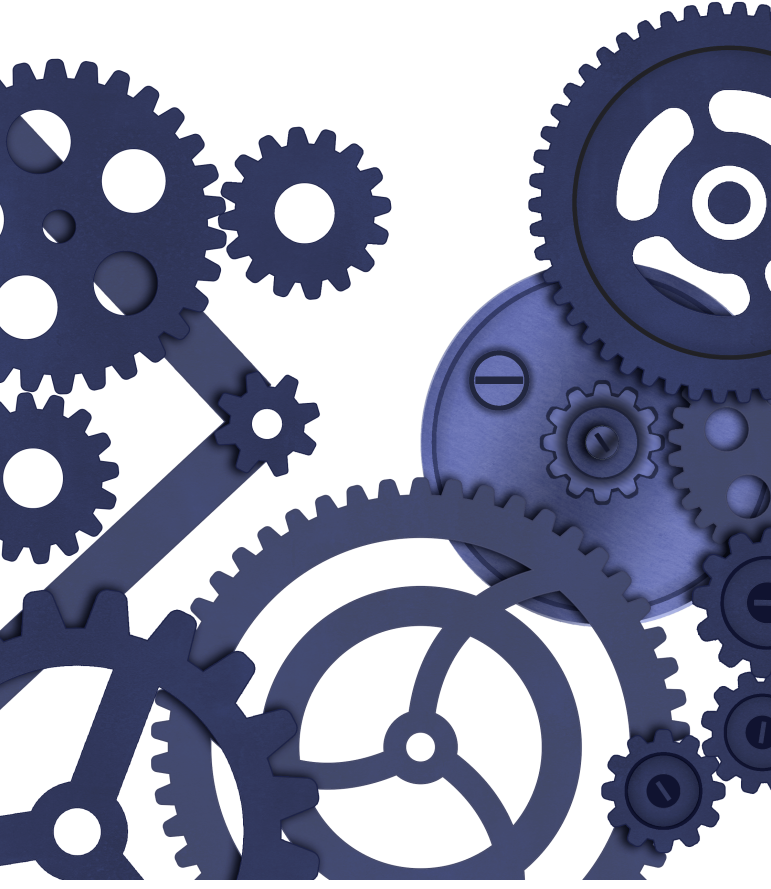
Сергей Константинов

API



Сергей Константинов. API.
yatwirl@gmail.com · linkedin.com/in/twirl ·
patreon.com/yatwirl

«API-first» подход — одна из самых горячих 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. В первом разделе мы поговорим о проектировании API: как грамотно выстроить архитектуру, от крупноблочного планирования до конечных интерфейсов. Второй раздел посвящён развитию существующих API с сохранением обратной совместимости. Наконец, в третьем разделе мы поговорим об API как о продукте.

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: vk · facebook · twitter · linkedin · reddit

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Глава 1. О структуре этой книги

Книга, которую вы держите в руках, состоит из введения и двух больших разделов (ещё один находится в разработке).

В первом разделе мы поговорим о проектировании API на стадии разработки концепции — как грамотно выстроить архитектуру, от крупноблочного планирования до конечных интерфейсов.

Второй раздел посвящён жизненному циклу API — как интерфейсы эволюционируют со временем и как развивать продукт так, чтобы отвечать потребностям пользователей.

Наконец, третий раздел будет касаться больше неразработческих сторон жизни API — поддержки, маркетинга, работы с комьюнити.

Первые два будут интересны скорее разработчикам, третий — и разработчикам, и менеджерам. При этом мы настаиваем, что как раз третий раздел — самый важный. Ввиду того, что API — продукт для программистов, перекладывать ответственность за его развитие и поддержку на неразработчиков неправильно: никто кроме вас самих не понимает так хорошо продуктовые свойства вашего API.

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 меня просят привести пример хорошего API, я обычно показываю 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 этой книги: 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 излагать рекомендации, нам следует определиться с тем, что мы считаем «хорошим» API, и какую пользу мы получаем от того, что наш API «хороший».

Начнём со второго вопроса. Очевидно, «хорошесть» API определяется в первую очередь тем, насколько он помогает разработчикам решать стоящие перед ними задачи. (Можно резонно возразить, что решение задач, стоящих перед разработчиками, не обязательно влечёт за собой выполнение целей, которые мы ставим перед собой, предлагая разработчикам API. Однако манипуляция общественным мнением не входит в область интересов автора этой книги: здесь и далее предполагается, что API существует в первую очередь для того, чтобы разработчики решали с его помощью свои задачи, а не ради каких-то завуалированных целей).

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; в идеале 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.

Однако статическое удобство и понятность API — это 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. Обратная совместимость

Обратная совместимость — это некоторая *временна́я* 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 5. О версионировании

Здесь и далее мы будем придерживаться принципов версионирования semver.

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».

Глава 6. Условные обозначения и терминология

В мире разработки программного обеспечения существует множество различных парадигм разработки, адепты которых зачастую настроены весьма воинственно по отношению к адептам 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 JSON-over-HTTP-эндпойтов. 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 однострочный 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 C-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.

РАЗДЕЛ I. ПРОЕКТИРОВАНИЕ API

Глава 7. 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 8. 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 9. 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 позволят нам в дальнейшем добавлять новую функциональность, не меняя интерфейс.

