ошибки, точно так же, как и клиентские, содержали подтип ошибки в машиночитаемом виде. Здесь по-прежнему применимы те же подходы — использование широкой номенклатуры кодов и/или передача типа ошибки заголовком — однако эта информация должна быть вырезана гейтвеем на границе внешней и внутренней систем, и заменена на общую информацию для разработчика и для конечного пользователя системы с описанием действий, которые необходимо выполнить при получении ошибки.

```
POST /v1/orders/?user_id=<user id> HTTP/1.1
If-Match: <ревизия>

{ "parameters" }
→
// Ответ, полученный гейтвеем
// от сервиса обработки заказов,
// метаданные которого будут
// использованы для мониторинга
HTTP/1.1 500 Internal Server Error
// Тип ошибки: получен таймаут от БД
X-OurCoffeeAPI-Error-Kind: db_timeout
{ /*
    * Дополнительные данные, например,
    * какой хост ответил таймаутом
*/ }
```

```

// Ответ, передаваемый клиенту.
// Детали серверной ошибки удалены
// и заменены на инструкцию клиенту.
// Поскольку гейтвей не знает, был
// ли в действительности сделан заказ,
// клиенту рекомендуется попробовать
// повторить запрос и/или попытаться
// получить актуальное состояние
HTTP/1.1 500 Internal Server Error
Retry-After: 5

{
  "reason": "internal_server_error",
  "localized_message": "Не удалось«
    получить ответ от сервера.«
    Попробуйте повторить операцию
    или обновить страницу.",
  "details": {
    "can_be_retried": true,
    "is_operation_failed": "unknown"
  }
}

```

Вот здесь мы, однако, вступаем на очень скользкую территорию. Современная практика реализации HTTP-клиентов такова, что безусловно повторяются только немодифицирующие (GET, HEAD, OPTIONS) запросы. В случае модифицирующих запросов *разработчик должен написать код*, который повторит запрос — и для этого разработчику нужно очень внимательно прочитать документацию к API, чтобы убедиться, что это поведение допустимо и не приведёт к побочным эффектам.

Теоретически идемпотентные методы PUT и DELETE можно вызывать повторно. Практически, однако, ввиду того, что многие разработчики упускают требование идемпотентности этих методов, фреймворки работы с HTTP API по умолчанию перезапросов модифицирующих методов, как правило, не делают, но некоторую выгоду из следования стандарту мы всё же можем извлечь — по крайней мере, сама сигнатура индицирует, что запрос *можно* повторять.

Что касается более сложных ситуаций, когда мы хотим указать разработчику, что он может безопасно повторить потенциально неидемпотентную операцию, то мы могли бы предложить формат описания доступных действий в теле ошибки... но практически никто не ожидает найти такое описание в самой ошибке. Возможно, потому, что с ошибками 5xx, в отличие от 4xx,

программисты практически не сталкиваются при написании клиентского кода, и мало какие тестовые среды позволяют такие ошибки эмулировать. Так или иначе, описывать необходимые действия при получении серверной ошибки вам придётся в документации. (Имейте в виду, что эти инструкции с большой долей вероятности будут проигнорированы. Таков путь.)

Организация системы ошибок в HTTP API на практике

Как понятно из вышесказанного, фактически есть три способа работать с ошибками HTTP API:

1. Расширительно трактовать номенклатуру статус-кодов и использовать новый код каждый раз, когда требуется индицировать новый вид ошибки. (Автор этой книги неоднократно встречал ситуации, когда при разработке API просто выбирался «похоже выглядящий» статус безо всякой оглядки на его описание в стандарте.)
2. Полностью отказаться от использования статус-кодов и вкладывать описание ошибки в тело и/или метаданные ответа с кодом 200. Этим путём идут почти все RPC-фреймворки.
 - 2a. Вариантом этой стратегии можно считать использование всего двух статус-кодов ошибок (400 для любой клиентской ошибки, 500 для любой серверной), дополнительно трёх (те же плюс 404 для статуса неопределённости).
3. Применить смешанный подход, то есть использовать статус-код согласно его семантике для индикации *рода* ошибки и вложенные (мета)данные в специально разработанном формате для детализации (подобно фрагментам кода, предложенным нами в настоящей главе).

Как нетрудно заметить, считать соответствующим стандарту можно только подход (3). Будем честны и скажем, что выгоды следования ему, особенно по сравнению с вариантом (2a), не очень велики и состоят в основном в чуть лучшей читабельности логов и большей прозрачности для промежуточных прокси.

Примечания

¹ RFC 9457 Problem Details for HTTP APIs

<https://www.rfc-editor.org/rfc/rfc9457.html>

Глава 40. Заключительные положения и общие рекомендации

Подведём итог описанному в предыдущих главах. Чтобы разработать качественный HTTP API, необходимо:

1. Описать happy path, т.е. составить диаграмму вызовов для стандартного цикла работы клиентского приложения.
2. Определить каждый вызов как операцию над некоторым ресурсом и, таким образом, составить номенклатуру URL и применимых методов.
3. Понять, какие ошибки возможны при выполнении операций и каким образом клиент должен восстанавливаться из какого состояния.
4. Решить, какая функциональность будет передана на уровень протокола HTTP [какие стандартные возможности протокола будут использованы в сопряжении с какими инструментами разработки] и в каком объёме.
5. Опираясь на решения 1-4, разработать конкретную спецификацию.
6. Проверить себя: пройти по пунктам 1-3, написать псевдокод бизнес-логики приложения согласно разработанной спецификации, и оценить, насколько удобным, понятным и читабельным оказался результирующий API.

Позволим себе так же дать несколько советов по code style:

1. Не различайте пути с / на конце и без него и примите какую-то рекомендацию по умолчанию (мы рекомендуем все пути заканчивать на / — по простой причине, это позволяет разумно описать обращение к корню домена как ГЛАГОЛ /). Если вы решили запретить один из вариантов (скажем, пути без слэша в конце), при обращении по второму варианту должен быть или редирект или однозначно читаемая ошибка.
2. Включайте в ответы стандартные заголовки — Date, Content-Type, Content-Encoding, Content-Length, Cache-Control, Retry-After — и вообще старайтесь не полагаться на то, что клиент правильно догадывается о параметрах протокола по умолчанию.
3. Поддержите метод OPTIONS и протокол CORS¹ на случай, если ваш API захотят использовать из браузеров.
4. Определитесь с правилами выбора кейсинга параметров (и преобразований кейсинга при перемещении параметра между различными частями запроса) и придерживайтесь их.

5. Всегда оставляйте себе возможность обратно-совместимого расширения операции API. В частности, всегда возвращайте корневой JSON-объект в ответах эндпойтов — потому что приписать новые поля к объекту вы можете, а к массивам и примитивам — нет.

- Отметим также, что пустая строка не является валидным JSON, поэтому корректнее возвращать пустой объект {} там, где ответа не подразумевается (или статус 204 No Content с пустым телом, но тогда эндпойнт нельзя будет расширить в будущем).

6. Для всех GET-запросов указывайте политику кэширования (иначе всегда есть шанс, что клиент или промежуточный агент придумает её за вас).

7. Не эксплуатируйте известные возможности оперировать запросами в нарушение стандарта и не изобретайте свои решения для «серых зон» протокола. В частности:

- не размещайте модифицирующие операции за методом GET и неидемпотентные операции за PUT / DELETE;
- соблюдайте симметрию GET / PUT / DELETE методов;
- не позволяйте GET / HEAD / DELETE-запросам иметь тело, не возвращайте тело в ответе метода HEAD или совместно со статус-кодом 204 No Content;
- не придумывайте свой стандарт для передачи массивов и вложенных объектов в query — лучше воспользоваться HTTP-глаголом, позволяющим запросу иметь тело, или, в крайнем случае, передать параметры в виде Base64-кодированного JSON- поля;
- не размещайте в пути и домене URL параметры, по формату требующие эскейпинга (т.е. могущие содержать символы, отличные от цифр и букв латинского алфавита); для этой цели лучше воспользоваться query-параметрами или телом запроса.

8. Ознакомьтесь хотя бы с основными видами уязвимостей в типичных имплементациях HTTP API, которыми могут воспользоваться злоумышленники:

- CSRF²
- SSRF³
- HTTP Response Splitting⁴
- Unvalidated Redirects and Forwards⁵

и заложите защиту от этих векторов атак на уровне вашего серверного ПО. Организация OWASP предоставляет хороший обзор лучших security-практик для HTTP API⁶.

В заключение хотелось бы сказать следующее: HTTP API — это способ организовать ваше API так, чтобы полагаться на понимание семантики операций как разнообразным программным обеспечением, от клиентских фреймворков до серверных гейтвеев, так и разработчиком, который читает спецификацию. В этом смысле экосистема HTTP предоставляет пожалуй что наиболее широкий (и в плане глубины, и в плане распространённости) по сравнению с другими технологиями словарь для описания самых разнообразных ситуаций, возникающих во время работы клиент-серверных приложений. Разумеется, эта технология не лишена своих недостатков, но для разработчика *публичного* API она является выбором по умолчанию — на сегодняшний день скорее надо обосновывать отказ от HTTP API чем выбор в его пользу.

Примечания

¹ Fetch Living Standard. CORS protocol

<https://fetch.spec.whatwg.org/#http-cors-protocol>

² Cross Site Request Forgery (CSRF)

<https://owasp.org/www-community/attacks/csrf>

³ Server Side Request Forgery

https://owasp.org/www-community/attacks/Server_Side_Request_Forgery

⁴ HTTP Response Splitting

https://owasp.org/www-community/attacks/HTTP_Response_Splitting

⁵ Unvalidated Redirects and Forwards Cheat Sheet

https://cheatsheetseries.owasp.org/cheatsheets/Unvalidated_Redirects_and_Forwards_Cheat_Sheet.html

⁶ REST Security Cheat Sheet

https://cheatsheetseries.owasp.org/cheatsheets/REST_Security_Cheat_Sheet.html

РАЗДЕЛ V. SDK И UI

Глава 41. Терминология. Обзор технологий разработки SDK

Как мы отмечали во Введении, аббревиатура «SDK» («Software Development Kit»), как и многие из обсуждавшихся ранее терминов, не имеет конкретного значения. Считается, что SDK отличается от API тем, что помимо программных интерфейсов содержит и готовые инструменты для работы с ними. Определение это, конечно, лукавое, поскольку почти любая технология сегодня идёт в комплекте со своим набором инструментов.

Тем не менее, у термина SDK есть и более узкое значение, в котором он часто используется: это клиентская библиотека, которая предоставляет высокоуровневый (обычно, нативный) интерфейс для работы с некоторой нижележащей платформой (и в частности с клиент-серверным API). Чаще всего речь идёт о библиотеках для мобильных ОС или веб браузеров, которые работают поверх HTTP API сервиса общего назначения.

Среди подобных клиентских SDK особо выделяются те, которые предоставляют не только программные интерфейсы для работы с API, но также и готовые визуальные компоненты, которые разработчик может использовать. Классический пример такого SDK — это библиотеки картографических сервисов; в силу исключительной сложности самостоятельной реализации движка работы с картами (особенно векторными) вендоры API карт предоставляют и «обёртки» к HTTP API (например, поисковому), и готовые библиотеки для работы с географическими сущностями, которые часто включают в себя и визуальные компоненты общего назначения — кнопки, метки, контекстные меню — которые могут применяться и совершенно самостоятельно вне контекста API (SDK) как такового.

Настоящий раздел будет посвящён именно двум этим видам программных инструментов:

- клиентским «обёрткам» поверх клиент-серверных API;
- клиентским библиотекам, предоставляющие визуальные компоненты, с которыми конечный пользователь может взаимодействовать напрямую.

Во избежание нагромождения подобных оборотов мы будем называть первый тип инструментов просто «SDK», а второй — «UI-библиотеки».

NB: вообще говоря, UI-библиотека может как включать в себя обёртку над клиент-серверным API, так и предоставлять чистый API к графическому движку. В рамках этой книги мы будем говорить, в основном, о первом варианте как о более общем случае (и более сложном с точки зрения проектирования API). Многие паттерны дизайна SDK, которые мы опишем далее, применимы и к «чистым» библиотекам без клиент-серверной составляющей.

Выбор фреймворка для разработки UI-компонентов

Поскольку UI — высокоуровневая абстракция над примитивами ОС, почти для всех платформ существуют специализированные фреймворки для разработки визуальных компонентов. Выбор такого фреймворка, увы, может быть непростым занятием. Например, в случае веб-платформы её низкоуровневость и популярность привели к тому, что на сегодня количество конкурирующих технологий, предоставляющих фреймворки для разработки SDK, превосходит всякое воображение. Можно упомянуть наиболее распространённые на сегодня React¹, Angular², Svelte³, Vue.js⁴ и не сдающие своих позиций Bootstrap⁵ и Ember⁶. Из перечисленных технологий наибольшее проникновение имеет React, но оно всё ещё измеряется в единицах процентов⁷. При этом компоненты на «чистом» JavaScript/CSS при этом часто критикуются как неудобные к использованию в перечисленных фреймворках, так как каждый из них реализует весьма строгие методологии. Примерно такая же ситуация наблюдается, например, с разработкой визуальных библиотек для Windows. Вопрос «на каком фреймворке разрабатывать UI-компоненты для этих платформ», увы, не имеет простого ответа — фактически, вам придётся измерять рынки и принимать решения по каждому фреймворку отдельно.

Лучше дела обстоят с актуальными мобильными платформами (а также MacOS), которые гораздо более гомогенны. Однако здесь возникает другая проблема — современные приложения, как правило, поддерживают сразу несколько таких платформ, что приводит к дублированию кода (и номенклатуры API).

Решением этой проблемы может быть использование кросс-платформенных мобильных (React Native⁸, Flutter⁹, Xamarin¹⁰) и десктопных (JavaFX¹¹, QT¹²) фреймворков, а также узкоспециализированных решений для конкретных задач (например, Unity¹³ для разработки игр). Несомненным преимуществом таких технологий является скорость разработки и универсальность (как кода, так и программистов). Недостатки также достаточно очевидны — от таких приложений может быть сложно добиться оптимальной производительности, и к ним часто неприменимы многие стандартные инструменты, доступные для конкретной платформы; например, отладка и профайлинг могут быть затруднены. На сегодня скорее наблюдается паритет между двумя этими подходами (несколько фактически независимых приложений, написанных на поддерживаемых платформой языках vs. одно кросс-платформенное приложение).

Примечания

¹ React

<https://react.dev/>

² Angular

<https://angular.io/>

³ Svelte

<https://svelte.dev/>

⁴ Vue.js

<https://vuejs.org/>

⁵ Bootstrap

<https://getbootstrap.com/>

⁶ Ember

<https://emberjs.com/>

⁷ How Many Websites Use React in 2023? (Usage Statistics)

<https://increditoools.com/react-usage-statistics/>

⁸ React Native

<https://reactnative.dev/>

⁹ Flutter

<https://flutter.dev/>

¹⁰ Xamarin

<https://dotnet.microsoft.com/en-us/apps/xamarin>

¹¹ JavaFX

<https://openjfx.io/>

¹² QT

<https://www.qt.io/>

¹³ Unity

<https://docs.unity3d.com/Manual/index.html>

Глава 42. SDK: проблемы и решения

Первый вопрос об SDK (напомним, так мы будем называть нативную клиентскую библиотеку, предоставляющую доступ к technology-agnostic клиент-серверному API), который мы должны прояснить — почему вообще такое явление как SDK существует. Иными словами, почему использование обёртки для фронтенд-разработчика является более удобным, нежели работа с нижележащим API напрямую.

Некоторые причины лежат на поверхности:

1. Протоколы клиент-серверных API, как правило, разрабатываются так, что не зависят от конкретного языка программирования и, таким образом, без дополнительных действий полученные из API данные будут представлены в не самом удобном формате. Например в JSON нет типа данных «дата и время», и его приходится передавать в виде строки; или, скажем, поддержка (де)сериализации хэш-таблиц в протоколах общего назначения отсутствует.
2. Большинство языков программирования императивные (и чаще всего — объектно-ориентированные), в то время как большинство форматов данных — декларативные. Работать с сырьими данными, полученными из API, таким образом почти всегда неудобно с точки зрения написания кода, программистам зачастую было бы удобнее работать с полученными из API данными как с объектами.
3. Разные языки программирования предполагают разный стиль кодирования (кейсинг, организация неймспейсов и т.п.), в то время как концепция API не предполагает адаптацию форматирования под запрашивающую платформу.
4. Как правило, платформа/язык программирования диктуют свою парадигму работы с возникающими ошибками (в виде исключений и/или механизмов defer/panic), что опять же неприменимо в концепции универсального для всех клиентов сетевого API.

5. API идёт в комплекте с рекомендациями (машино- или человекочитаемыми) по организации перезапросов в случае недоступности эндпойнтов. Эту логику необходимо реализовать разработчику клиента, поскольку библиотеки работы с сетью её, как правило, не предоставляют (и в общем-то не могут этого делать для потенциально неидемпотентных запросов). Этот пункт, при всей видимой малозначительности, является критически важным для любого крупного API, поскольку именно на этом уровне разработчики API могут заложить предохранители от потенциальной перегрузки серверов API лавиной перезапросов, поскольку разработчики клиентов этой частью традиционно пренебрегают: * читать заголовок `Retry-After` и не пытаться перезапросить эндпойнт раньше, чем указал сервер; * ввести увеличивающие интервалы между перезапросами.

Наличие собственного SDK устранило бы указанные проблемы, которые в некотором смысле являются тривиальными: для их решения не требуется изменять порядок работы с API (каждому вызову и каждому ответу в API однозначно соответствует какая-то конструкция на языке платформы, и достаточно описать правила построения такого соответствия) — достаточно адаптировать платформо-независимый формат API к правилам конкретного языка программирования, что часто можно автоматизировать.

Однако, помимо тривиальных проблем при разработке SDK к клиент-серверному API мы сталкиваемся и с проблемами более высокого порядка:

1. В клиент-серверных API данные передаются только по значению; чтобы сослаться на какую-то сущность, необходимо использовать какие-то внешние идентификаторы. Например, если у нас есть два набора сущностей — рецепты и предложения кофе — то нам необходимо будет построить карту рецептов по `id`, чтобы понять, на какой рецепт ссылается какое предложение:

```

// Запрашиваем информацию о рецептах
// лунго и латте
let recipes = await api
  .getRecipes(['lungo', 'latte']);
// Строим карту, позволяющую обратиться
// к данным о рецепте по его id
let recipeMap = new Map();
recipes.forEach((recipe) => {
  recipeMap.set(recipe.id, recipe);
});
// Запрашиваем предложения
// лунго и латте
let offers = await api.search({
  recipes: ['lungo', 'latte'],
  location
});
// Для того, чтобы показать предложения
// пользователю, нужно из каждого
// предложения извлечь id рецепта,
// и уже по id найти описание
promptUser(
  'Найденные предложения',
  offers.map((offer) => {
    let recipe = recipeMap
      .get(offer.recipe_id);
    return {offer, recipe};
  })
);

```

Указанный код мог бы быть вдвое короче, если бы мы сразу получали из метода `api.search` предложения с заполненной ссылкой на рецепт:

```

// Запрашиваем информацию о рецептах
// лунго и латте
let recipes = await api
  .getRecipes(['lungo', 'latte']);
// Запрашиваем предложения
// лунго и латте
let offers = await api.search({
  // Передаём не идентификаторы
  // рецептов, а ссылки на объекты,
  // описывающие рецепты
  recipes,
  location
});

promptUser(
  'Найденные предложения',
  // offer уже содержит
  // ссылку на рецепт
  offers
);

```

2. Клиент-серверные API, как правило, стараются декомпозировать так, чтобы одному запросу соответствовал один тип возвращаемых данных. Даже если эндпойнт композитный (т.е. позволяет при запросе с помощью параметров указать, какие из дополнительных данных необходимо вернуть), это всё ещё ответственность разработчика этими параметрами воспользоваться. Код из примера выше мог бы быть ещё короче, если бы SDK взял на себя инициализацию всех нужных связанных объектов:

```
// Запрашиваем предложения
// лунго и латте
let offers = await api.search({
  recipes: ['lungo', 'latte'],
  location
});

// SDK сам обратился к эндпойнту
// `getRecipes` и получил данные
// по лунго и латте
promptUser(
  'Найденные предложения',
  offers
);
```

При этом SDK может также заполнять программные кэши сущностей (если мы не полагаемся на кэширование на уровне протокола) и/или позволять «лениво» инициализировать объекты.

Вообще, хранение данных (таких, как токены авторизации, ключи идемпотентности при перезапросах, идентификаторы черновиков при двухфазных коммитах и т.д.) между запросами также является ответственностью клиента и с трудом поддаётся формализации. Если SDK возьмёт на себя эти функции, в коде приложений, использующих API, будет допущено намного меньше ошибок.

3. Получение обратных вызовов в клиент-серверном API, даже если это дуплексный канал, с точки зрения клиента выглядит крайне неудобным в разработке, поскольку вновь требует наличия карт объектов. Даже если в API реализована push-модель, код выходит чрезвычайно громоздким:

```

// Получаем текущие заказы
let orders = await api
    .getOngoingOrders();
// Строим карту заказов
let orderMap = new Map();
orders.forEach((order) => {
    orderMap.set(order.id, order);
});
// Подписываемся на события
// изменения состояния заказов
api.subscribe(
    'order_state_change',
    (event) => {
        let order = orderMap
            .get(event.order_id);
        // Выполняем какие-то
        // действия с заказом,
        // например, обновляем UI
        // приложения
        UI.update(order);
    }
);

```

Если же API требует поллинга изменений состояний объектов, то разработчику придётся ещё где-то реализовать периодический опрос эндпойнта со списком изменений, и ещё следить за тем, чтобы не перегружать сервер запросами.

Кроме того, обратите внимание, что в вышеприведённом фрагменте кода [разработчиком приложения] допущены множественные ошибки:

- сначала получается список заказов, а затем происходит подписывание на их изменения; если между двумя этими вызовами какой-то из заказов изменился, приложение об этом не узнает;
- если пришло событие изменения какого-то неизвестного приложению заказа (который, например, был создан с другого устройства или в другом потоке исполнения), поиск в карте заказов вернёт пустой результат, и обработчик события выбросит исключение, которое никак не обработано.

И вновь мы приходим к тому, что недостаточно продуманный SDK приводит к ошибкам в работе использующих его приложений. Разработчику было бы намного удобнее, если бы объект заказа позволял подписаться на свои события, не задумываясь о том, как эта подписка технически работает и как не пропустить события:

```

let order = await api
  .createOrder(...)
  // Нет нужды подписываться
  // на *все* события и потом
  // фильтровать их по id
  .subscribe(
    'state_change',
    (event) => { ... }
  );

```

NB: код выше предполагает, что объект `order` изменяется консистентным образом: даже если между вызовами `createOrder` и `subscribe` состояние заказа успело измениться на сервере, состояние объекта `order` будет консистентно списку событий `state_change`, полученных наблюдателем. Как это организовать технически — как раз забота разработчика SDK.

4. Восстановление после ошибок в бизнес-логике, как правило, достаточно сложная операция, которую сложно описать в машиночитаемом виде. Разработчику клиента необходимо самому продумать эти сценарии.

```

// Получаем предложения
let offers = await api.search(...);
// Пользователь выбирает
// подходящее предложение
let selectedOffer = await promptUser(offers);
let order;
let offer = selectedOffer;
let number0fTries = 0;
do {
  // Пытаемся создать заказ
  try {
    number0fTries++;
    order = await api.createOrder(offer, ...);
  } catch (e) {
    // Если количество попыток пересоздания
    // заказа превысило какое-то разумное
    // значение следует бросить попытки
    if (number0fTries > TRY_LIMIT) {
      throw new NoRetriesLeftError();
    }
    // Если произошла ошибка
    // «предложение устарело»
    if (e.type == api.OfferExpiredError) {
      // если попытки ещё остались,
      // пытаемся получить новое предложение
      offer = await api.renewOffer(offer);
    } else {
      // Обработка других видов ошибок
      ...
    }
  }
} while (!order);

```

Как мы видим, простая операция «попробовать продлить предложение» выливаются в громоздкий код, в котором легко ошибиться, и, что ещё важнее, который совершенно не нужен разработчику приложения, поскольку он не добавляет никакой новой функциональности для конечного пользователя. Было бы гораздо проще, если бы этой ошибки *вовсе не было в SDK*, т.е. попытки обновления и перезапросы выполнялись бы автоматически.

Аналогичные ситуации возникают и в случае нестрогого-консистентных API или оптимистичного управления параллелизмом — и вообще в любом API, в котором фон ошибок является ожидаемым (что в случае распределённых клиент-серверных API является нормой жизни). Для разработчика приложения написание кода, имплементирующего политики типа «*read your writes*» (т.е. передачу токенов последней известной операции в последующие запросы) — попросту напрасная трата времени.

5. Наконец, ещё одна важная функция, которая может быть доверена SDK — это изоляция нижележащего API и смена парадигмы версионирования. Доступ к функциональности API может быть скрыт (т.е. разработчики не будут иметь доступ к низкоуровневой работой с API), тем самым обеспечивая определённую свободу работы с API изнутри SDK, вплоть до бесшовного перехода на новые мажорные версии API. Этот подход, несомненно, предоставляет вендору API намного больше контроля над приложениями клиентов, но требует и намного больше ресурсов на разработку, и, что важнее, грамотного проектирования SDK — такого, чтобы у разработчиков не было необходимости обращаться к API напрямую в обход SDK по причине отсутствия в нём необходимых функций или их плохой реализации, и при этом SDK могут пережить смену мажорной версии низкоуровневого API.

