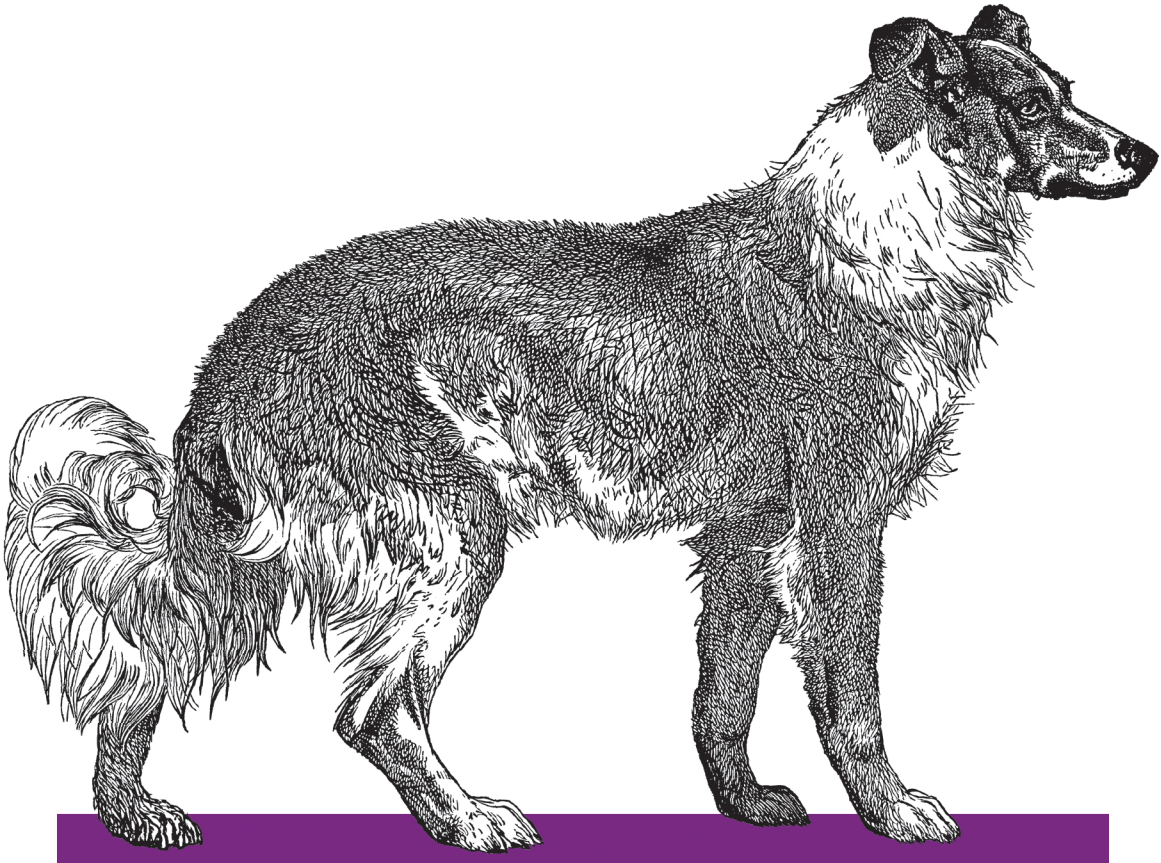
O'REILLY®



Непрерывное развитие API

ПРАВИЛЬНЫЕ РЕШЕНИЯ В ИЗМЕНЧИВОМ ТЕХНОЛОГИЧЕСКОМ ЛАНДШАФТЕ

Мехди Меджуи, Эрик Уайлд
Ронни Митра, Майк Амундсен

ПИТЕР®

Continuous API Management

Making the Right Decisions in an Evolving Landscape

Mehdi Medjaoui, Erik Wilde, Ronnie Mitra, and Mike Amundsen

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

Мехди Меджуи, Эрик Уайлд
Ронни Митра, Майк Амундсен

Непрерывное развитие API

ПРАВИЛЬНЫЕ РЕШЕНИЯ
В ИЗМЕНЧИВОМ ТЕХНОЛОГИЧЕСКОМ ЛАНДШАФТЕ

ПИТЕР®

Санкт-Петербург • Москва • Екатеринбург • Воронеж
Нижний Новгород • Ростов-на-Дону
Самара • Минск

2020

Мехди Меджуи, Эрик Уайлд, Ронни Митра, Майк Амундсен

Непрерывное развитие API. Правильные решения в изменчивом технологическом ландшафте

Перевела с английского А. Григорьева

Заведующая редакцией	*Ю. Сергиенко*
Руководитель проекта	*С. Давид*
Ведущий редактор	*Н. Гринчик*
Литературный редактор	*Н. Рощина*
Художественный редактор	*С. Заматевская*
Верстка	*О. Богданович*

ББК 32.988.02-018 УДК 004.438.5

Мехди Меджуи, Эрик Уайлд, Ронни Митра, Майк Амундсен

Н53 Непрерывное развитие API. Правильные решения в изменчивом технологическом ландшафте. — СПб.: Питер, 2020. — 272 с.: ил. — (Серия «Бестселлеры O'Reilly»).

ISBN 978-5-4461-1232-6

Для реализации API необходимо провести большую работу. Чрезмерное планирование может стать пустой тратой сил, а его недостаток приводит к катастрофическим последствиям. В этой книге вы получите решения, которые позволят вам распределить необходимые ресурсы и достичь требуемого уровня эффективности за оптимальное время. Как соблюсти баланс гибкости и производительности, сохранив надежность и простоту настройки? Четыре эксперта из Академии API объясняют разработчикам ПО, руководителям продуктов и проектов, как максимально увеличить ценность их API, управляя интерфейсами как продуктами с непрерывным жизненным циклом.

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

ББК 32.988.02-018
УДК 004.438.5

ISBN 978-1492043553 англ. Authorized Russian translation of the English edition of Continuous API Management.

ISBN 978-5-4461-1232-6

Изготовлено в России. Изготовитель: ООО «Прогресс книга». Место нахождения и фактический адрес: 194044, Россия, г. Санкт-Петербург, Б. Сампсониевский пр., д. 29А, пом. 52. Тел.: +78127037373.
Дата изготовления: 09.2019. Наименование: книжная продукция. Срок годности: не ограничен.
Налоговая льгота — общероссийский классификатор продукции ОК 034-2014, 58.11.12 — Книги печатные профессиональные, технические и научные.
Импортер в Беларусь: ООО «ПИТЕР М», 220020, РБ, г. Минск, ул. Тимирязева, д. 121/3, к. 214, тел./факс: 208 80 01.
Подписано в печать 16.09.19. Формат 70×100/16. Бумага офсетная. Усл. п. л. 21,930. Тираж 500. Заказ 0000.
Отпечатано в ОАО «Первая Образцовая типография». Филиал «Чеховский Печатный Двор» .
142300, Московская область, г. Чехов, ул. Полиграфистов, 1.
Сайт: www.chpk.ru. E-mail: marketing@chpk.ru Факс: 8(496) 726-54-10, телефон: (495) 988-63-87

Краткое содержание

Оглавление

Вступление

Системы API необходимы любым компании, организации, учреждению или государственному органу, которые учатся правильно управлять своими цифровыми ресурсами в условиях постоянно расширяющейся и изменяющейся цифровой среды. Цифровая трансформация, происходящая в последние годы, начала приводить к изменениям в среде API: компании уже перестают задавать вопрос: «Надо ли нам использовать API?» — и начинают искать информацию о том, как правильно это делать. В организациях понимают, что недостаточно просто создать API, нужно разработать весь его жизненный цикл. Авторы из Академии API, написавшие книгу «Непрерывное развитие API», хорошо понимают, что требуется, чтобы провести API по пути от идеи до реализации последовательно, в нужном масштабе и неоднократно, их опыт позволяет им стать практически уникальными наставниками в данной сфере.

Большинство специалистов по API работают с системой API, включающей в себя только один набор API. Меджуи, Уайлд, Митра и Амундсен обладают уникальным опытом работы с системой, охватывающей тысячи API, несколько отраслей и с десяток крупнейших современных предприятий. Высококлассных, талантливых специалистов по API по всему миру можно пересчитать по пальцам, и Меджуи, Уайлд, Митра и Амундсен всегда будут первыми, кого я назову, загибая пальцы на правой руке. Богатый опыт авторов помогает понять, как провести API по пути от идеи до проекта, от разработки до производства. Не существует другой команды экспертов по API, у которых есть такие же уникальные знания, благодаря чему эта книга станет настольной — тем зачитанным до дыр томиком, который вы будете перечитывать снова и снова.

Я прочитал множество книг по техническим аспектам создания API, книг на такие темы, как гипермедиа и все, что нужно знать о REST, или как реализовать свое видение с помощью различных языков программирования и платформ. Но это первая прочитанная мною книга об API, подходящая для разработки API целиком, от начала до конца, учитывая не только технологические детали, но и важные моменты

использования API в бизнесе, в том числе человеческий фактор при изучении API, их реализацию и активацию на крупных предприятиях. В издании методично описываются структурные элементы, которые понадобятся любому разработчику архитектуры API для создания надежных, безопасных и целостных API в нужном масштабе. Книга поможет любой команде по созданию API определить количество необходимых операций и обдумать улучшения и изменения API с критической точки зрения. В то же время авторы предлагают структурированный, но гибкий подход к выпуску API по одному стандарту для разных команд.

Отложив эту книгу, я почувствовал, что у меня поменялись взгляды на современный жизненный цикл API. Но что более важно, появилось много идей о том, как я могу оценить и измерить свои операции API и стратегию жизненного цикла API, которую сейчас использую для управления этими операциями. Даже с моим знанием этой области книга заставила меня посмотреть на данную среду по-новому. В итоге я почувствовал, что насытился информацией, которая не только освежила мои знания, но и изменила представления о том, что мне *казалось* известным, заставив меня расти над собой и прогрессировать в своей сфере деятельности. Для меня самое главное на пути разработки API — постоянно бросать себе вызов, учиться, планировать, выполнять, оценивать и повторять, пока не добьешься желаемого результата. Книга «Непрерывное развитие API» отражает реальность разработки API и представляет собой многоразовое руководство по технологии и тактике создания API в масштабе целого предприятия.

Советую прочитать эту книгу больше одного раза. Прочтите, потом пойдите и реализуйте свои идеи. Улучшите свою стратегию API, определите собственную версию жизненного цикла API, используя то, что узнали у Меджуи, Уайлда, Митры и Амундсена, и применяя это в работе. Гарантирую, что по всей книге разбросаны маленькие самородки знаний, которые будут представляться вам в новом свете при каждом перечитывании, — то, что поможет лучше понять происходящее в среде API и увереннее заявить о себе или даже занять лидирующую позицию в развивающейся онлайн-индустрии.

Ким Лейн, евангелист API

Предисловие

Когда общество и бизнес естественным образом тесно переплелись с цифровыми технологиями, спрос на взаимосвязанное программное обеспечение резко вырос. В свою очередь, интерфейс для программирования приложений (API) стал важным ресурсом для современных компаний, потому что он упрощает установление связей между ПО. Но управление этими API стало вызывать новые сложности. Чтобы получить от API максимум, необходимо научиться управлять их разработкой, запуском, ростом, качеством и безопасностью, учитывая сложные факторы контекста, времени и масштаба.

Кому следует прочесть эту книгу

Если вы только начинаете разрабатывать программу внедрения и использования API и хотите понять, какой объем работы вам предстоит выполнить, или если уже создали некие API и стремитесь узнать, как управлять ими более эффективно, тогда эта книга для вас.

В ней мы попытались построить систему управления API, применимую в различных контекстах. На ее страницах вы найдете указания, как управлять одним API, которым вы хотите поделиться с разработчиками по всему миру, а также советы по созданию сложной системы API в архитектуре микросервисов, предназначенной только для внутренних разработчиков, и информацию обо всех промежуточных вариантах.

Кроме того, мы постарались сделать книгу максимально технологически нейтральной. Советы и анализ, которые мы представляем, применимы к любой архитектуре, основанной на API, включая HTTP CRUD, REST, GraphQL, и событийно-ориентированных стилях взаимодействия. Книга предназначается каждому, кто хочет принимать правильные и эффективные решения в работе с API.

Что вы найдете в книге

Материал книги основан на наших коллективных знаниях, полученных за много лет создания, разработки и улучшения API — как своих, так и чужих. В ней изложен весь наш опыт. Мы определили два ключевых фактора для эффективной разработки API: необходимость продуктоориентированного подхода и формирование правильной команды. Мы также определили три важных фактора для управления этой работой: руководство, развитие продукта и разработка системы API.

Эти пять элементов формируют фундамент, на котором можно построить успешную программу по управлению API. Мы знакомим читателя со всеми этими темами и предоставляем руководство по тому, как вписать их в контекст вашей организации.

Структура издания

Мы организовали книгу так, чтобы список вопросов, требующих внимания, увеличивался по мере чтения. Начнем с основных понятий руководства, основанного на решениях, и концепции «API как продукт». Затем сделаем обзор всей работы, необходимой для создания продукта с API.

Далее к простому обзору одного API добавим аспект времени и тщательнее рассмотрим, как работать над изменениями API и как на это влияет развитие API. Потом вас ждет более подробное изучение команд и людей, занимающихся подобными изменениями. Наконец, во второй половине книги мы обсудим сложности масштаба системы API и управления ею.

Вот краткое содержание всех глав.

Глава 1 знакомит вас со сферой управления API и объясняет, почему так сложно эффективно управлять ими.

В главе 2 мы изучаем руководство с точки зрения принятия решений — краеугольный камень управления API.

В главе 3 рассказываем о концепции «API как продукт» и о том, почему это важнейшая часть любой стратегии API.

Глава 4 освещает десять базовых принципов работы в сфере продуктов с API. Эти принципы формируют систему задач по принятию решений, которыми нужно управлять.

Глава 5 проливает свет на то, каково это — непрерывно менять API. Мы рассказываем о необходимости принять идею постоянных изменений и знакомим с разными типами изменений в API и их последствиями.

Глава 6 рассматривает жизненный цикл продукта с API — структуру, которая поможет управлять работой API, опираясь на базовые принципы в течение всего времени использования продукта.

Глава 7 рассказывает о человеческом факторе в управлении API — типичных ролях, зонах ответственности и схемах разработки для команды, создающей API.

Глава 8 добавляет к проблеме управления API перспективу развития. В ней приведены «восемь V» — разнообразие (variety), словарь (vocabulary), объем (volume), скорость (velocity), уязвимость (vulnerability), видимость (visibility), контроль версий (versioning) и изменяемость (volatility), — которые нужно учитывать при одновременном изменении нескольких API.

В главе 9 мы освещаем подход непрерывной разработки системы, необходимый для постоянного и согласованного управления изменениями в API.

Глава 10 очерчивает систему с точки зрения подхода «API как продукт» и изменения работы API во время развития системы, в которую он входит.

Глава 11 суммирует изложенную здесь информацию по управлению API и дает советы, как подготовиться к будущей работе и сразу же начать ее.

