Сергей Константинов
API


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

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Глава 1. О структуре этой книги

Книга, которую вы держите в руках, состоит из введения и трех больших разделов.

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

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

Наконец, третий раздел будет касаться больше не-разработческих 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 для разработчика API. Ввиду того, что API — продукт для разработчиков, перекладывать ответственность за его развитие и поддержку на не-разработчиков 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 отработать корректно 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:

Photo 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 также хорошо иллюстрирует 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 может контролировать, и проблемы с водой, вызванные некомпетентностью подрядчика, неизбежно будут валить на вас.

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 наше API «хорошее».

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 не декларируемых явно целей.)

Как же дизайн API может помочь разработчику? Очень просто: API должно 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 его задачу;
- API должно 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 должно быть консистентно: 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 начинаются, когда мы начинаем API развивать. Добавление новой функциональности рано или поздно приводит к тому, что некогда простое и понятное 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 решает API и как лучше это сделать), 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 развивалось во времени и не превращалось 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 было спроектировано ваше 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.

Более подробно о политиках версионирования будет рассказано в разделе II. В разделе I мы ограничимся лишь указанием версии API в формате `v1`, `v2`, etc.

Глава 6. Условные обозначения и терминология

Разработка программного обеспечения характеризуется, помимо прочего, существованием множества различных парадигм разработки, адепты которых зачастую настроены весьма воинственно по отношению к адептам других парадигм. Поэтому при написании этой книги мы намеренно избегаем слов «метод», «объект», «функция» и так далее, используя нейтральный термин «сущность». Под «сущностью» 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
X-Idempotency-Token: <токен идемпотентости>
{
  …
  // Это однострочный комментарий
  "some_parameter": "value",
  …
}
→ 404 Not Found
Cache-Control: no-cache
{
  /* А это многострочный
     комментарий */
  "error_message"
}
```

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.

Здесь термин «клиент» означает «приложение, установленное на устройстве пользователя, использующее рассматриваемое 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 имеют 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 путем, вы получите на выходе готовое 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 нужно 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 нужно наше 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 чем 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 кофе-машины.

    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 кофе-машин 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 кофе-машин? 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?

Т.к. 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 чем-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 хорошее 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 взымаем 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 *кофе-машине*, 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 наше API к другим платформам, не меняя высокоуровневый интерфейс.

Допустим, мы имеем следующий интерфейс:

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

```
// размещает на указанной кофе-машине
// 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 представить

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 кофе-машин 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 кофе-машинах).

Что немаловажно, от разработчиков потребуется изменить логику своего приложения: если раньше они могли предлагать сначала выбрать объём, а потом кофе-машину, 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 кофе-машины»

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 кофе-машины марки «Доброе утро» в процессе приготовления напитка.