Суммируя написанное выше, хорошо спроектированный SDK служит, помимо поддержания консистентности платформе и предоставления «синтаксического сахара», трём важным целям:

- снижение количества ошибок в клиентском коде путём имплементации хелперов, покрывающих неочевидные и слабоформализуемые аспекты работы с API;
- избавление клиентских разработчиков от необходимости писать код, который им совершенно не нужен;

- предоставление разработчику API большего контроля над интеграциями.

Кодогенерация

Как мы убедились, список задач, стоящих перед разработчиком SDK (если, конечно, его целью является качественный продукт) — очень и очень значительный. Учитывая, что под каждую целевую платформу необходим отдельный SDK, неудивительно, что многие вендоры API стремятся полностью или частично заменить ручной труд машинным.

Одно из основных направлений такой автоматизации — кодогенерация, то есть разработка технологии, которая позволяет по спецификации API сгенерировать готовый код SDK на целевом языке программирования для целевой платформы. Многие современные стандарты обмена данными (в частности, gRPC) поставляются в комплекте с генераторами готовых клиентов на различных языках; к другим технологиям (в частности, OpenAPI/Swagger) такие генераторы пишутся энтузиастами.

Генерация кода позволяет решить типовые проблемы: стиль кодирования, обработка исключений, (де)серIALIZАЦИЯ сложных типов — словом все те задачи, которые зависят не от особенностей высокоуровневой бизнес-логики, а от конвенций конкретной платформы. Относительно недорого разработчик SDK может дополнить такой автоматизированный «перевод» удобными хелпера: обеспечить автоматические перезапросы для идемпотентных эндпоинтов (с реализацией какой-то политики управления интервалами повторных вызовов), кэширование результатов, сохранение данных (например, токенов авторизации) в системном хранилище и т.д.

Такой сгенерированный SDK часто называют термином «клиент к API». Удобство использования и функциональные возможности кодогенерации столь привлекательны, что многие вендоры API только ей и ограничиваются, предоставляя свои SDK в виде сгенерированных клиентов.

Как мы, однако, видим из написанного выше, проблемы более высокого порядка — получение серверных событий, обработка ошибок в бизнес-логике и т.п. — никак не может быть покрыта кодогенерацией, во всяком случае — стандартным модулем без его доработки применительно к конкретному API. В случае нетривиальных API со сложным основным циклом работы очень

желательно, чтобы SDK решал также и высокоуровневые проблемы, иначе вы просто получите множество разработанных поверх API приложений, раз за разом повторяющие одни и те же «детские ошибки». Тем не менее, это не повод отказываться от кодогенерации полностью — её можно использовать как базис, на котором будет разработан высокоуровневый SDK.

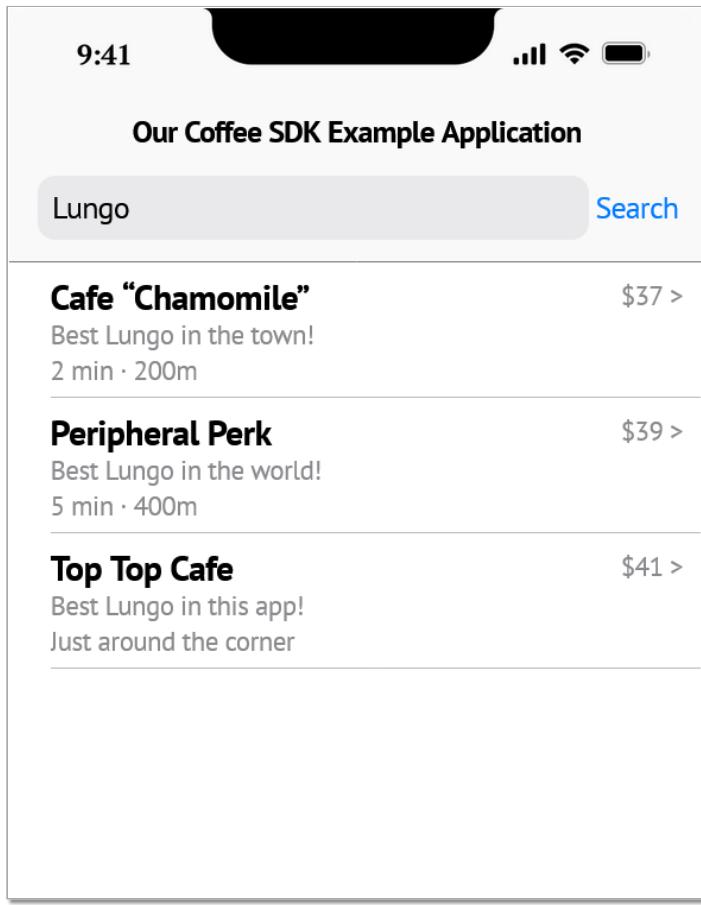
Другие инструменты

Слово «Kit» в «Software Development Kit» подразумевает, что в комплекте с технологией поставляются вспомогательные инструменты, такие как эмулятор / симулятор, песочница, плагины для популярных IDE и т.д. В рамках настоящего раздела мы не будем фокусироваться на этом аспекте, и обсудим его подробнее в разделе «API как продукт».

Глава 43. Проблемы встраивания UI-компонентов

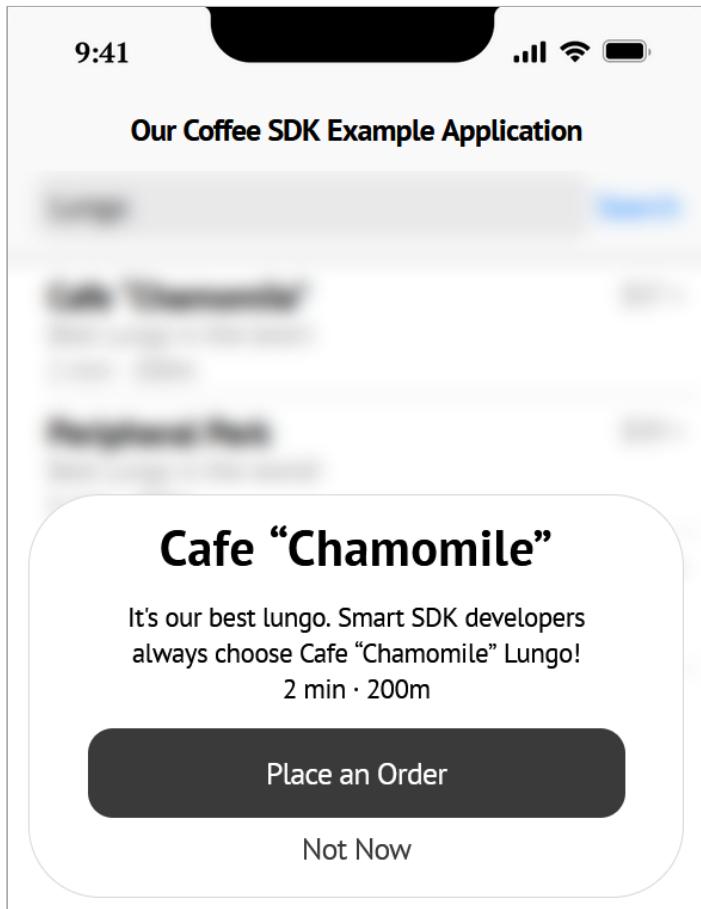
Введение в состав SDK UI-компонентов обогащает и так не самую простую конструкцию из клиент-серверного API и клиентской библиотеки дополнительным измерением: теперь с вашим API взаимодействуют одновременно и разработчики (которые написали код приложения), и пользователи (которые используют приложение). Хотя это изменение на первый взгляд может показаться не очень значительным, с точки зрения дизайна API добавление конечного пользователя — огромная проблема, которая требует на порядок более глубокой и качественной проработки дизайна программных интерфейсов по сравнению с «чистым» клиент-серверным API. Попробуем объяснить, почему так происходит, на конкретном примере.

Пусть мы решили поставлять в составе нашего кофейного API также и клиентский SDK, который предоставляет готовые компоненты для разработчиков приложений. Достаточно простая функциональность: пользователь вводит поисковый запрос и видит результаты в виде списка.



Основной экран приложения с результатами поиска

Пользователь может выбрать какой-либо из объектов, и тогда откроется экран просмотра предложения с панелью доступных действий.



Панель просмотра предложения

Для реализации этого сценария мы предоставим объектно-ориентированный API в виде, ну скажем, класса `SearchBox`, который реализует описанную функциональность поверх клиент-серверного метода `search` нашего клиент-серверного API.

Проблемы

С одной стороны нам может показаться, что наш UI — это просто надстройка над клиент-серверным search, визуализирующая результаты поиска или, иными словами, пользовательский интерфейс — всего лишь иное представление интерфейса программного. Но можно ли сказать, что показанный выше интерфейс — с кнопками и всплывающими панелями (и это мы оставляем за скобками анимации) — является просто проекцией API из двух методов `search` и `createOrder`? Мы бы назвали такое утверждение очень большой натяжкой. Для построения хорошего UI нам нужно пересечь две плоскости:

- собственно функциональность нижележащего API;
- принятые в рамках платформы концепции визуализации данных и взаимодействия с элементами управления (возможно, творчески развитые нашими разработками).

Эти две предметных области могут находиться весьма далеко друг от друга. Более того, **чем более интерфейс приближен к сырым данным, тем он, как правило, менее удобен для пользователя** (как классический пример — огромные формы ввода данных¹). Если мы хотим построить эргономичный интерфейс, нам придётся заменить формы сложными интерфейсными надстройками над данными и над графическими примитивами платформы, причём имеющими собственное состояние. Это, в свою очередь, ведёт к накоплению проблем в архитектуре SDK:

1. Объединение в одном объекте разнородной функциональности

Посмотрим на панель действий с предложениями. Допустим, мы размещаем на ней две кнопки — «заказать» и «показать на карте» — плюс действие «отменить». Эти кнопки выглядят одинаково и реагируют на действия пользователя одинаково — но при этом осуществляют абсолютно не имеющие ничего общего друг с другом действия.

Допустим, мы предоставили программисту возможность добавить свои кнопки действий на панель, для чего предоставим в составе SDK класс `Button`. Достаточно быстро мы выясним, что этой функциональностью будут пользоваться в двух основных диаметрально противоположных сценариях:

- для размещения на панели дополнительных кнопок, ну скажем, «позвонить в кафе», выполненных в том же дизайне, что и стандартные;
- для изменения дизайна стандартных кнопок в соответствии с фирменным стилем заказчика, сохраняя ту же самую функциональность в неизменном виде.

Более того, возможен и третий сценарий: разработчики заходят сделать кнопку «позвонить», которая будет и выглядеть иначе, и программно выполнять другие действия, но при этом будет *наследовать UX* кнопки — т.е. нажиматься при клике, располагаться в ряд с другими кнопками и так далее.

С точки зрения разработчика SDK это означает, что класс `Button` должен позволять независимо переопределять и внешний вид кнопки, и реакцию на действия пользователя, и элементы UX — или, иначе говоря, каждая из трёх подсистем может быть заменена альтернативной имплементацией так, чтобы две остальные системы продолжили работать без изменений.

2. Общее владение ресурсами

Предположим, что мы хотим разрешить разработчику подставить в наш `SearchBox` свой поисковый запрос — например, чтобы дать возможность разместить в приложении баннер «найти лунго рядом со мной», по нажатию на который происходит переход к нашему компоненту с введённым запросом «лунго». Для этого разработчику потребуется программно показать соответствующий экран и вызвать нужный метод `SearchBox`-а — допустим, не мудрствуя лукаво, мы назовём его просто `search`.

Два наших метода `search` («чистый» клиент-серверный и компонентный `SearchBox.search`) принимают одни и те же параметры и выдают один и тот же результат. Но *ведут себя* эти методы совершенно по-разному:

- если вызвать несколько раз `SearchBox.search`, не дожидаясь ответа сервера, то все запросы, кроме последнего во времени, должны быть проигнорированы; даже если ответы пришли вразнобой, только тот из них, который соответствует новейшему запросу, должен быть показан в UI;
 - дополнительная задача — что должен вернуть вызов метода `SearchBox.search`, если он был прерван выполнением другого запроса? Если неуспех, то в чём состоит ошибка вызывающего? Если успех, то почему результат не был отражён в UI?

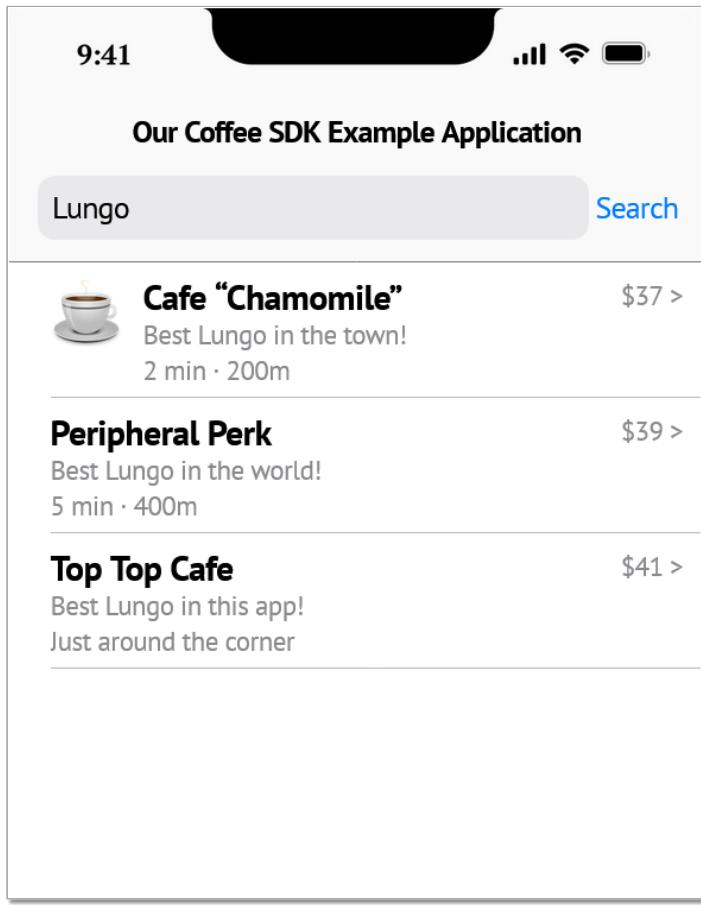
- что порождает другую проблему: а если в момент вызова `SearchBox.search` уже исполнялся какой-то запрос, инициированный пользователем — *что должно произойти?* Какой из вызовов приоритетнее — выполненный разработчиком или выполненный самим пользователем?

В реализации клиент-серверного API такой проблемы у нас нет — каждый актор,зывающий функцию поиска, получит свой ответ независимо. Но с UI-компонентами этот подход не работает, поскольку все они, в конечном итоге, разделяют один общий ресурс — экран приложения и внимание пользователя на нём.

Любая асинхронная операция в UI-компонентах, особенно если она индицируется визуально (с помощью анимации или другого длящегося действия), может помешать любой другой визуальной операции — в том числе вследствие действий пользователя.

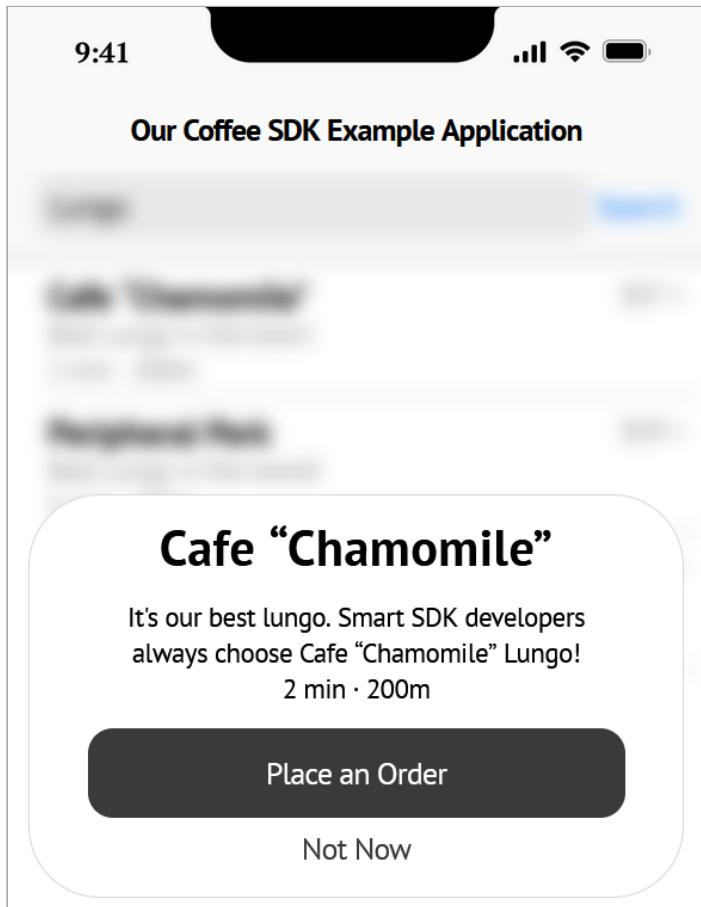
3. Множественная иерархия подчинения сущностей

Предположим, что разработчик хочет обогатить дизайн списка предложений иконками сетей кофеен. Если изображение известно, оно должно быть показано всюду, где происходит работа с предложением конкретной кофейни.



Результаты поиска с иконкой кофейни

Теперь предположим, что разработчик также переопределил внешний вид всех кнопок в SDK, добавив иконки действий.



Панель показа предложения с иконками действий

Возникает вопрос: если выбрано предложение сетевой кофейни, какая иконка должна быть на кнопке подтверждения заказа — та, что унаследована из данных предложения (логотип кофейни) или та, что унаследована от «рода занятий» самой кнопки? Элемент управления «создать заказ», таким образом, встроен в две иерархии сущностей [по визуальному отображению и по данным (семантическую)] и в равной степени наследует обоим.

Можно легко продемонстрировать, как пересечение нескольких предметных областей в одном объекте быстро приводит к крайне запутанной и неочевидной логике. Поскольку те же соображения справедливы и для кнопки «Показать на карте», вроде бы очевидно, что по умолчанию более частные свойства должны побеждать более общие, т.е. тип кнопки должен быть приоритетнее какой-то абстрактной «иконки» в данных.

Но на этом история не заканчивается. Если разработчик всё-таки хочет именно этого, т.е. показывать иконку сети кофеен (если она есть) на кнопке создания заказа — как ему это сделать? Из той же логики, нам необходимо предоставить ещё более частную возможность такого переопределения. Например, представим себе следующую функциональность: если в данных предложения есть поле `createOrderButtonIconUrl`, то иконка будет взята из этого поля. Тогда разработчик сможет кастомизировать кнопку заказа, подменив в данных поле `createOrderButtonIconUrl` для каждого результата поиска:

```
let searchBox = new SearchBox({
  // Предположим, что мы разрешили
  // переопределять поисковую функцию
  searchFunction: function (params) {
    let res = await api.search(params);
    res.forEach(function (item) {
      item.createOrderButtonIconUrl =
        <URL нужной иконки>;
    });
    return res;
});
```

Формально этот подход корректен и никаких рекомендаций не нарушает. Но с точки зрения связности кода, его читабельности — это полная катастрофа, поскольку следующий разработчик, которого попросят заменить иконку `кнопке`, очень вряд ли пойдёт читать код *функции поиска предложений*.

Если бы возможность кастомизации вообще не предоставлялась, эту функциональность было бы гораздо проще поддерживать. Да, разработчики были бы не рады необходимости разработать с нуля собственную панель поиска просто для замены иконки. Но в их коде замена иконки хотя бы будет находиться в *ожидаемом* месте — где-то в функции рендеринга панели.

NB: существует много других возможностей позволить разработчику кастомизировать кнопку, запрятанную где-то глубоко в дебрях компонента: разрешить dependency injection или переопределение фабрик субкомпонентов, предоставить прямой доступ к отрендеренному представлению компонента, настроить пользовательские макеты кнопок и так далее. Все они страдают от той же проблемы: крайне сложно консистентно описать порядок и приоритет применения инъекций / обработчиков событий рендеринга / пользовательских шаблонов.

С решением вышеуказанных проблем, увы, всё обстоит очень сложно. В следующих главах мы рассмотрим паттерны проектирования, позволяющие в том числе разделить области ответственности составляющих компонента; но очень важно уяснить одну важную мысль: полное разделение, то есть разработка функционального SDK+UI, дающего разработчику свободу в переопределении и внешнего вида, и бизнес-логики, и UX компонентов — невероятно дорогая в разработке задача, которая в лучшем случае устроит вашу иерархию абстракций. Универсальный совет здесь ровно один: *три раза подумайте прежде чем предоставлять возможность программной настройки UI-компонентов*. Хотя цена ошибки дизайна программных интерфейсов для UI-библиотек, как правило, не очень высока (вряд ли клиент потребует рефанд из-за неработающей анимации нажатия кнопки), плохо структурированный, нечитабельный и глючный SDK вряд ли может рассматриваться как сильное клиентское преимущество вашего API.

Примечания

¹ Lepinsky, R. Google and Apple Versus Your Company's Application

<https://rodgersnotes.wordpress.com/2010/10/25/google-and-apple-versus-your-companys-application/>

Глава 44. Декомпозиция UI-компонентов

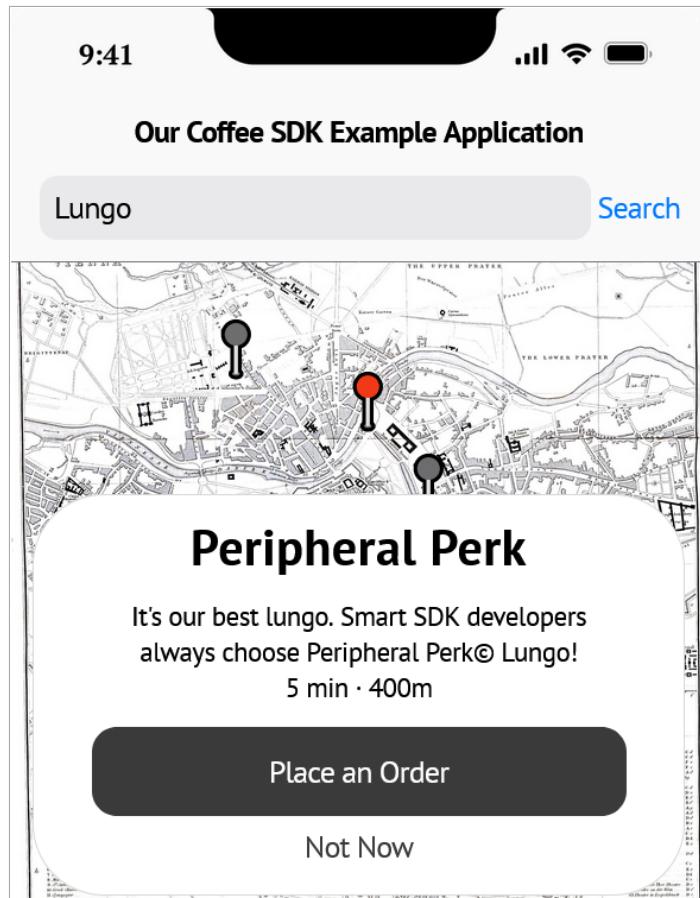
Перейдём к более предметному разговору и попробуем объяснить, почему требование возможности замены одной из подсистем компонента приводит к кратному усложнению интерфейсов. Продолжим рассматривать пример компонента `SearchBox` из предыдущей главы. Напомним, что мы выделили следующие факторы, осложняющие проектирование API визуальных компонентов:

- объединение в одном объекте разнородной функциональности, а именно — бизнес-логики, настройки внешнего вида и поведения компонента;
- появление разделяемых ресурсов, т.е. состояния объекта, которое могут одновременно читать и модифицировать разные акторы (включая конечного пользователя);
- неоднозначность иерархий наследования свойств и опций компонентов.