Чего нет в этой книге

Сфера управления API обширна, в ней гигантское количество вариантов контекстов, платформ и протоколов. Учитывая, что объем книги не бесконечен, как и время, отведенное на написание, было невозможно включить в нее все конкретные способы реализации API. Эта книга не руководство по разработке REST API или выбору шлюза безопасности. Если вы ищете пошаговую инструкцию по написанию кода для API или разработке HTTP API, здесь вы ее не найдете.

Хоть мы и приводим примеры, где говорится о конкретных способах, данная книга не о реализации API — в этом вам могут помочь другие книги, блоги и видео. Она затрагивает редко упоминаемую проблему: как эффективно управлять работой по созданию API в сложной, постоянно меняющейся организационной системе.

Условные обозначения

В этой книге применяются следующие шрифтовые обозначения.

Курсив

Обозначает новые термины и слова, на которых сделан акцент.

Рубленый шрифт

Им набраны веб-ссылки и адреса электронной почты.

`Моноширинный шрифт`

Обозначает программные элементы, такие как названия переменных или функций, типы данных, утверждения и ключевые слова. Им также выделены названия и расширения файлов.

Этот элемент обозначает совет или предложение.

Такой элемент обозначает примечание.

Этот элемент обозначает предупреждение.

Благодарности

Мы хотим поблагодарить многих людей за помощь и поддержку, которую мы получили за последний год, пока писали эту книгу. В первую очередь спасибо всем, кто отвечал на наши вопросы, консультировал нас и приходил на наши семинары. Мы получили от всех вас прекрасную обратную связь и отличные советы. Мы также благодарим сотрудников CA Technologies, которые поддерживали нас все эти годы и помогали организовывать семинары. Отдельная благодарность людям, вычитывавшим ранние черновики и помогавшим нам в создании конечного варианта книги, которая сейчас лежит перед вами. Мэтт Макларти, Джеймс Хиггинботэм и Крис Вуд выделили время в своем плотном графике, прочли нашу работу, написали отзывы и указали на моменты, требовавшие улучшения. И наконец, огромное спасибо команде O'Reilly Media. Спасибо Алисии Янг, Джастину Биллингу и всем сотрудникам издательства за помощь в превращении изначальных идей в эту книгу.

От издательства

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

Мы будем рады узнать ваше мнение!

На веб-сайте издательства www.piter.com вы найдете подробную информацию о наших книгах.

1 Сложности в управлении API

> Управление — это прежде всего занятие на стыке искусства, науки и ремесла.
>
> *Генри Минцберг*

Исследование, проведенное в 2017 году компанией Coleman Parkes (http://bit.ly/CP_APIs_survey), показало, что почти у каждой девятой из десяти международных компаний есть своего рода программы внедрения и использования API. Оно также показало, что эти программы приносят компаниям множество различных преимуществ, включая общее увеличение скорости выхода продукта на рынок примерно на 18 %. Однако только около 50 % этих же компаний утверждают, что обладают продвинутой программой управления API. Это указывает на важный пробел во многих программах внедрения и применения API корпоративного уровня — разницу между наличием функционирующих API, которые вносят основной вклад в доход, и навыками управления и инфраструктурой, необходимыми для поддержки этих приносящих доход API. Устранить этот пробел и призвана наша книга.

Хорошая новость: существует множество компаний, успешно управляющих своими программами внедрения и использования API. Новость похуже: либо их опытом и знаниями трудно поделиться, либо они не всегда доступны по нескольким причинам. Чаще всего организациям, успешно управляющим API, просто некогда поделиться опытом с другими. Несколько раз мы общались с компаниями, которые неохотно делятся своими знаниями по управлению API. Они уверены, что навыки в сфере API — это конкурентное преимущество, и редко открывают свои тайны. Наконец, даже когда компании делятся опытом на публичных конференциях, в статьях или блогах, информация обычно специфична для конкретной компании и ее тяжело перенести на все разнообразие программ внедрения и применения API в других организациях.

В книге мы пытаемся разобраться с последней из этих проблем — превращением специфичных примеров в универсальный опыт, которым могут воспользоваться любые организации. Мы посетили десятки компаний, разговаривали с множеством специалистов по API и попытались найти связующие нити между примерами,

которыми они поделились с нами и общественностью. Через всю книгу проходят несколько тем, которые мы представим в данной главе.

Главная сложность — понять, что именно люди имеют в виду, говоря об API. Во-первых, термин *API* может относиться просто к *интерфейсу* (например, URL HTTP-запроса и возвращаемый JSON). Во-вторых, он может применяться к тому коду и той процедуре развертки кода, которые необходимы для создания доступного сервиса (например, `customerOnBoarding` API). И в-третьих, иногда термин *API* относится к конкретному *экземпляру приложения*, предоставляющему доступ через API (например, `customerOnBoarding` API, запущенный в облаке AWS, в отличие от `customerOnBoarding` API, запущенного в облаке Azure).

Другая сложность в управлении API — это разница между созданием, разработкой и реализацией *одного API* и поддержкой *многих API* — так называемой *системы API* — и управлением ими. В этой книге мы подробно рассмотрим проблему со всех сторон. В работе с одним API могут помочь такие понятия, как *«API как продукт»* (AaaP), и навыки, необходимые для создания и поддержки API (они же *базовые принципы API*). Мы также поговорим о таких важных факторах, как роль модели развития API и работа с постепенными его изменениями.

Другая сторона проблемы — управление системой API. В данном случае система — это API всех бизнес-доменов, запущенные на всех платформах и управляемые всеми командами, разрабатывающими API в вашей компании. На сложности, порождаемые такой системой, нужно смотреть с различных точек зрения. Например, учитывать влияние масштаба и контекста на процесс дизайна и реализации API, а также то, что исключительно из-за своих размеров большие экосистемы могут быть значительно изменчивее и уязвимее остальных.

Наконец, мы коснемся процесса принятия решений во время управления экосистемой API. Наш опыт свидетельствует: это ключ к созданию успешного плана *руководства* вашими программами внедрения и использования API. Оказывается, что способ принятия решений должен меняться вместе со средой: то, что вы держитесь за старые модели управления, может помешать программе достичь успеха и даже поставить под удар существующие API.

Прежде чем начать изучение способов работы с индивидуальными API и системой API, рассмотрим два важных вопроса: что такое управление API и почему это так сложно?

Что такое управление API

Как уже говорилось, управление API включает в себя не только руководство разработкой, внедрением и реализацией одного API, но и управление экосистемой API, распределение решений внутри организации и даже перемещение уже существующих API в вашу растущую систему. В данном разделе мы остановимся на каждом из этих пунктов, но сначала кратко объясним, что же мы понимаем под API.

Что такое API

Иногда люди, использующие термин *API*, говорят не только об интерфейсе, но и о функциональности — о коде, скрывающемся за интерфейсом. Например, можно сказать: «Нам нужно поскорее выпустить обновленный клиентский API, чтобы другие команды смогли начать задействовать новые функции поиска, которые мы добавили». В других случаях этот термин может применяться только в отношении деталей самого интерфейса. Например, сотрудник вашей команды может сказать: «Я бы хотел разработать новый JSON API для существующих SOAP-сервисов добавления клиентов в систему (customer onboarding)». Конечно, оба раза термин употреблен правильно и в обоих случаях понятно, что имелось в виду, но иногда можно и запутаться.

Чтобы прояснить разницу и упростить разговор об интерфейсе и функциональности, мы введем несколько дополнительных терминов: интерфейс, реализация и экземпляр (приложения).

Интерфейс, реализация и экземпляр. Сокращение API обозначает *«программный интерфейс приложения»*. Мы используем интерфейс, чтобы получить доступ к тому, что работает «под капотом» API. Например, у вас может быть API, отображающий задачи управления учетными записями пользователей. Этот интерфейс может позволить разработчикам:

- открыть новую учетную запись;
- редактировать профиль уже существующей учетной записи;
- изменить (заморозить или активировать) статус учетной записи.

Данный интерфейс обычно предоставляется с помощью таких общеупотребимых протоколов, как HTTP, Thrift, TCP/IP и т. д., и опирается на стандартизированные форматы JSON, XML или HTML.

Но это только интерфейс. Что-то должно исполнять данные задачи. Это что-то мы в дальнейшем будем называть *реализацией*. Реализация — это тот элемент системы, который предоставляет саму ее функциональность. Часто реализация написана на языке программирования, например Java, C#, Ruby или Python. В рамках примера с пользовательской учетной записью реализация `UserManagement` (Управление пользователями) может включать в себя возможность создавать, добавлять, редактировать и удалять пользователей. Затем эта функциональность передается с помощью вышеупомянутого интерфейса.

Разъединение интерфейса и реализации

Следует отметить, что функциональность описанной реализации — это простой набор действий по схеме «создать, прочесть, обновить, удалить» (CRUD), тогда как в упомянутом интерфейсе есть три действия: OnboardAccount (Активировать учетную

запись), EditAccount (Редактировать учетную запись) и ChangeAccountStatus (Изменить статус учетной записи). Это кажущееся несовпадение между реализацией и интерфейсом широко распространено и может оказаться полезным. Оно разъединяет конкретную реализацию каждого сервиса и интерфейс, используемый для доступа к нему, облегчая изменения без разрыва связей.

Третий термин в нашем списке — *экземпляр*. Экземпляр приложения с API — это комбинация интерфейса и реализации. Это удобный способ говорить о существующем API, который уже запущен в производственной среде. Экземплярами управляют с помощью системы показателей, благодаря которым можно убедиться в их жизнеспособности. Экземпляры регистрируют и документируют, чтобы разработчикам было проще найти и использовать API для решения проблем реального мира. Экземпляры защищают, чтобы только авторизованные пользователи могли выполнять действия и читать/записывать данные, необходимые для этого.

Отношения между тремя элементами иллюстрирует рис. 1.1. В этой книге под API мы чаще всего понимаем экземпляр API — полностью рабочую комбинацию интерфейса и реализации. В тех случаях, когда хочется сделать акцент *только* на интерфейсе или *только* на реализации, мы подчеркнем это в тексте.

Рис. 1.1. Три элемента API

Больше чем просто API

Тема нашей книги не ограничивается лишь API — техническими деталями его интерфейса и реализации. Конечно, традиционные элементы «проект — разработка — реализация» важны для жизненного цикла ваших API. Но *управление* API включает в себя также тестирование, документирование и публикацию их на портале, чтобы пользователи (разработчики внутри организации, партнеры, сторонние анонимные разработчики приложений и т. д.) могли их найти и правильно использовать. Кроме того, API надо защищать, следить за ними во время их работы и поддерживать их работу (включая изменения в ней) в течение всего

жизненного цикла. Эти дополнительные элементы представляют собой так называемые *базовые принципы API*, необходимые для всех API, с которыми приходится сталкиваться всем специалистам в данной области. Мы подробно остановимся на базовых принципах в главе 4, где изучим список из десяти ключевых методик, необходимых для создания и поддержки качественных API.

Плюс этих методик в том, что они применимы к любым API. Например, навык хорошего документирования API может передаваться от одной команды API к другой. То же относится к изучению навыков тестирования, схем безопасности и т. д. Это также означает, что, даже если для каждого домена API у вас предусмотрены отдельные команды (по продажам, по продукту, вспомогательного офиса и т. д.), все равно есть пересекающиеся интересы, которые связывают членов этих команд между собой[1]. Так что создание и организация команд, которые разрабатывают API, — еще один важный аспект управления интерфейсами. О том, как это работает в разных организациях, мы поговорим в главе 7.

Стадии развития API

Однако понимание базовых принципов API — это еще не все. Каждый API в вашей программе проходит через собственный жизненный цикл — серию предсказуемых и полезных этапов. Если вы знаете, в какой точке пути API находитесь, то сможете определить, сколько времени и ресурсов инвестировать в API в данный момент. Понимание того, как *развивается* API, позволяет вам распознавать те же самые этапы в различных API и помогает удовлетворять меняющиеся запросы к времени и ресурсам на каждом.

На первый взгляд кажется правильным, что при планировании, разработке и релизе API надо учитывать все базовые принципы. Но это не так. Часто на ранних этапах API важнее всего сосредоточиться на планировании и разработке и тратить меньше сил, допустим, на документацию. На других этапах, например, когда прототип уже находится у бета-тестеров, важнее будет потратить больше времени на отслеживание использования API и его защиту от неправильного применения. Понимание этапов развития поможет вам использовать ограниченное количество ресурсов для достижения максимального результата. Мы подробно опишем этот процесс в главе 6.

Больше одного API

Как уже знают многие читатели, все меняется, когда начинаешь управлять большим количеством API. У нас есть клиенты, которым нужно разработать тысячи

[1] На стриминговом музыкальном сервисе Spotify эти пересекающиеся группы называются гильдиями. Больше информации по этой теме вы найдете в подразделе «Масштабирование команд» на с. 161.

API, а также отслеживать их и управлять ими в течение долгого времени. В этой ситуации люди меньше фокусируются на деталях реализации отдельного API и больше — на том, как эти API сосуществуют в постоянно растущей динамичной экосистеме. Как говорилось ранее, мы называем эту экосистему *системой API* и посвящаем ей несколько глав во второй половине книги.

В такой ситуации сложно обеспечить определенный уровень постоянного роста без пробок и задержек, связанных с централизованным управлением и изучением всех деталей API. Обычно такого уровня достигают, передавая ответственность за эти вопросы командам индивидуальных API и сосредотачивая усилия центрального управления/руководства на упорядочении взаимодействия API. Таким образом обеспечиваются наличие базового набора общих сервисов или инфраструктуры (безопасность, мониторинг и т. д.) и их доступность для всех команд по API, а более автономным командам предоставляются общее руководство и поддержка. Дело в том, что часто необходимо отойти от обычной модели централизованного руководства и контроля.

Одной из сложностей при передаче прав на принятие решений и расширении самостоятельности в организации является то, что вышестоящим лицам в таком случае легко потерять из виду важные операции, происходящие на уровне команд. В прошлом командам приходилось просить разрешения на то, чтобы предпринять какое-то действие, но теперь компании, распространяющие дополнительную автономию на индивидуальные команды, будут поощрять их действовать, не дожидаясь рассмотрения и разрешения начальства.

Сложность управления средой API связана в основном с *масштабом* и *объемом*. Оказывается, когда ваша программа с API растет, она не просто становится больше, но и меняет *форму*. Мы обсудим это более детально позже в данной главе (см. раздел «Почему управлять API — сложно» на с. 25).

API в бизнесе

Важно помнить не только о деталях создания API и управления ими в составе системы, но и о том, что вся эта работа должна поддерживать цели и задачи бизнеса. API — это не только технические детали JSON или XML, синхронности или асинхронности и т. д. Это способ связывать области бизнеса между собой, делиться важными функциями и знаниями, что помогает компании работать эффективнее. API часто представляют собой возможность раскрыть то ценное, что уже есть в организации, например, при создании новых приложений, открытии новых источников дохода и создании нового бизнеса.