Чем дальше находятся друг от друга программные контексты, которые соединяет наше 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 кофе-машины;
- с другой стороны, кофе-машина 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 кофе-машине
    ]
  }
  …
}
```

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 ваше 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 кофе-машинах, 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 кофе-машин и выяснили, что существует принципиально два класса устройств:

- кофе-машины 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;
- кофе-машины 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 данное API нарушает множество описанных нами принципов проектирования, начиная с отсутствия версионирования; оно приведено 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 такое API от производителей кофе-машин, 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
      // * shed_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 кофе-машиной 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 кофе-машин 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 кофе-машине 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/programs/match
{ "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 кофе-машины — 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/programs/{api_type}/match`
- `POST /v1/programs/{api_type}/{program_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 кофе-машины с передачей внутреннего идентификатор программ — 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 взят
  // * "terminated" – исполнение остановлено
  // * "technical_error" – ошибка при приготовлении
  // * "waiting_time_exceeded" – готовый заказ был
  //   утилизирован, т.к. 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, кофе-машине и программе и формирует stateless-запрос, в котором указаны все нужные данные (тип API кофе-машины и список команд в частности);
- либо в запросе содержатся только идентификаторы, и имплементация методов 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 кофе-машины) и в зависимости от типа API пойдёт по одной из двух веток исполнения:
  - либо вызовет `GET /execution/status` физического API кофе-машины, 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: сличит 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 получению 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 соответствующий этому уровню абстракции предметной области. Запрет «перепрыгивания» уровней приводит к тому, что нам необходимо дублировать статус на каждом уровне независимо.

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 кофе-машины) и в зависимости от типа API пойдёт по одной из двух веток исполнения:
  - либо вызовет `POST /execution/cancel` физического API кофе-машины;
  - 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 кофе-машину (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 оно выполняло свою функцию с минимальными накладными расходами, по сути — дать интерфейс к микроконтроллерам кофе-машины. 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 кофе-машины 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 кофе-машиной 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 кофе-машины 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 и 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 кофе-машине на выполнение указанной программы»).

В нашем умозрительном примере получится примерно так:

1. Сущности уровня пользователя (те, 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.
    - Кофе-машина `coffee-machine` — модель объекта реального мира. Из описания кофе-машины мы, в частности, должны извлечь её положение в пространстве и предоставляемые опции (о чём подробнее поговорим ниже).
2. Сущности уровня управления исполнением (те, работая с которыми, можно непосредственно исполнить заказ).
    - Программа `program` — описывает некоторый план исполнения для конкретной кофе-машины. 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 кофе-машине. Селектор работает только на выбор нужной программы.
    - Запуск программы `programs/run` — конкретный факт исполнения программы на конкретной кофе-машине. Запуски можно:
        - инициировать (создавать);
        - проверять состояние запуска;
        - отменять.
3. Сущности уровня программ исполнения (те, работая с которыми, можно непосредственно управлять состоянием кофе-машины 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 кофе-машина), дополняя их терминами своего уровня (идентификатор запуска). Это совершенно нормально: API должно связывать контексты.

Сценарии использования

На этом уровне, когда наше API уже в целом понятно устроено и спроектированы, 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 кофе-машины 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 кофе-машины
let coffeeMachines = api.getCoffeeMachines();
// Построить пространственный индекс
let coffeeMachineRecipesIndex = buildGeoIndex(recipes, coffee-machines);
// Выбрать кофе-машины, соответствующие запросу пользователя
let matchingCoffeeMachines = coffeeMachineRecipesIndex.query(
  parameters,
  { "sort_by": "distance" }
);
// Наконец, показать предложения пользователю
app.display(coffeeMachines);
```

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 кофе-машин 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/coffee-machines/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 кофе-машину.

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.

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

```
// Ищем кофе-машины, соответствующие запросу пользователя
let coffeeMachines = api.search(parameters);
// Показываем пользователю
app.display(coffeeMachines);
```

Хэлперы

Методы, подобные только что изобретённому нами `coffee-machines/search`, принято называть *хэлперами*. 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 цены, мы может просто закодировать это время в идентификаторе).

Альтернативно, кстати, можно было бы разделить функциональность поиска по заданным параметрам и получения офферов, 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 предусмотрено — 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 все 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 там, где это возможно, логически группировать сущности под одним именем — так, чтобы разработчику никогда не приходилось оперировать более чем 10 сущностями одновременно.

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

```
{
  "results": [
    {
      // Тип кофе-машины
      "coffee_machine_type": "drip_coffee_maker",
      // Марка кофе-машины
      "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 кофе-машину
      "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 кофе машины;
- данные о самой кофе-машине;
- 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 кофе-машине
    "coffee-machine": { "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 кофе-машине
      "options": { "volume" },
      // Метаданные предложения
      "offer": { "id", "valid_until" },
      // Цена
      "pricing": { "currency_code", "price", "localized_price" },
      "estimated_waiting_time"
    }
  }
}
```

Такое 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 одновременно в

моменте нужны 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 чем 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 — это всего лишь обобщения

Правила 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 надо.

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 неудобное, громоздкое, неочевидное 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.

1. Явное лучше неявного

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

Плохо:

```
// Отменяет заказ
GET /orders/cancellation
```

Неочевидно, что достаточно просто обращения к сущности `cancellation` (что это?), тем более немодифицирующим методом `GET`, чтобы отменить заказ.

Хорошо:

```
// Отменяет заказ
POST /orders/cancel
```

Плохо:

```
// Возвращает агрегированную статистику заказов за всё время
GET /orders/statistics
```

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:

```
// Возвращает агрегированную статистику заказов за указанный период
POST /v1/orders/statistics/aggregate
{ "begin_date", "end_date" }
```

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», что уж говорить о каких-то более сложных стандартах.

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"` — стандартов записи дат существует огромное количество, 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
`"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. 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.

4. Сущности должны именоваться конкретно

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

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

Хорошо: `user.get_id()`.

5. Не экономьте буквы

В XXI веке давно уже нет нужды называть переменные покороче.

Плохо: `order.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 str2
// 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 (lookup_character_set, str)` — 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.