Сделаем задачу более конкретной, и попробуем разработать наш `SearchBox` так, чтобы он допускал следующие модификации:

1. Замена списочного представления предложений, например, на представление в виде карты с подсвечиваемыми метками:

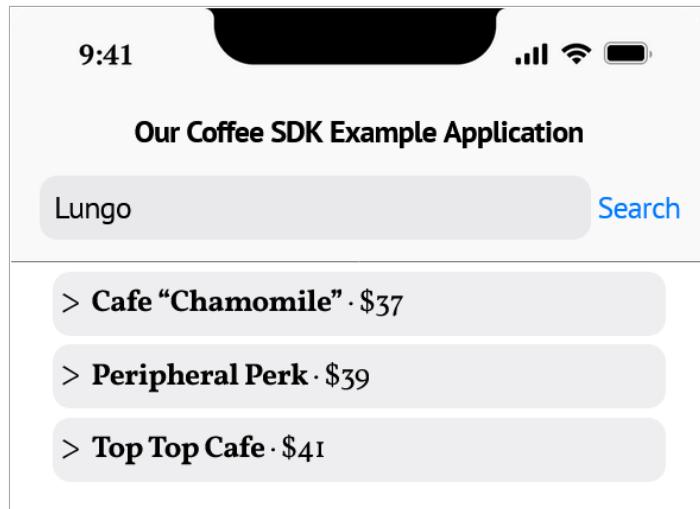
- иллюстрирует проблему полной замены одного субкомпонента (списка заказов) при сохранении поведения и дизайна остальных частей системы, а также сложности имплементации разделяемого состояния;



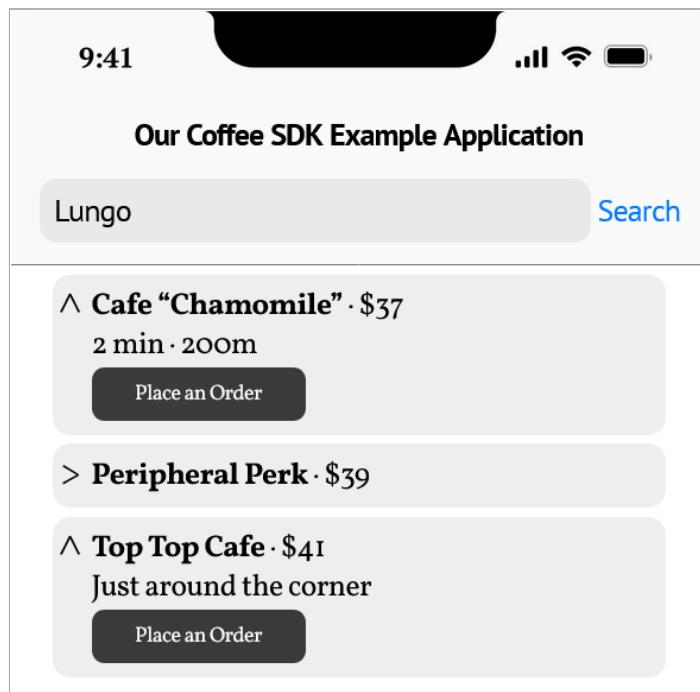
Результаты поиска на карте

2. Комбинирование краткого и полного описания предложения в одном интерфейсе (предложение можно развернуть прямо в списке и сразу сделать заказ):

- иллюстрирует проблему полного удаления одного из субкомпонентов с передачей его бизнес-логики другим частям системы;

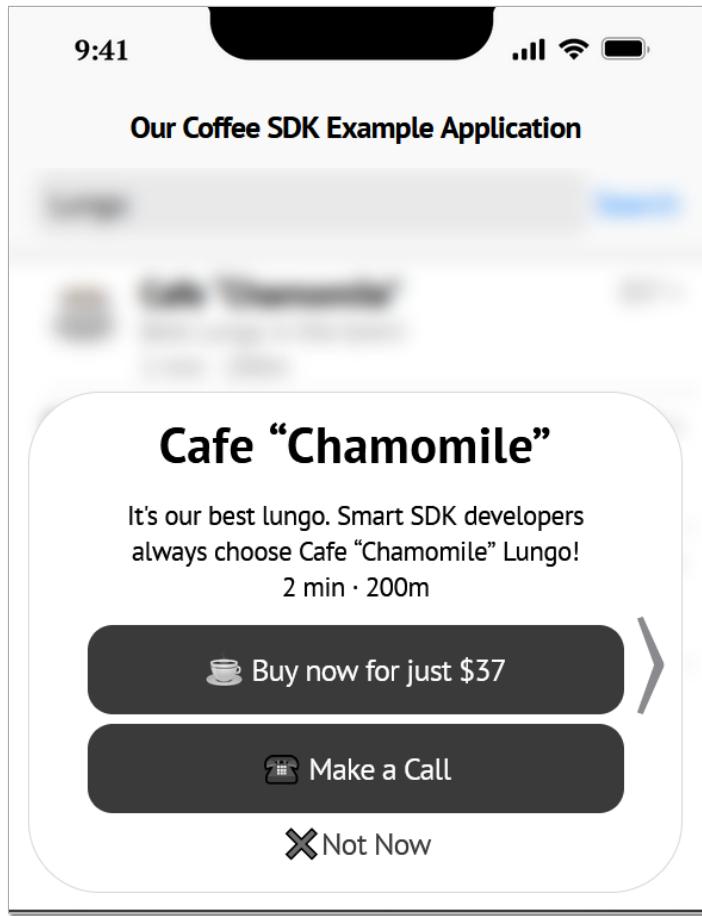


Список результатов поиска с короткими описаниями предложений



Список результатов поиска, в котором некоторые предложения развернуты

3. Манипуляция доступными действиями для предложения через добавление новых кнопок (вперёд, назад, позвонить) и управление их содержимым.

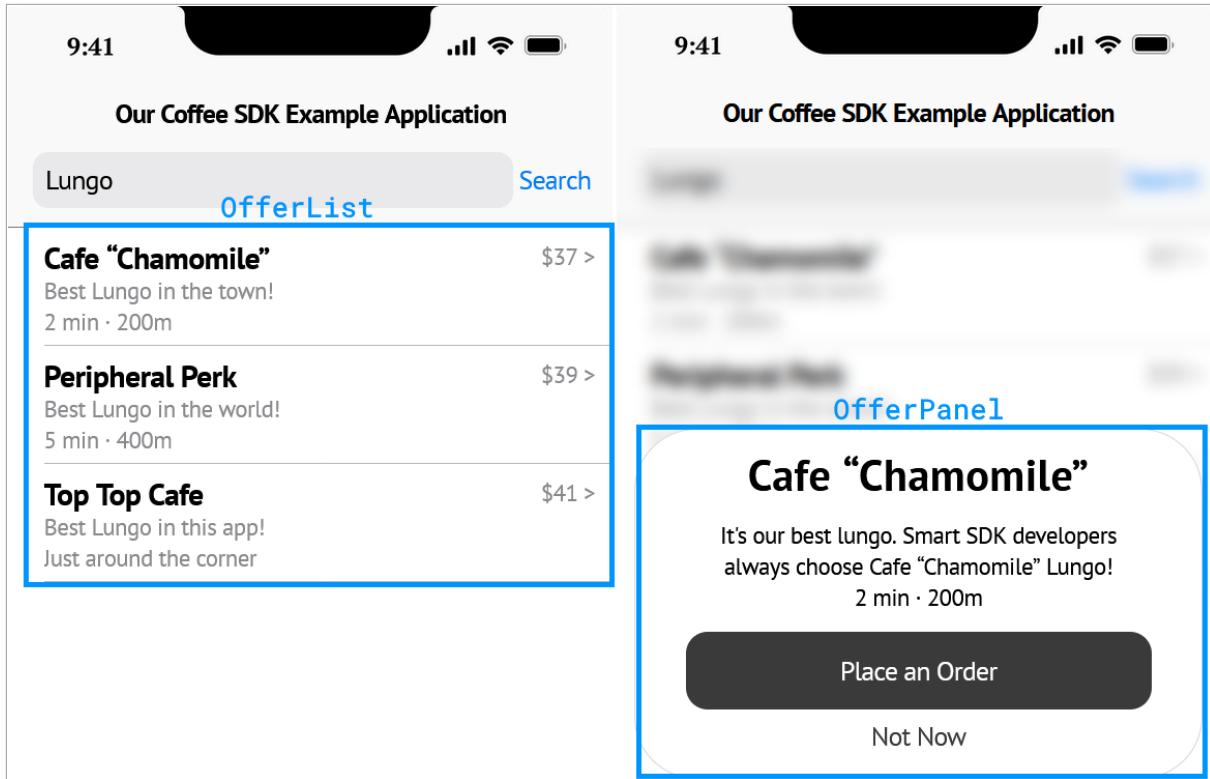


Панель предложения с дополнительными кнопками и иконками

В этом сценарии мы рассматриваем различные цепочки пропагирования информации и настройки до панели предложения и динамическое построение UI на их основе:

- часть данных является свойствами реального объекта (логотип, номер телефона), полученными из API поиска предложений;
- часть данных имеет смысл только в рамках конкретного UI и отражает механику его построения (кнопки «Вперёд» и «Назад»);
- часть данных (иконки отмены и звонка) связаны с типом кнопки (бизнес-логикой, которую она несёт в себе).

Решение, которое напрашивается в данной ситуации — это выделение двух дополнительных компонентов, отвечающих за представление списка предложений и за панель показа конкретного предложения, назовём их `OfferList` и `OfferPanel`.



'SearchBox' и его суб-компоненты

В случае отсутствия требований кастомизации, псевдокод, имплементирующий взаимодействие всех трёх компонентов, выглядел бы достаточно просто:

```
class SearchBox implements ISearchBar {
    // Ответственность `SearchBox`:
    // 1. Создать контейнер для визуального
    // отображения списка предложений,
    // сгенерировать опции и создать
    // сам компонент `OfferList`.
    constructor(container, options) {
        ...
        this.offerList = new OfferList(
            this,
            offerListContainer,
            offerListOptions
        );
    }
    // 2. Выполнять поиск предложений
    // при нажатии пользователем на кнопку поиска
    // и предоставлять аналогичный программный
    // интерфейс для разработчика
    onSearchButtonClick() {
        this.search(this.searchInput.value);
    }
    search(query) {
        ...
    }
    // 3. При получении новых результатов поиска
    // оповестить об этом
    onSearchResultsReceived(searchResults) {
        ...
        this.offerList.setOfferList(searchResults)
    }
    // 4. Создавать заказы (и выполнять нужные
    // операции над компонентами)
    createOrder(offer) {
        this.offerList.destroy();
        ourCoffeeSdk.createOrder(offer);
        ...
    }
    // 5. Самоуничтожаться
    destroy() {
        this.offerList.destroy();
    }
}
```

```

class OfferList implements IOfferList {
    // Ответственность OfferList:
    // 1. Создать контейнер для визуального
    // отображения панели предложений,
    // сгенерировать опции и создать
    // сам компонент `OfferPanel`
    constructor(searchBox, container, options) {
        ...
        this.offerPanel = new OfferPanel(
            searchBox,
            offerPanelContainer,
            offerPanelOptions
        );
        ...
    }
    // 2. Предоставлять метод для изменения
    // списка предложений
    setOfferList(offerList) { ... }
    // 3. При выборе предложения, вызывать метод
    // его показа в панели предложений
    onOfferClick(offer) {
        this.offerPanel.show(offer)
    }
    // 4. Самоуничтожаться
    destroy() {
        this.offerPanel.destroy();
    }
    ...
}

```

```

class OfferPanel implements IOfferPanel {
    constructor(searchBox, container, options) { ... }
    // Ответственность панели показа предложения:
    // 1. Собственно, показывать предложение
    show(offer) {
        this.offer = offer;
        ...
    }
    // 2. Создавать заказ по нажатию на кнопку
    // создания заказа
    onCreateOrderButtonClick() {
        this.searchBox.createOrder(this.offer);
    }
    // 3. Закрываться по нажатию на кнопку
    // отмены
    onCancelButtonClick() {
        ...
    }
    // 4. Самоуничтожаться
    destroy() { ... }
}

```

Интерфейсы ISearchBar / IOfferPanel / IOfferView также очень просты (конструкторы и деструкторы опущены):

```

interface ISearchBar {
    search(query);
    createOrder(offer);
}
interface IOfferList {
    setOfferList(offerList);
}
interface IOfferPanel {
    show(offer);
}

```

Если бы мы не разрабатывали SDK и у нас не было бы задачи разрешать кастомизацию этих компонентов, подобная реализация была бы стопроцентно уместной. Попробуем, однако, представить, как мы будем решать описанные выше задачи:

1. Показ списка предложений на карте: на первый взгляд, мы можем разработать альтернативный компонент показа списка предложений, скажем, OfferMap, который сможет использовать стандартную панель предложений. Но у нас есть одна проблема: если OfferList только отправляет команды для OfferPanel, то OfferMap должен ещё и получать обратную связь — событие закрытия панели, чтобы убрать выделение с метки. Наш интерфейс подобного не предусматривает. Имплементация этой функциональности не так и проста:

```

class CustomOfferPanel extends OfferPanel {
    constructor(
        searchBar, offerMap, container, options
    ) {
        super(searchBar, container, options);
        this.offerMap = offerMap;
    }
    onCancelButtonClick() {
        offerMap.resetCurrentOffer();
        super.onCancelButtonClick();
    }
}
class OfferMap implements IOfferList {
    constructor(searchBar, container, options) {
        ...
        this.offerPanel = new CustomOfferPanel(
            this,
            searchBar,
            offerPanelContainer,
            offerPanelOptions
        )
    }
    resetCurrentOffer() { ... }
}

```

Нам пришлось создать новый класс `CustomOfferPanel`, который, в отличие от своего родителя, теперь работает не с любой реализацией интерфейса `IOfferList`, а только с `IOfferMap`.

2. Полные описания и кнопки действий в самом списке заказов — в этом случае всё достаточно очевидно: мы можем добиться нужной функциональности только созданием собственного компонента. Даже если мы предоставим метод переопределения внешнего вида элемента списка для стандартного компонента `OfferList`, он всё равно продолжит создавать `OfferPanel` и открывать его по выбору предложения.
3. Для реализации новых кнопок мы можем только лишь предложить программисту реализовать свой список предложений (чтобы предоставить методы выбора предыдущего / следующего предложения) и свою панель предложений, которая эти методы будет вызывать. Даже если мы придумаем какой-нибудь простой способ кастомизировать, например, текст кнопки «Сделать заказ», его поддержка всё равно будет ответственностью компонента `OfferList`:

```
let searchBar = new SearchBox(..., {
  offerPanelCreateOrderButtonText:
    'Drink overpriced coffee!'
});

class OfferList {
  constructor(..., options) {
    ...
    // Вычленять из опций `SearchBox`
    // настройки панели предложений
    // вынужден конструктор класса
    // `OfferList`
    this.offerPanel = new OfferPanel(..., {
      createOrderButtonText: options
        .offerPanelCreateOrderButtonText
    })
  }
}
```

Неприятная особенность всех вышеперечисленных решений — их очень плохая расширяемость. Вернёмся к п.1: допустим, мы решили сделать функциональность реакции списка предложений на закрытие панели предложений частью интерфейса, чтобы программист мог ей воспользоваться. Для этого нам придётся объявить новый метод, который в целях обратной совместимости будет необязательным:

```
interface IOfferList {  
    ...  
    onOfferPanelClose?();  
}
```

и писать в коде OfferPanel что-то типа:

```
if (Type(this.offerList.onOfferPanelClose)  
    == 'function') {  
    this.offerList.onOfferPanelClose();  
}
```

Что, во-первых, совершенно не красит наш код и, во-вторых, делает связность OfferPanel и OfferList ещё более сильной.

Как мы описывали ранее в главе «[Слабая связность](#)», избавиться от такого рода проблем мы можем, если перейдём от сильной связности к слабой, например, через генерацию событий вместо вызова методов. Компонент IOfferPanel мог бы бросать событие 'close', и тогда OfferList мог бы на него подписаться:

```
class OfferList {  
    setup() {  
        ...  
        this.offerPanel.events.on(  
            'close',  
            function () {  
                this.resetCurrentOffer();  
            }  
        )  
    }  
    ...  
}
```

Код выглядит более разумно написанным, но никак не уменьшает взаимозависимость компонентов: использовать OfferList без OfferPanel, как этого требует сценарий #2, мы всё ещё не можем.

Заметим, что в вышеприведённых фрагментах кода налицо полный хаос с уровнями абстракции: OfferList инстанцирует OfferPanel и управляет ей напрямую. При этом OfferPanel приходится перепрыгивать через уровни, чтобы создать заказ. Мы можем попытаться разомкнуть эту связь, если начнём маршрутизировать потоки команд через сам SearchBox, например, так:

```

class SearchBox() {
  constructor() {
    this.offerList = new OfferList(...);
    this.offerPanel = new OfferPanel(...);
    this.offerList.events.on(
      'offerSelect', function (offer) {
        this.offerPanel.show(offer);
      }
    );
    this.offerPanel.events.on(
      'close', function () {
        this.offerList
          .resetSelectedOffer();
      }
    );
  }
}

```

Теперь `OfferList` и `OfferPanel` стали независимы друг от друга, но мы получили другую проблему: для их замены на альтернативные имплементации нам придётся переписать сам `SearchBox`. Мы можем абстрагироваться ещё дальше, поступив вот так:

```

class SearchBox {
  constructor() {
    ...
    this.offerList.events.on(
      'offerSelect', function (event) {
        this.events.emit('offerSelect', {
          offer: event.selectedOffer
        });
      }
    );
    ...
  }
}

```

То есть заставить `SearchBox` транслировать события, возможно, с преобразованием данных. Мы даже можем заставить `SearchBox` транслировать *любые* события дочерних компонентов, и, таким образом, прозрачным образом расширять функциональность, добавляя новые события. Но это совершенно очевидно не ответственность высокоуровневого компонента — состоять, в основном, из кода трансляции событий. К тому же, в этих цепочках событий очень легко запутаться. Как, например, должна быть реализована

функциональность выбора следующего предложения в offerPanel (п. 3 в нашем списке улучшений)? Для этого необходимо, чтобы OfferList не только генерировал сам событие 'offerSelect', но и прослушивал это событие на родительском контексте и реагировал на него. В этом коде легко можно организовать бесконечный цикл:

```
class OfferList {
  constructor(searchBox, ...) {
    ...
    searchBox.events.on(
      'offerSelect',
      this.selectOffer
    );
  }

  selectOffer(offer) {
    ...
    this.events.emit(
      'offerSelect', offer
    );
  }
}
```

```
class SearchBox {
  constructor() {
    ...
    this.offerList.events.on(
      'offerSelect', function (offer) {
        ...
        this.events.emit(
          'offerSelect', offer
        );
      }
    );
  }
}
```

Во избежание таких циклов мы можем разделить события:

```

class SearchBox {
  constructor() {
    ...
    // `OfferList` сообщает о низкоуровневых
    // событиях, а `SearchBox` – о высокоуровневых
    this.offerList.events.on(
      'click', function (target) {
        ...
        this.events.emit(
          'offerSelect',
          target.dataset.offer
        )
      }
    )
  }
}

```

Но тогда код станет окончательно неподдерживаемым: для того, чтобы открыть панель предложения, нужно будет сгенерировать 'click' на инстанции класса OfferList.

Итого, мы перебрали уже как минимум пять разных вариантов организации декомпозиции UI-компонентта в самых различных парадигмах, но так и не получили ни одного приемлемого решения. Вывод, который мы должны сделать, следующий: проблема не в конкретных интерфейсах и не в подходе к решению. В чём же она тогда?

Давайте сформулируем, в чём состоит область ответственности каждого из наших компонентов:

1. SearchBox отвечает за предоставление общего интерфейса. Он является точкой входа и для пользователя, и для разработчика. Если мы спросим себя: какой максимально абстрактный компонент мы всё ещё готовы называть SearchBox-ом? Очевидно, некоторый UI для ввода поискового запроса и его отображения, а также какое-то абстрактное создание заказа по предложениям.
2. OfferList выполняет функцию показа пользователю какого-то списка предложений кофе. Пользователь может взаимодействовать со списком — просматривать его и «активировать» предложения (т.е. выполнять *какие-то* операции с конкретным элементом списка).

3. `OfferPanel` представляет одно конкретное предложение и отображает *всю* значимую информацию для пользователя. Панель предложения всегда ровно одна. Пользователь может взаимодействовать с панелью, активируя различные действия, связанные с этим конкретным предложением (включая создание заказа).

Следует ли из определения `SearchBox` необходимость наличия суб-компоненты `OfferList`? Никоим образом: мы можем придумать самые разные способы показа пользователю предложений. `OfferList` — частный случай, каким образом мы могли бы организовать работу `SearchBox`-а по предоставлению UI к результатами поиска.

Следует ли из определения `SearchBox` и `OfferList` необходимость наличия суб-компонента `OfferPanel`? Вновь нет: даже сама концепция существования какой-то *краткой* и *полной* информации о предложении (первая показана в списке, вторая в панели) никак не следует из определений, которые мы дали выше. Аналогично, ниоткуда не следует и наличие действия «выбор предложения» и вообще концепция того, что `OfferList` и `OfferPanel` выполняют *разные* действия и имеют *разные* настройки. На уровне `SearchBox` вообще не важно, *как* результаты поиска представлены пользователю и в каких *состояниях* может находиться соответствующий UI.

Всё это приводит нас к простому выводу: мы не можем декомпозировать `SearchBox` просто потому, что мы не располагаем достаточным количеством уровней абстракции и пытаемся «перепрыгнуть» через них. Нам нужен «мостик» между `SearchBox`, который не зависит от конкретной имплементации UI работы с предложениями и `OfferList/OfferPanel`, которые описывают конкретную концепцию такого UI. Введём дополнительный уровень абстракции (назовём его, скажем, «composer»), который позволит нам модерировать потоки данных:

```

class SearchBoxComposer
  implements ISearchBarComposer {
  // Ответственность `composer`-а состоит в:
  // 1. Создании собственного контекста
  // для дочерних компонентов
  constructor(searchBox, container, options) {
    ...
    // Контекст состоит из показанного списка
    // предложений (возможно, пустого) и
    // выбранного предложения (возможно, пустого)
    this.offerList = null;
    this.currentOffer = null;
    // 2. Создании конкретных суб-компонентов
    // и трансляции опций для них
    this.offerList = this.buildOfferList();
    this.offerPanel = this.buildOfferPanel();
    // 3. Управлении состоянием и оповещении
    // суб-компонентов о его изменении
    this.searchBox.events.on(
      'offerListChange', this.onOfferListChange
    );
    // 4. Прослушивании событий дочерних
    // компонентов и вызове нужных действий
    this.offerListComponent.events.on(
      'offerSelect', this.selectOffer
    );
    this.offerPanelComponent.events.on(
      'action', this.performAction
    );
  }
  ...
}

```

Здесь методы-строители подчинённых компонентов, позволяющие переопределять опции компонентов (и, потенциально, их расположение на экране) выглядят как:

```

class SearchBoxComposer {
  ...
  buildOfferList() {
    return new OfferList(
      this,
      this.offerListContainer,
      this.generateOfferListOptions()
    );
  }

  buildOfferPanel() {
    return new OfferPanel(
      this,
      this.offerPanelContainer,
      this.generateOfferPanelOptions()
    );
  }
}

```

Мы можем придать `SearchBoxComposer`-у функциональность трансляции любых контекстов. В частности:

1. Трансляцию данных и подготовку данных. На этом уровне мы можем предположить, что `OfferList` показывает краткую информацию о предложений, а `OfferPanel` — полную, и предоставить (потенциально переопределяемые) методы генерации нужных срезов данных:

```
class SearchBoxComposer {  
    ...  
    onContextOfferListChange(offerList) {  
        ...  
        // `SearchBoxComposer` транслирует событие  
        // `offerListChange` как `offerPreviewListChange`  
        // специально для компонента `OfferList`,  
        // таким образом, исключая возможность  
        // зацикливания, и подготавливает данные  
        this.events.emit('offerPreviewListChange', {  
            offerList: this.generateOfferPreviews(  
                this.offerList,  
                this.contextOptions  
            )  
        });  
    }  
}
```

2. Логику управления собственным состоянием (в нашем случае полем `currentOffer`):

```
class SearchBoxComposer {  
    ...  
    onContextOfferListChange(offerList) {  
        // Если в момент ввода нового поискового  
        // запроса пользователем показано какое-то  
        // предложение, его необходимо скрыть  
        if (this.currentOffer !== null) {  
            this.currentOffer = null;  
            // Специальное событие для  
            // компонента `OfferPanel`  
            this.events.emit(  
                'offerFullViewToggle',  
                { offer: null }  
            );  
        }  
    }  
}
```

3. Логику преобразования действий пользователя на одном из субкомпонентов в события или действия над другими компонентами или родительским контекстом:

```

class SearchBoxComposer {
    ...
    public performAction({
        action, offerId
    }) {
        switch (action) {
            case 'createOrder':
                // Действие «создать заказ»
                // нужно оттранслировать `SearchBox`-у
                this.createOrder(offerId);
                break;
            case 'close':
                // Действие «закрытие панели предложения»
                // нужно распространить для всех
                if (this.currentOffer != null) {
                    this.currentOffer = null;
                    this.events.emit(
                        'offerFullViewToggle', { offer: null }
                    );
                }
                break;
        }
    }
}

```

Если теперь мы посмотрим на кейсы, описанные в начале главы, то мы можем наметить стратегию имплементации каждого из них:

1. Показ компонентов на карте не меняет общую декомпозицию компонентов на список и панель. Для реализации альтернативного IOfferList-а нам нужно переопределить метод buildOfferList так, чтобы он создавал наш кастомный компонент с картой.
2. Комбинирование функциональности списка и панели меняет концепцию, поэтому нам необходимо будет разработать собственный ISearchBoxComposer. Но мы при этом сможем использовать стандартный OfferList, поскольку Composer управляет и подготовкой данных для него, и реакцией на действия пользователей.
3. Обогащение функциональности панели не меняет общую декомпозицию (значит, мы сможем продолжать использовать стандартный SearchBoxComposer и OfferList), но нам нужно переопределить подготовку данных и опций при открытии панели, и реализовать дополнительные события и действия, которые SearchBoxComposer транслирует с панели предложения.

Ценой этой гибкости является чрезвычайное усложнение взаимодействия. Все события и потоки данных должны проходить через цепочку таких `Composer`-ов, удлиняющих иерархию сущностей. Любое преобразование (например, генерация опций для вложенного компонента или реакция на события контекста) должно быть параметризуемым. Мы можем подобрать какие-то разумные хелперы для того, чтобы пользоваться такими кастомизациями было проще, но никак не сможем убрать эту сложность из кода нашего SDK. Таков путь.