3. Поддержание интероперабельности. Правильно выделенные низкоуровневые абстракции позволят 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.

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 варьировать 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 поддержать 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.

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 кофемашинах). 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: высокоуровневые сущности и номенклатура их методов должны напрямую отражать сценарии использования API; низкоуровневый — 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 сырым байтовый 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,
  "operation_state": {
    "status": "executing",
    "operations": [
      // описание операций, запущенных на
      // физической кофемашине
    ]
  }
  …
}
```

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 сырыми 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"
    },
    …
  ]
}
```

А `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 обратятся 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 дожидаясь 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 10. 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. `/v1/offers/search` — более логичное имя для такого эндпойнта.

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 *хелперами*. 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 может 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 совершал 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*. Всё остальное вторично; если бы ошибка была программно читаема,

мы могли бы вовсе не снабжать её никаким сообщением для пользователя.

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 пп. 12-13 следующей главы.

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».

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": [{
    "coffee_machine_id",
    // Тип кофемашины
    "coffee_machine_type":
      "drip_coffee_maker",
    // Марка кофемашины
    "coffee_machine_brand",
    // Название заведения
    "place_name": "Кафе «Ромашка»",
    // Координаты
    "place_location_latitude",
    "place_location_longitude",
    // Флаг «открыто сейчас»
    "place_open_now",
    // Часы работы
    "working_hours",
    // Сколько идти: время и расстояние
    "walking_distance",
    "walking_time",
    // Как найти заведение и кофемашину
    "place_location_tip",
    "offers": [{
      "recipe": "lungo",
      "recipe_name":
        "Наш фирменный лунго®™",
      "recipe_description",
      "volume": "800ml",
      "offer_id",
      "offer_valid_until",
      "localized_price":
        "Большая чашка↵
          всего за 19 баксов",
      "price": "19.00",
```

```
        "currency_code": "USD",
        "estimated_waiting_time": "20s"
      }, …]
    }, …]
  }
```

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;
- особенности рецепта в конкретном заведении;
- 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" },
      // 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"
```

```
      }, …]
    }, …]
  }
```

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 II.

Глава 11. 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.

Важное уточнение под номером ноль:

0. 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.

Например, требование консистентности номенклатуры существует затем, чтобы разработчик тратил меньше времени на чтение документации; если вам *необходимо*, чтобы разработчик обязательно прочитал документацию по какому-то методу, вполне разумно сделать его сигнатуру нарочито неконсистентно.

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.

Обеспечение читабельности и консистентности

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.

1. 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, не может быть модифицирующих операций за `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.

2. 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 спокойствия», который будет описан в разделе II.

3. 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()`.

4. Не экономьте буквы

В XXI веке давно уже нет нужды называть переменные покороче.

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

Хорошо:

```
order
  .get_estimated_delivery_time()
```

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.

5. 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 примитивных 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-не функционирует).

Хорошо:

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

6. 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.

7. 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 семантически двойных отрицаний, даже если вы придумали «негативное» слово без явной приставки «не».

Стоит также отметить, что в использовании законов де Моргана 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:

```
{
  "beans_absence": false,
  "cup_absence": false
}
```

— то разработчику потребуется вычислить флаг `!beans_absence && !cup_absence`, что эквивалентно `!(beans_absence || cup_absence)`, 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».

8. 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:

```
const orderParams = {
  contactless_delivery: false
};
const 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:

```
if (
  Type(
    orderParams.contactless_delivery
  ) == 'Boolean' &&
  orderParams
    .contactless_delivery == 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 самое произойдёт, если для индикации отсутствия значения поля использовать специальное значение типа `null` или `-1`.

NB. Это замечание не распространяется на те случаи, когда платформа и протокол однозначно и без всяких дополнительных абстракций поддерживают такие специальные значения для сброса значения поля в значение по умолчанию. Однако полная и консистентная поддержка частичных операций со сбросом значений полей практически нигде не

имплементирована. Пожалуй, единственный пример такого API из имеющих широкое распространение сегодня — SQL: в языке есть и концепция `NULL`, и значения полей по умолчанию, и поддержка операций вида `UPDATE … SET field = DEFAULT` (в большинстве диалектов). Хотя работа с таким протоколом всё ещё затруднена (например, во многих диалектах нет простого способа получить обратно значение по умолчанию, которое выставил `UPDATE … DEFAULT`), логика работы с умолчаниями в SQL имплементирована достаточно хорошо, чтобы использовать её как есть.

Если протоколом явная работа со значениями по умолчанию не предусмотрена, универсальное правило — все новые необязательные булевы флаги должны иметь значение по умолчанию false.