Такой подход позволяет сосредоточиться на потребностях тех, кто использует API, а не тех, кто их создает и публикует. Этот клиентоориентированный подход обычно называют «работа, которую нужно сделать», или JTBD (Jobs to Be Done). Его предложил Клейтон Кристенсен из Гарвардской школы бизнеса, чьи книги *The Innovator's Dilemma* («Дилемма инноватора») и *The Innovator's Solution* («Решение

инноватора») (Harvard Business Review Press) тщательно исследуют возможности такого подхода. Он служит напоминанием о том, что API нужны для решения задач бизнеса, и важен для запуска успешной программы с API и управления ею. Наш опыт свидетельствует о том, что компании, удачно применяющие API в задачах своего бизнеса, относятся к ним как к продуктам, которые должны выполнить работу в том же смысле, в каком концепция JTBD Кристенсена решает проблемы клиентов.

Один из способов, которым программа с API может помочь бизнесу, — это создание набора инструментов (API) для новых решений, не требующих больших расходов. Например, если у вас есть API `OnlineSales` (Онлайн-продажи), позволяющий ключевым партнерам управлять продажами и наблюдать за их активностью, и API `MarketingPromotions` (Маркетинговое продвижение), который позволяет команде по маркетингу разрабатывать и отслеживать кампании по продвижению продуктов, появляется возможность создать новое партнерское решение — приложение `SalesAndPromotions` (Продажи и продвижение).

Другой способ помощи бизнесу — облегчение доступа к важным данным клиента или рынка, которые можно сопоставить с новыми тенденциями или уникальным поведением в новых клиентских сегментах. API предоставляет безопасный и легкий доступ к этим данным (анонимизированным и отфильтрованным), тем самым помогая вашему бизнесу быстрее и дешевле реализовать новые возможности, продукты или услуги и даже создать новые инициативы.

Мы рассматриваем этот важный аспект (AaaP) в главе 3.

Почему управлять API — сложно

Как мы сказали в начале главы, многие компании уже запустили программы с API, однако только около 50 % из них считают, что хорошо *управляют* своими API. Почему так? В чем заключаются проблемы и как вы можете помочь своей компании справиться с ними?

Посетив множество компаний по всему миру и обсудив с ними управление жизненным циклом API, мы выделили несколько основных тем.

- *Объем*. На чем должны сосредоточиться команды по архитектуре центрального программного обеспечения, управляя API в течение долгого времени.
- *Масштаб*. Зачастую то, что работает, когда компании только начинают заниматься API, перестает работать, когда программа из нескольких небольших команд разрастается до глобальной инициативы.
- *Стандарты*. Мы обнаружили, что с развитием программ усилия по управлению и руководству ими надо перенести с подробных советов по разработке

и реализации API на более общую стандартизацию среды API, позволяя командам принимать больше самостоятельных решений на детальном уровне.

Фактически именно постоянный баланс этих трех элементов — объема, масштаба и стандартов — создает жизнеспособную и растущую программу по управлению API. Поэтому стоит остановиться на них подробнее.

Объем

Одна из самых больших трудностей в ходе работе с жизнеспособной программой управления API — достичь оптимального уровня централизованного контроля. Усложняет дело то, что оптимальный уровень меняется вместе с развитием программы.

На ранних этапах развития программы имеет смысл сосредоточиться на деталях разработки API. Когда API еще находится в зачаточном состоянии, они могут исходить напрямую от команды, создающей его, — сотрудники изучают существующие «в дикой природе» программы, выбирают инструменты и библиотеки, подходящие для стиля того API, который они планируют создать, и затем разрабатывают его.

На первоначальном этапе развития API все в новинку. Все проблемы возникают (и решаются) впервые. Эти случаи часто в итоге заносятся в хроники в качестве лучших методик API компании или попадают в ее основные инструкции и т. д. Они имеют значение для маленькой команды, работающей над несколькими API в первый раз. Однако эти изначальные инструкции могут оказаться неполными.

С увеличением количества команд, работающих над API в компании, растет и разнообразие стилей, ситуаций и точек зрения. Становится сложнее поддерживать стабильность во всех командах, и не только потому, что некоторые из них не соблюдают инструкции, опубликованные компанией. Может быть, новая команда работает с другим набором готовых продуктов и это мешает ей следовать изначальным инструкциям. Может быть, ее члены не работают в среде с потоком событий, а поддерживают API, основанные на XML, в режиме «запрос — ответ». Конечно, им тоже нужно руководство, но оно должно соответствовать *их* сфере и потребностям *их* клиентов.

Разумеется, существуют инструкции, которые должны быть универсальными и общими для всех команд, но они должны соответствовать сферам их задач и запросам пользователей их API. С расширением вашего сообщества увеличивается и разнообразие. Очень важно не допустить ошибку и не попытаться его уничтожить. Здесь от вас требуется сместить фокус контроля с *директивных распоряжений* (например, «Все API *должны* использовать следующие схемы URL...») на указание *направлений* (например, «API, работающие на HTTP, *могут* использовать один из следующих шаблонов URL...»).

Другими словами, с расширением объемов вашей программы необходимо соответственно расширять и инструкции. Это особенно важно для международных

предприятий, где местная культура, язык и история играют важную роль в том, как команды думают, творят и решают задачи.

Это подводит нас к следующему ключевому элементу — масштабу.

Масштаб

Еще одна сложность при создании и поддержке жизнеспособной программы управления API заключается в работе с изменениями в масштабе, происходящими с течением времени. Как мы обсудили в предыдущем разделе, рост количества команд и созданных ими API может вызывать трудности. Процессы, необходимые для мониторинга API и управления ими в период выполнения программы, также будут меняться с развитием системы. Инструменты, которые нужны для мониторинга нескольких API, разработанных одной командой в одной географической локации, очень отличаются от инструментов, требующихся для отслеживания работы сотен или тысяч API с точками доступа, разбросанными по нескольким часовым поясам и странам.

В книге мы рассказываем об этом аспекте управления API как о системе. При расширении вашей программы необходимо будет следить за множеством процессов у множества команд во множестве локаций. Вы начнете больше опираться на контроль характера выполнения программы, чтобы понять, насколько жизнеспособна ваша система в каждый отдельно взятый момент. Во второй части этой книги (начиная с главы 8) мы рассмотрим, как представление об управлении API поможет понять, на каких элементах стоит сосредоточиться и какие инструменты и процессы нужны для контроля растущей платформы API.

Система API ставит перед вами новые задачи. Процессы, используемые для разработки, реализации и поддержки одного API, не всегда совпадают с теми, которые нужны при расширении экосистемы. Здесь работает закон больших чисел: чем больше в системе API, тем вероятнее их взаимодействие между собой. Это повышает вероятность того, что некоторые из этих взаимодействий приведут к неожиданным последствиям (или ошибкам). Так работают все крупные системы: больше взаимодействий — больше неожиданных результатов. Попытки избавиться от неожиданных результатов не приведут к решению проблемы. Удалить все баги невозможно.

Это ведет к еще одной сложности, с которой сталкивается большинство растущих программ с API: как уменьшить количество неожиданных изменений, придерживаясь соответствующего уровня стандартов?

Стандарты

Одно из ключевых изменений, происходящих при переходе с уровня API на уровень системы, — это изменение количества стандартов при непрерывном руководстве командами, разрабатывающими, реализующими и использующими API в вашей организации.

Увеличение количества групп, в том числе появление группы команд, ответственных за API организации, влечет за собой увеличение расходов на координацию их совместных действий. Увеличение масштаба требует изменений в объеме работ. Разобраться с этим будет проще, если больше полагаться на общие стандарты, а не на конкретные ограничения.

Например, одна из причин того, что Интернет так хорошо работает все время с момента создания в 1990 году, заключается в том, что разработчики уже на самом раннем этапе решили опираться на общие стандарты, применимые ко всем типам платформ программного обеспечения и языкам, а не создавать узкоспециальные инструкции по реализации, основанные на каком-то одном языке или платформе. Это позволяет креативным командам изобретать новые языки, архитектурные шаблоны и даже фреймворки среды исполнения, не нарушая работу существующих программ.

Все долгосрочные стандарты, которые помогали Интернету успешно функционировать, объединяет фокус на стандартизации *взаимодействия* между компонентами и системами. Вместо того чтобы стандартизировать то, как компоненты реализуются изнутри (например, используйте эту библиотеку, определенную модель данных и т. д.), веб-стандарты стремятся облегчить понимание сторонами друг друга. Когда ваша программа с API поднимается на более высокий уровень, указания вашему сообществу API также должны быть сосредоточены на более широких стандартах взаимодействия, а не на конкретных деталях реализации.

Этот переход может показаться непростым, но он необходим для того, чтобы получить жизнеспособную систему, где команды смогут разрабатывать API, которые способны легко взаимодействовать как с уже существующими в системе, так и с будущими API.

Управление системой API

Как было сказано в начале главы, двумя основными и наиболее сложными задачами в управлении API являются управление жизненным циклом одного API и управление системой всех API. Посещая различные компании и исследуя эту сферу в целом, мы видели много версий управления одним экземпляром API. Мы обнаружили много видов жизненных циклов и моделей развития, помогающих распознать сложности создания, разработки и применения API и справиться с ними. А указаний, относящихся к экосистемам API, получили гораздо меньше.

У систем API есть собственные сложности, свое поведение и тенденции. Информация, требующаяся при разработке одного API, отличается от той, о которой нужно помнить, когда вы должны поддерживать десятки, сотни или даже тысячи API. В экосистеме появляются проблемы с *масштабом*, которых не возникало в ходе работы с одной программой или реализацией API. Мы подробнее остановимся на

системе API позже, а здесь, в начале книги, хотим упомянуть три сферы, в управлении которыми проявляются уникальные для нее проблемы:

- расширение технологий;
- расширение команд;
- расширение руководства.

Рассмотрим каждый из этих аспектов управления API в отношении систем.

Технология

Когда вы впервые запускаете программу с API, требуется принять серию технических решений, которые повлияют на все ваши API. То, что «все API» в данном случае лишь небольшой набор, сейчас не имеет особого значения. Важно то, что у вас есть стабильный набор инструментов и технологий, на которые вы полагаетесь при разработке изначальной программы с API. Когда вы доберетесь до деталей жизненного цикла API (см. главу 6) и развития API, то увидите, что стоимость программ с API весьма высока и нужно внимательно отслеживать, сколько времени и энергии вы затратите на действия, которые значительно повлияют на успех вашего API, не рискуя большим капиталом на ранних стадиях. Обычно это означает, что нужно выбрать и поддерживать небольшой набор инструментов и предоставить очень четкие и детализированные документы с указаниями, чтобы помочь вашим специалистам создать и разработать те API, которые решат проблемы бизнеса и будут хорошо сочетаться друг с другом. Иными словами, на ранних этапах можно сэкономить, ограничивая технический объем.

На первых порах это хорошо работает по всем вышеупомянутым причинам. Но как только объем программы увеличивается и она расширяется (например, больше команд начинают разрабатывать больше API, чтобы обслужить больше сфер бизнеса в большем количестве мест и т. д.), появляются сложности. При росте программы с API привязка к ограниченному набору инструментов и технологий может значительно замедлить работу. Поначалу, при небольшом количестве команд, ограничение выбора ускоряло работу, но, когда команд много, ограничения — затратное и рисковое предприятие, особенно если вы организуете новые команды в географически отдаленных областях, или создаете новые подразделения бизнеса, или приобретаете новые компании, которые нужно включить в систему API. Здесь разнообразие становится гораздо более важным фактором на пути к успешному развитию вашей экосистемы.

Таким образом, в управлении технологиями системы API важно понять, когда она расширилась до размеров, позволяющих увеличивать разнообразие технологий, а не ограничивать их. Иногда это связано с уже применяемыми средствами. Если система API должна поддерживать существующие в организации сервисы SOAP-на-TCP/IP, нельзя требовать, чтобы все эти сервисы использовали указания по URL, которые вы составили для только что созданных API на CRUD-на-HTTP. То же самое относится

к созданию сервисов для событийно-ориентированных программ на Angular или унаследованному ПО для программ с удаленным вызовом процедур (RPC).

Больший объем обусловливает большее разнообразие технологий в вашей среде.

Команды

Технологии не единственный аспект управления API, который сталкивается с новыми задачами при росте программы. При изменении системы необходимо изменять и организацию самих команд. Опять же в начале работы над программой с API можно работать всего с несколькими специалистами, которые будут делать почти все. Это так называемые разработчики широкого профиля, или MEAN-разработчики (MongoDB, Express.js, Angular.js, Node.js). Сюда же относятся другие вариации на тему одного разработчика, чьи навыки подходят для всех аспектов программы. Наверное, вы часто слышите разговоры о командах-стартапах или самодостаточных командах. Все это сводится к наличию всех нужных вам навыков у членов одной команды.

Прибегать к услугам таких специалистов имеет смысл, когда у вас мало API и все они разрабатываются и реализуются с помощью одного набора инструментов. Но при увеличении масштаба и объема программы растет и количество навыков, необходимых для ее разработки и поддержки. Уже нельзя ожидать, что в каждой команде для работы с API будет нужное количество людей, имеющих навыки в областях разработки, баз данных, серверных и клиентских приложений, тестирования и развертывания программы. У вас может трудиться группа специалистов, чьей задачей будет разработать интерфейс панели управления, ориентированный на обработку данных, который станут применять другие команды. Они должны обладать такими навыками, как использование всех форматов данных и инструментов, необходимых для сбора данных. Или у вас может быть команда, призванная разрабатывать мобильные приложения, задействующие одну технологию, например, GraphQL или другую библиотеку, основанную на запросах. С ростом технологического разнообразия, возможно, стоит привлекать более специализированные команды. Мы исследуем этот вопрос подробнее в главе 7.

Другая сфера, в которой группам разработчиков придется измениться с ростом среды API, — это их участие в ежедневных процессах принятия решений. Когда у вас немного команд с небольшим опытом, может оказаться разумно централизовать принятие решений и делегировать его руководящей группе. В больших организациях она часто носит название группы архитектуры предприятия или подобное. Это работает при небольших масштабах и объемах, но оказывается серьезной проблемой, когда экосистема становится более разнообразной. Когда количество технологий растет, одна команда вряд ли сможет оказаться компетентной во всех деталях всех инструментов и фреймворков. Так что с добавлением новых команд должно быть перераспределено само принятие решений: центральный штаб редко разбирается в подробностях всех повседневных операций в международной компании.

Выход из положения заключается в том, чтобы раздробить процесс принятия решений на то, что мы называем элементами решения, и распределить их на соответствующих уровнях внутри компании. Рост экосистемы означает, что команды должны стать более специализированными на техническом уровне и более ответственными на уровне принятия решений.

Руководство

Последняя тема, которую мы хотим упомянуть, говоря о сложности системы API, — это общий подход к *руководству* программой. Как и в других упомянутых случаях, наши наблюдения показали, что при росте экосистемы роль и рычаги руководства меняются. Появляются новые задачи, и старые методы перестают быть эффективными. На самом деле, особенно на уровне предприятия, привязка к старым моделям руководства может замедлить достижение успеха вашими API или даже препятствовать этому процессу.