Пример реализации компонентов с описанными интерфейсами и имплементацией всех трёх кейсов вы можете найти в репозитории настоящей книги:

- исходный код доступен на www.github.com/twirl/The-API-Book/docs/examples
 - там же предложены несколько задач для самостоятельного изучения;
- песочница с «живыми» примерами доступна на twirl.github.io/The-API-Book.

Глава 45. MV*-фреймворки

Очевидным способом сделать менее сложными многослойные схемы, подобные описанным в предыдущей главе, является ограничение возможных путей взаимодействия между компонентами. Как мы описывали в главе «Слабая связность», мы могли бы упростить код, если бы разрешили нижележащим субкомпонентам напрямую вызывать методы вышестоящих сущностей. Например, так:

```
class SearchBoxComposer
  implements ISearchBarComposer {
  ...
  protected context: ISearchBar;
  ...
  public createOrder(offerId: string) {
    const offer = this.findOfferById(offerId);
    if (offer !== null) {
      // Вместо вызова
      // this.events.emit(
      //   'createOrder', { offer });
      this.context
        .createOrder(offer);
    }
  }
}
```

Кроме того, мы можем убрать Composer из цепочки подготовки данных так, чтобы подчинённые компоненты напрямую получали нужные поля напрямую из SearchBox:

```

class OfferListComponent
  implements IOfferListComponent {
  ...
  protected context: SearchBox;
  ...
  constructor () {
    ...
    // Список заказов напрямую
    // получает нужные данные
    // и оповещения из `SearchBox`:
    this.context.events.on(
      'offerListChange',
      () => {
        this.show(
          context.getOfferList()
        );
      }
    );
  ...
}

```

Тем самым мы утратили возможность подготавливать данные для их показа в списке и, тем самым, возможность встраивать их в любого родителя через соответствующий Composer. Но при этом мы сохранили возможность свободно использовать альтернативные реализации компонентов панели, поскольку реакция на взаимодействие пользователя с интерфейсом находится всё ещё находится под контролем Composer-а. Как бонус мы получили отсутствие двусторонних взаимодействий между тремя нашими сущностями:

- субкомпоненты *читают* состояние SearchBox-а, но не модифицируют его;
- Composer *получает оповещения* о взаимодействии пользователя с UI, но никак не влияет на сам UI;
- наконец, SearchBox никак не взаимодействует ни с тем, ни с другим — только лишь предоставляет контекст, методы его изменения и нотификации.

Сделав подобное упрощение, мы фактически получили компонент, следующий методологии «Model-View-Controller» (MVC)^I: OfferList и OfferPanel (а также код показа поля ввода) — это различные view, которые непосредственно наблюдает пользователь и взаимодействует с ними; Composer — это controller, который получает события от view и модифицирует модель (сам SearchBox).

NB: следуя букве подхода, мы должны выделить из SearchBox компонент-модель, который отвечает только за данные. Это упражнение мы оставим читателю.

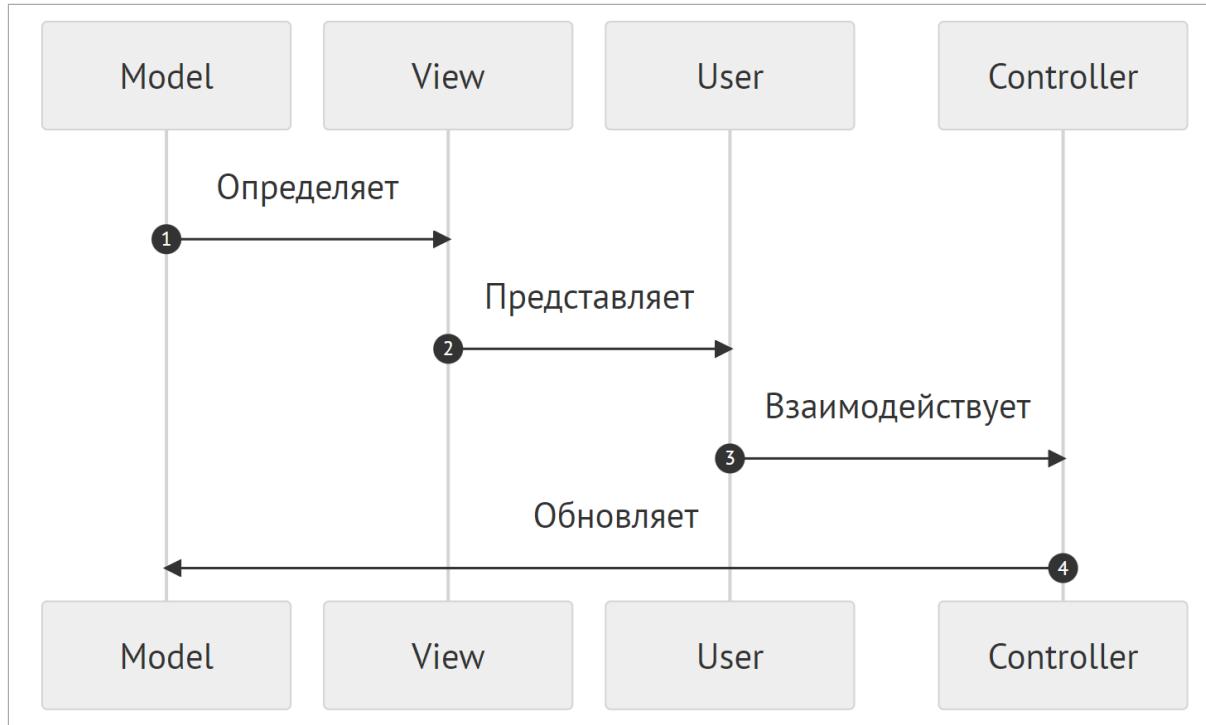


Схема взаимодействия компонентов в MVC

Если мы выберем другие направления редукции полного взаимодействия, мы получим другие варианты MV*-фреймворков (Model-View-ViewModel, Model-View-Presenter и т.д.) Все они, в конечно счёте, основаны на паттерне «модель», который мы обсудим ниже.

Паттерн «модель»

Общая черта, объединяющая все MV*-фреймворки — это требование к сущности «модель» (некоторого набора данных) *полностью детерминировано определять* внешний вид и состояние UI-компонента. Изменения в модели порождают и изменения в отображении компонента (или дерева компонентов; в некоторых подходах модель может быть одной на всё приложение и полностью определять весь интерфейс). В то же время визуальные представления не могут влиять на модель напрямую, так как им разрешено взаимодействовать только с контроллером.

SDK, реализованный в MV*-парадигмах, в теории получает несколько важных свойств:

- Принудительное разделение уровней абстракции, поскольку постулируется (но далеко не всегда выполняется, см. ниже), что модель содержит *семантически высоконивневые данные*.
- Практически исключены циклы в обработке событий, поскольку контроллер должен реагировать только на взаимодействие пользователя или разработчика с view, но не на изменения модели.
 - Дополнительно, события изменения состояния модели обычно генерируются только в том случае, если состояние действительно изменилось (новое значение поля не совпадает со старым), и, таким образом, чтобы зациклить обработку события, система должна бесконечно осциллировать между двумя разными состояниями, что достаточно сложно допустить случайно.
- controller транслирует низкоуровневые события (взаимодействие пользователя с view) в высокоуровневые, тем самым предоставляя нужную глубину уровней абстракции и позволяя полностью менять UI при сохранении бизнес-логики;
- Данные модели полностью определяют состояние системы, что очень удобно при реализации такой сложной функциональности как восстановление приложения в случае сбоя, совместное редактирование, отмена последних действий и т.д.

- Один из частных случаев использования этого свойства — сериализация модели в виде URL (или App Links в случае мобильных приложений). Тогда URL полностью определяет состояние приложения, и любые изменения состояния отражаются в виде изменений URL. Этот подход чрезвычайно удобен тем, что можно сгенерировать специальные ссылки, открывающие нужный экран в приложении.

Иными словами, MV*-фреймворки представляют собой жёсткий шаблон, который помогает писать качественный код и не запутываться в потоках данных.

Однако эта жёсткость влечёт за собой недостатки. Если задаться целью *полностью* описать состояние компонента, то мы обязаны внести в него и такие данные, как выполняющиеся сейчас анимации и даже процент их выполнения. Таким образом, модель обязана будет содержать в себе все данные всех уровней абстракции и, более того, каким-то образом включать в себя две или более иерархий подчинения (по семантической и визуальной иерархиям, а также, возможно, вычисляемые значения опций). В нашем примере это означает, например, что модель должна будет хранить и `currentSelectedOffer` для `OfferPanel`, и список показанных кнопок, и даже вычисленные значения иконок для кнопок.

Подобная полная модель представляет собой проблему не только теоретически и семантически (перемешивание в одной сущности разнородных данных), но и в практическом смысле — сериализация таких моделей окажется ограничена рамками конкретной версии API или приложения (поскольку они содержат все внутренние переменные, включая непубличные). Если мы в следующей версии изменим реализацию субкомпонентов, то старые ссылки и закэшированные состояния перестанут работать (либо нам потребуется держать слой совместимости, описывающий, как интерпретировать модели предыдущих версий).

Другая идеологическая проблема подхода — организация вложенных контроллеров. В системе с дочерними субкомпонентами все те проблемы, которые решил MV*-подход, возвращаются на новом уровне: нам придётся разрешить вложенным контроллерам либо перепрыгивать через уровни абстракции и модифицировать корневую модель, либо вызывать методы

контроллеров родительских компонент. Оба подхода влекут за собой сильную связность сущностей и требуют очень аккуратного проектирования, иначе переиспользование компонентов окажется затруднено.

Если мы внимательно посмотрим на современные UI библиотеки, выполненные в MV*-парадигмах, то увидим, что они следуют парадигме весьма нестрогого и лишь заимствуют из неё основной принцип: модель полностью определяет внешний вид компонента, и любые визуальные изменения инициируются через изменение модели контроллером. Дочерние компоненты при этом обычно имеют собственные модели (часто в виде подмножества родительской модели, дополненного собственным состоянием компонента), и в глобальной модели приложения находится только ограниченный набор полей. Этот подход адаптирован во многих современных UI-фреймворках, даже тех, которые от MV*-парадигм откращиваются (например, React² ³).

На эти же проблемы MVC-подхода обращает внимание в своём эссе Мартин Фаулер в своём эссе «Архитектуры GUI»⁴, и предлагает решение в виде фреймворка *Model-View-Presenter* (MVP), в котором место controller-а занимает сущность-presenter, в обязанности которой входит не только трансляция событий, но и подготовка данных для view, что позволяет разделить уровни абстракции (модель хранит только семантические данные, описывающие предметную область, presenter — низкоуровневые данные, описывающие UI, в терминологии Фаулера — application model или presentation model).

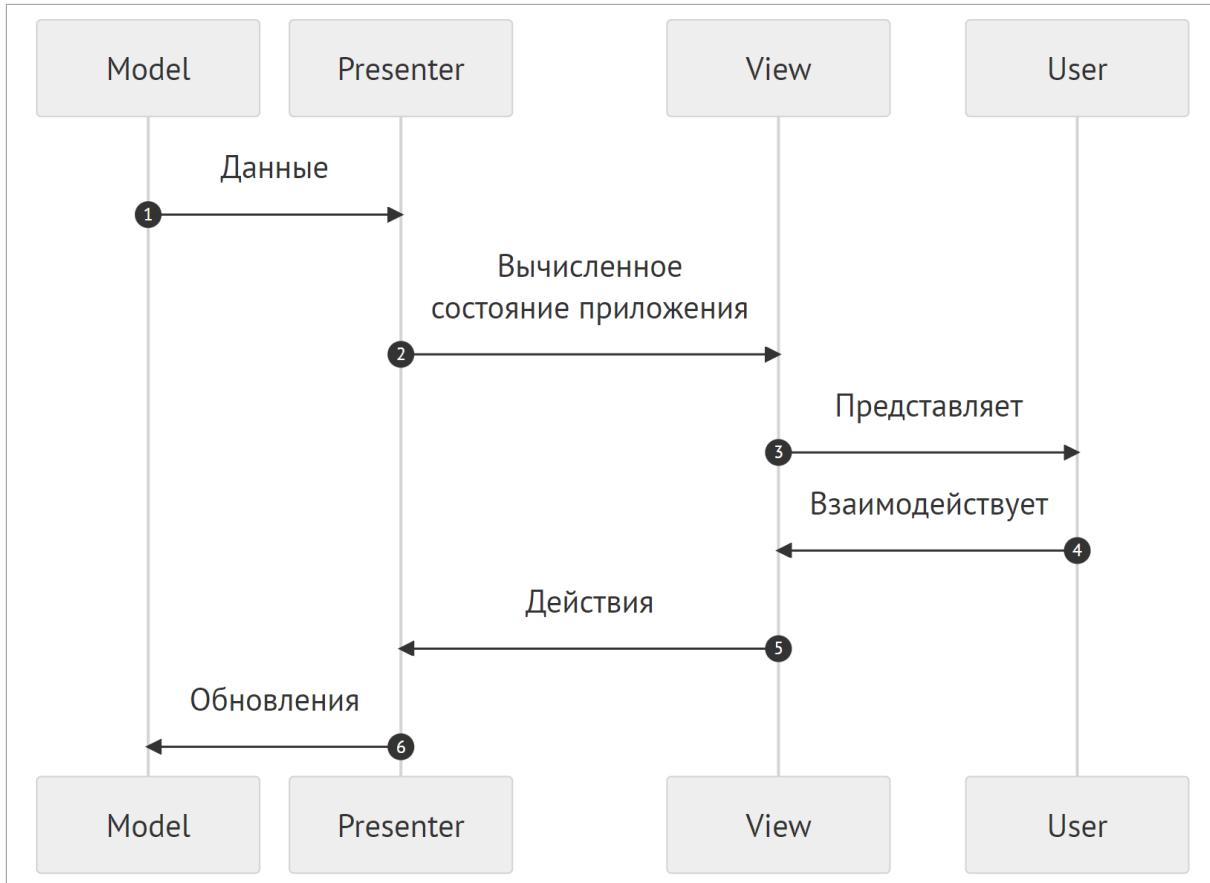


Схема взаимодействия компонентов в MVP

Предложенная Фаулером парадигма во многом схожа с концепцией Composer-a, которую мы обсуждали в предыдущей главе, но с одним заметным различием. По мысли Фаулера собственного состояния у presenter-а нет (за исключением, возможно, кэшей и замыканий), он только вычисляет данные, необходимые для показа визуального интерфейса, из данных модели. Если необходимо манипулировать каким-то низкоуровневым свойством, например, цветом текста в интерфейсе, то нужно разработать модель так, чтобы цвет текста вычислялся presenter-ом из какого-то высокоДуровневого поля в модели (возможно, искусственно введённого), что ограничивает возможности альтернативных реализаций субкомпонентов.

NB: на всякий случай уточним, что автор этой книги не предлагает Composer как альтернативную MV*-методологию. Идея предыдущей главы состоит в том, что сложные сценарии декомпозиции UI-компонентов решаются *только* искусственным введением мостиков-уровней абстракции. Неважно, как мы этот мостик назовём и какие правила для него придумаем.

Примечания

¹ MVC

<https://en.wikipedia.org/wiki/Model%E2%80%93view%E2%80%93controller>

² Why did we build React?

<https://legacy.reactjs.org/blog/2013/06/05/why-react.html>

³ Mattiazz, R. How React and Redux brought back MVC and everyone loved it

<https://rangle.io/blog/how-react-and-redux-brought-back-mvc-and-everyone-loved-it>

⁴ Fowler, M. GUI Architectures

<https://www.martinfowler.com/eaaDev/uiArchs.html>

Глава 46. Backend-Driven UI

Другой способ обойти сложность «переброса мостов» между несколькими предметными областями, которые нам приходится сводить в рамках одного UI-компоненты — это убрать одну из них. Как правило, речь идёт о бизнес-логике: мы можем разработать компоненты полностью абстрактными, и скрыть все трансляции UI-событий в полезные действия вне контроля разработчика.

В такой парадигме код поиска предложений выглядел бы так:

```
class SearchBox {  
    ...  
    search(query) {  
        const markup = await api.search(query);  
        this.render(markup);  
    }  
    ...  
}
```

То есть вместо получения машиночитаемых результатов поиска, мы получаем с сервера их готовое представление в виде кода на HTML либо другого (возможно, разработанного специально для нашего SDK) языка разметки.

Этот подход можно обобщить ещё сильнее:

```
class SearchBox {  
    constructor (...) {...}  
  
    stateChange (patch) {  
        // Получаем от сервера  
        // список действий, которые  
        // необходимо выполнить при  
        // запрошенном изменении  
        let actions = await api  
            .getActions(  
                this.model,  
                patch  
            );  
        // Применяем действия  
        ...  
    }  
}
```

В этом примере подразумевается, что SearchBox не содержит вообще никакой другой логики, кроме отсылки на сервер событий, с ним произошедших (действий пользователя или разработчика) и показа присланного с сервера контента.

(Примером реализации этой идеи можно считать т.н. «Web I.O» — сервер присыпает готовый контент страницы, а вся интерактивность сводится к переходам по ссылкам.)

Хотя Backend-Driven UI обладает очевидным недостатком в виде требовательности к качеству соединения с сервером, у него есть два очень важных преимущества:

- возможность с сервера регулировать поведение клиента, в том числе устранять возможные ошибки в бизнес-логике в реальном времени без необходимости обновлять приложение;
- возможность сэкономить на разработке консистентной и читабельной номенклатуры сущностей публичного SDK, ограничившись минимальным набором доступной функциональности.

Тем не менее, мы не можем не отметить: при том, что любая крупная ИТ-компания проходит через эту фазу — разработки Backend-Driven UI (они же — «тонкие клиенты») для своих приложений или своих публичных SDK — мы не знаем ни одного заметного на рынке продукта, разработанного в этой парадигме (кроме, разве что, протоколов удалённых терминалов), хотя во многих случаях возникающими сетевыми задержками вполне можно было бы пренебречь. Рискнём сказать, что причины сложившегося положения вещей таковы:

- разработать серверный код управления UI ничуть не проще, нежели клиентский;
- современные клиентские устройства предоставляют широкий спектр функциональности, доступной только клиентскому разработчику, от кэшей до анимаций;
 - гибридный код, который частично получает состояние с сервера, частично обогащает его на клиенте, писать гораздо сложнее, чем чистый клиентский (в силу того, что в Backend-Driven UI содержимое ответа сервера для клиента представляет собой «чёрный ящик», для взаимодействия с которым приходится изобретать дополнительные

протоколы);

- отказ от гибридного кода, во-первых, сужает возможности приложения и, во-вторых, фактически переводит клиентское устройство в режим «облачного гейминга», что на сегодня очень дорого и не очень удобно для пользователя;
- сложившая практика такова, что разработка бэкенда и клиентского кода — разные специализации, требующие разной экспертизы и разных подходов к написанию кода.

NB: следует различать Backend-Driven UI и технологию серверного рендеринга (SSR). Второе подразумевает, что одно и то же состояние UI (как правило, в виде HTML-разметки или аналогичного декларативного описания) может быть сгенерировано как сервером, так и клиентом. В SSR, однако, подразумевается, что клиент может интерпретировать ответ сервера и извлекать из него семантические данные.

С точки зрения предложения SDK сторонним разработчикам, Backend-Driven UI фактически вынуждает разрабатывать гибридный код (поскольку практика предоставления партнёрам возможности встроиться в функции серверного рендеринга на стороне провайдера API выглядит весьма далёкой от жизнеспособности идеей), и, таким образом, будет страдать от той же проблемы непрозрачности — разработчик не может со своей стороны определить, в каком сейчас состоянии находится визуальный компонент. Как ни странно, этот недостаток одновременно является и достоинством, поскольку вендор API сохраняет возможность на своей стороне производить любые манипуляции с контентом без нарушения обратной совместимости (мы обсудим этот вопрос в главе «[Линейка сервисов API](#)»). Таких клиентских API в мире существует довольно много (в первую очередь, разнообразные виджеты, рекламные API и т.д.), но их сложно назвать полноценными SDK.

Глава 47. Разделяемые ресурсы и асинхронные блокировки

Другой важный паттерн, который мы должны рассмотреть — это доступ к общим ресурсам. Предположим, что в нашем учебном приложении открытие экрана предложения стало требовать выполнения дополнительного запроса к серверу и, таким образом, стало асинхронным. Модифицируем код OfferPanelComponent:

```
class OfferPanelComponent {  
    ...  
    show (offer) {  
        let fullData = await api  
            .getFullOfferData(offer);  
        ...  
    } ...  
}
```

Возникает вопрос: а что должно произойти, если пользователь или разработчик пытается выбрать другой offerId, пока ответ сервера ещё не пришёл? Очевидно, нам нужно выбрать, какое из двух открытий панель мы должны запретить. Предположим, что мы решили блокировать интерфейс на время подгрузки данных и, таким образом, не давать выбирать другое предложение. Чтобы реализовать эту функциональность, нам нужно оповестить вышестоящие компоненты о начале и окончании загрузки:

```
class OfferPanelComponent {  
    ...  
    show () {  
        this.events.emit('beginDataLoad');  
        let fullData = await api  
            .getFullOfferData(offer);  
        this.events.emit('endDataLoad');  
        ...  
    } ...  
}
```

```

// `Composer` прослушивает события
// на панели предложений и выставляет
// значения соответствующего флага
class SearchBoxComposer {
    ...
    constructor () {
        ...
        this.offerPanel.events.on(
            'beginDataLoad', () => {
                this.isLoading = true;
            }
        );
        this.offerPanel.events.on(
            'endDataLoad', () => {
                this.isLoading = false;
            }
        );
    }

    selectOffer (offer) {
        if (this.isLoading) {
            return;
        }
        ...
    }
}

```

Но этот код очень плох по множеству причин:

- непонятно, как его модифицировать, если у нас появятся разные виды загрузок данных, причём некоторые из них будут требовать блокировки интерфейса, а некоторые — нет;
- этот код просто плохо читается: совершенно непонятно, почему события загрузки данных на одном компоненте влияют на пользовательскую функциональность в другом компоненте;
- если при загрузке произойдёт какая-то ошибка, событие endDataLoad не произойдёт, и интерфейс останется заблокированным навсегда.

Если вы внимательно читали предыдущие главы, решение этих двух проблем должен быть очевидным. Необходимо абстрагироваться от самого факта загрузки данных и переформулировать проблемы в высокогенерализованных терминах. У нас есть разделяемый ресурс — место на экране. Мы можем показывать в один момент времени только одно предложение. Следовательно, если какому-то актору требуется длящийся доступ к панели, он должен этот доступ явно получить. Отсюда следует, что:

- флаг такого доступа должен именоваться явно (например, offerFullViewLocked, а не isLoading);

- флаг контролируется Composer-ом, но никак не самой панелью предложения (ещё и потому, что подготовка данных для показа — также ответственность Composer-a).

```
class SearchBoxComposer {
  constructor () {
    ...
    this.offerFullViewLocked = false;
  }
  ...
  selectOffer (offer) {
    if (this.offerFullViewLocked) {
      return;
    }
    this.offerFullViewLocked = true;
    let fullData = await api
      .getFullOfferData(offer);
    this.events.emit(
      'offerFullViewChange',
      this.generateOfferFullView(fullData)
    );
    this.offerFullViewLocked = false;
  }
}
```

Такой подход улучшает читабельность, но не помогает с проблемами параллельного доступа и ошибочно неснятых флагов. Чтобы решить их, нам нужно сделать ещё один шаг: не просто ввести флаг, но и процедуру его *захвата* (вполне классическим образом по аналогии с управлением разделяемыми ресурсами в системном программировании):

```

class SearchBoxComposer {
    ...
    selectOffer (offer) {
        let lock;
        try {
            // Пытаемся захватить ресурс
            // `offerFullView`
            lock = await this.acquireLock(
                'offerFullView', '10s'
            );
            let fullData = await api
                .getFullOfferData(offer);
            this.events.emit(
                'offerFullViewChange',
                this.generateOfferFullView(fullData)
            );
            lock.release();
        } catch (e) {
            // Если получить доступ не удалось
            return;
        } finally {
            // Не забываем освободить ресурс
            // в случае ошибки
            if (lock) {
                lock.release();
            }
        }
    }
}

```

NB: вторым параметром в `acquireLock` мы передали максимальное время жизни блокировки — 10 секунд. Если в течение этого времени блокировка не снята (например, в случае, если мы забыли обработать какое-то исключение или выставить таймаут на запрос к серверу), она будет отменена автоматически, и интерфейс будет разблокирован.

В таком подходе мы можем реализовать не только блокировки, но и различные сценарии, которые позволяют нам более гибко ими управлять. Добавим в функцию захвата ресурса дополнительные данные о целях захвата:

```

lock = await this.acquireLock(
    'offerFullView', '10s', {
        // Добавляем описание,
        // кто и зачем пытается
        // выполнить блокировку
        reason: 'userSelectOffer',
        offer
    }
);

```

Тогда текущий владелец ресурса (или диспетчер блокировок, если мы реализуем такой объект) может, в зависимости от ситуации, отдавать владение ресурсом или, наоборот, запрещать перехват. Скажем, если открытие панели инициировано программистом через вызов API компонента (а не пользователем через выбор предложения в списке), оно может иметь более высокий приоритет и быть разрешено:

```
lock.events.on('tryAcquire', (actor) => {
  if (sender.reason == 'apiSelectOffer') {
    lock.release();
  } else {
    // Иначе запрещаем перехват
    return false;
  }
});
```

Дополнительно мы можем ввести и обработку потери контроля ресурса — например, отменить загрузку данных, которые больше не нужны.

```
lock.events.on('lost', () => {
  this.cancelFullOfferDataLoad();
});
```

Паттерн контроля разделяемых ресурсов также хорошо сочетается с паттерном «модель»: акторы могут захватывать доступ на чтение и/или изменение свойств или групп свойств модели.