Хорошо

```
const orderParams = {
  force_contact_delivery: true
};
const 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.

9. 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.)

10. Ошибки должны быть информативными

При написании кода разработчик неизбежно столкнётся с ошибками, в том числе самого примитивного толка: неправильный тип параметра или неверное значение. Чем понятнее ошибки, возвращаемые вашим API, тем меньше времени разработчик потратит на борьбу с ними, 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": "constraint_violation",
        "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.

11. Соблюдайте правильный порядок ошибок

Во-первых, всегда показывайте неразрешимые ошибки прежде разрешимых:

```
POST /v1/orders
{
  "recipe": "lngo",
  "offer"
}
→ 409 Conflict
{
  "reason": "offer_expired"
}
// Повторный запрос
// с новым `offer`
POST /v1/orders
{
  "recipe": "lngo",
  "offer"
}
→ 400 Bad Request
{
  "reason": "recipe_unknown"
}
```

— какой был смысл получать новый `offer`, если заказ всё равно не может быть создан?

Во-вторых, соблюдайте такой порядок разрешимых ошибок, который приводит к наименьшему раздражению пользователя и разработчика. В частности, следует начинать с более значимых ошибок, решение которых требует более глобальных изменений.

Плохо:

```
POST /v1/orders
{
  "items": [{
    "item_id": "123",
    "price": "0.10"
  }]
}
→
409 Conflict
{
  "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
{
  "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.

В-третьих, постройте схему: разрешение какой ошибки может привести к появлению другой, иначе вы можете показать одну и ту же ошибку несколько раз, а то и вовсе зациклить разрешение ошибок.

```
// 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.

Правила разработки машиночитаемых интерфейсов

В погоне за понятностью API для людей мы часто забываем, что работать с API всё-таки будут не сами разработчики, а написанный ими код. Многие концепции, которые хорошо работают для визуальных интерфейсов, плохо подходят для интерфейсов программных: в частности, разработчик не может в коде принимать решения, ориентируясь на текстовые сообщения, и не может «выйти и зайти снова» в случае нештатной ситуации.

12. Состояние системы должно быть понятно клиенту

Часто можно встретить интерфейсы, в которых клиент не обладает полнотой знаний о том, что происходит в системе от его имени — например, какие операции сейчас выполняются и каков их статус.

Плохо:

```
// Создаёт заказ и возвращает его id
POST /v1/orders
{ … }
→
{ "order_id" }
```

```
// Возвращает заказ по его id
GET /v1/orders/{id}
// Заказ ещё не подтверждён
// и ожидает проверки
→ 404 Not Found
```

— хотя операция будто бы выполнена успешно, клиенту необходимо самостоятельно запомнить идентификатор заказа и периодически проверять состояние `GET /v1/orders/{id}`. Этот паттерн плох сам по себе, но ещё и усугубляется двумя обстоятельствами:

- клиент может потерять идентификатор, если произошёл системный сбой в момент между отправкой запроса и получением ответа или было повреждено (очищено) системное хранилище данных приложения;
- потребитель не может воспользоваться другим устройством; фактически, знание о сделанном заказе привязано к конкретному юзер-агенту.

В обоих случаях потребитель может решить, что заказ по какой-то причине не создался — и сделать повторный заказ со всеми вытекающими отсюда проблемами.

Хорошо:

```
// Создаёт заказ и возвращает его
POST /v1/orders
{ <параметры заказа> }
→
{
  "order_id",
  // Заказ создаётся в явном статусе
  // «идёт проверка»
  "status": "checking",
  …
}
```

```
// Возвращает заказ по его id
GET /v1/orders/{id}
→
{ "order_id", "status" … }
```

```
// Возвращает все заказы пользователя
// во всех статусах
GET /v1/users/{id}/orders
```

Это правило также распространяется и на ошибки, в первую очередь, клиентские. Если ошибку можно исправить, информация об этом должна быть машиночитаема.

Плохо: `{ "error": "email malformed" }` — единственное, что может с этой ошибкой сделать разработчик — показать её пользователю

Хорошо:

```
{
  // Машиночитаемый статус
  "status": "validation_failed",
  // Массив описания проблем;
  // если пользовательский ввод
  // некорректен в нескольких
  // аспектах, пользователь сможет
  // исправить их все
  "failed_checks": [
     {
       "field: "email",
       "error_type": "malformed",
       // Локализованное
       // человекочитаемое
       // сообщение
       "message": "email malformed"
     }
  ]
}
```

13. 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; каким 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.

```
// Возвращает предложение: за какую сумму
// наш сервис готов приготовить лунго
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"
    }
  }
}
```

14. Пагинация, фильтрация и курсоры

Любой эндпойнт, возвращающий массивы данных, должен содержать пагинацию. Никаких исключений в этом правиле быть не может.

Любой эндпойнт, возвращающий изменяемые данные постранично, должен обеспечивать возможность эти данные перебрать.

Плохо:

```
// Возвращает указанный limit записей,
// отсортированных по дате создания
// начиная с записи с номером offset
GET /v1/records?limit=10&offset=100
```

На первый взгляд это самый что ни на есть стандартный способ организации пагинации в API. Однако зададим себе три вопроса.

1. Каким образом клиент узнает о появлении новых записей в начале списка? Легко заметить, что клиент может только попытаться повторить первый запрос и сверить идентификаторы с запомненным началом списка. Но что делать, если добавленное количество записей превышает `limit`? Представим себе ситуацию:
    - клиент обрабатывает записи в порядке поступления;
    - произошла какая-то проблема, и накопилось большое количество необработанных записей;

    - клиент запрашивает новые записи (`offset=0`), однако не находит на первой странице известных идентификаторов — новых записей накопилось больше, чем `limit`;
    - клиент вынужден продолжить перебирать записи (увеличивая `offset`) до тех пор, пока не доберётся до последней известной ему; всё это время клиент простаивает;
    - таким образом может сложиться ситуация, когда клиент вообще никогда не обработает всю очередь, т.к. будет занят беспорядочным линейным перебором.
2. Что произойдёт, если при переборе списка одна из записей в уже перебранной части будет удалена?

   Произойдёт следующее: клиент пропустит одну запись и никогда не сможет об этом узнать.
3. Какие параметры кэширования мы можем выставить на этот эндпойнт?

   Никакие: повторяя запрос с теми же `limit-offset`, мы каждый раз получаем новый набор записей.

Хорошо: в таких однонаправленных списках пагинация должна быть организована по тому ключу, порядок сортировки по которому фиксирован. Например, вот так:

```
// Возвращает указанный limit записей,
// отсортированных по дате создания,
// начиная с первой записи,
// созданной позднее,
// чем запись с указанным id
GET /v1/records↵
  ?older_than={record_id}&limit=10
// Возвращает указанный limit записей,
// отсортированных по дате создания,
// начиная с первой записи,
// созданной раньше,
// чем запись с указанным id
GET /v1/records↵
  ?newer_than={record_id}&limit=10
```

При такой организации клиенту не надо заботиться об удалении или добавлении записей в уже перебранной части списка: он продолжает перебор по идентификатору известной записи — первой известной, если надо получить новые записи; последней известной, если надо продолжить перебор. Если операции удаления записей нет, то такие запросы можно свободно кэшировать — по одному и тому же URL будет всегда возвращаться один и тот же набор записей.

Другой вариант организации таких списков — возврат курсора `cursor`, который используется вместо `record_id`, что делает интерфейсы более универсальными.

```
// Первый запрос данных
POST /v1/records/list
{
  // Какие-то дополнительные
  // параметры фильтрации
  "filter": {
    "category": "some_category",
    "created_date": {
      "older_than": "2020-12-07"
    }
  }
}
→
{ "cursor" }
```

```
// Последующие запросы
GET /v1/records?cursor=<курсор>
{ "records", "cursor" }
```

Достоинством схемы с курсором является возможность зашифровать в самом курсоре данные исходного запроса (т.е. `filter` в нашем примере), и таким образом не дублировать его в последующих запросах. Это может быть особенно актуально, если инициализирующий запрос готовит полный массив данных, например, перенося его из «холодного» хранилища в горячее.

Вообще схему с курсором можно реализовать множеством способов (например, не разделять первый и последующие запросы данных), главное — выбрать какой-то один.

NB: в некоторых источниках такой подход, 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.

Плохо:

```
// Возвращает указанный limit записей,
// отсортированных по полю sort_by
// в порядке sort_order,
// начиная с записи с номером offset
GET /records?sort_by=date_modified↵
  &sort_order=desc&limit=10&offset=100
```

Сортировка по дате модификации обычно означает, что данные могут меняться. Иными словами, между запросом первой порции данных и запросом второй порции данных какая-то запись может измениться; она просто пропадёт из перечисления, т.к. автоматически попадает на первую страницу. Клиент никогда не получит те записи, которые менялись во время перебора, и у него даже нет способа узнать о самом факте такого пропуска. Помимо этого отметим, что

такой API нерасширяем — невозможно добавить сортировку по двум и более полям.

Хорошо: в представленной постановке задача, собственно говоря, не решается. Список записей по дате изменения всегда будет непредсказуемо изменяться, поэтому необходимо изменить сам подход к формированию данных, одним из двух способов.

Вариант 1: фиксировать порядок в момент обработки запроса; т.е. сервер формирует полный список и сохраняет его в неизменяемом виде:

```
// Создаёт представление по указанным параметрам
POST /v1/record-views
{
  sort_by: [{
      "field": "date_modified",
      "order": "desc"
  }]
}
→
{ "id", "cursor" }
```

```
// Позволяет получить часть представления
GET /v1/record-views/{id}↵
  ?cursor={cursor}
```

Поскольку созданное представление уже неизменяемо, доступ к нему можно организовать как угодно: через курсор, limit/offset, заголовок Range и т.д. Однако надо иметь в виду, что при переборе таких списков порядок может быть нарушен: записи, изменённые уже после генерации представления, будут находиться не на своих местах (либо быть неактуальны, если запись копируется целиком).

Вариант 2: гарантировать строгий неизменяемый порядок записей, например, путём введения понятия события изменения записи:

```
// Курсор опционален
GET /v1/records/modified/list↵
  ?[cursor={cursor}]
→
{
  "modified": [
    { "date", "record_id" }
  ],
  "cursor"
}
```

Недостатком этой схемы является необходимость заводить отдельное индексированное хранилище событий, а также появление множества событий для одной записи, если данные меняются часто.

Техническое качество API

Хороший API должен не просто решать проблемы разработчиков и пользователей, но и делать это максимально качественно, т.е. не содержать в себе логических и технических ошибок (и не провоцировать на них разработчика), экономить вычислительные ресурсы и вообще имплементировать лучшие практики в своей предметной области.

15. 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. предпочесть формат `00:20` формату `0.333333…`), либо предоставить SDK работы с такими данными, либо (в крайнем случае) описать в документации принципы округления.

16. 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.

Также стоит упомянуть, что добавление токенов идемпотентности к эндпойнтам, 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;
- клиенты склонны 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.

17. Избегайте неатомарных операций

С применением массива изменений часто возникает вопрос: что делать, если часть изменений удалось применить, а часть — нет? Здесь правило очень простое: если вы можете обеспечить атомарность, т.е. выполнить либо все изменения сразу, либо ни одно из них — сделайте это.

Плохо:

```
// Возвращает список рецептов
api.getRecipes();
→
{
  "recipes": [{
    "id": "lungo",
    "volume": "200ml"
  }, {
    "id": "latte",
    "volume": "300ml"
  }]
}

// Изменяет параметры
api.updateRecipes({
  "changes": [{
    "id": "lungo",
    "volume": "300ml"
  }, {
    "id": "latte",
    "volume": "-1ml"
  }]
});
→
Bad Request

// Перечитываем список
api.getRecipes();
→
{
  "recipes": [{
    "id": "lungo",
    // Это значение изменилось
    "volume": "300ml"
```

```
    }, {
      "id": "latte",
      // А это нет
      "volume": "300ml"
    }]
  }
```

— клиент никак не может узнать, что операция, которую он посчитал ошибочной, на самом деле частично применена. Даже если индицировать это в ответе, у клиента нет способа понять — значение объёма лунго изменилось вследствие запроса, или это конкурирующее изменение, выполненное другим клиентом.

Если способа обеспечить атомарность выполнения операции нет, следует очень хорошо подумать над её обработкой. Следует предоставить способ получения статуса каждого изменения отдельно.

Лучше:

```
api.updateRecipes({
  "changes": [{
    "recipe_id": "lungo",
    "volume": "300ml"
  }, {
    "recipe_id": "latte",
    "volume": "-1ml"
  }]
});
// Можно воспользоваться статусом
// «частичного успеха»,
// если он предусмотрен протоколом
→
{
  "changes": [{
    "change_id",
    "occurred_at",
    "recipe_id": "lungo",
    "status": "success"
  }, {
    "change_id",
    "occurred_at",
    "recipe_id": "latte",
    "status": "fail",
    "error"
  }]
}
```

Здесь:

- `change_id` — уникальный идентификатор каждого атомарного изменения;

- `occurred_at` — время проведения каждого изменения;
- `error` — информация по ошибке для каждого изменения, если она возникла.

Не лишним будет также:

- ввести в запросе `sequence_id`, чтобы гарантировать порядок исполнения операций и соотнесение порядка статусов изменений в ответе с запросом;
- предоставить отдельный эндпойнт `/changes-history`, чтобы клиент мог получить информацию о выполненных изменениях, если во время обработки запроса произошла сетевая ошибка или приложение перезагрузилось.

Неатомарные изменения нежелательны ещё и потому, что вносят неопределённость в понятие идемпотентности, даже если каждое вложенное изменение идемпотентно. Рассмотрим такой пример:

```
api.updateRecipes({
  "idempotency_token",
  "changes": [{
    "recipe_id": "lungo",
    "volume": "300ml"
  }, {
    "recipe_id": "latte",
    "volume": "400ml"
  }]
});
→
{
  "changes": [{
    …
    "status": "success"
  }, {
    …
    "status": "fail",
    "error": {
      "reason":
        "too_many_requests"
    }
  }]
}
```

Допустим, клиент не смог получить ответ и повторил запрос с тем же токеном идемпотентности.

```
api.updateRecipes({
  "idempotency_token",
  "changes": [{
    "recipe_id": "lungo",
    "volume": "300ml"
  }, {
    "recipe_id": "latte",
    "volume": "400ml"
  }]
});
→
{
  "changes": [{
    …
    "status": "success"
  }, {
    …
    "status": "success",
  }]
}
```

По сути, для клиента всё произошло ожидаемым образом: изменения были внесены, и последний полученный ответ всегда корректен. Однако по сути состояние ресурса после первого запроса отличалось от состояния ресурса после второго запроса, что противоречит самому определению идемпотентности.

Более корректно было бы при получении повторного запроса с тем же токеном ничего не делать и возвращать ту же разбивку ошибок, что была дана на первый запрос — но для этого придётся её каким-то образом хранить в истории изменений.

На всякий случай уточним, что вложенные операции должны быть сами по себе идемпотентны. Если же это не так, то следует сгенерировать внутренние ключи идемпотентности на каждую вложенную операцию в отдельности.

18. 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 параметры 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 Man-in-the-Middle, как соединение по протоколу TLS с взаимной проверкой сигнатур сертификатов.

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 времени, а веб-сервер — атаке с разделением запросов.

19. 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.

20. Считайте трафик

В современном мире такой ресурс, как объём пропущенного 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:

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

Если первые две проблемы решаются чисто техническими средствами (см. соответствующие разделы), то третья проблема скорее логическая: каким образом разумно организовать канал обновления состояния клиента так, чтобы найти баланс между отзывчивостью системы и затраченными на эту

отзывчивость ресурсами. Здесь мы можем дать несколько рекомендаций:

- не злоупотребляйте асинхронными интерфейсами;
    - с одной стороны, они позволяют нивелировать многие технические проблемы с производительностью API, что, в свою очередь, позволяет поддерживать обратную совместимость: если метод изначально асинхронный, то можно без проблем увеличивать время обработки и менять модель консистентности данных;
    - с другой стороны, количество генерируемых клиентами запросов становится трудно предсказуемым, поскольку для получения результата клиенту необходимо сделать заранее неизвестное число обращений;
- объявляйте явную политику перезапросов (например, посредством заголовка `Retry-After`);
    - да, какие-то клиенты будут её игнорировать, т.к. разработчики поленятся её имплементировать, но какие-то не будут (особенно если вы сами предоставляете SDK);

- если вы ожидаете значительного количества асинхронных операций в API, изначально дайте разработчику выбор между моделями poll (клиент самостоятельно производит новые запросы к API чтобы проверить, не изменился ли статус асинхронной операций) и push (сервер уведомляет клиентов об изменениях статусов посредством отправки специального запроса, например, через webhook-и или server push-механизмы);

- 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.

21. Избегайте неявных частичных обновлений

Один из самых частых антипаттернов в разработке API — попытка сэкономить на подробном описании изменения состояния.

Плохо:

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