Как и в любой области лидерства, при небольших объеме и масштабе самым эффективным может стать подход, основанный на прямых указаниях. Обычно это работает не только с небольшими, но и с *новыми* командами. Когда их рабочий опыт невелик, быстрее всего обеспечить его приобретение, дав детальные указания и/или предоставив документацию по процессу. Например, мы обнаружили, что руководство программой с API на ранней стадии часто принимает форму длинного документа по процессу, объясняющего конкретные задачи: как создать URL для API, какие названия подходят для URL или где в заголовке HTTP должен находиться номер версии. Имея четкие указания по выполнению небольшого количества действий, разработчики почти не могут отклониться от утвержденного способа реализации API.

Но с ростом программы, при добавлении новых команд и поддержке большего количества сфер бизнеса становится все сложнее вести один документ с указаниями для всех команд. И даже если возможно перепоручить кому-то работу по написанию и поддержке детализированных документов, регламентирующих процессы на всем предприятии, это все равно не лучшая идея — как говорилось ранее, разнообразие технологий становится сильной стороной больших экосистем, и попытки сузить спектр применяемых технологий на уровне руководства предприятием могут замедлить прогресс программы.

Поэтому с расширением системы API необходимо изменить суть руководящих документов: это должны быть не прямые инструкции, а изложение общих принципов. Например, вместо детального описания того, какие URL считаются подходящими для вашей компании, можно указать разработчикам на руководство Инженерного совета Интернета по созданию и владению URL (RFC 7320) и привести общие принципы использования этого стандарта в организации. Другой прекрасный пример *принципиальных указаний* можно найти в большинстве руководств по UI/UX, например «10 удобных в применении эвристических алгоритмов для разработки пользовательского интерфейса» от Nielsen Norman Group (https://www.nngroup.com/

articles/ten-usability-heuristics/). В таких документах приводится четкое логическое обоснование необходимости использовать один шаблон UI, а не другой. Они разъясняют разработчикам, почему и когда что-либо применять, а не просто диктуют требования, которым нужно подчиняться.

Наконец, в очень больших организациях, особенно в компаниях, которые работают в нескольких местах и разных часовых поясах, надо изменить способ руководства с раздачи указаний на выработку рекомендаций. Это кардинально отличается от обычной модели централизованного руководства. Основной задачей центрального штаба руководства становится не строгое регламентирование того, что следует делать командам, а сбор рабочей информации в полевых условиях, поиск совпадений и передача указаний, которые отражают избранные методики всей организации.

Таким образом, при росте системы API модель руководства должна измениться с директивных указаний через общие принципы на сбор и передачу методик, используемых опытными командами, внутри компании. Как будет показано в главе 2, вы можете задействовать много различных принципов и методик для создания модели руководства, подходящей именно для вашей организации.

Выводы

В этой главе, открывающей книгу, мы затронули многие важные аспекты управления API, которые обсудим на ее страницах. Рассказали, что API продолжают оставаться движущей силой, однако лишь около 50 % опрошенных компаний уверены, что правильно ими управляют. Мы также пояснили, что у термина API много значений и это может усложнить создание стабильной модели руководства программой.

А важнее всего то, что управление одним API очень отличается от управления системой API. В первом случае вы можете полагаться на модели «API как продукт», «жизненный цикл API» и «развитие API». Управление изменениями в API также сосредоточено на понятии одного API. Но это далеко не все.

Затем мы обсудили управление системой API — целой экосистемой внутри вашей организации. Управление растущей системой API требует другого набора навыков и параметров — навыков работы с разнообразием, объемом, изменяемостью, уязвимостью и другими характеристиками. Все эти характеристики системы влияют на жизненный цикл API, и мы подробнее рассмотрим их позже.

Наконец, мы подчеркнули, что даже способ принятия решений по поводу вашей программы со временем придется менять. С ростом системы понадобится распределять принятие решений так же, как и элементы IT, например хранение данных, вычислительные ресурсы, средства безопасности и другие части инфраструктуры компании.

Запомнив это введение, сосредоточимся на понятии *руководства* и на том, как можно использовать принятие и распределение решений в качестве основного элемента общего подхода к управлению API.

2 Руководство API

> Эй, правила есть правила, и давайте признаем: без правил наступает хаос.
>
> *Космо Крамер*

Мало кто хочет, чтобы им руководили, — у большинства был отрицательный опыт работы под руководством тех, кто плохо планировал свои действия и устанавливал бессмысленные правила. Плохое руководство, как и плохое планирование, усложняет жизнь. Но наш опыт свидетельствует: тяжело говорить об управлении API, не упоминая руководство.

Мы даже позволим себе сказать, что управлять API без руководства *невозможно*.

Иногда в компаниях не используется термин «руководство». И это совершенно нормально. Названия имеют большое значение, а в некоторых организациях под руководством понимают стремление к чрезмерной централизации и авторитарности. Это может идти наперерез с корпоративной культурой, ценящей децентрализацию и самостоятельность сотрудников, поэтому логично, что в этом случае слово «руководство» — неподходящее. Неважно, как это называется, но даже при такой децентрализованной культуре производства так или иначе происходит принятие решений — конечно, скорее всего, оно радикально отличается от системы руководства в более традиционной, иерархически четко выстроенной компании.

Вопрос «Нужно ли руководить API?» кажется нам не очень интересным, потому что, по нашему мнению, ответ на него всегда утвердительный. Вместо этого спросите себя: «Какими решениями нужно руководить?» и «Где должно происходить это руководство?». В поиске ответов на такие вопросы и заключается работа по созданию системы руководства. От стиля руководства зависят показатели продуктивности, качество продукции и стратегическая ценность. Вы должны будете разработать систему, подходящую именно вам. Наша цель в этой главе — предоставить вам структурные элементы для этой разработки.

Начнем с обзора трех фундаментальных элементов хорошего руководства API: решений, управления ими и сложности. Разобравшись в них, рассмотрим поближе, как можно распределять решения в организации и как это влияет на ее работу. Таким образом, мы рассмотрим централизацию, децентрализацию и составные элементы решения. Наконец, поговорим о формировании системы руководства и исследуем три стиля руководства.

Руководство является центральной частью управления API, и понятия, представленные в этой главе, будут всплывать на протяжении всей книги. Поэтому стоит потратить время, чтобы понять, что на самом деле означает «руководство API» и как оно может помочь вам создать эффективную систему управления API.

Что такое руководство API

Работа с технологиями — это работа по принятию решений, множества решений. Одни из них жизненно важны, а другие — просто текучка. Именно из-за этого можно сказать, что работа технологической команды — *умственный труд*. Ключевой навык для работника умственного труда — принятие многих высококачественных решений одного за другим. Это довольно очевидно, но об этом часто забывают при управлении API.

Неважно, с какими технологиями вы работаете, как разрабатываете архитектуру и с какими компаниями решили сотрудничать, судьбу вашего бизнеса определяют способности к принятию решений у тех, кто в нем занят. Поэтому руководство имеет большое значение. Всем этим решениям нужно придать форму, которая поможет вам достичь целей, поставленных перед организацией.

Конечно, это проще сказать, чем сделать. Чтобы получить больше шансов на успех, вам понадобится понимание фундаментальных понятий руководства и их соотношения. Начнем с быстрого обзора темы решений в API.

Решения

Ваша работа и работа многих сотрудников организации состоит в основном из принятия решений. Поэтому руководство имеет такое значение. Если вы как группа можете принимать оптимальные решения, то есть лучшие из возможных, вы начнете показывать лучшие результаты. Но не забывайте, что эти решения касаются не только выбора технологий — в сфере API они могут быть самыми разными. Рассмотрите список вопросов, на которые может понадобиться ответить команде по API.

1. Какой URL будет у нашего API — `/payments` или `/PaymentCollection`?
2. В каком облачном сервисе следует разместить наш API?
3. У нас есть два API по клиентской информации. От которого из них отказаться?

4. Кто войдет в команду по разработке?
5. Как назвать эту переменную на Java?

На основании этого краткого списка вопросов можно сделать несколько выводов. Во-первых, выбор в управлении API может быть связан с широким спектром тем и его принятие потребует хорошей координации между отдельными людьми и командами. Во-вторых, индивидуальные решения, принимаемые сотрудниками, имеют разную степень воздействия — выбор облачного сервиса, скорее всего, повлияет на вашу стратегию управления API сильнее, чем название переменной на Java. В-третьих, при расширении масштаба небольшие решения могут стать более важными: если 10 000 переменных на Java названы неудачно, поддерживать реализации API будет гораздо сложнее.

Все эти решения в различных сферах, принимаемые сообща и с учетом масштаба, нужно собрать воедино для достижения лучшего результата. Это тяжелая и грязная работа. Позже в данной главе мы отдельно рассмотрим эту проблему и дадим некоторые указания по изменению системы решений. Но сначала поближе познакомимся с тем, что представляет собой *руководство* этими решениями и почему оно так важно.

Руководство решениями

Занимаясь небольшим проектом в одиночку, вы знали, что успех или провал этой работы зависит исключительно от вас. Если вы постоянно принимаете хорошие решения, это дает хорошие результаты. Один программист с отличными навыками может создавать потрясающие вещи. Но такой способ работы плохо масштабируется. Когда вашу продукцию начинают использовать другие люди, растет потребность в изменениях и новых характеристиках. Таким образом, вам понадобится принимать гораздо больше решений за меньший промежуток времени, то есть нужно больше сотрудников, которые будут этим заниматься. Расширение круга тех, кто принимает решения, требует осторожного подхода. Нельзя рисковать потерей качества решений только из-за того, что ими занимается больше людей.

Здесь и надо подключить руководство. *Руководство — это процесс управления принятием решений и их реализацией.* Заметьте, мы не утверждаем, что руководство связано с контролем или властью. Это не так. Оно связано с улучшением качества принятия решений вашими сотрудниками. В сфере API качественное руководство приводит к производству таких API, которые помогают вашей организации добиться успеха. Да, вам может понадобиться определенный уровень контроля и власти, чтобы достичь этого, но это не самоцель.

Вы можете руководить работой с API множеством разных способов. Например, можно ввести принцип, согласно которому все команды по API в компании должны использовать одну и ту же стандартную технологическую платформу. Или

установить правило, по которому все API перед запуском должны соответствовать набору стандартных параметров качества. Одна из этих политик жестче другой, но обе могут привести к одинаковым результатам. На практике вам придется управлять множеством различных типов решений одновременно, и ваша система руководства будет состоять из множества ограничений, поощрений, правил и процессов.

Имейте в виду, что у руководства всегда есть стимулы. Ограничения нужно донести до сотрудников, внедрить и поддерживать. Поощрения, влияющие на принятие решений, должны казаться им ценными и привлекательными. Стандарты и процессы должны быть документированы, мотивированы и актуальны. В придачу к этому нужно постоянно собирать информацию, чтобы наблюдать, как эти стимулы влияют на систему. Возможно, потребуется даже нанять больше сотрудников, чтобы заниматься руководством.

Кроме этих общих расходов на поддержание механизма, есть и скрытые убытки от применения руководства в вашей системе. Существуют первичные расходы, которые появляются, как только вы начинаете руководить системой. Например, если вы назначаете технологическую платформу, которую должны использовать все разработчики, какими окажутся организационные убытки из-за отсутствия инноваций? А сколько будет стоить недовольство сотрудников? Станет ли после этого труднее привлекать новые таланты?

Оказывается, такие убытки трудно предсказать. Это происходит потому, что в реальности вы руководите сложной системой, включающей в себя людей, процессы и технологии. Чтобы руководить системой API, сначала нужно научиться управлять сложной системой в целом.

Управление сложными системами

Хорошая новость: для получения хороших результатов не обязательно контролировать каждое решение в организации. Плохая новость: вам необходимо понять, какие решения все же требуют контроля. Данную проблему решить нелегко, и в книге вы не найдете однозначного ответа. Потому что невозможно дать ответ, который будет полностью соответствовать вашей конкретной среде и цели.

Если бы вы хотели испечь бисквитный торт, мы могли бы дать вам довольно четкий рецепт. Рассказали бы, сколько нужно муки и яиц, какую температуру устанавливать в духовке. Мы даже могли бы точно объяснить, как проверить готовность бисквита, потому что в современной выпечке очень мало вариативности. Ингредиенты одни и те же вне зависимости от места покупки. Духовки запрограммированы на выпекание при определенных стандартных температурах. А главное, одна и та же цель — конкретный вид торта.

Но вы не печете торты, а эта книга — не кулинарная. Так что придется разбираться с невероятным количеством вариантов. Например, у вас будут сотрудники с разным

уровнем способностей принимать решения. Регулирующие ограничения будут уникальными для вашей сферы деятельности и места размещения. Вам также предстоит обслуживать собственный динамично меняющийся клиентский рынок с его особой потребительской культурой. И в придачу ко всему этому ваши организационные цели и стратегии будут полностью уникальными.

Из-за разнообразия вариантов тяжело прописать один правильный «рецепт» руководства API. Осложняющим фактором является эффект неожиданных последствий. Каждый раз, когда вы вводите правило, создаете новый стандарт или применяете какую-то форму руководства, вам придется иметь дело с такими последствиями. Это потому, что все части организации переплетены и взаимосвязаны. Например, чтобы улучшить стабильность и качество кода своего API, вы вводите стандартную технологическую платформу. Это может привести к увеличению пакетов кода, потому что программисты начнут добавлять новые библиотеки и фреймворки. А это, в свою очередь, способно вызвать изменения в процессе развертывания, потому что существующая система не сможет поддерживать увеличившиеся пакеты кода.

Зная это заранее, возможно, вы могли бы предсказать и предотвратить такие последствия. Но это невозможно сделать для каждой непредвиденной ситуации, особенно за короткое время. Вместо этого нужно принять тот факт, что вы работаете со сложной адаптивной системой. И это не проблема, а просто особенность. И следует понять, как воспользоваться этим с выгодой для себя.

Сложная адаптивная система. Говоря, что ваша организация представляет собой сложную адаптивную систему, мы имеем в виду следующее.

- В ней много взаимозависимых элементов, например, люди, технологии, процессы, культура производства.
- Элементы могут менять свое поведение и адаптироваться к изменениям системы. Таково, например, изменение методик развертывания с вводом контейнеризации.

Во Вселенной существует множество подобных систем, и изучение их сложности уже стало признанной научной дисциплиной. Даже вы сами — сложная адаптивная система. Вы можете воспринимать себя цельной единицей — собой, но это всего лишь абстракция. На самом деле вы — скопление органических клеток, пусть даже способных на удивительные подвиги: мышление, движение, ощущения и реакцию на внешние события в качестве цельного существа. На клеточном уровне отдельные клетки специализированы, старые клетки отмирают и заменяются новыми, а группы клеток функционируют вместе. Сложность биологической системы, которой вы являетесь, обусловливает высокую устойчивость и приспособляемость вашего тела. Вы, скорее всего, не бессмертны, но очень вероятно, что адаптируетесь к глобальным переменам в окружающей среде

и даже выдержите некоторые физические повреждения благодаря сложности своей биологической системы.