NB: мы могли бы решить проблему подгрузки данных иначе:

- открыть панель предложения;
- вместо настоящих данных отобразить спиннер или какую-то другую индикацию загрузки;
- асинхронно обновить отображение при получении ответа от сервера.

Однако в постановке проблемы это ничего не меняет: нам всё ещё нужно разработать политику разрешения конфликтов для случая, если какой-то актор пытается открыть панель предложения, пока загрузка данных ещё не закончена, для чего нам вновь нужны разделяемые ресурсы и их захват.

Отметим, что в современном фронтенде (к нашему большому сожалению) подобные упражнения с захватом контроля на время загрузки данных или анимации компонентов практически не производятся (считается, что такие асинхронные операции происходят быстро, и коллизии доступа не представляют собой проблемы). Однако, если асинхронные операции выполняются долго (происходят длительные или многоступенчатые загрузки данных, сложные анимации), пренебрежение организацией доступа может быть очень серьёзной UX-проблемой.

Глава 48. Вычисляемые свойства

Вернёмся к одной из проблем, описанных в главе «[Проблемы встраивания UI-компонентов](#)»: наличие множественных линий наследования усложняет кастомизация компонентов, поскольку подразумевает, что они могут наследовать важные свойства по любой из вертикалей.

Пусть у нас имеется кнопка, которая получает одно и то же свойство `iconUrl` по двум вертикалям — из данных [т.е., в случае нашего примера, из результатов поиска предложений] и из настроек отображения:

```
class Button {
    static DEFAULT_OPTIONS = {
        ...
        iconUrl: <иконка по умолчанию>
    }

    constructor (data, options) {
        this.data = data;
        // Разрешаем переопределять
        // опции по умолчанию
        this.options = extend(
            Button.DEFAULT_OPTIONS,
            options
        )
    }

    render() {
        ...
        this.iconElement.src =
            this.data.iconUrl ||
            this.options.iconUrl
    }
}
```

При этом мы можем легко представить себе, что по обеим иерархиям свойство `iconUrl` было получено от кого-то из родителей по любой из вертикалей:

- опции по умолчанию могут быть определены в базовом классе, от которого унаследован `Button`;
- данные, на которых строится кнопка, могут быть общими для группы кнопок или сами по себе быть иерархическими (например, если мы будем группировать предложения сети кофеен и наследовать иконку именно родительской группы);

- в целях облегчить кастомизацию визуального стиля компонент мы можем разрешить переопределять иконку вообще всем кнопкам через задание свойств по умолчанию для всего SDK.

В этой ситуации у нас возникает вопрос: если значение определено сразу в нескольких иерархиях (например, и в данных предложения, и в опциях по умолчанию), каким образом задавать приоритеты, чтобы выбирать одно из них?

Простой подход «в лоб» к этому вопросу — попросту запретить наследование и заставить разработчика копировать все нужные ему свойства. То есть в нашем примере сам разработчик должен написать что-то типа:

```
const button = new Button(data);
if (data.createOrderButtonIconUrl) {
  button.view.iconUrl =
    data.createOrderButtonIconUrl;
} else if (data.parentCategory.iconUrl) {
  button.view.iconUrl =
    data.parentCategory.iconUrl;
}
```

Достоинства простого решения очевидны — разработчик сам имплементирует ту логику, которая ему нужна. Недостатки тоже очевидны — во-первых, это лишний и зачастую дублирующийся код; во-вторых, разработчик быстро запутается в том, какие правила он реализовал и почему.

Чуть более сложный подход к проблеме — разрешить наследование, но строго зафиксировать приоритеты (скажем, заданное в опциях отображения значение всегда важнее заданного в данных, и они оба всегда важнее любого унаследованного свойства). Однако в достаточно сложном API результат будет тот же самым: если разработчику необходим другой порядок приоритетов, ему придётся задавать нужные свойства вручную, т.е. в итоге писать код, подобный вышеприведённому.

Альтернативный подход — это предоставить возможность задавать правила, каким образом для конкретной кнопки определяется её иконка, декларативно или императивно:

```
// Декларативный подход:  
// описываем правила в каком-то формате  
{  
  "button.checkout.iconUrl": "@data.iconUrl"  
}
```

```
// Императивный подход – программно  
// добавляем функцию вычисления значения  
api.options.addRule(  
  'button.checkout.iconUrl',  
  (data, options) => data.iconUrl  
)
```

Наиболее последовательная реализация этого подхода — CSS¹. Мы не то чтобы рекомендуем использовать CSS-подобные правила в библиотеках компонентов (в силу потрясающей сложности их имплементации и, в большинстве случаев, избыточности), но осторожно замечаем, что поддержка какого-то простого подмножества подобного рода правил *значительно* облегчает кастомизацию визуальных компонент для разработчиков.

Вычисленные значения

Очень важно не забыть предоставить разработчику не только способы задать приоритеты параметров, но и возможность узнать, какой же из вариантов значения был применён. Для этого мы должны разделить заданные и вычисленные значения:

```
// Задаём значение в процентах  
button.view.width = '100%';  
// Получаем реально применённое  
// значение в пикселях  
button.view.computedStyle.width;
```

Хорошей практикой также будет предоставлять доступ не только к вычисляемым значениям, но и к событию изменения этого значения.

Примечания

¹ CSS

<https://www.w3.org/Style/CSS/>

Глава 49. Заключение

Предыдущие восемь глав были написаны нами, чтобы раскрыть две очень важные мысли:

- разработка качественной UI-библиотеки — это отдельная и весьма непростая инженерная задача;
- и эта задача не сводится к автоматической генерации SDK по спецификации / модели данных.

Оглядываясь на всё написанное, мы с трудом можем сказать, что нашли лучшие примеры и самые понятные слова для описания такой сложной предметной области. Мы, тем не менее, надеемся, что сделали вашу жизнь — и жизнь ваших пользователей — чуточку проще.

РАЗДЕЛ VI. API КАК ПРОДУКТ

Глава 50. Продукт API

Когда мы говорим об API как о продукте, необходимо чётко зафиксировать два важных тезиса.

1. API — это *полноценный продукт*, как и другое ПО. Вы «продаёте» его точно так же, и к нему полностью применимы принципы управления продуктом. Весьма сомнительно, что вы сможете качественно развивать API, не изучив потребности аудитории, спрос и предложение, рынок и конкурентов.
2. API — это *очень специфический продукт*. Вы продаёте возможность некоторые действия выполнять автоматизированно через написание кода, и этот факт накладывает большие ограничения на управление продуктом.

Для того, чтобы успешно развивать API, необходимо уметь отвечать именно на этот вопрос: почему ваши потребители предпочут выполнять те или иные действия *программно*. Вопрос этот не праздный, поскольку, по опыту автора настоящей книги, именно отсутствие у руководителей продукта и маркетологов экспертизы в области работы с API и есть самая большая проблема развития API.

Конечный пользователь взаимодействует не с вашим API напрямую, а с приложениями, которые поверх API написали разработчики в интересах какого-то стороннего бизнеса (причём иногда в цепочке между вами и конечным пользователем находится ещё и более одного разработчика). С этой точки зрения целевая аудитория API — это некоторая пирамида, напоминающая пирамиду Маслоу:

- основание пирамиды — это пользователи; они ищут удовлетворение каких-то своих потребностей и не думают о технологиях;
- средний ярус пирамиды — бизнес-заказчики; соотнеся потребности пользователей с техническими возможностями, озвученными разработчиками, бизнес строит продукты;
- вершиной же пирамиды являются разработчики; именно они непосредственно работают с API, и они решают, какой из конкурирующих API им выбрать.

Непосредственный вывод из этой модели звучит так: в достоинствах вашего API в первую очередь необходимо убедить разработчиков; они, в свою очередь, примут решение о выборе технологии, ну а бизнес оттранслирует концепцию пользователям. Если руководителями продукта API являются бывшие или действующие разработчики, они зачастую склонны оценивать конкуренцию на рынке API именно в этом аспекте, и направлять основные усилия по продвижению продукта именно на аудиторию программистов.

«Постойте!» — воскликнет здесь внимательный читатель. Но ведь дела здесь обстоят с точностью до наоборот:

- разработчики, как правило, являются наёмными сотрудниками, выполняющими поставленные заказчиком задачи (и даже если мы говорим о каком-то разработчике-одиночке, он всё же выбирает API в интересах какого-либо проекта, т.е. выступает здесь как бизнес-заказчик для самого себя);
- бизнес, в свою очередь, ставит задачи разработчикам не из чувства стиля и не из соображений красоты кода — заказчика интересует реализация конкретной функциональности, необходимой для решения задач пользователя;
- наконец, пользователь вообще не разбирается в технических аспектах решения: он выбирает тот продукт, который ему нужен, и требует наличия в нём определённой функциональности.

Получается, что на вершине пирамиды, таким образом, находится конечный пользователь: это его нам нужно убедить, что он хочет не какую попало кружку кофе, а именно приготовленную через наш API (и отдельный вопрос, как же мы будем доносить информацию о том, что за API работает под капотом, и почему пользователь должен своими деньгами проголосовать именно за наш API!); тогда бизнес поставит разработчику задачу интегрировать наш API, ну а разработчик уже никуда не денется и напишет код (что, кстати, означает, что вкладываться в читабельность и консистентность API не так уж и обязательно).

Истина, разумеется, лежит где-то посередине. В каких-то предметных областях и на каких-то рынках все решения принимаются разработчиками (например, какой фреймворк выбрать); в других областях и рынках последнее слово остаётся за бизнес-заказчиком и конечным пользователем. К тому же, многое зависит от конкурентной ситуации — если вход на рынок фронтенд-

разработки с новым фреймворком не сталкивается ни с каким противодействием, то разработка, например, новой мобильной операционной системы требует многомиллионных расходов на продвижение и стратегические партнерства.

Здесь и далее мы будем описывать некоторый «усреднённый» случай, в котором все три яруса нашей двойной пирамиды важны: и пользователи (которые выбирают подходящий продукт), и бизнес (которому важны гарантии качества и стоимость разработки), и разработчики (которых интересует удобство работы с API и его функциональность).

Глава 51. Бизнес-модели API

Прежде, чем переходить непосредственно к принципам продуктового управления API, позволим себе заострить внимание читателя на вопросе, каким образом наличие API как продукта приносит пользу компании, а также соответствующие модели монетизации, прямой и косвенной. Вопрос этот, как мы покажем в следующих главах, далеко не праздный, так как напрямую влияет на KPI API и принятие решений. В этой главе мы разберём основные модели монетизации API. [В квадратных скобках будем приводить примеры подобных моделей применительно к нашему учебному примеру с API кофе-машин.]

1. Разработчик = конечный пользователь

Самая простая и понятная ситуация — в том случае, если разработчики и есть конечные пользователи. В первую очередь здесь речь идёт о всевозможных инструментах для программиста: API языков программирования, фреймворков, операционных систем, библиотек компонентов, игровых движков и так далее, — иными словами, об интерфейсах общего назначения. [В нашем кофейном примере это означает следующее: мы разработали библиотеку для заказа кофе, возможно, с визуальными компонентами, и теперь продаём всем желающим-владельцам сети кофе-машин, чтобы облегчить им разработку собственного приложения.] В этом случае ответ на вопрос «зачем предоставлять программный интерфейс» самоочевиден.

Видов монетизации такого API существует множество — по сути, речь идёт о моделях монетизации ПО для разработчиков как таковом.

1. Фреймворк / библиотека / платформа могут быть платными сами по себе, т.е. распространяться под платной лицензией; в настоящее время такие модели становятся всё менее популярны в связи с растущим проникновением открытого программного обеспечения, но, тем не менее, всё ещё широко распространены.

2. API может быть лицензирован под открытой лицензией с определёнными ограничениями, которые могут быть сняты путём покупки расширенной лицензии; это может быть как ограничение функциональности API (например, запрет публикации приложения в соответствующем магазине приложений или невозможность сборки приложения в продакшен-режиме без приобретения лицензии), так и ограничения на использование (например, открытая лицензия может быть «заразной», т.е. требовать распространения написанного поверх платформы кода под той же лицензией, или же использование бесплатного API может быть запрещено для определённых целей).
3. API может быть бесплатен, но компания-разработчик API может оказывать платные услуги по его консультированию и внедрению или просто продавать расширенную техническую поддержку.
4. Разработка API может спонсироваться явно или неявно владельцем платформы или операционной системы [в нашем кофейном примере — производителем «умных» кофе-машин], который заинтересован в том, чтобы разработчики имели как можно больше удобных инструментов для работы с платформой.
5. Наконец, компания-разработчик API путём публикации его под открытой лицензией тем самым привлекает к себе внимание и надеется на повышение продаж других своих продуктов для разработчиков.

Примечательно, что подобные API — чуть ли не единственный «чистый» случай, когда при выборе API разработчик оперирует исключительно соображениями о том, насколько хорош дизайн API, понятна ли документация, продуманы сценарии использования и так далее. Известны случаи, когда сторонние компании или даже пользователи-энтузиасты самостоятельно реализовывали альтернативные имплементации популярного API — так произошло, например, с API языков Java (альтернативная реализация от Google) и C# (проект Mono) — или некоторых отдельных удачных решений в дизайне функциональности (как, например, концепция работы с DOM-деревом с помощью выборки элементов CSS-селекторами, изначально появившаяся в проекте cssQuery, позднее была реализована в jQuery, и уже на волне популярности последнего адаптирована непосредственно разработчиками спецификации DOM).

2. API = основной и/или единственный способ доступа к сервису

Этот случай примыкает к предыдущему в том смысле, что потребителем API является разработчик, а не конечный пользователь — с той разницей, что продуктом является не сам API, а сервис, доступ к которому он предоставляет. Чистый пример — это API различных облачных платформ, например, Amazon AWS или API Braintree. Да, какая-то работа с платформой возможна и через UI, но без API такие сервисы практически бесполезны. [В нашем кофейном примере — если мы являемся оператором сети «облачных» кофе-машин и доставки кофе дронами, и предоставляем к ним доступ только через API.]

Как правило, тарифицируется в этом случае использование самого сервиса, а не API как такового. Часто, однако, тарифы исчисляются именно по сущностям API, т.е. тарифицируется количество вызовов методов.

3. API = партнёрская программа

Многие коммерческие сервисы предоставляют доступ к своей платформе для сторонних разработчиков с целью увеличения продаж или привлечения аудитории. Примером такого API могут служить, например, партнёрские программы Google Books, Skyscanner Travel APIs или Uber API. [Нашему учебному примеру здесь соответствует вот такая модель: мы — крупная известная сеть кофеен, и мы мотивируем сторонних разработчиков продавать наш кофе через свои сайты и приложения; фактически, позволяем разместить более интерактивную и глубоко интегрированную рекламу нашего сервиса — то, что сегодня принято называть «нативной рекламой».] В этом случае партнёрство является полностью и исключительно коммерческим: потребители API монетизируют свою собственную аудиторию, а компания — провайдер API таким образом надеется получить доступ к расширенной аудитории, дополнительным рекламным каналам. Как правило, владелец API выплачивает вознаграждение партнёрам за каждое целевое действие и устанавливает требования к эффективности интеграции (например, в виде минимального click-target ratio) чтобы избежать нецелевого использования API.

4. API = дополнительный доступ к сервису

Если некоторая компания располагает некоторой уникальной экспертизой, чаще всего — набором данных, который трудно и/или очень дорого получить самостоятельно, вполне логично возникает спрос на предоставление этой экспертизы через API. Самый классический пример API такого рода — это картографические API; собрать подробные и точные геоданные и поддерживать их в актуальном виде — очень дорого; при этом чуть ли не все сервисы, которые только можно себе придумать, станут существенно лучше и удобнее, если добавить карту. [На наш кофейный пример такой подход ложится с трудом, поскольку аккумулируемые нами в этом случае данные — расположения кофе-машин и типы приготавливаемых ими напитков — не являются чем-то, имеющим самостоятельную ценность вне контекста заказа кофе.]

Этот кейс — пожалуй что самый интересный с точки зрения разработчика API, поскольку существование программного интерфейса в этом случае действительно является мультипликатором возможностей: компания-владелец экспертизы чисто физически не может сама реализовать все на свете сервисы, эту экспертизу использующие. Предоставление API здесь является «win-win» стратегией: функциональность сторонних сервисов улучшается, а компания-провайдер зарабатывает на этом деньги.

Доступ к API в этом случае может быть безусловно платным, хотя чаще встречаются смешанные модели монетизации: API предоставляется бесплатно до достижения какого-то лимита либо при соблюдении определённых условий (например, для некоммерческих проектов). В отдельных случаях API предоставляется бесплатно с минимальными ограничениями с целью популяризации определённой платформы (например, Apple Maps).

Отдельно здесь следует отметить B2B-сервисы. Так как провайдеру сервиса выгодно дать клиенту как можно более разнообразные возможности, а клиенту, в свою очередь, часто требуется максимально гибко использовать имеющуюся функциональность, часто предоставление API — оптимальный выход для обеих сторон. Крупным компаниям, располагающим своими

отделами разработки, чаще необходимо запрограммировать свои бизнес-процессы через API и интегрировать их со своими внутренними системами. Часто таким B2B-сервисом выступает сама компания — разработчик API, если собственные сервисы компании строятся поверх API же, а внешний API существует в дополнение к внутреннему.

NB: мы всячески не рекомендуем при этом предоставление API «на сдачу», т.е. публикацию внутренних API без какой-либо дополнительной продуктовой и технической подготовки. Главная проблема таких API заключается в том, что интересы партнёров при планировании разработки никак не учитываются, что приводит к множественным проблемам.

- API плохо покрывает основные сценарии интеграции:
 - внутренние потребители, как правило, используют вполне конкретный технический стек, и API плохо оптимизирован под любые другие языки программирования / операционные системы / фреймворки;
 - внутренние потребители гораздо лучше погружены в контекст, могут посмотреть исходный код или пообщаться напрямую с разработчиком API, и для них порог входа в технологию находится на совершенно другом уровне по сравнению с внешними потребителями;
 - документация покрывает какой-то наиболее востребованный внутренними потребителями срез сценариев;
 - линейка сервисов API, о которой мы расскажем ниже, зачастую попросту отсутствует, т.к. внутренним потребителям она не нужна.
- Любые ресурсы выделяются в первую очередь на поддержку внутренних потребителей. Это означает, что:
 - планы развития API для партнёров совершенно непрозрачны и бывает что попросту абсурдны; очевидные недоработки не исправляются годами;
 - техническая поддержка внешних пользователей осуществляется по остаточному принципу.
- Разработчики внутренних сервисов часто ломают обратную совместимость или выпускают новые мажорные версии, совершенно не заботясь о последствиях этих действий для внешних партнёрах.

Всё это приводит к тому, что наличие внешнего API зачастую работает не в плюс компании, а в минус: фактически, вы предоставляете крайне критически и скептически настроенной аудитории очень плохой продукт. Если у вас нет ресурсов на грамотное развитие API как продукта для внешних пользователей — лучше за него не браться совсем.

5. API = площадка для рекламы

В основном речь здесь идёт о разного рода виджетах и поисковиках: чтобы размещать какую-то рекламу, необходимо иметь прямой доступ к конечному пользователю. Наиболее распространённые API такого типа — это собственно API рекламных сетей, однако встречаются и комбинированные подходы, когда вместе с некоторым API, например, поисковым, «в нагрузку» идёт показ рекламы. [В нашем кофейном примере — если наша функция поиска предложений напитков начнёт каким-то образом выделять на странице результатов оплаченные показы.]

6. API = самореклама и самопиар

Если никакой — ни прямой, ни косвенной — монетизации API не имеет, он всё ещё может приносить доход, развивая знание о компании через брендирование — отображение логотипов и других узнаваемых элементов при работе пользователя с API, нативное (если визуальные интерфейсы отрисовываются непосредственно самим API) или договорное (потребители API обязаны по контракту размещать определённые элементы брендирования там, где используется функциональность API или предоставленные через него данные). Целью компании-разработчика API в этом случае является или привлечение аудитории на свои основные сервисы, либо продвижение своего бренда в целом. [В случае нашего кофейного API — представим, что мы предоставляем некоторый совершенно иной полезный сервис, допустим, продаём автомобильные шины, а через предоставление API кофе-машин пытается увеличить узнаваемость и заработать себе репутацию технологической компании.]

В этом случае возможны вариации относительно целевой аудитории саморекламы:

- вы можете работать на узнаваемость бренда среди конечных пользователей (размещением своих логотипов и ссылок на сайтах и в приложениях партнёров, использующих API), и даже строить сам бренд таким образом [например, если наш кофейный API будет предоставлять рейтинги кофеен, и мы будем работать на то, чтобы потребитель сам требовал от кофеен указывать рейтинг по версии нашего сервиса];
- вы можете работать на распространение знания о себе среди бизнес-аудитории [например, чтобы партнёры, размещающие у себя виджет заказа кофе, заодно и изучили каталог ваших шин];
- наконец, вы можете распространять API только и исключительно ради того, чтобы заработать себе репутацию среди разработчиков — ради информирования о других ваших продуктах или ради улучшения вашего имиджа как работодателя для ИТ-специалистов (эту деятельность иногда называют словом «технопиар»).

Дополнительно в этом разделе можно говорить о формировании комьюнити, т.е. сообщества пользователей или разработчиков, лояльных к продукту. Выгоды от существования таких комьюнити бывают довольно существенны: это и снижение расходов на техническую поддержку, и удобный канал информирования о новых сервисах и релизах, и возможность получить бета-тестеров разрабатываемых продуктов.

7. API = инструмент получения обратной связи и UGC

Если компания располагает какими-то большими данными, то оправданной может быть стратегия выпуска публичного API для того, чтобы конечные пользователи вносили исправления в данные или иным образом вовлекались в их разметку. Например, провайдеры картографических API часто разрешают сообщить об ошибке или исправить неточность прямо в стороннем приложении. [А в случае нашего кофейного API мы могли бы собирать обратную связь, как пассивно — строить рейтинги заведений, например, — так и активно — контактировать с владельцами заведений чтобы помочь им исправить недостатки; находить через UGC ещё не подключенные к API кофейни и проактивно работать с ними.]

8. Терраформирование

Наконец, наиболее альтруистический подход к разработке API — предоставление его либо полностью бесплатно, либо в формате открытого кода и открытых данных просто с целью изменения ландшафта: если сегодня за API никто не готов платить, то можно вложиться в популяризацию и продвижение функциональности, рассчитывая найти коммерческие ниши (в любом из перечисленных форматов) в будущем или повысить значимость и полезность API в глазах конечных пользователей (а значит и готовность потребителей платить за использование API). [В случае нашего кофейного примера — если компания-производитель умных кофе-машин предоставляет API полностью бесплатно в расчёте на то, что со временем наличие у кофе-машин API станет нормой, и появится возможность разработать новые монетизируемые сервисы за счёт этого.]

9. Серая зона

Отдельный источник дохода для разработчика API — это анализ запросов, которые делают конечные пользователи или, иными словами, сбор и дальнейшая продажа какой-то информации. Следует иметь в виду, что граница между допустимыми способами обработки информации (например, агрегирование поисковых запросов с целью выделения трендов или потенциально прибыльных точек для открытия кофейни) и недопустимыми здесь весьма неочевидна и имеет тенденцию меняться как во времени, так и в пространстве (в смысле, одни и те же действия могут быть легальны по одну сторону государственной границы и нелегальны по другую), так что принимать решения о монетизации API подобными способами следует с очень большой осторожностью.

Подход API-first

В последние годы набирает силу тенденция предоставлять некоторую функциональность в виде API (т.е. продукта для разработчиков) там, где теоретически можно было бы предоставить сервис напрямую конечным пользователям. Этот подход известен как «API-first» и отражает нарастающую специализацию IT-сфера в целом: разработка API выделяется в отдельную компетенцию, которую бизнес готов отдавать сторонним компаниям на аутсорс вместо того, чтобы тратить ресурсы на разработку внутренних API для приложений своими силами. Тем не менее, пока этот подход не является универсально общепринятым, следует помнить о факторах принятия решений, когда можно запускать сервис в формате API-first.

1. Целевой рынок должен быть достаточно «прогрет» — на нём уже должны действовать компании, обладающие достаточным ресурсом для разработки сервисов поверх вашего API, и готовых, к тому же, оплачивать его использование (если только вашей целью не является самореклама или тераформирование);
2. Качество сервиса не должно страдать от того, что он предоставляется через API;
3. Необходимо реально обладать пресловутой экспертизой в разработке API, иначе велик шанс наделать неисправимых ошибок.

Иногда раздача API является своеобразным «прощупыванием почвы» для того, чтобы компания-разработчик могла оценить рынок и решить, стоит ли выводить на него полноценный пользовательский сервис (мы такую практику скорее осуждаем, поскольку она неизбежно приведёт к закрытию API в будущем: либо потому, что рынок оказался не столь привлекателен, как казалось априори; либо потому, что API начнёт создавать конкуренцию материнскому сервису и будет со временем закрыт или существенно ограничен).