```
// Частично перезаписывает заказ
// обновляет объём второго напитка
PATCH /v1/orders/{id}
{
  "items": [null, {
    "volume": "800ml"
  }]
}
→
{ /* изменения приняты */ }
```

Эта сигнатура плоха сама по себе, поскольку является нечитабельной. 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 ещё одно «зануляемое» значение специально для удаления? Аналогично, если значения неуказанных полей остаются без изменений — как сбросить их в значения по умолчанию?

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.

Все эти соображения, однако, на поверку оказываются мнимыми:

- причины увеличенного расхода трафика мы разбирали выше, 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 полениться 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 нашим 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 входящем трафике, возвращая пустой ответ сервера для модифицирующих операций; таким образом, два клиента, редактирующих одну и ту же сущность, не видят изменения друг друга.

Лучше: разделить эндпойнт. Этот подход также хорошо согласуется с принципом декомпозиции, который мы рассматривали в предыдущем разделе.

```
// 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`?).

Также в ответах операций `PUT` можно возвращать объект заказа целиком, а не перезаписываемый суб-ресурс (однако следует использовать какую-то конвенцию именования).

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 вся концепция не только перестаёт работать, но и выглядит как плохой дизайн. 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).

Ещё лучше: разработать формат описания атомарных изменений.

```
POST /v1/order/changes
X-Idempotency-Token: <токен идемпотентности>
{
  "changes": [{
    "type": "set",
    "field": "delivery_address",
    "value": <новое значение>
  }, {
    "type": "unset_item_field",
    "item_id",
    "field": "volume"
  }],
  …
}
```

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

Продуктовое качество API

Помимо технологических ограничений, любой реальный API скоро столкнётся и с несовершенством окружающей действительности. Конечно, мы все хотели бы жить в мире розовых единорогов, свободном от накопления legacy, злоумышленников, национальных конфликтов и происков конкурентов. Но, к сожалению или к счастью, живём мы в реальном мире, в котором хороший API должен учитывать всё вышеперечисленное.

22. Используйте глобально уникальные идентификаторы

Хорошим тоном при разработке API будет использование для идентификаторов сущностей глобально уникальных строк, либо семантичных (например, "lungo" для видов напитков), либо случайных (например UUID-4). 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 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.

NB: в этой книге часто используются короткие идентификаторы типа "123" в примерах — это для удобства чтения на маленьких экранах, повторять эту практику в реальном API не надо.

23. 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.

24. 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.

25. 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 примере из п. 12 сообщение `localized_message` адресовано пользователю — его должно показать приложение, если в коде обработка 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.

Глава 12. 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
{
  "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
{
  "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. ОБРАТНАЯ СОВМЕСТИМОСТЬ

Глава 13. 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.

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.

   Что касается мажорных версий, то *теоретический* ответ мы дали выше: в идеальной ситуации жизненный цикл мажорной версии должен быть чуть длиннее жизненного цикла платформы. Для стабильных ниш типа десктопных операционных систем это порядка 5-10 лет, для новых и динамически развивающихся — меньше, но всё равно измеряется в годах. *Практически* следует

смотреть на долю потребителей, реально продолжающих пользоваться версией.

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: серверный API находится полностью под вашим контролем, и вы можете оперативно исправить любые проблемы с логикой;
   - 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.

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 сосуществуют. 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 14. 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 асинхронной репликации базы данных и разрешили чтение из реплики. Это приведёт к тому, что после создания заказа следующее обращение к его статусу по id может вернуть `404`, если оно пришлось на асинхронную реплику, до которой ещё не дошли последние изменения из мастера. Фактически, вы сменили политику консистентности со strong на eventual.

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.httpStatusCode != 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 нестрого 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-а.

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 совершает 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 неопределённое 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 исполнять 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 граф 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 эндпойнты, 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 15. 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, каким 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 вот такие методы.

```
// 1. Зарегистрировать новый тип API
PUT /v1/api-types/{api_type}
{
  "order_execution_endpoint": {
    // Описание функции обратного вызова
  }
}
```

```
// 2. Предоставить список кофемашин с разбивкой
// по типу API
PUT /v1/partners/{partnerId}/coffee-machines
{
  "coffee_machines": [{
    "id",
    "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 /cofee-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 14.

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 «странности» 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 16. 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_value"
    // 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 11: 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 стандартизированных 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 есть функция форматирования строк для отображения объёма напитка:

```
l10n.volume.format(
  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, 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_preparation": {
    "action": "divide",
    "divisor": 30
  },
  "template": "{volume} ун."
}
```

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:

```
POST /v1/layouts
{
  "properties"
}
→
{ "id", "properties" }
```

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",
    "l10n"
  },
  // 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 11.