6. Тип поля должен быть ясен из его названия

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

То же касается и примитивных типов. Сущности-массивы должны именоваться во множественном числе или собирательными выражениями — `objects`, `children`; если это невозможно (термин неисчисляемый), 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 из названия это должно быть очевидно, т.е. именование должно описывать некоторое качественное состояние, например, `is_ready`, `open_now`.

Плохо: `"task.status": true` — неочевидно, что статус бинарен, плюс такое API будет нерасширяемым.

Хорошо: `"task.is_finished": true`.

Отдельно следует оговорить, что на разных платформах эти правила следует дополнить по-своему с учетом специфики first-class citizen-типов. Например, объекты типа `Date`, если таковые имеются, разумно индицировать с помощью, например, постфикса `_at` (`created_at`, `occurred_at`, etc) или `_date`.

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

Плохо:

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

Слово "functions" многозначное; может означать и встроенные функции, и написанный код, и состояние (функционирует-не функционирует).

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

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

Плохо: `begin_transition` / `stop_transition`
— `begin` и `stop` — непарные термины; разработчик будет вынужден рыться в документации.

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

Плохо:

```
// Находит первую позицию 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 подчеркивания;
- близкие по смыслу методы имеют разный порядок аргументов `needle`/`haystack`;
- первый из методов находит только первое вхождение строки `needle`, а другой — все вхождения, и об этом поведении никак нельзя узнать из сигнатуры функций.

Упражнение «как сделать эти интерфейсы хорошо» предоставим читателю.

8. Клиент всегда должен знать полное состояние системы

Правило можно ещё сформулировать так: не заставляйте клиент гадать.

Плохо:

```
// Создаёт комментарий и возвращает его id
POST /comments
{ "content" }
→
{ "comment_id" }
```

```
// Возвращает комментарий по его id
GET /comments/{id}
→
{
  // Комментарий не опубликован
  // и ждёт прохождения капчи
  "published": false,
  "action_required": "solve_captcha",
  "content"
}
```

— хотя операция будто бы выполнена успешно, клиенту необходимо сделать дополнительный запрос, чтобы понять необходимость решения капчи. Между

вызовами `POST /comments` и `GET /comments/{id}` клиент находится в состоянии кота Шрёдингера: непонятно, опубликован комментарий или нет, и как отразить это пользователю.

Хорошо:

```
// Создаёт комментарий и возвращает его
POST /v1/comments
{ "content" }
→
{ "comment_id", "published", "action_required", "content" }
```

```
// Возвращает комментарий по его id
GET /v1/comments/{id}
→
{ /* в точности тот же формат,
    что и в ответе POST /comments */
    …
}
```

Вообще в 9 случаях из 10 (а фактически — всегда, когда размер ответа не оказывает значительного влияния на производительность) во всех отношениях лучше из любой модифицирующей операции возвращать полное состояние сущности в том же формате, что и из операции доступа на чтение.

То же соображение применимо и к значениям по умолчанию. Не заставляйте клиент гадать эти значения, или хуже — хардкодить их. Возвращайте заполненные значения необязательных полей в ответе операции создания (перезаписи) сущности.

9. Идемпотентность

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}
{ "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 нативно 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 (если, конечно, вы следовали правилу #9 при разработке API) 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.

10. Кэширование

В клиент-серверном API, как правило, сеть и ресурс сервера не бесконечны, поэтому кэширование на клиенте результатов операции является стандартным действием.

Желательно в такой ситуации внести ясность; если не из сигнатур операций, то хотя бы из документации должно быть понятно, каким образом можно кэшировать результат.

Плохо:

```
// Возвращает цену лунго в кафе,
// ближайшем к указанной точке
GET /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 нужен и по временной, и по пространственной координате следует поступить примерно так:

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

11. 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 сличить 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`), 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 которых 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=<значение курсора>
{ "records", "cursor" }
```

Достоинством схемы с курсором является возможно 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.

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 такое API нерасширяемо — невозможно добавить сортировку по двум или более полям.

Хорошо: в представленной постановке задача, вообще 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}
```

Т.к. созданное представление уже неизменяемо, доступ к нему можно организовать как угодно: через курсор, limit/offest, заголовок Range и т.д. Однако надо иметь в виду, что при переборе таких списков может получиться так, что порядок будет 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:

```
POST /v1/records/modified/list
{
  // Опционально
  "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.

12. 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 все 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. Do you mean 'lungo'?"
      },
      {
        "field": "position.latitude",
        "error_type": "constraint_violation",
        "constraints": {
          "min": -180,
          "max": 180
        },
        "message":
          "'position.latitude' value must fall in [-180, 180] interval"
      }
    ]
  }
}
```

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

13. 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 должно 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.