Глава 52. Формирование продуктового видения

Описанная выше фрагментация целевой аудитории API, триада «разработчики — бизнес — конечные пользователи», делает управление продуктом API весьма нетривиальной проблемой. Да, базовый принцип — выяснить потребности аудитории и удовлетворить их — всё тот же; только аудиторий у вашего продукта три, причём их интересы далеко не всегда коррелируют. Из потребности конечного пользователя в качественном и недорогом кофе отнюдь не следует потребность бизнеса в API для работы с кофе-машинами.

В общем случае продуктовое видение будущего API тоже должно включать в себя те же звена:

- представление об идеальном решении проблем конечного пользователя;
- предположения о том, каким образом бизнес подошёл бы к решению этих проблем, располагая техническими инструментами;
- понимание доступного спектра технических решений и их применимости.

На разных рынках и в разных ситуациях «вес» каждой ступени различен. Если вы производите API-first продукт для разработчиков (без визуальной составляющей), то вы вполне можете обойтись без анализа проблем конечных пользователей; и наоборот, если вы предоставляете API к чрезвычайно ценной для пользователя функциональности в условиях, близких к монопольным, — вам в общем-то всё равно, насколько разработчикам нравится ваша архитектура и удобно ли им работать с вашими интерфейсами, выбора у них все равно нет.

В большинстве же случаев мы имеем дело с некоторой двухуровневой эвристикой, которая идёт или от технических возможностей, или от бизнес-потребностей:

- либо, исходя из вашего понимания спектра доступных технических решений, вы пытаетесь сформировать понимание, как вы могли бы помочь бизнесу (первый шаг эвристики), а уже из него — общее представление о том, как ваш API будет помогать конечным пользователям (второй шаг эвристики);

- либо, исходя из вашего понимания проблем, стоящих перед бизнес партнёрами, вы можете сделать «шаг вправо», обрисовав будущую функциональность для конечных пользователей, и «шаг влево», прикинув возможные технические решения.

Из эвристичности обоих подходов неизбежно следует и неопределённость продуктового видения API; в большинстве случаев это нормально: если бы вы могли иметь полное и чёткое представление о том, какие продукты для пользователей можно разработать поверх вашего API, вы могли бы разработать их сами, опустив промежуточное звено в виде партнёров. Здесь также важно и то, что многие API проходят стадию «терраформирования» (см. предыдущую главу), то есть «подготавливают почву» для новых рынков и видов сервисов — таким образом, ваше идеализированное представление о светлом будущем, когда доставка готового кофе дронами будет нормой жизни, будет постепенно детализироваться по мере появления на этом рынке новых компаний, предоставляющих новые виды услуг. (Что, в свою очередь, отразится и на моделях монетизации: по мере детализации облика грядущего вы будете переходить от всё более абстрактных KPI и теоретических выгод наличия API ко всё более конкретным.)

Ту же неопределённость следует иметь в виду и при проведении интервью и сборе обратной связи. Программисты будут, в основном, сообщать вам о своих проблемах, возникающих при разработке сервиса — редко о проблемах бизнеса; бизнесу, в свою очередь, мало дела до неудобств разработчиков. И те, и другие обладают при этом каким-то представлением о проблемах пользователя, но зачастую это представление сильно ограничено сегментом рынка, в котором оперирует партнёр.

Если у вас есть доступ к инструментам отслеживания действий конечных пользователей (см. главу «[Ключевые показатели эффективности API](#)»), то вы можете попробовать через их логи восстановить типичное поведение пользователей и понять, как они взаимодействуют с приложениями партнёров. Но вам вновь придётся эти данные анализировать для каждого приложения по отдельности и попытаться кластеризовать общие кейсы и частотные сценарии.

Проверка продуктовых гипотез

Помимо общих сложностей с формированием продуктового видения API есть и более частные сложности с проверкой продуктовых гипотез. «Святой грааль» управления продуктом — создание максимально недорогого с точки зрения затраченных ресурсов *minimal viable product* (MVP) — обычно недоступен для менеджера API. Дело в том, что вы не можете так просто *проверить* MVP, даже если вам удалось его разработать: для проверки MVP API партнёры должны *написать код* (читай — вложить свои деньги); если по итогам этого эксперимента будет принято решение о беспersпективности продукта, эти деньги окажутся потрачены впустую. Разумеется, партнёры к подобного рода предложениям относятся с некоторым скептицизмом. Таким образом «дешёвый» MVP включает в себя либо компенсацию расходов партнёрам, либо затраты на разработку референсного приложения (т.е. в дополнение к MVP API разрабатывается сразу и MVP приложения, использующего этот API).

Частично эту проблему можно решить, если выпустить MVP от имени сторонней компании (например, в виде модуля с открытым кодом, опубликованного от лица разработчика). Однако тогда вы получите проблемы с собственно проверкой гипотезы, так как подобные модули рискуют быть просто оставленными без внимания.

Другой вариант проверки гипотез — это собственный сервис (или сервисы) компании-разработчика API, если такие есть. Внутренние заказчики обычно более лояльно относятся к трате ресурсов на проверку гипотез, и с ними легче договориться о сворачивании или замораживании MVP. Однако таким образом можно проверить далеко не всякую функциональность — а только то, на которую имеется хоть в каком-то приближении внутренний заказ.

Вообще, концепция “eat your own dogfood” применительно к API означает, что у команды продукта есть какие-то свои тестовые приложения (т.н. “pet project”ы) поверх API. Учитывая трудоёмкость разработки подобных приложений, имеет смысл поощрять их наличие, например, предоставлением бесплатных квот на API и вычислительные ресурсы членам команды.

Подобные pet project-ы также дают уникальный опыт: каждый может попробовать себя в новой роли. Разработчик узнает о типичных проблемах менеджера продукта: недостаточно написать приложение хорошо, нужно ещё и изучить своего потребителя, понять его потребности, сформулировать привлекательное предложение и донести его. Менеджеры продукта же,

соответственно, получат какое-то представление о том, насколько технически просто или сложно воплотить их продуктовое видение в жизнь, и какие проблемы возникнут при реализации. Наконец, и тем, и другим будет исключительно полезно взглянуть со стороны документацию API, на user story разработчика, который впервые услышал о продукте API и пытается в нём разобраться.

Глава 53. Взаимодействие с разработчиками

Как мы описали в предыдущих главах, управление продуктом API требует выстраивания отношений и с бизнес-партнёрами, и с разработчиками. (В идеале и с конечными пользователями, но эта опция для провайдеров API крайне редко доступна.)

Начнём с разработчиков. Специфика программистов как аудитории API заключается в том, что:

- разработчики — высокообразованные люди с практически-прикладным рациональным мышлением; как правило, выбор продукта ими осуществляется предельно рационально (если только вы не раздаёте бесплатные рюкзаки с крутой символикой вашего сервиса);
 - это не исключает определённой вкусовщины в части выбора, скажем, языков программирования или вендоров мобильной ОС; однако повлиять на эти вкусы крайне сложно и часто выходит за пределы возможностей поставщика API;
- в частности, разработчики скептически относятся к громким рекламным заявлениям и готовы разбираться в том, насколько эти заявления соответствуют истине;
- к разработчикам крайне сложно обращаться через стандартные маркетинговые каналы; помимо того, что они получают информацию в основном из узкоспециализированных сообществ, программисты также смотрят в первую очередь на мнения, подтверждённые конкретными цифрами и примерами (желательно — примерами кода);
 - мнения «инфлюэнсеров» в такой среде значат очень мало, поскольку ничьё мнение не принимается на веру;
- идеи Open Source и бесплатного ПО распространены среди разработчиков достаточно широко; попытки в том или ином виде зарабатывать на вещах, которые должны быть бесплатными и/или открытыми (например, путём лицензирования интеллектуальной собственности на интерфейсы), будут встречать сопротивление, причём представление об этих «должны быть» существенно варьируется.

В силу этих особенностей аудитории (в первую очередь — малой роли инфлюэнсеров и критического отношения к рекламным заявлениям) доносить информацию до разработчиков приходится через специфические каналы:

- коллективные блоги (такие, например, как субреддит “r/programming” или dev.to);
- сайты вопросов и ответов (Stack Overflow, Experts Exchange);
- обучающие сервисы (Codecademy, Udemy и другие);
- технические конференции и вебинары.

Во всех этих каналах прямая реклама вашего сервиса затруднена или невозможна вовсе. (Точнее, конечно, вы можете купить баннерную рекламу, но мы выражим очень большие сомнения в том, что это поможет вам выстроить отношения с разработчиками.) Вам необходимо генерировать какой-то ценный и/или интересный контент для улучшения знания о вашем API, и эта работа тоже ложится на ваших разработчиков: писать тексты, отвечать на вопросы, записывать обучающие семинары, выступать с докладами.

Программисты любят делиться опытом, и с удовольствием будут заниматься всеми вышеперечисленными задачами (в рабочее время); однако хорошее выступление на конференции, не говоря уже об обучающем курсе, требует многих часов подготовки. По опыту автора настоящей книги две вещи критически важны для технопиара:

- поощрения, пусть даже номинальные — работа по продвижению должна как-то вознаграждаться;
- методичность и стандарты качества — ревью презентаций столь же важно как и ревью кода.

Почти ничто не сделает вам худшей антирекламы, нежели плохо подготовленное выступление (см. выше — ошибки в вашей презентации непременно найдут) или плохо замаскированная реклама (по той же причине). Над текстами надо работать: следить за структурой, логикой и темпом повествования. И технический рассказ должен быть хорошо выстроен; по его окончании у слушателей должно сложиться чёткое понимание того, какую мысль им хотели донести (и хорошо бы эта мысль была как-то увязана с тем, что ваш API великолепно подходит для их нужд).

Отдельно следует упомянуть о «евангелистах»: так называют людей, обычно, обладающих определённым авторитетом в ИТ-сообществе, которые продвигают ту или иную технологию или компанию — то есть делают всё вышеперечисленное (пишут в блоги, записывают курсы, выступают на конференциях) за вас (чаще всего будучи внештатными, а иногда и штатными сотрудниками компании). Евангелист, таким образом, разгружает команду от необходимости заниматься технопиаром. Мы же всё-таки склонны считать, что эту функцию лучше иметь внутри команды, поскольку прямое взаимодействие с разработчиками чрезвычайно полезно для формирования продуктового видения. (Что вовсе не означает отказа от евангелистов — вы вполне можете совмещать две стратегии.)

Open Source

Важный вопрос, который рано или поздно встанет перед любым провайдером API — это выкладывание кода в Open Source. У этого действия есть как достоинства, так и недостатки:

- вы повысите узнаваемость вашего бренда и приобретёте какое-то уважение в среде разработчиков;
 - если, конечно, ваш API достаточно хорошо написан и прокомментирован;
- вы получите какой-то дополнительный фидбек, в идеальном случае даже pull request-ы от сторонних разработчиков;
 - а ещё вы получите какое-то количество обращений и комментариев, от бесполезных до откровенно провокационных, на которые нужно будет корректно отвечать;
- открытый код повышает доверие разработчиков и их готовность положиться на вашу платформу;
 - открытый код также означает и повышенные риски, как с точки зрения безопасности, так и с точки зрения того, что недовольное комьюнити может создать форк вашего репозитория и породить конкурирующий API.

Наконец, просто подготовка к открытию кода API сама по себе может быть весьма затратна: во-первых, код надо «причесать», во-вторых, перейти на открытые же инструменты сборки и тестирования, убрав из кода все ссылки на проприетарные ресурсы. Решение здесь следует принимать максимально осторожно, взвесив все «за» и «против». Добавим, что многие компании

пытаются снизить перечисленные выше риски, разделив API на две части, открытую и проприетарную, а также путём подбора специфической лицензии, которая не позволит нанести вред интересам компании через использование открытого кода (например, запрещая продавать hosted-решения или требуя обязательного раскрытия производного кода).

Фрагментация аудитории

Ко всему вышесказанному есть одно очень важное дополнение: в связи с огромной востребованностью информационных технологий в мире значительное количество ваших потребителей не будут профессиональными разработчиками. Огромное количество людей находятся в той или иной фазе освоения профессии: кто-то занимается разработкой в дополнение к основной деятельности, кто-то переучивается, кто-то осваивает азы компьютерных наук самостоятельно. Многие из этих непрофессиональных разработчиков при этом имеют прямое влияние на решение о выборе API — например, владельцы небольших бизнесов, которые по совместительству ещё и занимаются программной реализацией несложных задач.

Более правильно было бы сказать, что вы работаете на две основные аудитории:

- профессиональных программистов, имеющих широкий опыт интеграции различного рода систем;
- начинающих и полупрофессиональных разработчиков, для которых каждая такого рода интеграция является новой и неизведанной территорией.

Этот факт напрямую влияет на всё, что мы обсуждали выше (кроме, может быть, Open Source — разработчики-любители редко обращают на него внимание):

- Ваши лекции, семинары и выступления на конференциях должны как-то подходить и тем, и другим.
- Абсолютное большинство нагрузки на вашу техническую поддержку будет создавать первая категория: начинающим разработчикам гораздо сложнее найти ответы на свои вопросы или разобраться в проблеме самостоятельно, и они будут задавать их вам.

- При этом вторая категория потребителей будет гораздо более требовательна к качеству как продукта, так и поддержки, и при этом удовлетворить их запросы будет крайне нетривиально.

Наконец, практически невозможно в рамках одного продукта создать такой API, который одинаково хорошо подходит и начинающим разработчикам, и профессионалам; первым необходима максимальная простота реализации базовых сценариев использования API, вторым — возможность адаптировать использование API под конкретный стек технологий и парадигму разработки, и стоящие перед ними задачи, как правило, требуют глубокой кастомизации. Мы обсудим этот вопрос более подробно в главе «[Линейка сервисов API](#)».

Глава 54. Взаимодействие с бизнес-аудиторией

Основные принципы работы с партнёрами несколько парадоксально полностью противоположны принципам взаимодействия с разработчиками:

- с одной стороны, партнёры гораздо более лояльно и зачастую даже с энтузиазмом относятся к предлагаемым им возможностям, особенно бесплатным;
- с другой стороны, донести до бизнес-аудитории смысл вашего предложения несравненно сложнее, чем до разработчиков, так как представителям бизнеса в целом гораздо сложнее объяснить, в чём вообще состоят преимущества API (как концепции) по сравнению с сервисом для конечных пользователей.

Как итог, работа с бизнес-аудиторией в первую очередь сводится к тому, чтобы максимально доходчиво объяснить свойства и преимущества продукта. В остальных же смыслах API «продаётся» как и любое другое программное обеспечение.

Как правило, чем дальше некоторая отрасль находится от информационных технологий, тем с большим энтузиазмом её представители воспринимают рекламу возможностей API, и тем менее вероятно, что этот энтузиазм будет конвертирован в реальную интеграцию. Помочь решению этой проблемы должна интенсивная работа с комьюнити (см. предыдущую главу), благодаря которой появляется множество фрилансеров и аутсорсеров, готовых помочь не-ИТ бизнесам с интеграцией. Вы также можете помочь развитию рынка путём создания обучающих курсов для разработчика и выпуска сертификатов, подтверждающих навыки работы с вашим API (или более широким слоем технологий).

Аналогичным образом обстоит дело и с исследованиями рынка и получением обратной связи. Далёкие от ИТ бизнесы, как правило, не могут сформулировать свои потребности, поэтому к обработке полученных сведений следует подходить творчески (и критически).

Глава 55. Линейка сервисов API

Важное правило управления продуктом API, которое любой достаточно крупный поставщик API довольно быстро для себя откроет, звучит так: нет смысла поставлять всего лишь один какой-то API; есть смысл говорить о наборе продуктов, причём сразу в двух измерениях.

Горизонтальное разделение сервисов API

Как правило, любая функциональность, предоставляемая через API, делится на независимые блоки. Например, в нашем кофейном API есть функциональность поиска предложений и функциональность заказа кофе. Ничто не мешает нам как объявить эти эндпойнты разными API, так и считать их частями одного общего API.

Разные компании используют разные подходы к определению гранулярности сервисов API, что считать отдельным продуктом, а что нет; это до определённой степени вопрос вкусовщины и удобства. Стоит задуматься о разделении API на части, если:

- интеграция только с одной из частей имеет смысл, т.е. подсемейство API само по себе решает какую-то проблему пользователя без привлечения других API;
- части API могут версионироваться отдельно и независимо, и это осмысленно с точки зрения разработчика (обычно в таких ситуациях «отделившиеся» API должны либо быть полностью независимы, либо максимально поддерживать обратную совместимость и выпускать новые мажорные версии только в случае крайней необходимости, иначе поддержание в актуальном виде таблицы, какая версия API № 1 совместима с какой версией API № 2, быстро превратится в катастрофу);
- имеет смысл тарифицировать и устанавливать раздельные лимиты на каждый из сервисов по отдельности;
- аудитории различных сегментов API (разработчики, бизнес или конечные пользователи) не пересекаются, и «продать» гранулярное API потребителю проще, чем целый комбайн в нагрузку.

NB: при этом раздельные API могут предоставляться в рамках одного SDK, для удобства клиентской разработки.

Вертикальное разделение сервисов API

Часто, однако, имеет смысл предоставлять несколько сервисов API, оперирующих одной и той же функциональностью. Напомним, что все разработчики делятся на две категории — профессионалов, которые ищут возможности глубокой кастомизации (поскольку работают в крупных ИТ-компаниях со сложившимся подходом к разработке), и начинающих разработчиков, которым необходим максимально заниженный порог входа. Максимально полно покрыть нужды обеих групп можно только разработав множество продуктов с разным порогом входа и требовательностью к профессиональному уровню программиста. Можно выделить следующие подвиды API, по убыванию требуемого уровня разработчиков.

1. Самый сложный уровень — физического API и семейства абстракций над ними. [В нашем кофейном примере — та часть интерфейсов, которая описывает работу с физическим API кофе машин, см. главу «Разделение уровней абстракции» и главу «Слабая связность».]
2. Базовый уровень — работы с продуктами сущностями через формальные интерфейсы. [В случае нашего учебного API этому уровню соответствует HTTP API заказа.]
3. Упростить работу с продуктами сущностями можно, предоставив SDK для различных платформ, скрывающие под собой сложности работы с формальными интерфейсами и адаптирующие концепции API под соответствующие парадигмы (что позволяет разработчикам, знакомым только с конкретной платформой, не тратить время и не разбираться в формальных интерфейсах и протоколах).
4. Ещё более упростить работу можно с помощью сервисов, генерирующих код. В таком интерфейсе разработчик выбирает один из представленных шаблонов интеграции, кастомизирует некоторые параметры, и получает на выходе готовый фрагмент кода, который он может вставить в своё приложение (и, возможно, дописать необходимую функциональность с использованием API 1–3 уровней). Подобного рода подход ещё часто называют «программированием мышкой». [В случае нашего кофейного API примером такого сервиса мог бы служить визуальный редактор форм/экранов, в котором пользователь расставляет UI элементы и получает полный код приложения, или консольный скрипт, который генерирует «скелет» приложения.]

5. Ещё более упростить такой подход можно, если результатом работы такого сервиса будет уже не код поверх API, а готовый компонент / виджет / фрейм, подключаемый одной строкой. [Например, если мы дадим возможность разработчику вставлять на свой сайт iframe, в котором можно заказать кофе, кастомизированный под нужды заказчика, либо, ещё проще, описать правила формирования «deep link-а»¹, который приведёт пользователя на наш сервис.]

В конечном итоге можно прийти к концепции мета-API, когда готовые визуальные компоненты тоже будут иметь какое-то свой высокуюровневый API, который «под капотом» будет обращаться к базовым API.

Важным преимуществом наличия линейки продуктов API является не только возможность адаптировать его к возможностям конкретного разработчика, но и увеличение уровня вашего контроля над кодом, встроенным в приложение партнёра.

1. Приложения, использующие физические интерфейсы, полностью вне вашей досягаемости; вы не можете, например, форсировать переход на новые версии технологической платформы или, скажем, добавить в них рекламные размещения.
2. Приложения, оперирующие базовым уровнем API, позволяют вам менять нижележащие уровни абстракции — переходить на новые технологии, манипулировать выдачей и представлением данных.
3. SDK, особенно имеющие визуальные компоненты в составе, дают гораздо более широкий контроль над внешним видом партнерских приложений и их взаимодействием с пользователем, что позволяет развивать UI, добавлять новые виды интерактивных элементов и обогащать функциональность старых. [Например, если SDK нашего кофейного API содержит в себе карту кофеен, ничего не может нам помешать в новой версии SDK сделать объекты на карте кликабельными или, например, выделять оплаченные размещения цветом.]
4. Кодогенерация позволяет вам манипулировать желательным видом приложений. Например, если для вас важным показателем является количество поисков через сторонние приложения, вы можете добавить в генерированный код показ панели поиска на видном месте; пользователи, прибегающие к помощи генератора кода, как правило, не меняют сгенерированный результат.

5. Наконец, готовые компоненты и виджеты находятся полностью под вашим контролем, и вы можете экспериментировать с доступной через них функциональностью так же свободно, как если бы это было ваше собственное приложение. (Здесь следует, правда, отметить, что не всегда от этого контроля есть толк: например, если вы позволяете вставлять изображение по прямому URL, ваш контроль над этой интеграцией практически отсутствует; при прочих равных следует выбирать тот вид интеграции, который позволяет получить больший контроль над соответствующей функциональностью в приложении партнёра.)

NB. При разработке «вертикального» семейства API замечания, описанные в главе «[О ватерлинии айсберга](#)» особенно важны. Вы можете свободно манипулировать контентом и поведением виджета, если и только если у разработчика нет способа «сбежать из песочницы», т.е. напрямую получить низкоуровневый доступ к объектам внутри виджета.

Как правило, вы должны стремиться к тому, чтобы каждый партнёрский сервис использовал тот вид API, который вам как разработчику наиболее выгоден. Там, где партнёр не стремится создать какую-то уникальную функциональность и размещает типовое решение, вам выгодно иметь виджет, который полностью находится под вашим контролем и, с одной стороны, снимает с вас головную боль относительно обновления версий API, и, с другой стороны, даёт вам свободу экспериментировать с внешним видом и поведением интеграций с целью оптимизации ваших KPI. Там, где партнёр обладает экспертизой и желанием разработать какой-то уникальный сервис поверх вашего API, вам выгодно предоставить ему максимальную свободу действий, чтобы, во-первых, покрыть тем самым уникальные продуктовые ниши, и, во-вторых, обладать конкурентным преимуществом в виде возможности глубокой кастомизации относительно других API на рынке.

Примечания

¹ Mobile Deep Linking

https://en.wikipedia.org/wiki/Mobile_deep_linking

Глава 56. Ключевые показатели эффективности API

Как мы описали выше, существует большое количество различных моделей монетизации API, прямой и косвенной. Важной их особенностью является то, что, во-первых, большинство из них является частично или полностью бесплатными для партнёра, а, во-вторых, соотношение прямой и косвенной выгоды часто меняется в течение жизненного цикла API. Возникает вопрос, каким же образом следует измерять успех API и какие цели ставить продуктовой команде.

Разумеется, наиболее прямым измеримым показателем являются деньги: если ваш API имеет прямую монетизацию либо явно приводит пользователей на монетизируемый сервис, то остальная часть главы будет вам интересна разве что в рамках общего развития. Если же напрямую измерить вклад API в доходы компаний не получается, придётся прибегать к иным, синтетическим, показателям.

Очевидный ключевой показатель эффективности (Key Performance Indicator, KPI) №1 — это количество конечных пользователей и количество интеграций (читай, партнёров, использующих API). В нормальной ситуации он является в определённом смысле барометром состояния бизнеса: если предположить, что на рынке наблюдается здоровая конкуренция между API разных поставщиков, и все находятся в более-менее одинаковом положении, то количество использующих API разработчиков (и как производная — конечных пользователей) и есть главный показатель успеха продукта.

Однако чистые цифры количества пользователей и партнёров могут вводить в заблуждение, особенно в случае бесплатных инсталляций. Есть несколько факторов, искажающих статистику:

- сервисы в линейке API-продуктов, рассчитанные на простую интеграцию (см. предыдущую главу) существенно искажают статистику количества партнёров в сравнении с конкурентами, если у них таких сервисов нет; зачастую на одну глубокую интеграцию приходятся десятки, если не сотни более простых; таким образом, подсчёт партнёров следует как минимум делить по видам интеграции;
- партнёры склонны использовать API неоптимально:

- подключать его на всех страницах / экранах приложения, а не только там, где пользователь реально с ним взаимодействует;
 - размещать виджеты глубоко в «подвале» или за спойлером;
 - инициализировать широкий набор модулей, но использовать только тривиальную функциональность;
- чем шире распространён ваш API, тем менее значим показатель количества уникальных пользователей, поскольку в какой-то момент проникновение API приблизится к 100%; например, с сервисами Facebook или Google в виде счётчиков и виджетов средний пользователь Интернета встречается чуть ли не ежеминутно, и дневная аудитория этих API уже в принципе вырасти не может.

Вышеперечисленные проблемы приводят нас к простому выводу: считать нужно не только сырье цифры количества уникальных пользователей и партнёров, но и их вовлечённость, т.е. выделить целевые действия (например, количество поисков через платформу, показ определённых данных, события взаимодействия пользователя с виджетом и т.д.). В идеале эти целевые действия должны коррелировать с монетизацией:

- если API монетизируется за счёт рекламы — считать, сколько раз в день пользователи взаимодействуют с рекламой;
- если API приводит пользователя на сайт материнского сервиса — считать переходы;
- если API используется для сбора обратной связи и UGC — считать оставленные отзывы и исправления.