```
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.)

В предыдущей главе мы продемонстрировали, как разрыв сильной связности приводит к декомпозиции сущностей и схлопыванию публичных интерфейсов до минимума. Внимательный читатель может подметить, что этот приём уже был продемонстрирован в нашем учебном API гораздо раньше в главе 9 на примере сущностей «программа» и «запуск программы». В самом деле, мы могли бы обойтись без программ и без эндпойнта `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.

Вернёмся теперь к вопросу, который мы вскользь затронули в главе 15 — каким образом нам параметризовать приготовление заказа, если оно исполняется через сторонний API. Иными словами, что такое этот самый `program_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:

```
// Эндпойнт добавления списка
// кофемашин партнёра
PUT /v1/api-types/{api_type}
{
  "order_execution_endpoint":
    "program_run_endpoint": {
      /* Какое-то описание
          удалённого вызова эндпойнта */
      "type": "rpc",
      "endpoint": <URL>,
      "format"
    },
    "program_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_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 — ну, например, попросить посыпать кофе корицей или 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 уметь понимать в

реальном времени, можно ли этот конкретный кофе посыпать корицей). 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 обвязка для этого типа API бесполезны — автомат не умеет посыпать кофе корицей, а требование бесконтактной выдачи попросту ничего не значит. 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 трёх 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 посыпку корицей 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.

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 легаси-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, с корицей, выдать такому-то пользователю;
- нижележащий API исполнения программ 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».

В нашем конкретном примере нам нужно имплементировать следующие механизмы:

- запуск программы создаёт контекст её исполнения, содержащий все существенные параметры;
- существует способ обмена информацией об изменении данных: исполнитель может читать контекст, узнавать о всех его изменениях и сообщать обратно о изменениях своего состояния.

Организовать и то, и другое можно разными способами, однако по сути мы имеем два описания состояния (верхне- и низкоуровневое) и поток событий между ними. В случае SDK эту идею можно было бы выразить так:

```
/* Имплементация партнёром интерфейса
   запуска программы на его кофемашинах */
registerProgramRunHandler(
  apiType,
  (context) => {
    // Инициализируем запуск исполнения
    // программы на стороне партнёра
    let execution =
      initExecution(context, …);
    // Подписываемся на события
    // изменения контекста
    context.on(
      'takeout_requested',
      () => {
        // Если запрошена выдача напитка,
        // инициализируем выдачу
        execution.prepareTakeout(() => {
            // как только напиток
            // готов к выдаче,
            // сигнализируем об этом
            execution.context
              .emit('takeout_ready');
        });
      }
    );

    return execution.context;
  }
);
```

NB: в случае HTTP API соответствующий пример будет выглядеть более громоздко, поскольку потребует создания отдельных эндпойнтов чтения очередей событий типа `GET /program-run/events` и `GET /partner/{id}/execution/events`, это упражнение мы оставляем читателю. Следует также отметить, что в реальных системах потоки событий часто направляют через внешнюю шину типа Apache Kafka или Amazon SNS/SQS.

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 сущности `context` и событий, которые она 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 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; и эта имплементация будет состоять из какого-то (хорошо если явного!) способа ответить, что данная функциональность не поддерживается (или, наоборот, поддерживается всегда и безусловно).

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.

Важно также отметить, что, хотя количество сущностей (полей, событий) эффективно удваивается по сравнению с сильно связанным API, это удвоение является качественным, а не количественным. Контекст `program` содержит описание задания в своих терминах (вид напитка, объём, посыпка корицей); контекст `execution` должен эти термины переформулировать для своей предметной области (чтобы быть, в свою очередь, таким же информационным контекстом для ещё более низкоуровневого API). Что важно, `execution`-контекст имеет право эти термины конкретизировать, поскольку его нижележащие объекты будут уже работать в рамках какого-то конкретного API, в то время как `program`-контекст обязан выражаться в общих терминах, применимых к любой возможной нижележащей технологии.

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, посвящённом разработке SDK.

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,
  (context) => {
    // Инициализируем запуск исполнения
    // программы на стороне партнёра
    let execution =
      initExecution(context, …);
    // Подписываемся на события
    // изменения контекста
    context.on(
      'takeout_requested',
      () => {
        // Если запрошена выдача напитка,
        // инициализируем выдачу
        execution.prepareTakeout(() => {
          /* как только напиток
             готов к выдаче,
             сигнализируем об этом,
             вызовом метода контекста,
             а не генерацией события */
          // execution.context
          //   .emit('takeout_ready')
          context.set('takeout_ready');
          // Или ещё более жёстко:
          // context.setTakeoutReady();
        }
      );
    }
  );
  // Так как мы сами
  // изменяем родительский контекст
  // нет нужды что-либо возвращать
```

```
  // return execution.context;
}
```

Вновь такое решение выглядит контринтуитивным, 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 сверху вниз.