Обычно, говоря о системах в технологиях, имеют в виду системы программного обеспечения и сетевую архитектуру. Такие системы определенно могут усложняться со временем. Например, Интернет — идеальный пример сложности и эмерджентности (интегральное свойство сложных систем, которое не присуще ни одному из компонентов системы в отдельности, а проявляется только в их совокупности). Сеть состоит из отдельных серверов, работающих независимо друг от друга, но благодаря их взаимосвязям появляется нечто целое, называемое Интернетом. Однако бóльшая часть программного обеспечения не *адаптивна*.

Программное обеспечение API, которое вы пишете сейчас, довольно бестолковое. Это не значит, что его код плохого качества или что оно не выполняет задачи, для которых разрабатывалось. Напротив, бóльшая часть выпускаемого вами API будет в точности выполнять свои задачи. В этом и проблема. Можно создать API, способный адаптироваться к меняющимся схемам трафика или растущему количеству ошибок, но непрактично было бы создавать API, который мог бы добавить новую функцию без вмешательства человека, или самостоятельно исправить сложный баг, или обновить свою документацию.

В будущем все это может измениться. Но в нынешней ситуации поведением системы программного обеспечения управляют ваши сотрудники. Плюс здесь в том, что люди очень хорошо адаптируются, особенно в сравнении с программным обеспечением. Ваша организация API является сложной адаптивной системой. Все люди в ней принимают множество локальных решений — какие-то из них коллективно, а какие-то — индивидуально. Когда таких решений много и принимаются они на протяжении длительного времени, система прогрессирует. Она, как и ваше тело, способна адаптироваться ко многим изменениям.

Но работа со сложной системой требует особого подхода. Влияние изменений на сложную систему трудно предсказать — изменения в одной ее части могут привести к неожиданным последствиям в другой. Это происходит потому, что сотрудники постоянно адаптируются к меняющейся среде. Например, установление правила, запрещающего реализацию программного обеспечения в контейнерах, может произвести сильнейший эффект, повлияв на разработку ПО, набор сотрудников, процессы реализации и культуру предприятия.

Все это означает, что желаемого нельзя достичь, просто внося значительные изменения и ожидая результатов. Вместо этого необходимо «подталкивать» систему небольшими изменениями и наблюдать за их влиянием. Для этого нужны постоянные улучшение и адаптация, так же как при уходе за садом следует обрезать ветки, сеять семена, поливать, постоянно наблюдая и подстраивая очередные свои действия под результат предыдущих. В главе 5 мы исследуем концепцию постоянного улучшения более детально.

Руководство решениями

В последнем разделе мы ввели понятие руководства решениями внутри сложной системы. Надеемся, что это помогло вам осознать его важнейшую роль в руководстве API: если вы хотите, чтобы система руководства была эффективной, необходимо лучше управлять решениями. Мы считаем, что один из лучших способов — сосредоточиться на том, где и кем принимаются решения. Оказывается, что одного оптимального способа распределить решения не существует. Например, рассмотрим вымышленный пример, как руководство дизайном API реализуется в двух разных компаниях.

- *Компания Pendant Software*. Всем командам, занятым API, предоставляется доступ к электронной книге *Pendant Guidelines for API Design* («Указания Pendant по разработке API»). Эти указания публикуются ежеквартально Центром по высокому качеству и передаче опыта в API компании Pendant — небольшой командой экспертов по API, работающих в данной компании. Они содержат предписывающие и очень четкие правила разработки API. От всех команд ожидается неукоснительное выполнение указаний, а API перед публикацией автоматически тестируются на соответствие им.

  В результате подобной политики компания Pendant смогла опубликовать ряд ведущих в своей области и очень согласованных API, которые были высоко оценены другими разработчиками. Эти API помогли Pendant выделиться среди конкурентов на рынке.

- *Компания Vandelay Insurance*. Командам по API сообщают цели их компании в бизнесе и просто ожидают результатов от создаваемых ими продуктов. Эти цели и результаты определяются исполнительными командами и регулярно обновляются. У каждой команды по API есть возможность достигать общей цели в бизнесе любым выбранным ею способом. Несколько команд могут выбрать одну и ту же цель. Команды по API могут разрабатывать и реализовывать API, как им угодно, но каждый продукт должен соответствовать параметрам и стандартам отслеживания компании Vandelay. Стандарты определяются Системным обществом Vandelay — группой, состоящей из участников каждой команды по API, которые вступают в нее добровольно и определяют набор стандартов, которому должны следовать все.

  В результате этой политики Vandelay смогла создать инновационную и подстраиваемую архитектуру API. Эта система API позволила компании опередить конкурентов с помощью инновационных бизнес-методик, которые на данной технологической платформе создаются очень быстро.

В вымышленном кейсе и Pendant, и Vandelay очень эффективно управляли принятием решений. Но руководили они этой работой совершенно по-разному. Компания Pendant пришла к успеху с помощью высокоцентрализованного авторитарного

подхода, а Vandelay предпочла метод ориентирования на результат. Ни один из этих подходов не является единственно правильным, у обоих стилей руководства есть свои преимущества.

Чтобы эффективно руководить решениями, необходимо ответить на три ключевых вопроса.

1. Какими решениями следует управлять?
2. Кто и где должен принимать эти решения?
3. Как стратегия управления решениями повлияет на систему?

Первый и третий вопросы мы рассмотрим чуть позже. Сейчас же сосредоточимся на втором: где следует принимать самые важные решения системы? Чтобы помочь разобраться в распределении решений, углубимся в тему руководства решениями. Мы затронем компромисс между централизованным и децентрализованным принятием решений и рассмотрим поближе, что такое передача решения.

Централизация и децентрализация

Ранее в этой главе мы ввели понятие сложной адаптивной системы и в качестве примера рассматривали тело человека. Природа изобилует такими системами, вы окружены ими повсюду. Например, об экосистеме какого-нибудь маленького пруда можно говорить как о сложной адаптивной системе. Она выживает благодаря действиям и взаимосвязям обитающих в ней животных и растений. Экосистема адаптируется к меняющимся условиям благодаря локализованному принятию решения каждым из живых организмов.

Но у пруда нет руководителя, и вряд ли лягушки, змеи и рыбы проводят квартальные совещания по управлению. Вместо этого каждый участник системы принимает собственные решения и демонстрирует свое поведение. Эти индивидуальные решения и действия формируют коллективное сложившееся целое, которое может выжить, даже если отдельные части системы меняются или исчезают и вновь появляются с течением времени. Как и бóльшая часть природных сообществ, система пруда выживает, потому что решения на уровне системы *децентрализованы* и *распределены между участниками*.

Как мы утверждали ранее, ваша организация также представляет собой сложную адаптивную систему. Она продукт всех индивидуальных решений, принятых ее сотрудниками. По аналогии с телом человека или экосистемой пруда, если вы позволите отдельным работникам иметь полную свободу и автономию, организация в целом станет более жизнеспособной и адаптивной. Вы получите децентрализованную компанию без начальника, которая сможет выжить благодаря индивидуальным решениям сотрудников (рис. 2.1).

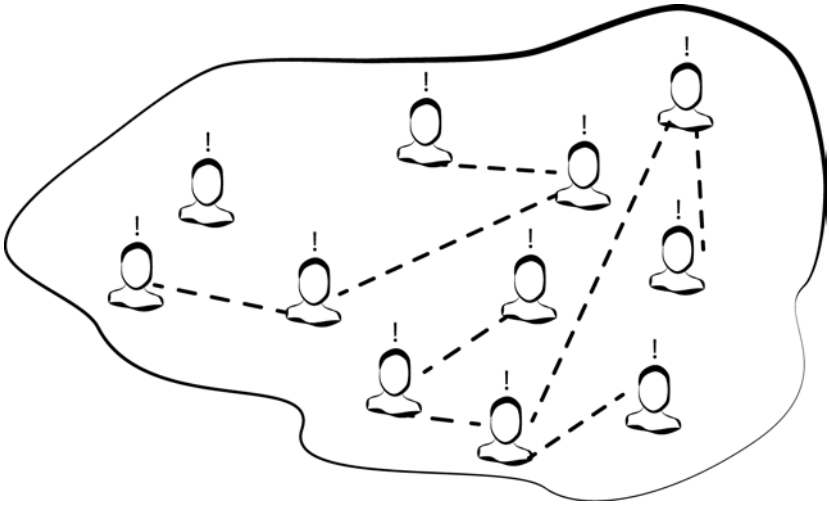


Рис. 2.1. Децентрализованная организация

Вы можете так поступить, но это способно вызвать проблемы, в первую очередь потому, что в свободных рыночных отношениях тяжело преуспеть так же, как преуспевают сложные системы в природе. В биосистеме пруда ведущую роль играет естественный отбор. Каждое звено в системе оптимизировано для выживания своего вида. Но у выживания вида нет цели на системном уровне. Кроме того, в природе системы могут гибнуть. Например, если запустить в пруд новый агрессивный вид живых организмов, вся прежняя система может вымереть. Для природы это нормально, потому что ее место займет что-то другое и система в целом останется жизнеспособной.

Однако руководители бизнеса вряд ли одобрят такой уровень вероятности и бесконтрольности. Скорее всего, у вашей системы есть известные вам цели, которые включают в себя не только выживание. Вы также вряд ли захотите допустить ликвидацию компании ради того, чтобы ее место заняла другая, более успешная. И практически точно вы хотите уменьшить риск краха компании из-за неудачного решения одного сотрудника. Таким образом, необходимо урезать свободу принятия индивидуальных решений и ввести какую-то отчетность. Одним из способов сделать это является *централизация* решений (рис. 2.2).

Здесь имеется в виду то, что в организации принятие решений поручено конкретным человеку или команде. Эта централизованная группа принимает решение, которое должны исполнять все остальные. Децентрализация означает обратное: отдельные команды могут принимать решения, которые обязаны выполнять только они.

В реальности идеально централизованных или децентрализованных организаций не существует. Разные типы решений распределяются в организации по-разному: одни более, другие менее централизованно. Вам придется решать, как распределять

решения, которые влияют на систему сильнее всего. Так какие из них должны быть более централизованными?

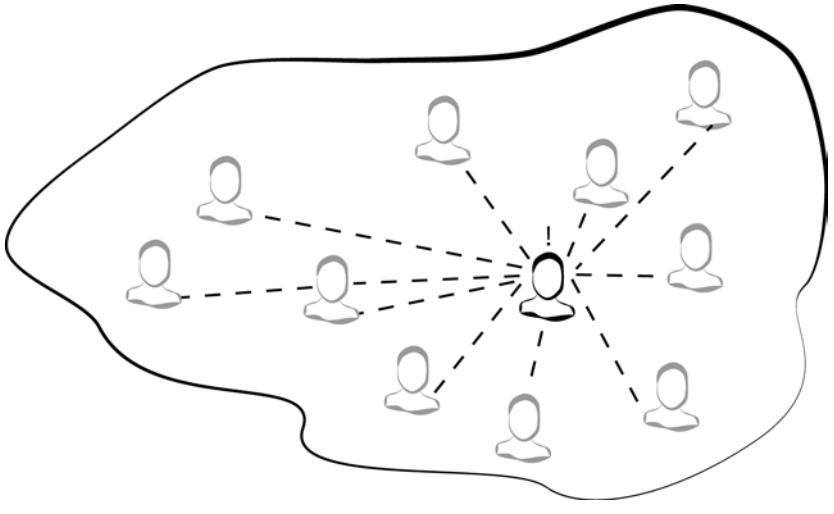


Рис. 2.2. Централизованная организация

Запомните: первоначальная цель руководства решениями — помочь организации достичь успеха и выжить. Что конкретно имеется в виду, зависит от специфики вашего бизнеса, но обычно это означает, что решения должны быть своевременными, чтобы бизнес был гибким, и качественными, чтобы он улучшался или как минимум не ухудшался. Вот три фактора, влияющие на способность принимать решения.

- *Доступность и точность информации.* Принять правильное решение очень трудно, если оно основано на неточной информации или она отсутствует. Вы можете не знать цели решения или сопутствующих факторов, а также быть не в курсе его будущего влияния на систему. Обычно мы предполагаем, что ответственность за сбор информации, важной для принятия решения, лежит на том, кто его принимает. Но для распределения решений мы должны учесть, как их централизация или децентрализация повлияет на доступность информации.
- *Талант в области принятия решений.* Обычно качество решений улучшается, если человек, принимающий их, делает это хорошо и качественно. Проще говоря, талантливые люди с большим опытом принимают решения лучше, чем менее талантливые и неопытные. При распределении решений важно рассредоточить ценных сотрудников так, чтобы они были на своем месте.
- *Затраты на координацию.* Нельзя своевременно принять сложное решение, если процесс принятия решений не будет коллективным. Но в этом случае придется учесть затраты на координацию. Если они слишком сильно растут, вы не сможете быстро принимать решения. Централизация и децентрализация решений могут очень серьезно влиять на эти затраты.

Если вы примете во внимание эти факторы, вам будет проще определить, когда решение должно быть централизовано, а когда — децентрализовано. Чтобы помочь разобраться с этим, рассмотрим решение с двух точек зрения: масштаба оптимизации и объема работы.

Масштаб оптимизации

Централизованное и децентрализованное решения значительно различаются по своему масштабу. Централизованное решение принимается для целой организации. Объем оптимизации в этом случае включает в себя всю систему, и вашей целью является принятие решения, которое ее улучшит. Другими словами, это решение должно *оптимизировать объем системы*. Например, централизованная команда может решить, какой методологии разработки будет придерживаться вся компания. Та же команда может постановить, какой API удалить из системы. Оба решения должны быть приняты с учетом выгоды для всей системы.

Основной характеристикой децентрализованного решения является то, что оно *оптимизировано для локального объема*. В этом случае вы принимаете решение, которое улучшит частности — информацию, которая относится только к данной конкретной ситуации. Хотя это решение может повлиять и на всю систему, но вашей целью является улучшение локальных результатов. Например, команда по API может принять решение использовать каскадный процесс разработки, потому что она работает совместно с внешней компанией, которая на этом настаивает.

В децентрализованном принятии решений прекрасно то, что оно помогает всему вашему бизнесу сильно выиграть в эффективности, инновациях и гибкости. Так происходит потому, что люди, принимающие решения децентрализованно, могут ограничить объем информации локальной ситуацией, которую они лучше понимают. Таким образом, они могут сформировать решение, опираясь на точные сведения о своей предметной области. Для любого современного бизнеса, пытающегося применять стратегию гибкости и инноваций, подходом, задействуемым по умолчанию, должна быть именно схема децентрализованного принятия решений.