Довольно часто встречаются также такие KPI как использование той или иной функциональности API (в том числе для приоритизации планов разработки). По сути это те же целевые действия, но только совершаемые разработчиком, а не пользователем. Для библиотек, особенно клиентских, сбор этой информации бывает затруднён, но не невозможен (хотя крайне желательно подходить к вопросу максимально аккуратно, поскольку любая аудитория сегодня крайне нервно реагирует на автоматический сбор любой статистики).

Наиболее непростая ситуация с KPI наблюдается, если API в первую очередь способ (техно)пиара и (техно)маркетинга. В этом случае наблюдается накопительный эффект: наращивание аудитории не конвертируется в пользу компании моментально. *Сначала* вы набираете большую лояльную аудиторию разработчиков, *потом* репутация вашей компании улучшается, и *ещё* более

потом эта репутация начинает играть вам на руку при найме. Сначала логотип вашей компании появляется на сторонних сайтах и приложениях, потом top-of-mind знание бренда увеличится. Нет прямого способа отследить, как то или иное действие (например, релиз новой версии или проведение мероприятия) сказывается на целевых показателях. В этом случае приходится оперировать косвенными показателями — посещаемость ресурсов для разработчиков, количество упоминаний в тематических сообществах, популярность блогов и семинаров и т.п.

Кратко суммируем вышесказанное:

- считать прямые показатели, такие как общее число пользователей и партнёров, — необходимый минимум, без которого сложно двигаться дальше, но это не KPI;
- KPI для продукта API должен выставлять исходя из количества целевых действий, которые производятся через платформу;
- определение «целевых действий» зависит от модели монетизации и может быть как прямолинейным «количество платящих клиентов» или «количество кликов по рекламе», так и достаточно опосредованным и эвристическим типа «количество посетителей блога команды разработки».

SLA

Невозможно в этом разделе не упомянуть и о «гигиеническом KPI» — уровне предоставляемых услуг и доступности сервиса. Мы не будем здесь давать детального описания, поскольку SLA API ничем не отличается от SLA других видов цифровых сервисов, отметим лишь то, что следить за ним, разумеется, надо, особенно в случае платных API. Впрочем, во многих случаях провайдеры API обычно ограничиваются достаточно свободным SLA, трактуя тарифицируемые услуги как услуги доступа к информации или лицензирование контента.

Тем не менее, позволим себе ещё раз напомнить: любые проблемы вашего API автоматически умножаются на количество партнёров, особенно в тех случаях, когда ваш API критически для них важен, т.е. при неработоспособности API становится недоступной основная функциональность сервиса. (Впрочем, по упомянутым выше причинам качество интеграции большей части партнёров

почти неизбежно будет таково, что ошибки в работе их сервисов будут происходить, даже если ваш API не является формально для них критическим — а потому, что разработчики подключают API даже там, где оно формально не нужно, и пренебрегают обработкой ошибок.)

Важно отметить, что нагрузку на API, вообще говоря, крайне сложно предсказать. Неоптимальное использование API, т.е. его инициализация в тех разделах приложений, где он в реальности не нужен, может привести к колоссальному росту нагрузки вследствие перемещения одной-единственной строки кода партнёра. «Запас прочности» API-сервис должен быть гораздо выше, чем у обычных пользовательских сервисов — как минимум на уровне, достаточном для поддержания его работоспособности в том случае, если крупнейший партнёр начнёт вызывать API при загрузке любой страницы вебсайта / открытии любого экрана приложения. (Если партнёр уже так делает — то API должен переживать как минимум удвоение этой нагрузки на тот случай, если разработчики случайно начнут инициализировать API дважды.)

Другой важнейший гигиенический минимум — это обеспечение информационной безопасности API. В худшем из возможных сценариев, если посредством эксплуатации уязвимости в API можно будет наносить вред конечным пользователем, фактически дыра в безопасности будет создана *в каждом приложении партнёра*. Излишне уточнять, что цена такой ошибки может оказаться невероятно большой даже если само API достаточно невинно и никакого доступа к чувствительным данным не имеет (особенно если речь идёт о веб-страницах, где в принципе нет никакой «песочницы» для сторонних скриптов, и любой код на странице может, например, отследить вводимые данные в формы). Сервисы API обязаны как использовать максимальные меры защиты (например, с запасом выбирать надёжные криптографические протоколы), так и максимально оперативно реагировать на сообщения об уязвимостях.

Сравнение с конкурентами

При измерении КPI любого сервиса критически важно измерять не только свои собственные показатели, но и состояние рынка:

- какую долю рынка вы занимаете, и как она меняется во времени?
- растёт ли ваш сервис быстрее рынка, в темпе рынка или медленнее его?

- какая доля прироста обеспечена ростом самого рынка, а какая — вашими действиями?

Получить ответы на эти вопросы в случае сервисов API может быть достаточно нетривиально. В самом деле, как выяснить, сколько интеграций за тот же период времени выполнил конкурент, и какое количество целевых действий пользователя выполняется через конкурирующие API? Иногда вам могут помочь с этими данными компании-разработчики аналитических инструментов для приложений, но чаще всего вам нужно будет самостоятельно заниматься мониторингом крупных площадок, которые потенциально могли бы быть интегрированы с вашим API, и отмечать, какие конкурирующие сервисы они используют. Аналогичная ситуация наблюдается и ростом рынков: если ваша ниша недостаточно заметна для того, чтобы крупная независимая аналитическая компания выпустила его исследование, вам придётся или заказать такое исследование за свой счёт, или прикинуть нужные цифры самостоятельно — аналогичным образом, через исследование потенциальных потребителей.

Глава 57. Идентификация пользователей и борьба с фродом

В контексте работы с API мы говорим о двух видах пользователей системы:

- пользователи-разработчики, т.е. ваши партнёры, разрабатывающие код поверх вашего API;
- конечные пользователи, которые будут работать с приложениями, написанными партнерами с использованием вашего API.

И тех, и других в большинстве случаев необходимо уметь идентифицировать (в техническом смысле, т.е. уметь считать уникальные визиты), чтобы иметь ответы на следующие вопросы:

- сколько пользователей взаимодействуют с системой (одновременно, в течение дня, месяца, года);
- какое количество действий совершает каждый пользователь.

NB. Иногда, в случае больших и/или абстрактных API цепочка между вашим API и финальным пользователем может содержать более одного разработчика, т.е. крупные партнёры предоставляют сервис, разработанный поверх API, более мелким. Считать нужно уметь и прямых партнёров, и «производных».

Обладать этой информацией критически важно по двум основным причинам:

- чтобы понимать пределы прочности системы и уметь планировать её развитие;
- чтобы понимать количество ресурсов (в пределе — денег), расходуемых (и зарабатываемых) на каждого пользователя.

В случае коммерческих API точность и своевременность сбора этой информации важна вдвойне, поскольку от неё напрямую зависят параметры тарифов и бизнес-модель в целом; поэтому вопрос *как* мы идентифицируем пользователей — отнюдь не праздный.

Идентификация приложений и их владельцев

Начнём с первой категории, то есть пользователей-клиентов API. Сделаем здесь важное уточнение: нам необходимо идентифицировать две различные сущности — приложения и их владельцев.

Приложение — это, грубо говоря, какой-то логически отдельный кейс использования API, чаще всего — в прямом смысле слова приложение (мобильное или десктопное) или веб-сайт, т.е. некоторая техническая сущность. В то же время владелец — это тот, с кем вы заключаете договор использования API, т.е. юридическая сущность. Если схема тарификации API подразумевает систему лимитов и/или тарифы зависят от вида сервиса или способа его использования, то это автоматически означает необходимость тарифицировать приложения одного владельца раздельно.

В современном мире фактический стандарт идентификации (и того, и другого) — это использование API-ключей: разработчик, желающий воспользоваться API, должен явно получить ключ, оставив свои контактные данные. Ключ, таким образом, идентифицирует приложение, а контактные данные — владельца.

Несмотря на широкое распространение этой практики мы не можем не отметить, что в большинстве случаев она бесполезна, а иногда и вредна.

Её несомненным преимуществом является обязанность каждого клиента предоставить актуальные контактные данные, что (теоретически) позволяет связываться с владельцем приложения. (Что в реальном мире не совсем так — в значительном проценте случаев владелец не читает почту, оставленную в качестве контактной; в случае корпоративных клиентов это вовсе может бытьничейный почтовый ящик или личная почта давно уволившегося сотрудника.)

Проблема же API-ключей заключается в том, что они *не позволяют* надёжно идентифицировать ни приложение, ни владельца.

Если API предоставляется с какими-то бесплатными лимитами, то велик соблазн завести множество ключей, оформленных на разных владельцев, чтобы оставаться в рамках бесплатных лимитов. Вы можете повышать стоимость заведения таких мультиаккаунтов, например, требуя привязки номера телефона или кредитной карты, однако и то, и другое — в настоящий момент широко распространённая услуга. Выпуск виртуальных телефонных

номеров или виртуальных кредитных карт (не говоря уже о нелегальных способах приобрести краденые) всегда будет дешевле, чем честная оплата использования API — если, конечно, это не API выпуска карт или номеров. Таким образом, идентификация пользователя по ключам (если только ваш API не является чистым B2B и для его использования нужно подписать физический договор) никак не освобождает от необходимости перепроверять, действительно ли пользователь соблюдает правила и не заводит множество ключей для одного приложения.

Другая опасность заключается в том, что ключ могут банально украсть у добросовестного партнёра; в случае клиентских и веб-приложений это довольно тривиально.

Может показаться, что в случае предоставления серверных API проблема воровства ключей неактуальна, но, на самом деле, это не так. Предположим, что партнёр предоставляет свой собственный публичный сервис, который «под капотом» использует ваше API. Это часто означает, что в сервисе партнёра есть эндпойнт, предназначенный для конечных пользователей, который внутри делает запрос к API и возвращает результат, и этот эндпойнт может использоваться злоумышленником как эквивалент API. Конечно, можно объявить такой фрод проблемой партнёра, однако было бы, во-первых, наивно ожидать от каждого партнёра реализации собственной антифрод-системы, которая позволит выявлять таких недобросовестных пользователей, и, во-вторых, это попросту неэффективно: очевидно, что централизованная система борьбы с фрорами всегда будет более эффективной, нежели множество частных любительских реализаций. К тому же, и серверные ключи могут быть украдены: это сложнее, чем украсть клиентские, но не невозможно. Популярный API рано или поздно столкнётся с тем, что украденные ключи будут выложены в свободный доступ (или владелец ключа просто будет делиться им со знакомыми по доброте душевной).

Так или иначе, встаёт вопрос независимой валидации: каким образом можно проконтролировать, действительно ли API используется потребителем в соответствии с пользовательским соглашением.

Мобильные приложения удобно отслеживаются по идентификатору приложения в соответствующем сторе (Google Play, App Store и другие), поэтому разумно требовать от партнёров идентифицировать приложение при подключении API. Вебсайты с некоторой точностью можно идентифицировать по заголовкам `Referer` или `Origin` (и для надёжности можно также

потребовать от партнёра указывать домен сайта при инициализации API).

Эти данные сами по себе не являются надёжными; важно то, что они позволяют проводить кросс-проверки:

- если ключ был выпущен для одного домена, но запросы приходят с Referer-ом другого домена — это повод разобраться в ситуации и, возможно, забанить возможность обращаться к API с этим Referer-ом или этим ключом;
- если одно приложение инициализирует API с указанием ключа другого приложения — это повод обратиться к администрации стора с требованием удалить одно из приложений.

NB: не забудьте разрешить безлимитное использование с Referer-ом localhost и 127.0.0.1 / [::1], а также из вашей собственной песочницы, если она есть. Да, в какой-то момент злоумышленники поймут, что на такие Referer-ы не действуют ограничения, но это точно произойдёт гораздо позже, чем вы по неосторожности забаните локальную разработку или собственный сайт документации.

Общий вывод из вышеизложенного таков:

- очень желательно иметь формальную идентификацию пользователей (API-ключи как самая распространённая практика, либо указание контактных данных, таких как домен вебсайта или идентификатор приложения в сторе, при инициализации API);
- доверять этой информации безусловно ни в коем случае нельзя: должны существовать механизмы проверки, которые позволяют найти подозрительные запросы.

Идентификация конечных пользователей

Если к партнёрам вы можете предъявлять какие-то требования по самоидентификации, то от конечных пользователей требовать раскрытия информации о себе в большинстве случаев не представляется возможным. Все методы контроля, описанные ниже, являются неточными и зачастую эвристическими. (Даже если функциональность партнёрских приложений предоставляется только после регистрации пользователя и вы имеете к этой регистрации доступ, вы всё ещё гадаете, т.к. аккаунт это не то же самое, что и отдельный пользователь: несколько различных людей могут пользоваться одним профилем или, наоборот, у одного человека может быть множество профилей.) Кроме того, следует иметь в виду, что сбор подобного рода информации может регулироваться законодательно (хотя большей частью речь пойдёт об анонимизированных данных, но и они могут быть регламентированы).

1. Самый простой и очевидный показатель — это IP-адреса; их невозможно подделать (в том смысле, что сервер API всегда знает адрес вызывающего клиента), и поэтому статистика по уникальным IP довольно показательна.

Если API предоставляется как server-to-server сервис, доступа к IP-адресу конечного пользователя может и не быть, однако весьма разумно в такой ситуации требовать от партнёра пробрасывать IP-адрес клиента (например, в виде заголовка X-Forwarded-For) — в том числе для того, чтобы помочь партнёрам бороться с фрэном и неправомерным использованием API.

До недавнего времени ip-адрес как единица подсчёта статистики был ещё и удобен тем, что обзавестись большим пулом уникальных адресов было достаточно дорого. Однако с распространением ipv6 это ограничение перестало быть актуальным; скорее, ipv6 ярко подсветил тот факт, что не стоит ограничиваться только подсчётом уникальных ip. Необходимо следить за несколькими агрегатами:

- суммировать статистику по подсетям, т.е. вести иерархические подсчёты (количество уникальных сетей /8, /16, /24 и так далее);
- наблюдать за агрегированной статистикой по автономным сетям (autonomous networks, AS);
- мониторить использование известных публичных прокси и TOR Network.

Необычно высокое количество запросов из одной подсети может свидетельствовать о том, что API активно используется во внутренкорпоративной сети (или в данном регионе доступ в Интернет в основном предоставляется через NAT).

2. Дополнительным способом идентификации служат уникальные идентификаторы пользователей, в первую очередь — cookie. Однако в последние годы этот способ ведения статистики подвергается атаке с нескольких сторон: производители браузеров ограничивают возможности установки cookie третьей стороной, пользователи активно защищаются от слежения, и законодатели начали выдвигать требования в отношении сбора данных. В рамках текущего законодательства проще отказаться от использования cookie, чем соблюсти все необходимые требования.

Всё это приводит к тому, что публичным API, особенно используемым в бесплатных сайтах и приложениях, очень тяжело вести статистику, а значит и тяжело анализировать поведение пользователей. И речь здесь не только о борьбе с разного рода фрэном, но и банальном анализе сценариев использования API. Таков путь.

NB. В некоторых юрисдикциях IP-адреса считаются персональными данными, и их сбор также запрещён. Мы не берёмся давать советы, каким образом поставщик API должен одновременно уметь бороться с незаконным контентом на платформе и при этом не иметь доступа к IP-адресам пользователей. Предполагаем, что для соответствия такого рода законодательству необходимо будет хранить статистику (и банить пользователей) по хэшам IP-адресов. (На всякий случай, мы не будем здесь уточнять, что построение радужной таблицы SHA-256 хэшей для 4 млрд возможных IPv4 адресов — задача на несколько часов работы обычного офисного компьютера.)

Глава 58. Технические способы борьбы с несанкционированным доступом к API

Реализация парадигмы, описанной в предыдущей главе — централизованной борьбы с фрода, осуществляемым через клиентские API партнёра — на практике сталкивается с достаточно нетривиальными проблемами.

Задача отсеивания нежелательных запросов, в общем случае, состоит из трёх шагов:

- идентификация подозрительных пользователей;
- опционально, запрос дополнительного фактора аутентификации;
- вынесение и применение решения об ограничении доступа.

1. Идентификация подозрительных пользователей

По большому счёту, здесь есть всего два подхода, которые мы можем применить — статический и динамический (поведенческий).

Статически мы отслеживаем подозрительную концентрацию активности (как описано в предыдущей главе), отмечая нехарактерно высокое количество запросов из определённых подсетей или Referer-ов (на самом деле, нам подойдёт любая информация, которая как-то делит пользователей на более-менее независимые группы — такая как, например, версия ОС или язык системы, если такие данные нам доступны).

При *поведенческом* анализе мы анализируем историю запросов одного конкретного пользователя и отмечаем нетипичное поведение — «нечеловеческий» порядок обхода эндпоинтов, слишком быстрый их перебор, etc.

Важно: когда мы здесь говорим о «пользователе», нам почти всегда придётся дублировать анализ для работы по ip-адресу, поскольку злоумышленник вовсе не обязан будет сохранять cookie или другой идентификационный токен, или будет ротировать набор таких токенов, чтобы затруднить идентификацию.

2. Запрос дополнительного фактора аутентификации

Поскольку и статический, и поведенческий анализ эвристические, очень желательно не просто выносить решение на их основе, но предлагать подозрительным пользователям дополнительно доказать, что они совершают легитимные запросы. Если такие механизмы есть, качество работы анти-фрод системы существенно возрастает, поскольку тогда допустимо будет снизить порог срабатывания или вовсе включить проактивную защиту, т.е. предлагать пользователям пройти дополнительную проверку превентивно.

В случае сервисов для конечных пользователей основным методом дополнительной аутентификации является перенаправление на страницу с капчей. В случае API это может быть весьма проблематично, особенно если вы пренебрегли советом «[Предусмотрите ограничения](#)» — во многих случаях вам придётся переложить имплементацию этого сценария на партнёра (т.е. это партнёр должен будет показывать капчу и идентифицировать пользователя, основываясь на сигналах, поступающих от эндпойнтов API) что, конечно, сильно снижает комфортность работы с таким API.

NB. Вместо капчи здесь могут быть любые другие действия, вводящие дополнительные факторы аутентификации. Это может быть, например, подтверждение номера телефона или второй шаг протокола 3D-Secure. Важно здесь то, что запрос второго шага аутентификации должен быть предусмотрен в API, поскольку добавить его обратно совместимым образом к существующим endpoint-ам нельзя.

Другие популярные способы распознать робота — предложить ему приманку (honeypot) или использовать методы проверки среды исполнения (начиная от достаточно простых вроде исполнения JavaScript на странице и заканчивая технологиями проверки целостности приложения).

3. Ограничение доступа

Видимость богатства способов технической идентификации пользователей, увы, разбивается о суровую реальность наличия у вас очень скромных средств ограничения доступа. Бан по cookie / Referer-у / User-Agent-у практически не работает по той причине, что эти данные передаёт клиент, и он же легко может их подменить. По большому счёту, способов ограничения доступа у вас четыре:

- бан пользователя по ip (подсети, автономной системе);
- требование обязательной идентификации пользователя (возможно, прогрессивной: логин в системе / логин с подтверждённым номером телефона / логин с подтверждением личности / логин с подтверждением личности и биометрией);
- отдача ложного ответа;
- борьба административными методами.

Вариант номер один плох тем, что наносит огромный сопутствующий ущерб, особенно если вам придётся банить подсети.

Второй вариант, при всём его удобстве, мало применим в случае реальных API, поскольку на него будут согласны далеко не все партнёры и уж точно далеко не все пользователи, и к тому же потребует от вас дополнительных мер по соблюдению требований законодательства о персональных данных.

Третий вариант с точки зрения эффективности противоборства атаке является наиболее предпочтительным, поскольку перекидывает мяч на ту сторону: теперь уже злоумышленнику нужно каким-то образом определять, был ли он пойман. Но с точки зрения морали (и буквы закона) этот способ весьма сомнителен — особенно если учесть, что всегда возможны ложноположительные срабатывания, и некорректные данные будут отданы честному пользователю.

Поэтому по факту у вас есть только один действительно работающий метод борьбы с фродом — жалоба провайдеру, хостеру или правоохранительным органам. Излишне уточнять, что способ этот несёт репутационные издержки и при этом реакция наступает отнюдь не молниеносно.

В большинстве случаев вы на самом деле не боретесь с фродом — вы повышаете атакующему стоимость атаки, выигрывая себе время на принятие решения о преследовании нарушителя в административном порядке. Предотвратить атаку полностью невозможно, поскольку у злоумышленника всегда есть в запасе дорогой, но работающий способ: посадить реальных людей с реальными приложениями, чтобы они выполняли нужные запросы к API и были неотличимы от обычных пользователей.

Существует мнение, разделяемое автором настоящей книги, что, ввязываясь в эту борьбу щита с мечом, нужно очень аккуратно использовать технически продвинутые методы борьбы — только в том случае, когда вы уверены, что оно того стоит (читай — если злоумышленники воруют реальные деньги или данные). Вводя сложные алгоритмы, вы тем самым проводите своеобразный «эволюционный отбор», направленный на выявление самых умных и хитрых злоумышленников, противодействовать которым будет гораздо сложнее, чем наивным попыткам вызывать методы API curl-ом. Что ещё важнее, в финальной фазе — т.е. при обращении в контролирующие инстанции — вам придётся предъявить доказательства нарушения, и сделать это в отношении продвинутого противника будет не в пример сложнее. Поэтому лучше держать нарушителей на карандаше, т.е. мониторить их и регулярно слать жалобы, и эскалировать ситуацию (т.е. переходить к техническим мерам защиты и юридическим действиям) только в случае наличия реальной угрозы. Это, кстати, означает, что все приборы и механизмы у вас должны быть готовы и ожидать своего часа в пассивном режиме.

По опыту автора этой книги играть в игры со злоумышленниками по принципу на каждое улучшение их скрипта отвечать минимальным изменением в линии защиты можно очень долго. Такая стратегия — т.е. заставлять фродера каждый раз пытаться понять, по какому признаку он был забанен теперь, а не выводить сразу всю тяжёлую артиллерию — чрезвычайно раздражает «хакеров»-любителей, которые страдают от недостатка навыков разработки и часто в итоге просто сдаются и прекращают свои попытки.

Противодействие краже ключей

Рассмотрим теперь второй вариант несанкционированного использования, когда злоумышленник крадёт API-ключ добросовестного партнёра и вставляет его в своё приложение. Запросы при этом генерируются настоящими пользователями, а значит капча никак не поможет — но помогут другие методы.

1. Ведение статистики по ip-адресам и сетям может помочь и здесь. Если приложение злоумышленника всё-таки не обычное приложение для честных потребителей, а какой-то закрытый сервис для ограниченного круга пользователей, этот факт будет виден на приборах (а если повезёт — то вы увидите ещё и подозрительные Referer-ы, закрытые для внешнего доступа).

2. Предоставление возможности партнёрам ограничивать функциональность, которая доступна по ключу:

- устанавливать диапазон допустимых IP-адресов для серверных API, идентификаторов приложений и хостов в клиентских API;
- разрешать использование конкретного ключа только для конкретных методов API;
- вводить иные разумные ограничения (например, для ключа нашего кофейного API можно установить ограничения по странам и городам, в которых работает партнёр).

3. Дополнительное подписывание запроса:

- например, если на странице вебсайта партнера осуществляется поиск лучших предложений лунго, для чего клиент обращается к URL вида `/v1/search?recipe=lungo&api_key={apiKey}`, то API-ключ может быть заменён на сгенерированную сервером подпись вида `sign = HMAC("recipe=lungo", apiKey)`; такая подпись может быть украдена, но будет бесполезна для злоумышленника, так как позволяет найти только лунго;
- вместо API-ключа можно использовать одноразовые пароли (Time-Based One-Time Password, TOTP); такие токены действительны, как правило, в течение короткого времени, порядка минуты, что чрезвычайно затрудняет злоумышленнику работу с украденными ключами.

4. Обращаться к администрации (хостерам и владельцам магазинов приложений) — в случае, если нарушитель распространяет своё приложение легально или пользуется услугами добросовестного хостера, рассматривающего такого рода обращения. Обращение в правоохранительные органы и суды тоже вполне осмысленно, и даже более разумно, чем в случае фрода от имени пользователей, так как несанкционированный доступ к компьютерным системам с использованием украденных ключей в большинстве юрисдикций чётко подпадает под уголовный кодекс.

5. Банить скомпрометированные ключи; эта операция почти всегда вызовет негативную реакцию партнёра, но, в конце концов, для многих бизнесов лучше временно лишиться какой-то функциональности в приложении, чем получить многомиллионный счёт.

Глава 59. Поддержка пользователей API

Прежде всего сделаем важную оговорку: когда мы говорим о поддержке пользователей API, мы имеем в виду поддержку разработчиков и отчасти — бизнес-партнёров. Конечные пользователи, как правило, напрямую с API не взаимодействуют, за исключением некоторых нестандартных сценариев.

1. Если до партнёров, неправильно использующих API, невозможно «достучаться» по иным каналам, и приходится отображать в их приложениях видимую конечным пользователям ошибку. Такое случается, если в фазе роста API предоставлялся бесплатно и с минимальными требованиями по идентификации партнёров, и позднее условия изменились (популярная версия API перестала поддерживаться или стала платной).
2. Если разработчик API не может самостоятельно воспроизвести некоторую проблему, и вынужден обращаться напрямую к конечному пользователю с целью сбора обратной связи.
3. Если через API осуществляется сбор UGC-контента.

Первые два сценария по сути представляют собой ошибки продуктового или технического развития API, и их желательно не допускать. Третий сценарий не имеет особенной API-специфики, он по факту мало отличается от поддержки пользователей на основном UGC-сервисе.