Другой аспект 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:

```
execution.prepareTakeout(() => {
  // Вместо обращения к вышестоящей сущности
  // или генерации события на себе,
  // компонент обращается к глобальному
  // состоянию и вызывает действия над ним
  dispatch(takeoutReady());
});
```

Надо отметить, что такой подход *в принципе* не противоречит описанному принципу, 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.

```
execution.prepareTakeout(() => {
  // Вместо обращения к вышестоящей сущности
  // или генерации события на себе,
  // компонент обращается к вышестоящему
  // объекту
  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)
  )
}
```

Проверим себя

Описав указанным выше образом взаимодействие со сторонними API, мы можем (и должны) теперь рассмотреть вопрос, совместимы ли эти интерфейсы с нашими собственными абстракциями, которые мы разработали в главе 9; иными словами, можно ли запустить исполнение такого заказа, оперируя не высокоуровневым, а низкоуровневым API.

Напомним, что мы предложили вот такие абстрактные интерфейсы для работы с произвольными типами API кофемашин:

- `POST /v1/program-matcher` возвращает идентификатор программы по идентификатору кофемашины и рецепта;
- `POST /v1/programs/{id}/run` запускает программу на исполнение.

Как легко убедиться, добиться совместимости с этими интерфейсами очень просто: для этого достаточно присвоить идентификатор `program_id` паре (тип API, рецепт), например, вернув его из метода `PUT /coffee-machines`:

```
PUT /v1/partners/{partnerId}/coffee-machines
{
  "coffee_machines": [{
    "id",
    "api_type",
    "location",
    "supported_recipes"
  }, …]
}
→
{
  "coffee_machines": [{
    "id",
    "recipes_programs": [
      {"recipe_id", "program_id"},
      …
    ]
  }, …]
}
```

И разработанный нами метод

```
POST /v1/programs/{id}/run
```

будет работать и с партнёрскими кофемашинами (читай, с третьим видом API).

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, бóльшая 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 9 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 18. 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 SOLID — 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.

Остаётся, 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 поддерживаемыми кофемашиной опциями, 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 16.

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 имплементаций 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.

Глава 19. 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, тестироваться 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 может 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 пусть условную, но отличную от базовой реализацию каждого интерфейса.

3. Изолируйте зависимости

В случае, если API является гейтвеем, предоставляющим доступ к какому-то нижележащему API или агрегирующим несколько различных API за одним фасадом, велик соблазн предоставить оригинальный интерфейс as is, не внося в него изменений и не усложняя себя жизнь разработкой слабо связанного взаимодействия. Например, разрабатывая интерфейс для запуска программ, описанный в главе 9, 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 16, 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 мультипликатором 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 III. API КАК ПРОДУКТ

Глава 20. 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 предпочтут 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, 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 21. 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 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. внутренними 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.

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 продуктов 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-сферы 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 терраформирование);

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 22. 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 три 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 «KPI 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 бесперспективности 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 23. 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 ускоспециализированных 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 (StackOverflow, Experts Exchange);
- обучающие сервисы (CodeAcademy, 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 выразим 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. Существует мнение, что угодить одновременно обеим аудиториям невозможно: первые ищут возможности глубокой кастомизации (поскольку работают в крупных ИТ-компаниях со сложившимся подходом к разработке), вторым необходим максимально заниженный порог входа. Мы с этим мнением склонны не согласиться по причинам, которые будут обсуждаться в главе «Линейка сервисов API».

Глава 24. 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.

соответствующую 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 25. 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. Дело в том, что описанная в предыдущих главах фрагментация пользователей API

по профессиональному уровню имеет несколько важных следствий:

- 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;
- 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.

Самый важный вывод здесь такой: максимально полно покрыть нужды всех категорий пользователей 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 9 и главу 17.]
2. Базовый уровень — работы с продуктовыми сущностями через формальные интерфейсы. [В случае нашего учебного API этому уровню соответствует HTTP API заказа.]
3. Упростить работу с продуктовыми 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 в конечном итоге получает полный код 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 URL изображения, который позволит показать в приложении партнёра баннер с наиболее релевантным предложением для конечного пользователя.]

В конечном итоге можно прийти к концепции мета-API, когда готовые визуальные компоненты тоже будут иметь какое-то свой высокоуровневый API, который «под капотом» будет обращаться к базовым API.

Важнейшим 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.

Глава 26. 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 компании 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 сырые 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 бóльшей 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 KPI 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 27. 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 быть ничейный 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 фродерами всегда будет более эффективной, нежели множество частных любительских реализаций. К томе же, и серверные ключи могут быть украдены: это сложее, 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 так же 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 фродом 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 внутрикорпоративной 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 фродом, 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 28. Технические способы борьбы с несанкционированным доступом к API

Реализация парадигмы, описанной в предыдущей главе — централизованной борьбы с фродом, 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 эндпойнтов, 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 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 29. 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. соответствующую главу). 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 30. 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 глоссария 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 ухудшат 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 (туториал)

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. Офлайн-документация

Хотя мы и живём в мире победившего онлайна, 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 осложняется, 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 31. 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 симулируется 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 кофемашин. 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 третье — 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 32. 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. Любая программа рано или поздно устареет и будет

переписана; если в рамках этого переписывания будет осуществлён и переход на свежую версию API, партнёр, скорее всего, отнесётся к этому с пониманием. Разумеется, в разных областях этот интервал различен, и зависит от скорости эволюции самой нижележащей платформы.

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 неопределённость 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).