Однако принятие решений, сосредоточенных только на оптимизации локального объема, может создать проблемы, в частности, если эти решения способны отрицательно и необратимо повлиять на систему. Когда генеральный директор Amazon Джефф Безос (http://bit.ly/2NxRswZ) говорил о влиянии решений, он разделил их на два типа: если оказались неправильными решения типа 1, их легко отменить, а после реализации неверных решений типа 2 восстановиться почти невозможно. Так, многие крупные компании предпочли централизовать решения по поводу конфигурации безопасности API, чтобы локальная оптимизация не привела к уязвимости в системе.

Кроме опасности для системы, есть и другие причины, по которым единство на системном уровне может быть ценнее локальной оптимизации. Например, отдельная команда по API может выбрать самый удобный для их предметной сферы стиль API. Но если у каждой команды будет свой стиль, станет гораздо сложнее использовать каждый API из-за недостатка единообразия, особенно когда для выполнения одного задания необходимо применять много API. В этом случае стоит оптимизировать решение по поводу стиля API на системном уровне.

Понадобится как следует обдумать объем оптимизации при планировании, когда будет принято решение. Если решение может необратимо повлиять на систему, начните с его централизации, чтобы оптимизировать его в объеме системы. Если качество решения улучшится благодаря локальным сопутствующим факторам или информации, начните с его децентрализации. Но если децентрализация решения может привести к недопустимому отсутствию единообразия на уровне системы, подумайте над его централизацией.

Объем работы

Если бы ресурсы для принятия качественных решений были неограниченными, вам нужно было бы думать только о масштабе. Но это не так. Поэтому, кроме масштаба, следует думать и об объеме принимаемых решений. Если их требуется много, будет высока нагрузка на людей, способных принимать решения, и еще более высокими окажутся затраты на координацию. Если вы хотите, чтобы API увеличивался соответственно росту организации, необходимо внимательно распланировать схему распределения решений.

Децентрализация принятия решений создает большой спрос на тех, кто способен это делать, если работать в больших объемах. Проводя децентрализацию, вы делаете принятие решений прерогативой не одной команды. Если хотите, чтобы все они были высококачественными, необходимо внедрить во все команды людей с даром в области принятия решений. Если вы не можете себе этого позволить, то количество неудачных результатов окажется велико. Поэтому стоит нанять лучших специалистов на каждую должность, на которой в вашей компании придется принимать решения.

К сожалению, хороших специалистов немного — это не секрет. За право нанять талантливых и опытных людей соревнуются многие компании. Некоторые из них готовы потратить любые деньги, чтобы точно получить самых лучших. Если вам повезло, вы сможете децентрализовать больше решений, потому что ими будут заниматься талантливые специалисты. Но если нет, придется подходить к распределению более прагматично.

Если у вас ограниченное количество высококвалифицированных специалистов, вы можете собрать их вместе и отдать самые важные решения на откуп этой группе

людей. Таким образом, будет высока вероятность выработки лучших решений более быстрыми темпами. Но эта модель перестанет работать, когда вырастет спрос на решения, потому что централизованной команде придется расти вместе с ним. С увеличением команды возрастут и затраты на координированное принятие решений. Независимо от таланта сотрудников затраты на координацию растут при добавлении новых людей в группу. И рано или поздно вы не сможете себе этого позволить.

Все это означает, что при распределении решений придется идти на компромиссы. Если решение может так сильно влиять на работу компании, как описанные Джеффом Безосом решения типа 1, вам придется его централизовать и заплатить за это снижением производительности. Напротив, если важнее всего скорость и локальная оптимизация, можно децентрализовать решение и либо заплатить за прием на работу лучших специалистов, либо смириться со снижением уровня качества решений.

Однако существует способ справиться с этой задачей более детально и гибко. Он заключается в распределении и передаче *частей* решения, а не решения в целом. Именно на нем мы и сосредоточимся в следующем разделе.

Элементы решения

Непросто распределять решения описанным ранее способом, потому что это отчасти сводится к подходу «все или ничего». Позволить командам самим решить, какой метод разработки они будут использовать, или выбрать самостоятельно и заставить все команды его применять? Позволить им решать, какой API устарел и подлежит удалению, или лишить их права выбора? В реальности руководство требует больше деталей. Далее мы исследуем способ более гибкого распределения решений путем разбиения их на части.

Вместо распределения решения целиком можно передавать командам его части. Таким образом реально получить преимущества оптимизации как на системном уровне, так и на локальном, когда каждая команда понимает контекст своего решения. Одни части решения могут быть централизованы, а другие — децентрализованы. Чтобы помочь вам точно распределять их, мы разбили решения по API на шесть *элементов* (рис. 2.3), таких как:

- зарождение идеи;
- поиск вариантов;
- выбор;
- одобрение;
- применение;
- проверка.

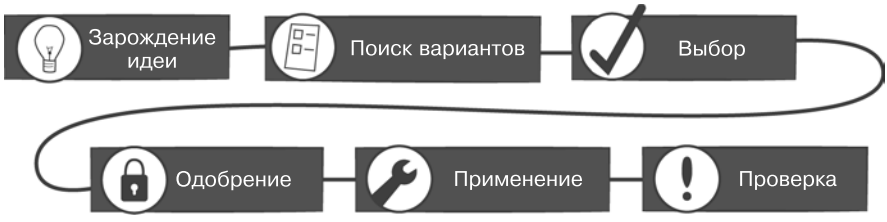


Рис. 2.3. Элементы решения

Эта модель не должна вас ограничивать и не может подойти для любого принятия решений. Мы разработали ее для того, чтобы выделить части решения, сильнее всего влияющие на систему вне зависимости от степени их централизации. Она основана на распространенных в сфере управления бизнесом моделях принятия решений из пяти, шести или семи шагов. Несмотря на то что описанные нами шаги могут применяться и к решению, принимаемому одним человеком, они наиболее актуальны при рассмотрении решений, принимаемых в группе.

Первым делом рассмотрим, как влияет на систему этап зарождения идеи.

Зарождение идеи

Каждое решение принимается потому, что кто-то считает, что это нужно сделать. Это означает: он определил, что проблему можно решить несколькими способами. Иногда это очевидно, но зачастую увидеть возможность принятия решения — дело, требующее таланта и опыта. Необходимо обдумать, какие решения возникают естественно, а какие требуют особого обращения.

Принятие решений по поводу работы с API происходит естественно в процессе ежедневного разрешения проблем. Например, выбор того, какую базу данных использовать для хранения постоянных данных, среднестатистическому разработчику было бы непросто игнорировать. Решение принимают, потому что без него нельзя продолжать работу. Но существуют ситуации, когда нужно инициировать зарождение идеи. Обычно это происходит по одной из двух причин.

- *Привычное принятие решений*. Если команда принимает одно и то же решение раз за разом, то с течением времени необходимость делать это может исчезнуть. Другие возможности более не рассматриваются, вместо этого появляется предположение, что работа будет продолжаться так же, как всегда. Например, если все реализации API написаны на языке программирования Java, никому может не прийти в голову выбрать другой язык.
- *Невнимательность к принятию решений*. Иногда команды упускают возможность принять решение, которое могло бы на что-то повлиять. Обычно это вызвано привычкой, но иногда нехваткой информации, опыта или таланта. Например, команда может сосредоточиться на выборе базы данных и не

понять, что API можно разработать так, что постоянное хранение не понадобится.

Не каждое решение необходимо принимать, и это нормально, если вы упустили возможность или по привычке считаете его действующим по умолчанию. Проблемой это становится только тогда, когда непринятие решения отрицательно влияет на результаты, получаемые от API. Если слепо требовать увеличения количества решений, это может отрицательно повлиять на продуктивность. Роль руководства API заключается в том, чтобы генерировать больше решений, которые дадут нужные результаты, и меньше тех, которые ни к чему не приведут.

Поиск вариантов

Трудно выбрать, если не знаешь, какие существуют варианты. Поиск вариантов — это работа по определению возможностей, из которых вы будете выбирать.

Если вы принимаете решение в сфере, в которой хорошо разбираетесь, поиск вариантов может стать довольно простой задачей. Но если неизвестных слишком много, придется потратить больше времени на определение возможностей. Например, у опытного программиста на C всегда есть хорошие варианты развития событий, когда принимается решение по циклической структуре, но начинающему, скорее всего, придется разобраться, где можно использовать оператор цикла `for` или `while` и какая между ними разница.

Даже если вы довольно хорошо разбираетесь в своей сфере деятельности, то, скорее всего, потратите больше времени на поиск вариантов, если значимость и влияние этого решения очень серьезны. Например, вы можете досконально разбираться в различных облачных хостингах, но все равно предпочтете перепроверить информацию, когда придет время подписывать договор с одним из них. Не появились ли новые предложения, о которых вы не знали? Точно ли вы запомнили цены и условия?

С точки зрения руководства поиск вариантов важен потому, что здесь устанавливаются *границы* принятия решений. Это особенно полезно, когда список вариантов разрабатывают одни, а выбор делают другие. Например, можно стандартизировать перечень возможных форматов описания API, но позволить отдельным командам решать, какой из них им больше нравится. При этом подходе необходимо быть уверенными в качестве «меню», которое вы предлагаете. Если список вариантов слишком ограниченный и не очень качественный, появятся проблемы.

Выбор

Выбор осуществляется из нескольких возможных вариантов. Это сердце принятия решений, и большинство людей фокусируется именно на данном шаге, но его важность зависит от количества доступных вариантов. Если оно очень велико, то процесс выбора сильно влияет на качество решения. Но если выбор был ограничен

безопасными вариантами, различия между которыми невелики, этот шаг может быть быстрым и почти ни на что не влиять.

Рассмотрим пример из практики. Представьте, что вы ответственны за конфигурацию безопасности транспортного уровня (TLS) для вашего API на HTTP. Задание включает в себя выбор набора шифров (наборов криптографических алгоритмов), который должен поддерживать сервер. Это важное решение, потому что некоторые наборы шифров со временем стали уязвимыми и неправильный выбор может снизить безопасность вашего API. Если вы выберете наборы шифров, которые не поддерживаются программным обеспечением пользователей, то никто не сможет воспользоваться вашим API.

Один сценарий развития ситуации предполагает, что вам дадут список всех известных наборов шифров и попросят выбрать те, которые должен поддерживать сервер. В этом случае к выбору нужно отнестись с особым вниманием. Вы, скорее всего, будете долго разбираться и почувствуете уверенность в выборе, только если соберете всю возможную информацию. Наверное, если у вас нет большого опыта в области безопасности серверов, вы найдете человека, у которого он есть, и попросите сделать выбор за вас.

А если вместо списка всех возможных наборов шифров вам дадут заранее отобранные? Список вариантов может включать также релевантную информацию о качестве поддержки каждого набора и его известных уязвимостях. Вооружившись этой информацией, вы, скорее всего, сможете сделать выбор быстрее. К тому же он явно будет безопаснее, потому что ваше решение ограничено вариантами, которые уже сочли достаточно безопасными для использования. В этом случае вы станете принимать решение, основываясь на том, что знаете о клиентах, работающих с вашим API, и его важности для бизнеса.

Наконец, вам могут дать только один вариант — один набор шифров, который вы должны применять. Решение с одним вариантом превращает выбор в формальность — оно уже было принято за вас. В таком случае качество решения полностью зависит от людей, которые нашли этот вариант. Будем надеяться, что он хорошо соответствует вашим требованиям.

Таким образом, важность выбора во многом зависит от количества предложенных вариантов. Здесь возможен некоторый компромисс. Если больше инвестировать в поиск вариантов, то будет потрачено меньше времени на выбор, и наоборот. Это влияет на распределение элементов решения и ответственности за них. Тот элемент, который становится более важным, потребует подходящего высококвалифицированного сотрудника для принятия решения.

Это также означает, что можно комбинировать системный и локальный масштаб, распределяя *поиск* и *выбор* вариантов. Например, можно централизовать поиск вариантов метода разработки, основанных на окружении системы, но при этом позволить отдельным командам выбрать себе метод с использованием локальных

сопутствующих факторов. Это исключительно полезная схема для управления растущими большими системами API и сохранения безопасности и скорости изменений.

Одобрение

Выбор варианта еще не означает принятия решения. Перед реализацией выбор должен быть одобрен. Одобрение — это работа по определению правильности выбранного варианта. Правилен ли этот выбор? Осуществим ли он? Безопасен ли? Имеет ли он смысл в контексте принятых ранее решений?

Одобрение может быть скрытым или явным. Явным называется такое одобрение, когда какой-то человек или команда должны прямо утвердить решение, прежде чем оно будет запущено в работу. Это шаг одобрения в процессе принятия решения. Наверняка вам часто доводилось принимать решения, которые требовали какого-либо одобрения. Например, во многих компаниях работники могут выбрать даты своих выходных из списка рабочих дней, но окончательное решение по утверждению расписания лежит на их руководителе.

Скрытое одобрение происходит автоматически, когда выбранный вариант соответствует определенному набору критериев. Примеры критериев: роль человека, делающего выбор, значимость выбранного варианта или соответствие конкретной стратегии. В частности, одобрение может стать скрытым, когда один и тот же человек и делает выбор, и одобряет его. То есть он одобряет сам себя.

Явное одобрение полезно, потому что в дальнейшем оно может улучшить безопасность решения. Но если принимается много решений и все они одобряются *из центра*, скорость этого процесса уменьшится. В итоге слишком много решений будут ждать одобрения. Скрытое одобрение резко увеличивает скорость принятия решений с помощью переноса ответственности на этап выбора, но в то же время оно сопряжено с большим риском.

Распределение этапа одобрения — важное решение в вашей схеме руководства. Необходимо принять во внимание качества сотрудников, принимающих решения, влияние неудачных решений на бизнес и степень риска действий в соответствии с предложенными вариантами. Очень важные решения, скорее всего, лучше одобрять явно. Для срочных решений большого масштаба необходимо ввести скрытую систему одобрения.

Применение

Процесс решения не заканчивается одобрением выбора. Решение не реализовано, пока кто-то не исполнит или не применит выбранный вариант. Применение — также важная часть работы по управлению API. Если решения воплощаются слишком медленно или некачественно, значит, весь процесс их принятия пошел насмарку.

Зачастую решение реализуется не теми людьми, которые делали выбор. В этих случаях важно понимать, как при этом собирать точную информацию. Например, вы можете решить ввести в свою систему API стиль гипермедиа, но если задействовать его слишком сложно для разработчиков, то решение стоит пересмотреть. При хорошей схеме руководства такие важные для практического использования вещи учитываются. Неверно управлять решениями, улучшая их только в *теории*. Оценивая качество решения, необходимо учитывать такой критерий, как его применимость на практике.

Тестирование

Решения не являются аксиомами, и каждое из тех, которые вы принимаете в системе управления API, должно быть готово к тестированию. Зачастую мы не принимаем во внимание, что наши решения могут быть в будущем пересмотрены, изменены или даже полностью отменены. Элемент проверки позволяет нам планировать непрерывные изменения на уровне принятия решений.

Например, если вы определили «меню» вариантов для команд по API, нелишним будет также выяснить, как его можно обойти. Таким образом вы сможете поддерживать высокий уровень инноваций и предотвратить принятие неверных решений. Но если все могут проверять на прочность решение ограничить их выбор, тогда это не ограничение. Поэтому необходимо определить, кто может проверять решение и при каких условиях.