Поддержку же собственно API можно разделить на два больших раздела:

- юридическая и организационная поддержка по вопросам условий использования и SLA сервиса (здесь скорее речь идёт об ответах на вопросы бизнес-партнёров);
- поддержка разработчиков по возникающим техническим вопросам.

Первый раздел, несомненно, критически важен для любого здорового продукта (включая API), но вновь не содержит в себе какой-то особенной специфики. С точки зрения содержания настоящей книги нас гораздо больше интересует второй раздел.

Поскольку API — программный продукт, разработчики фактически будут задавать вопросы о работе того или иного фрагмента кода, который они пишут. Этот факт сам по себе задирает планку требований к качеству специалистов поддержки до очень высокого уровня, поскольку прочитать код и понять причину проблемы может только разработчик же. Но это полбеды: другая сторона проблемы заключается в том, что, как мы упоминали в предыдущих главах, львиная доля этих запросов будет задана неопытными или непрофессиональными разработчиками, что в случае любого сколько-нибудь популярного API приводит к тому, что 9 из 10 запросов *будут вовсе не про работу API*. Начинающие разработчики плохо владеют языком, обладают фрагментарными знаниями об устройстве платформы и не умеют правильно формулировать свои проблемы (и как следствие не могут поискать ответ на свой вопрос в Интернете перед тем; хотя, будем честны, в большинстве случаев даже и не пробуют).

Вариантов работы с такими обращениями может быть несколько.

1. Наиболее дружелюбный сценарий — набор людей с базовыми техническими навыками в первую линию поддержки. Эти сотрудники должны достаточно хорошо разбираться в механике работы API, чтобы опознавать непрофильные вопросы и отвечать на них по FAQ или переадресовывать вовне, если это требуется (например, в техподдержку платформы или комьюнити языка), и перенаправлять действительно релевантные вопросы разработчикам API.
2. Обратный сценарий — когда техподдержка предоставляется только на платной основе, и на вопросы отвечают непосредственно разработчики; пусть на качество и релевантность запросов такая модель не оказывает большого влияния (вашим API продолжают пользоваться, в основном, новички; вы лишь отсекаете тех из них, у кого нет денег на платную поддержку), но, по крайней мере, вы не будете испытывать проблем с наймом, поскольку сможете позволить себе роскошь поставить технического специалиста на первую линию поддержки.
3. Частично или полностью проблему с поддержкой новичков может снять развитое комьюнити (см. главу «[Взаимодействие с разработчиками](#)»). Как правило, члены комьюнити в состоянии ответить на вопросы новичков, особенно если им активно помогают модераторы.

Важный момент заключается в том, что, какой вариант оказания техподдержки вы ни выберете, финально на вопросы пользователей придётся отвечать разработчикам API просто в силу того факта, что полноценно разобраться в коде партнёра может только программист. Из этого следует два важных вывода.

1. Время на оказание технической поддержки необходимо закладывать при планировании. Чтение совершенно незнакомого кода и удалённая его отладка — крайне сложная и отнимающая большое количество времени задача. Чем больше функциональности вы публикуете и чем больше платформ поддерживаете, тем выше будет нагрузка на команду разработки в части разбора обращений.
2. При этом программисты, как правило, не испытывают никакого восторга, занимаясь разбором обращений. Фильтр в лице первой линии поддержки всё равно не спасает от дилетантских и/или плохо сформулированных вопросов, что вызывает заметное раздражение дежурных разработчиков API. Выходов из этой ситуации несколько:
 - по возможности старайтесь найти людей, которым по складу ума нравится заниматься такой деятельностью, и поощряйте их заниматься поддержкой (в т.ч. материально); это может быть кто-то из команды (причём вовсе не обязательно разработчик) или кто-то из активных членов комьюнити;
 - остаточная нагрузка на команду должна быть распределена максимально равномерно и честно, вплоть до введения календаря дежурств.

Разумеется, полезным упражнением будет анализ вопросов и ответов с целью дополнения FAQ-ов, внесения изменений в документацию и скрипты работы первой линии поддержки.

Внешние платформы

Рано или поздно вы обнаружите, что пользователи задают вопросы о работе с вашим API не только через официальные каналы, но и на многочисленных публичных интернет-площадках, начиная от специально предназначенных для этого сервисов типа StackOverflow и заканчивая социальными сетями и личными блогами. Тратить ли на поиск подобных обращений время — решать вам; мы бы рекомендовали оказывать техническую поддержку на тех площадках, которые предоставляют для этого удобные инструменты (типа возможности подписаться на новые вопросы по конкретным тэгам).

Глава 60. Документация

К сожалению, многие разработчики API уделяют справочной документации прискорбно мало внимания; между тем документация является ни много ни мало лицом продукта и точкой входа в него. Проблема усугубляется тем, что написать хотя бы удовлетворительную с точки зрения разработчиков документацию невероятно сложно.

Прежде, чем мы перейдём к описанию видов и форматов документации, хотелось бы проговорить очень важную мысль: пользователи взаимодействуют со справкой по вашему API совершенно не так, как вы себе это представляете. Вспомните, как вы сами работаете над проектом: вы выполняете вполне конкретные шаги.

1. Необходимо установить (чем быстрее, тем лучше!) подходит ли в принципе данный сервис для вашего проекта.
2. Если да, то нужно найти, как с его помощью решить конкретную задачу.

Фактически, новички (т.е. те разработчики, которые не знакомы с вашим API), как правило, хотят ровно одного: скомпоновать из имеющихся примеров код, решающий их конкретную задачу, и больше никогда к этому не возвращаться. Звучит, конечно, не очень-то привлекательно с точки зрения усилий и времени, которые вы затратили на разработку API и документации к нему, но суровая реальность выглядит именно так. Кстати, поэтому же разработчики всегда будут недовольны качеством вашей документации — поскольку решительно невозможно предусмотреть все возможные варианты, какие комбинации каких примеров нужны новичкам, и какие конкретно термины и концепции в примере окажутся им непонятны. Добавим к вышесказанному то соображение, что не-новичкам, то есть разработчикам, уже знакомым с системой и ищущим решения каких-то сложных специфических кейсов, как раз совершенно бесполезны комбинации простых примеров, и им, ровно наоборот, нужны подробные материалы по продвинутой функциональности API.

Вводные замечания

Документация зачастую страдает от канцелярита: её пишут, стараясь использовать строгую терминологию (зачастую требующую изучения гlosсария перед чтением такой документации) и беспричинно раздувают — так, чтобы вместо простого ответа из двух слов на вопрос пользователя получался абзац текста. Мы такую практику категорически осуждаем: идеальная документация должна быть проста и лаконична, а все термины должны быть снабжены расшифровками или ссылками прямо в тексте (однако, простая не значит неграмотная: помните, документация — лицо вашего продукта, грамматические ошибки и вольности использования терминов здесь недопустимы).

Однако следует учесть, что документация будет использоваться также и для поиска по ней — таким образом, каждая страница должна содержать достаточные наборы ключевых слов, чтобы находиться в поиске. Это несколько противоречит требованию лаконичности и компактности, но таков путь.

Виды справочных материалов

1. Спецификация / справочник / референс

Любая документация начинается с формального описания доступной функциональности. Да, этот вид документации будет максимально бесполезен с точки зрения удобства использования, но не предоставлять его нельзя — справочник является гигиеническим минимумом. Если у вас нет документа, в котором описаны все методы, параметры и настройки, типы всех переменных и их допустимые значения, зафиксированы все опции и поведения — это не API, а просто какая-то самодеятельность.

Сегодня также стало стандартом предоставлять референс в машиночитаемом виде — согласно какому-либо стандарту, например, OpenAPI.

Спецификация должна содержать не только формальные описания, но и документировать неявные соглашения, такие как, например, последовательность генерации событий или неочевидные побочные эффекты методов. Референс следует рассматривать как наиболее полный из возможных видов документации: изучив референс разработчик должен будет узнать

абсолютно обо всей имеющейся функциональности. Его важнейшее прикладное значение — консультативное: разработчики будут обращаться к нему для прояснения каких-то неочевидных вопросов.

Важно: формальная спецификация *не является* документацией сама по себе; документация — это слова, которые вы напишете в поле `description` для каждого поля и метода. Без словесных описаний спецификация годится разве что для проверки, достаточно ли хорошо вы назвали сущности, чтобы разработчик мог догадаться об их смысле самостоятельно.

В последнее время часто описания номенклатуры методов выкладываются также в виде готовых коллекций запросов или фрагментов кода — в частности, для Postman или аналогичных инструментов.

2. Примеры кода

Исходя из вышесказанного, примеры кода — самый важный инструмент привлечения и поддержки новых пользователей вашего API. Исключительно важно подобрать примеры так, чтобы они облегчали новичкам работу с API; неправильно выстроенные примеры только ухудшают качество вашей документации. При составлении примеров следует руководствоваться следующими принципами:

- примеры должны покрывать актуальные сценарии использования API: чем лучше вы угадаете наиболее частые запросы разработчиков, тем более дружелюбным и простым для входа будет выглядеть ваш API в их глазах;
- примеры должны быть лаконичными и атомарными: смешивание в одном фрагменте кода множества разных трюков из различных предметных областей резко снижает его понятность и применимость;
- код примеров должен быть максимально приближен к реальному приложению; автор этой книги сталкивался с ситуацией, когда синтетический фрагмент кода, абсолютно бессмысленный в реальном мире, был бездумно растиражирован разработчиками в огромном количестве.

В идеале примеры должны быть провязаны со всеми остальными видами документации. В частности, референс должен содержать примеры, релевантные описанию сущностей.

3. Песочницы

Примеры станут намного полезнее для разработчиков, если будут представлены в виде «живого» кода, который можно модифицировать и запустить на исполнение. В случае библиотечных API это может быть просто онлайн-песочница с заранее заготовленными примерами (в качестве такой песочницы можно использовать и существующие онлайн-сервисы типа JSFiddle); в случае других API разработка песочницы может оказаться сложнее:

- если через API предоставляется доступ к данным, то и песочница должна позволять работать с настоящими данными — либо своими собственными (привязанными к профилю разработчика), либо с некоторым тестовым набором данных;
- если API представляет собой интерфейс (визуальный или программный) к некоторой не-онлайн среде (например, графические библиотеки для мобильных устройств), то песочница должна представлять собой симулятор или эмулятор такой среды, в виде онлайн сервиса или standalone-приложения.

4. Руководство (тutorиал)

Под руководством мы понимаем специально написанный человекочитаемый текст, в котором изложены основные принципы работы с API. Руководство, таким образом, занимает промежуточную позицию между справочником и примерами: подразумевает более глубокое, нежели просто копирование кусков кода из примеров, погружение в API, но требует меньших инвестиций времени и внимания, нежели чтение полного референса.

По смыслу руководство — это такая «книга», в которой вы доносите до читателя, как работать с вашим API. Правильно составленное руководство, таким образом, должно примерно следовать принципам написания книг по программированию, т.е. излагать концепции консистентно и последовательно от простых к сложным. Кроме того, в руководстве должны быть зафиксированы:

- общие сведения по предметной области; например, для картографических API руководство должно содержать вводную часть про координаты и работу с ними;

- правильные сценарии использования, т.е. «happy path» API;
- описания корректной реакции на те или иные ошибки;
- подробные учебные материалы по продвинутой функциональности API (разумеется, с подробными примерами).

Как правило, руководство представляет собой общий блок (основные термины и понятия, используемые обозначения) и набор блоков по каждому роду функциональности, предоставляемой через API.

Часто в составе руководства выделяется т.н. “quick start” (“Hello, world!”): максимально короткий пример, позволяющий новичку собрать хотя бы какое-то минимальное приложение поверх API. Целей его существования две:

- стать точкой входа по умолчанию, максимально понятным и полезным текстом для тех, кто впервые услышал о вашем API;
- вовлечь разработчиков, дать им «пощупать» сервис на живом примере.

Quick start-ы также являются отличным индикатором того, насколько хорошо вы справились с определением частотных кейсов и разработкой хелперных методов. Если ваш quick start содержит более десятка строк кода, вы точно что-то делаете не так.

5. Часто задаваемые вопросы и база знаний

После того, как вы опубликуете API и начнёте поддерживать пользователей (см. предыдущую главу), у вас также появится понимание наиболее частых вопросов пользователей. Если интегрировать ответы на эти вопросы в документацию так просто не получается, имеет смысл завести отдельный раздел с часто задаваемыми вопросами (ЧаВо, англ. FAQ). Раздел ЧаВо должен отвечать следующим критериям:

- отвечать на реальные вопросы пользователей
 - часто можно встретить такие разделы, составленные не по реальным обращениям и отражающие в основном стремление владельца API ещё разок донести какую-то важную информацию; конечно, такой FAQ в лучшем случае бесполезен, а в худшем раздражает; за идеальными примерами реализации этого антипаттерна можно обратиться на сайт любого банка или авиакомпании);

- и вопрос, и ответ должны быть сформулированы лаконично и понятно; в ответе допустимо (и даже желательно) дать ссылку на соответствующие разделы руководства и справочника, но сам по себе ответ не должен превышать пары абзацев.

Кроме того, раздел ЧаВо очень хорошо подходит для того, чтобы явно донести главные преимущества вашего API. В виде пары вопрос-ответ можно наглядно рассказать о том, как ваш API решает сложные проблемы красиво и удобно (или хотя бы решает вообще, в отличие от API конкурентов).

Если вы оказываете техническую поддержку публично, имеет смысл сохранять вопросы и ответы в виде отдельного сервиса, чтобы сформировать базу знаний, т.е. набор «живых» вопросов и ответов.

6. Оффлайн-документация

Хотя мы и живём в мире победившего онлайн, офлайн-версия документации в виде сгенерированного документа, тем не менее, бывает полезной — в первую очередь как «слепок» актуального состояния API на определённый момент времени.

Проблемы дублирования контента

Большую проблему для читабельности документации представляет версионирование API: многие тексты для разных версий похожи до неразличимости. Организовать качественный поиск по такому массиву данных очень сложно как внутренними, так и внешними средствами. Правилами хорошего тона в связи с этим являются:

- явное и заметное выделение на страницы документации версии API, к которой она относится;
- явное и заметное указание на существование более актуальной версии страницы для новых API;
- пессимизация (вплоть до запрета индексирования) документации к устаревшим версиям API.

Если вы поддерживаете обратную совместимость, то можно попытаться поддерживать единую документацию для всех версий API. В этом случае для каждой сущности нужно указывать, начиная с какой версии API появилась её поддержка. Здесь, однако, возникает проблема с тем, что получить документацию для какой-то конкретной (устаревшей) версии API (и вообще понять, какие возможности предоставляла определённая версия API) крайне затруднительно. (Но с этим может помочь офлайн-документация, о чём мы упоминали выше.)

Проблема усложняется, если вы поддерживаете документацию не только для разных версий API, но и для разных сред / платформ / языков программирования — скажем, ваша визуальная библиотека поддерживает и Android, и iOS. В этой ситуации обе версии документации полностью равноправны, и выделить одну из них в ущерб другой невозможно.

В этом случае необходимо выбрать одну из двух стратегий:

- если контент справки полностью идентичен для всех платформ, т.е. меняется только синтаксис кода — придётся подготовить возможность писать документацию обобщённым образом: статьи документации должны содержать примеры кода (и, возможно, какие-то примечания) сразу для всех поддерживаемых платформ;
- если контент, напротив, существенно различен (как в упомянутом кейсе Android/iOS), мы можем только предложить максимально разнести площадки, вплоть до заведения разных сайтов: хорошая новость состоит в том, что разработчикам практически всегда нужна только одна из версий, другая платформа их совершенно не интересует.

Качество документации

Важно отметить, что документация получается максимально удобной и полезной, если вы рассматриваете её саму как один из продуктов в линейке сервисов API — а значит, анализируете поведение пользователей (в том числе автоматизированными средствами), собираете и обрабатываете обратную связь, ставите KPI и работаете над их улучшением.

Была ли эта статья полезна для вас?

[Да / Нет](#)

Глава 61. Тестовая среда

Если через ваш API исполняются операции, которые имеют последствия для пользователей или партнёров (в частности, стоят денег), то вам необходимо иметь тестовую версию этого API. В тестовом API реальные действия либо не происходят совсем (например, заказ создаётся, но никем не исполняется), либо симулируются дешёвыми способами (например, вместо отправки SMS на номер пользователя уходит электронное письмо на почту разработчика).

Однако во многих случаях этого недостаточно — как, например, в нашем выдуманном примере с API кофе-машин. Если заказ просто создаётся, но не исполняется — партнёры не смогут протестировать, как в их приложении работает функциональность выдачи заказа или, скажем, запроса возврата денег. Для проведения полного цикла тестирования необходимо, чтобы этот виртуальный заказ можно было переводить в другие статусы — так, как это будет происходить в реальности.

Решение этой проблемы «в лоб» — это предоставление полного комплекта тестовых API и административных интерфейсов. Т.е. разработчик должен будет запустить параллельно второе приложение — то, которое вы предоставляете кофейням, чтобы они могли принять и исполнить заказ (а если предусмотрена курьерская доставка, то ещё и третья — приложение курьера) — и выполнять в этом приложении действия, которые в норме выполняет сотрудник кофейни. Очевидно, что это далеко не самый удобный вариант, по многим причинам:

- разработчику пользовательского приложения придётся дополнительно разбираться в работе приложений для кофеен и курьеров, которые, вообще говоря, никакого отношения к его задачам не имеют;
- необходимо будет придумать и реализовать какие-то алгоритмы связывания: заказ, сделанный через тестовое приложение, должен попадать на исполнение нужному виртуальному курьеру; это фактически означает, что в тестовом API вам нужно будет создавать виртуальную «песочницу» (читай — полный набор сервисов) для каждого партнёра в отдельности;
- полный happy path заказа будет занимать минуты, иногда десятки минут, и требовать выполнения множества действий в нескольких интерфейсах.

Избежать подобного рода проблем вы можете двумя основными способами.

1. API среды тестирования

Первый вариант — это предоставление мета-API к самой тестовой среде. Вместо того, чтобы запускать приложение кофейни в отдельном эмуляторе, разработчику предоставляются хелперные методы (типа `simulateOrderPreparation`) или специальный визуальный интерфейс, позволяющие управлять сценарием исполнения заказа с минимальными усилиями.

В идеале вы должны иметь хелперные методы среды тестирования на все действия, которые в реальном продакшн-окружении выполняются людьми. Имеет смысл поставлять такие мета-API сразу с готовыми скриптами или коллекциями запросов, демонстрирующих правильную последовательность вызовов разных API для симуляции стандартных сценариев.

Недостаток этого подхода заключается в том, что разработчику клиентского приложения всё ещё необходимо разбираться в том, как работает «изнанка» системы, пусть и в упрощённых терминах.

2. Симулятор предопределённых сценариев

Альтернативой API среды тестирования является симуляция сценариев работы, когда тестовая среда берёт на себя управление «подводной» частью системы. В нашем кофейном примере это будет выглядеть так: после размещения заказа система автоматически симулирует все шаги его приготовления, а потом и получение заказа конечным пользователем.

Плюсом такого подхода является наглядная демонстрация, каким образом система работает согласно задумке провайдера API, какие события в какой последовательности генерируются и через какие стадии проходит заказ. Кроме того, уменьшается возможность ошибки в скриптах тестирования, поскольку провайдером API гарантируется, что действия будут выполнены в правильном порядке и с правильными параметрами.

Основным минусом является необходимость разработать отдельный сценарий для всех возможных unhappy path, т.е. для каждой возможной ошибки, причём также необходимо и предоставить возможность разработчику указать, какой именно из сценариев он хочет воспроизвести. (Например, следующим образом: если заказ создан с комментарием определённого вида, будет эмулирована ошибка его исполнения, и разработчик сможет отладить правильную реакцию на такого рода ошибку.)

Автоматизация тестирования

Ваша конечная цель при разработке тестового API, независимо от выбранного варианта — это позволить партнёрам автоматизировать тестирование их продуктов. Разработку тестовой среды нужно вести именно с этим прицелом; например, если для оплаты заказа необходимо перевести пользователя на страницу 3-D Secure, то в API тестовой среды должен быть предусмотрена возможность программно симулировать прохождение (или непрохождение) пользователем этого вида проверки. Кроме того, в обоих вариантах возможно (и скорее даже желательно) выполнение сценариев в ускоренном масштабе времени, что позволяет производить автоматическое тестирование гораздо быстрее ручного.

Конечно, далеко не все партнёры этой возможностью смогут воспользоваться (что помимо прочего означает, что «ручной» способ протестировать пользовательские сценарии тоже должен поддерживаться наряду с программным) просто потому что далеко не все бизнесы могут позволить себе нанять автоматизатора тестирования. Тем не менее, сама возможность такие автотесты писать — огромное конкурентное преимущество вашего API в глазах технически продвинутых партнёров.

Глава 62. Управление ожиданиями

Наконец, последний аспект, который хотелось бы осветить в рамках данного раздела — это управление ожиданиями партнёров в отношении развития вашего API. С точки зрения коммуникации потребительских качеств API мало отличается от любого другого B2B программного обеспечения: и там, и там вам нужно как-то сформировать у разработчиков и бизнеса понимание о допустимом SLA, объёме функциональности, отзывчивости интерфейсов и прочих пользовательских характеристиках. Однако у API как продукта есть и специфические особенности.

Версионирование и жизненный цикл приложений

Конечно, в идеальном случае однажды выпущенный API должен жить вечно; но, как разумные люди, мы понимаем, что в реальной жизни это невозможно. Даже если мы продолжаем поддерживать старые версии, они все равно морально устаревают: партнёры должны потратить ресурсы на переписывание кода под новые версии API, если хотят получить доступ к новой функциональности.

Автор этой книги формулирует для себя золотое правило выпуска новых версий API так: период времени, после которого партнеру потребуется переписать свой код, должен совпадать с жизненным циклом приложений в вашей предметной области (см. главу [«Постановка проблемы обратной совместимости»](#)). Помимо переключения мажорных версий API, рано или поздно встанет вопрос и о доступе к минорным версиям. Как мы упоминали в главе [«О ватерлинии айсберга»](#), даже исправление ошибок в коде может привести к неработоспособности какой-то интеграции. Соответственно, может потребоваться и сохранение возможности зафиксировать *минорную* версию API до момента обновления затронутого кода партнёром.

В этом аспекте интеграция с крупными компаниями, имеющими собственный отдел разработки, существенно отличается от взаимодействия с одиночными разработчиками-любителями: первые, с одной стороны, с гораздо большей вероятностью найдут в вашем API недокументированные возможности и неисправленные ошибки; с другой стороны, в силу большей бюрократичности

внутренних процессов, исправление проблем может затянуться на месяцы, а то и годы. Общая рекомендация здесь — поддерживать возможность подключения старых минорных версий API достаточно долго для того, чтобы самый забюрократизированный партнёр успел переключиться на новую версию.

Поддержка платформ

Ещё один аспект, критически важный во взаимодействии с крупными интеграторами — это поддержка зоопарка платформ (браузеров, языков программирования, протоколов, операционных систем) и их версий. Как правило, большие компании имеют свои собственные стандарты, какие платформы они поддерживают, и эти стандарты могут входить в заметное противоречие со здравым смыслом. (Например, от TLS 1.2 в настоящий момент уже желательно отказываться, однако многие интеграторы продолжают работать именно через этот протокол, и это ещё в лучшем случае.)

Формально говоря, отказ от поддержки какой-либо версии платформы — это слом обратной совместимости и может привести к неработоспособности какой-то интеграции у части пользователей. Исключительно желательно иметь чётко сформулированные политики, какие платформы по какому принципу поддерживаются. В случае массовых публичных API ситуация, обычно, достаточно простая (провайдер API обещает поддерживать платформы с долей более N%, или, ещё проще, последние M версий платформы); в случае коммерческих API это всегда некоторый торг — сколько отсутствие поддержки той или иной версии платформы будет стоить компании. Крайне желательно результат этого торга фиксировать в договорах — что конкретно вы обещаете поддерживать и в течение какого времени.

Движение вперёд

Наконец, помимо частных проблем поддержки, ваших пользователей почти наверняка волнуют и более общие вопросы: насколько вам можно доверять; насколько можно рассчитывать, что ваш API продолжит расти, развиваться, будет вбирать в себя современные тренды, и не окажется ли в один прекрасный день интеграция с вашим API на свалке истории. Будем честны: учитывая неопределенность продуктового видения API этот вопрос и нас самих весьма интересует. Даже древнеримский акведук, сохраняющий обратную совместимость вот уже две тысячи лет, уже давно является весьма архаичным и ненадёжным способом решать проблемы пользователя.

Работать с этими ожиданиями клиентов можно через публичные роадмапы. Не секрет, что многие компании избегают открыто сообщать о своих планах (и не без причины). Тем не менее, в случае API мы всячески рекомендуем роадмапы публиковать, пусть и условные и без конкретных дат, *особенно* если речь идёт о закрытии или прекращении поддержки какой-то функциональности. Наличие таких обещаний (при условии, что разработчик API их выполняет, конечно) — очень важное конкурентное преимущество для всех видов ваших пользователей.

На этом нам хотелось бы закончить настоящую книгу. Мы надеемся, что изложенные здесь принципы и концепции помогут вам создавать API, максимально удобные и для разработчиков, и для бизнеса, и для конечных пользователей, и развивать их без потери обратной совместимости следующие две тысячи лет (а может и больше).