Важно также позволить проверять выбор по прошествии некоторого времени. Поскольку бизнес-стратегии и окружение могут меняться, решения в вашей системе тоже должны пересматриваться. Чтобы запланировать такую приспособляемость, необходимо встроить в систему элемент тестирования. Это означает, что надо подумать над тем, кто в вашей организации сможет проверить уже существующее решение.

Планирование решений

Теперь мы знаем, что решения состоят из набора элементов. Понимание этого позволяет нам передавать отдельным командам части решения, а не весь процесс. Это важная характеристика организационной схемы, она позволяет вам сильнее влиять на баланс эффективности и основательности.

Например, очень важно решить, какой стиль должен иметь новый API. В неуклюжей бинарной схеме «централизация против децентрализации» разработчик схемы управления API должен понять, доверить ли решение по стилю членам команды по API (децентрализовать) или это решение должно контролировать руководство (централизовать). Плюс распределения власти над делегированием принятия решения командам по API в том, что каждая из них сможет принять решение

исходя из собственной ситуации. Преимущество централизации решения в одной стратегической команде — в уменьшении разнообразия в стилях API и контроле качества выбора стиля.

Это нелегкий компромисс. Но если вместо этого вы распределите *элементы* решения, станет возможно создать систему управления API, которая располагается где-то между двумя крайними вариантами. Например, можно решить, что для решения по стилю API элементы *исследования* и *поиска вариантов* будут в ведении централизованной команды по стратегии управления API, а элементы *выбора вариантов*, *одобрения* и *применения* — у самих команд по API. Таким образом, вы частично пожертвуете новизной, основанной на децентрализации выбора вариантов, в пользу преимуществ известного вам набора стилей API, используемого в компании. В то же время распределение элементов выбора стиля и одобрения позволяет командам по API работать максимально быстро, так как не нужно просить разрешения на выбор подходящего стиля.

Чтобы получить максимум преимуществ от планирования решений, необходимо распределять их, основываясь на общей ситуации и целях. Рассмотрим два довольно распространенных сценария, чтобы понять, как планирование решений может стать полезным инструментом.

Пример планирования решения: выбор языка программирования

Вы поняли, что решение по выбору языка программирования для реализации API будет очень важным, и предпочли бы руководить им. Ваша организация использует стиль архитектуры *«микросервисы»*, и обязательным требованием была свобода выбора языка программирования. Но после нескольких экспериментов вы заметили, что разнообразие в языках программирования усложняет для разработчиков переход из команды в команду, а группам по безопасности и разработке — поддержку приложений.

В итоге вы решили попробовать применить для выбора языка программирования распределение решений, показанное в табл. 2.1.

Таблица 2.1. План решения по языку программирования

Идея	Поиск вариантов	Выбор	Одобрение	Применение	Тестирование
Централизовать	Централизовать	Децентрализовать	Децентрализовать	Децентрализовать	Децентрализовать

Таким образом вы сужаете выбор языка программирования до набора вариантов, оптимизированных для вашей системы в целом, но позволяете индивидуальным командам оптимизировать его под конкретную ситуацию. Вы также позволяете командам по API тестировать решение, чтобы проверить новые варианты языков.

Пример планирования решения: выбор инструмента

Технический директор компании пытается улучшить уровень гибкости и инноваций вашей платформы программного обеспечения. Частью этой инициативы является решение позволить командам по API выбирать собственные платформы программного обеспечения, включая свободное ПО. Однако юридический отдел и отдел закупки обеспокоены этим из-за риска нарушения закона и ухудшения отношений с поставщиками. Чтобы начать вводить такие изменения, вы решили использовать для пробного выбора платформы ПО планирование решения, показанное в табл. 2.2.

Таблица 2.2. Планирование решения по выбору инструмента

Идея	Поиск вариантов	Выбор	Одобрение	Применение	Тестирование
Децентрализовать	Децентрализовать	Децентрализовать	Централизовать	Децентрализовать	Централизовать

Локальная оптимизация — один из ключей к стратегии, предложенной техническим директором, поэтому вы решили полностью децентрализовать идею, поиск вариантов и выбор. Но, чтобы уменьшить риск на уровне системы, предоставили возможность одобрения юридическому отделу и отделу закупки. Сейчас это должно сработать, но вас беспокоит, что по прошествии времени и с увеличением объема работ этот момент может стать узким местом системы, поэтому вы отметили, что необходимо наблюдать за ним и изменить его, если понадобится.

Разработка системы руководства

Мы потратили много времени на детали распределения решений, потому что считаем этот момент основным в системе руководства. Но это не единственное, на что нужно будет обратить внимание, если вы хотите эффективно руководить API. Хорошая система руководства API должна иметь следующие характеристики:

- распределение решений, основанное на влиянии на систему, объеме и масштабе;
- усиление системных ограничений и проверку качества (для централизованных решений);
- поощрение принятия решений (для децентрализованных решений);
- адаптивность за счет отслеживания уровня влияния и непрерывного улучшения.

Трудно получить преимущества от централизации решений, если вся остальная организация не поддерживает принятое решение. Поэтому одной из характеристик системы руководства API должны быть усиление ограничений и проверка качества. До сих пор мы намеренно обходили вопрос об авторитарных формах руководства, но в конце концов вам придется создать в своей системе хотя бы несколько огра-

ничений. Даже у самых децентрализованных организаций есть правила, которым нужно следовать. Конечно, проверка качества и усиление ограничений потребуют внедрения некоторого уровня подчинения. Если у централизованной команды, ответственной за принятие решений, нет власти, ее решения не будут иметь силы.

Если у вас не авторитарный стиль руководства, можете вместо ограничений использовать поощрения. Это особенно полезно, если вы решили децентрализовать решения, но при этом хотите контролировать их форму. Например, команда по архитектуре может так изменить процесс развертывания, что развертывание постоянных контейнерных классов станет намного дешевле и проще других типов развертывания. Целью здесь будет стимулирование команд по API, которые руководят собственными решениями по реализации, чаще делать выбор в пользу контейнеризации.

На самом деле ни пряник поощрения, ни кнут ограничений не могут полностью определять вашу систему — необходимо эффективно задействовать и то и другое. В общем, если элемент одобрения решения был децентрализован и вы хотите его контролировать, надо использовать поощрение. Если выбор и одобрение были централизованы, а применение — децентрализовано, необходимо убедиться, что введены определенный уровень ограничений или проверка качества. Таблица 2.3 показывает, когда нужно ограничивать или поощрять решение, основываясь на схеме его планирования.

Таблица 2.3. Когда ограничивать и поощрять?

Ограничивать или поощрять?	**Идея**	**Поиск вариантов**	**Выбор**	**Одобрение**	**Применение**	**Тестирование**
Ограничивать		Централизован	Централизован или децентрализован	Централизовано или децентрализовано		
Поощрять		Децентрализован	Децентрализован	Децентрализовано		

Независимо от того, как вы распределяете решения или меняете стиль их принятия, крайне важно определить, какое влияние вы имеете на саму систему. В идеале у компании должны быть какие-то показатели и параметры процесса, которые можно использовать для оценки влияния этих изменений. Если их нет, одним из основных приоритетов должно быть внедрение таких параметров на уровне организации. Позже, в главе 6, мы поговорим об измерениях (metrics) в системе API. Хотя мы сосредоточимся конкретно на параметрах продуктов с API, этот раздел можно использовать как пособие по разработке параметров руководства для вашей системы.

Чтобы связать вместе все сказанное ранее, рассмотрим три схемы руководства API. Они включают в себя разные подходы к руководству, но все придерживаются главных принципов: распределение решений, ограничения, поощрение и оценка показателей. Имейте в виду, что мы не предлагаем вам меню — вы не обязаны выбирать для себя одну из этих систем руководства. Мы приводим их в качестве

иллюстрации того, как на концептуальном уровне может быть реализована система руководства API.

Для каждой из описанных схем руководства определим несколько ключевых решений и способов планирования, а также то, как ограничиваются и поощряются нужные поступки, как распределяются особо ценные сотрудники, а также каковы расходы, доходы и параметры каждого подхода.

Схема руководства 1: контроль интерфейса

Здесь акцент сделан на важности для API модели интерфейса. Контроль (supervision) интерфейса централизует все решения, связанные с его разработкой, чтобы убедиться, что все интерфейсы согласованы, безопасны и удобны для использования (табл. 2.4).

Таблица 2.4. Планирование решений

Сфера решений	Идея	Поиск вариантов	Выбор	Одобрение	Применение	Тестирование
Разработка API	Централизована	Централизован	Децентрализован	Централизовано	Децентрализовано	Децентрализовано
Реализация API	Децентрализована	Децентрализован	Децентрализован	Децентрализовано	Децентрализовано	Централизовано

- *Ограничения и поощрение*. Реализация и развертывание API рассматриваются централизованной командой по разработке интерфейса. Хотя у команд есть возможность принимать собственные решения по реализации и развертыванию, центральная команда может отметить и удалить API, если он не соответствует заданной модели интерфейса.
- *Распределение кадров*. Профессионалы в области разработки собраны в центральной команде, а специалисты в области программирования и реализации могут быть децентрализованы.
- *Расходы и доходы*. Отделение друг от друга команд по разработке и реализации означает, что есть риск создания дизайна, который сложно или дорого реализовывать. Но имеются и преимущества — у команды разработки интерфейса есть перспектива «чистого дизайна», поэтому она может выполнить дизайн, больше сконцентрированный на пользователе. При увеличении масштаба вырастет риск появления узкого места из-за нехватки ресурсов у централизованной команды по разработке. Это может быть особенно проблематично, когда потребуется внести небольшие изменения во многие интерфейсы.
- *Необходимые измерения*:
  - оценка удобства использования API;
  - мониторинг реализации плана выполнения проекта;
  - контроль разработки и управления программой.

Схема руководства 2: автоматизированное руководство

Такое руководство задействует механизм стандартизации и автоматизации для ограничения принятия решений. В этой схеме централизованная команда пытается максимально увеличить контроль над системой с помощью автоматики, уменьшая влияние на количество принятых решений. Это происходит благодаря централизации пространства решений по работе API (то есть только элемента поиска вариантов). У команд есть возможность принимать любые решения при условии, что те соответствуют стандартным вариантам (табл. 2.5).

Таблица 2.5. Планирование решений

Сфера решений	Идея	Поиск вариантов	Выбор	Одобрение	Применение	Тестирование
Разработка API	Децентрализована	Централизован	Децентрализован	Централизовано	Децентрализовано	Децентрализовано
Реализация API	Децентрализована	Централизован	Децентрализован	Децентрализовано	Децентрализовано	Децентрализовано
Развертывание API	Децентрализована	Централизован	Децентрализован	Децентрализовано	Децентрализовано	Децентрализовано

- *Ограничения и поощрение*. Поскольку выбранные варианты применяются стандартизированно, все аспекты разработки, реализации и развертывания могут быть проверены автоматически с помощью инструментальных средств. Например, команды по API должны документировать дизайн интерфейсов на языке, который может прочесть автоматика, что проверяется с помощью инструмента «контроль качества кода».
- *Распределение кадров*. В центральную команду должны быть набраны опытные дизайнеры, разработчики архитектуры, специалисты по внедрению, чтобы они могли убедиться в том, что выбраны лучшие варианты. Если централизованные кадры выбраны комплексно и являются высококвалифицированными, в децентрализованные команды требуется меньше талантливых разработчиков.
- *Расходы и доходы*. Автоматику всегда дорого разрабатывать, создавать, поддерживать и настраивать. Потребуются крупные начальные инвестиции для формирования лучшего набора стандартов для этого типа системы, и возникнет постоянная проблема с поддержанием актуальности вариантов и инструментов. Но эти расходы компенсируются тем, что нужно меньше распределять решения, а благодаря автоматизации выработка решений увеличивается. Возможное влияние такой схемы на систему — недовольство в командах по API из-за отсутствия свободы: если варианты слишком регламентированы, будет трудно привлечь хороших сотрудников.

- *Необходимые измерения*:
  - мониторинг реализации плана выполнения проекта;
  - оценка популярности того или иного варианта (отслеживание того, когда и как используются стандартизированные варианты);
  - оценка эффективности команд по API.

Схема руководства 3: совместное руководство

В схеме совместного руководства решения по API принимаются индивидуально, но при этом общее понимание влияния на систему формируется совместно. Цель заключается в создании «общего мозга» в смысле образа на системном уровне при сохранении скорости и объема децентрализованной системы (табл. 2.6).

Таблица 2.6. Планирование решений

Сфера решений	Идея	Поиск вариантов	Выбор	Одобрение	Применение	Тестирование
Разработка API	Централизована	Децентрализован	Децентрализован	Централизовано	Децентрализовано	Децентрализовано
Реализация API	Децентрализована	Децентрализован	Децентрализован	Децентрализовано	Децентрализовано	Децентрализовано
Развертывание API	Децентрализована	Децентрализован	Децентрализован	Децентрализовано	Децентрализовано	Децентрализовано
Измерение API	Централизована	Централизован	Централизован	Централизовано	Децентрализовано	Децентрализовано

- *Ограничения и поощрение*. При совместном руководстве большинство решений полностью децентрализованы, за исключением идеи API и измерений. Так создается ориентированный на результаты образ API в системе. Следовательно, ограничения также полностью ориентированы на результат: если API не достигает ожидаемого результата, его удаляют, а команду могут распустить. Решения по разработке, реализации и развертыванию децентрализованы, однако на них обычно можно повлиять с помощью поощрения. Например, если решения, принятые командой, ведут к удачным результатам и ими делятся с организацией, команда может получить денежную премию. Сочетание премии и прозрачности может повлиять на решения других команд, функционирующих в компании.

  Поскольку бóльшая часть работы децентрализована, нужно поощрять сотрудничество между командами. Поощрять его (или принуждать к нему) необходимо на системном уровне.

- *Распределение кадров*. Совместное руководство требует множества талантливых сотрудников. Такой уровень децентрализации требует соответствующего количества одаренных людей, распределенных по командам. Это не означает, что все сотрудники должны быть звездами, но в каждой команде должно быть достаточно профессионалов, чтобы все решения оказывались безопасными и эффективными.
- *Затраты и доходы*. Децентрализованные команды, обладающие отличными навыками, могут создавать высококачественные инновационные API. Основными затратами для достижения этого результата будут расходы на талантливых работников и поддержку сотрудничества. При увеличении масштаба работы увеличатся и они.
- *Необходимые измерения*:
  - отслеживание разработки продукта с API;
  - оценка эффективности команд по API;
  - оценка удобства использования API.

Выводы

В этой главе вы познакомились с данным нами определением руководства: это управление принятием и реализацией решений. После этого мы более подробно рассмотрели, что значит принять решение и руководить им. Вы узнали, что решения в API могут быть незначительными («Какой будет следующая строчка моего кода?») или важными («С каким поставщиком нам сотрудничать?») и сильно различаются по объему. И что самое важное, вы узнали, что система, которой вы пытаетесь руководить, относится к *сложным адаптивным системам*, поэтому трудно заранее предсказать результаты применения любой стратегии управления решениями.

Далее мы рассмотрели распределение решений, сравнили централизацию с децентрализацией, оценили их плюсы и минусы. Чтобы помочь вам понять, в чем заключается разница между ними, мы сравнили их по *масштабу оптимизации* и *объему работы*. Затем обсудили, как можно раздробить решения на ключевые элементы: идею, поиск вариантов, выбор, одобрение, применение и тестирование. Объединив все эти понятия, а также используя ограничения и поощрения, можно построить эффективную систему руководства API.

Руководство — это сердце управления API, поэтому неудивительно, что это основное понятие всей книги. Нашей целью в этой главе было ввести основные понятия и рассказать о рычагах управления. Далее мы углубимся в сферу руководства API, затрагивая специфические проблемы, в которых решения имеют наибольшее значение. Расскажем, как управлять сотрудниками и что делать, когда API развивается и его масштаб растет. В следующей главе мы начнем этот путь с исследования того, как мышление с точки зрения продукта поможет вам определить самые важные решения в работе с API.

3 API как продукт

> Если вы создаете нечто потрясающее, клиенты будут рассказывать друг другу об этом. Сарафанное радио — очень мощное средство.
>
> *Джефф Безос, основатель и генеральный директор Amazon*

Фразу «API как продукт» (AaaP) часто произносят в разговоре о компаниях, создавших и поддерживающих успешные программы с API. Это перефразированное выражение «...как услуга», которое часто используют в технических кругах («программное обеспечение как услуга», «платформа как услуга» и т. д.). Обычно оно выражает важный взгляд на разработку, применение и реализацию API, который заключается в том, что API — это продукт, полностью заслуживающий продуманного дизайна, создания прототипов, исследования клиентской базы и тестирования, а также длительного контроля и технической поддержки. Обычно эта фраза означает: «Мы относимся к нашим API так же, как к любым другим предлагаемым нами продуктам».

В этой главе мы исследуем подход AaaP и его применение для улучшения эффективности разработки и реализации API, а также управления ими. Как вы могли понять из главы 2, этот подход включает в себя осознание того, какие решения важны для успеха ваших API и в каком отделе организации их должны принимать. Он поможет вам обдумать, какую работу надо централизовать, а какую можно успешно децентрализовать, где лучше всего задействовать ограничения и поощрения и как измерить степень влияния этих решений, чтобы при необходимости быстро адаптировать ваши продукты (API).

При создании новой продукции для клиентов приходится принимать множество решений. Неважно, разрабатываете ли вы плееры, ноутбуки или API для создания очереди сообщений. Во всех этих случаях вам нужно знать свою целевую аудиторию, понимать и решать ее самые животрепещущие проблемы и обращать внимание на

предложения клиентов об улучшении продукции. Эти три задачи можно сформулировать в виде трех важных тем, на которых мы сосредоточимся в этой главе.

- Дизайн-мышление, необходимое для того, чтобы знать свою целевую аудиторию и понимать ее проблемы.
- Поддержка новых клиентов как способ быстро продемонстрировать им, как успешно пользоваться вашим продуктом.
- Опыт разработчика, требуемый для управления жизненным циклом продукта после его выпуска и для выработки идей будущих модификаций.

Вскоре мы узнаем о возможностях дизайн-мышления и поддержки клиентов на примере таких компаний, как Apple. Увидим, как Джефф Безос помог отделу интернет-сервисов Amazon (AWS) создать устав реализации, обеспечивающий получение четкого и предсказуемого опыта разработчика. В большинстве компаний, со специалистами которых мы общались, понимают, что такое AaaP, но не везде могут применить это понимание на практике. Однако во всех организациях, на счету которых немало успешных продуктов с API, поняли, как разобраться с упомянутыми тремя важными задачами, и первая из них связана с тем, как ваши команды обдумывают то, что создают.

Дизайн-мышление

В кругах разработчиков компания Apple известна, в частности, благодаря своей способности задействовать *дизайн-мышление*. Например, описывая работу по созданию Apple Mac OS X, один из ключевых разработчиков ПО Корделл Ратцлафф сказал: «Мы сосредоточились на том, что, по нашему мнению, было нужно людям, и на том, как они взаимодействуют с компьютером» (http://bit.ly/2IPvqVQ). И этот подход затем отразился на практике. «Для создания идеи продукта были необходимы три отчета: о требованиях маркетинга, о технических требованиях и об опыте пользователей», — объяснил бывший вице-президент Apple и один из первопроходцев в области разработки взаимодействия человека и компьютера Дональд Норман (http://bit.ly/2Pk0LFI).

Такое внимание к потребностям пользователей привело Apple к созданию стабильного бизнеса. Непрерывная цепочка продуктов, выпускаемых в течение нескольких десятилетий, создала им репутацию компании, определяющей новые тенденции в технологиях, и помогла неоднократно занимать большой сегмент рынка.

Тим Браун, генеральный директор IDEO — расположенной в Калифорнии фирмы по разработке и консультированию, определяет термин «дизайн-мышление» следующим образом (http://bit.ly/2OEP6St): «Раздел дизайна, использующий чутье и методы разработчика для того, чтобы продукт соответствовал требованиям потребителей и был технологически осуществимым, а конкурентная бизнес-стратегия могла извлечь из него потребительскую ценность и возможность для рынка».

В этом определении необходимо пояснить несколько пунктов. Сосредоточимся на понятиях «соответствие требованиям потребителей» и «конкурентная бизнес-стратегия».

Соответствие требованиям потребителей

Одна из целей разработки API — соответствие требованиям потребителей, то есть решение их проблем. Поиск проблем, требующих решения, и определение их приоритетности является частью задачи подхода AaaP — это ответ на вопрос «Что делать?» в области API. Еще более фундаментальный элемент — ответ на вопрос «Для кого?». Кому вы облегчите жизнь с помощью этого API? Правильное определение целевой аудитории (ЦА) и ее проблем — первый шаг на долгом пути к тому, чтобы убедиться, что вы создаете нужный продукт: он эффективно работает и часто используется ЦА.

Клейтон Кристенсен из Гарвардской бизнес-школы называет процесс осознания потребностей вашей ЦА теорией «работы, которую нужно сделать» (Jobs to Be Done, http://bit.ly/2EFxUaV). Он говорит: «Люди не просто покупают товары или услуги, они "нанимают" их, чтобы продвинуться в определенных обстоятельствах». Люди (ваши потребители) хотят продвинуться (решить проблемы), и они будут применять (или арендовать) те продукты и услуги, которые, по их мнению, помогут им этого добиться.

СТОИТ ЛИ ИСПОЛЬЗОВАТЬ AAAP И ДЛЯ ВНУТРЕННИХ, И ДЛЯ ВНЕШНИХ API

Да. Может быть, следует вкладывать в них разное количество времени и ресурсов — это мы обсудим в следующем разделе, — но это одна из тех вещей, которым учит Джефф Безос в «Уставе Безоса», позволившего Amazon открыть изначально внутреннюю платформу AWS для использования в качестве приносящего доход внешнего API. Поскольку Amazon с самого начала применял подход AaaP, предложить внешним пользователям внутренний API было не только возможно (то есть безопасно), но и выгодно.

В большинстве компаний отдел IT помогает решать проблемы. Обычно их клиенты — сотрудники той же компании (внутренние разработчики). Иногда это важные партнеры по бизнесу или даже анонимные разработчики сторонних приложений (внешние разработчики). Каждая из этих групп разработчиков (внутренняя, партнерская и внешняя) сталкивается со своим набором проблем и по-своему обдумывает и решает их. Планирование дизайна поощряет команды к тому, чтобы узнать свою ЦА перед тем, как начать создавать API в качестве решения проблемы. Мы исследуем эту тему далее в разделе «Познакомьтесь с целевой аудиторией».

Конкурентная бизнес-стратегия

Другой важной частью планирования дизайна является определение *конкурентной бизнес-стратегии* вашего продукта с API. Бессмысленно вкладывать много денег в продукт, который не окупится, и тратить на него много времени. Даже когда вы стараетесь разработать правильный продукт для правильной ЦА, необходимо убедиться, что потратили должное количество времени и денег и у вас есть четкое представление, каким будет доход после запуска API.

У большинства компаний ограничено количество времени, денег и энергии, которое можно выделить на создание API для решения проблем. Таким образом, огромное значение приобретает выбор этих проблем. Иногда мы сталкиваемся с компаниями, где API не решают важных для бизнеса задач. Вместо этого они решают хорошо известные проблемы отдела IT, например открытие таблиц данных или автоматизацию внутренних процессов отдела. Обычно все это важно, но, возможно, такие решения не очень сильно влияют на ежедневные бизнес-операции и не приближают компанию к достижению ежегодных целей в сфере продаж или производства.

Непросто понять, какие проблемы значимы для вашего бизнеса. Может быть сложно сформулировать цели компании так, чтобы они были понятны сотрудникам IT-отдела. И даже когда они поняли, какие проблемы важны для компании, у них может не оказаться подходящих методик измерения, чтобы подтверждать свои предположения и отслеживать прогресс. Поэтому важно наладить стандартизированный способ донести до них ключевые задачи бизнеса и обеспечить наличие релевантных показателей прогресса в работе. Этот аспект пути к успеху вашего API мы обсудим подробнее в разделе «Измерения и ключевые моменты» главы 6.

Устав Безоса

Запускать успешную программу с API — ту, которая способна преобразить вашу компанию, — непросто, независимо от возраста организации. Одна из самых уважаемых компаний, проработавших этот процесс и не прекращающих преобразования даже десять лет спустя, — это Amazon, создавшая платформу AWS. Разработанная в начале 2000-х, эта платформа считается шедевром, виртуозно исполненным изобретательной командой специалистов в области IT и бизнеса. Хотя платформа AWS сейчас имеет огромный успех, она была создана из-за внутренней необходимости — большого недовольства тем, что IT-программы Amazon тратили слишком много времени на обработку и доставку запросов бизнес-команды. Команда AWS работала слишком медленно, и вначале продукт не соответствовал требованиям и на техническом (объем работы), и на деловом (качество продукта) уровне.

Как рассказывает нынешний генеральный директор AWS Энди Джасси (https://tcrn.ch/2R5dh9n), команда AWS, а также генеральный директор Amazon Джефф Безос

и другие сотрудники потратили немало времени на определение того, что у них получается удачно и что потребуется для разработки и создания базового набора общедоступных сервисов на межоперационной платформе. На разработку плана ушло более трех лет, но в итоге он лег в основу знаменитой новой платформы, действующей по принципу «инфраструктура как услуга» (IaaS). Бизнес, который принес своим разработчикам 17 млрд долларов, был создан только благодаря вниманию к деталям и неустанным попыткам улучшить изначальную идею. Примерно так же, как Apple изменила представление покупателей о переносных устройствах, AWS изменила представление бизнеса о серверах и другой инфраструктуре.

AWS смогла изменить мнение об API *внутри* компании во многом благодаря тому, что сейчас известно как «Устав Безоса». Стив Йегги, бывший руководитель разработки программного обеспечения в Amazon, описал этот устав в статье «Тирада о Google-платформах» (http://bit.ly/2OI2c1c) в 2005 году. Один из важнейших пунктов этой записи из блога состоит в том, что Безос выпустил устав, по которому все команды должны раскрывать свой функционал с помощью API и пользоваться функционалом других команд тоже через API. Иными словами, что-то делать можно только через API. Он также потребовал, чтобы все API были созданы и разработаны *так, словно они будут представлены за пределами компании*. Мысль о том, что «API должны иметь возможность экстернализации», была еще одним важным требованием, повлиявшим на создание и разработку API, а также управление им.

Таким образом, дизайн-мышление включает в себя способность учитывать потребности вашей целевой аудитории и конкурентную бизнес-стратегию, принимая решение о том, на какие API стоит потратить ограниченные ресурсы и внимание. Как это выглядит на практике? Как можно применить знания о продукте к процессу управления API, чтобы он соответствовал подходу «API как продукт»?

Применение дизайн-мышления к API

Вы можете поднять статус API со вспомогательных систем до продуктов, используя в процессе их создания и разработки дизайн-мышление[1]. Так поступают несколько компаний, с сотрудниками которых мы общались в последние несколько лет. Они приняли решение, что их API, включая предназначенные для внутреннего пользования, заслуживают того же уровня внимания, изучения и разумного дизайна, как и любые другие их продукт или услуга. Для некоторых компаний это означает, что они должны обучить своих разработчиков API и других сотрудников IT-отдела принципам дизайн-мышления. Для других это создание внутри организации мостика между группами, занимающимися дизайном продукта и API. В нескольких компаниях, с которыми мы работали, наблюдались сразу оба этих процесса: обучение разработчиков дизайн-мышлению и усиление связи между командой продукта и разработчиками.

[1] Советуем почитать: *Леврик М., Линк П., Лейфер Л.* Дизайн-мышление. От инсайта к новым продуктам и рынкам. — СПб.: Питер, 2020. — *Примеч. ред.*

Полное содержание программы по дизайн-мышлению не уместится в эту книгу. Однако большая часть курсов предлагает комбинацию тем, уже упомянутых в данной главе, например:

- навыки дизайн-мышления;
- понимание нужд потребителя;
- разработка сервиса/потока работ;
- создание прототипов и тестирование;
- факторы, которые необходимо учитывать в бизнесе;
- измерение и оценка.

Если у вас в компании уже есть сотрудники, занимающиеся дизайном продукта, они могут обучить команды разработчиков тому, как начать мыслить и действовать, как они. А если таких сотрудников нет, обычно можно найти курсы по дизайну продукта в местном колледже или университете. Многие из этих учреждений могут предложить адаптировать курс для вашей компании и провести его у вас. Наконец, даже если вы просто сотрудник небольшой организации, интересующийся этой темой, можете найти онлайн-курсы по дизайн-мышлению сами.

Одна из компаний, с сотрудниками которой мы беседовали (большой клиентский банк), решила создать собственный курс по дизайн-мышлению, и сотрудники отдела дизайна продукта проводили занятия для команд по API в различных отделениях компании. Впоследствии команды по API могли обращаться к своим наставникам за советом по улучшению дизайна API. Их целью не было превращение всех разработчиков и архитекторов программного обеспечения в профессиональных дизайнеров. Они стремились к тому, чтобы команды по API лучше понимали процесс дизайна и научились применять эти навыки в работе.

Важно помнить, что результатами дизайн-мышления являются не только удобство в использовании или улучшенный внешний вид ваших API, но и лучшее понимание целевой аудитории (клиентов), создание API, соответствующих конкурентной бизнес-стратегии, и более надежный процесс измерения относительного успеха API, которые ваша команда развертывает в экосистеме.

Как бы ни был важен дизайн в подходе AaaP, это только начало. Необходимо также обратить внимание на первые впечатления пользователей после реализации API и открытия доступа к нему. Об этом поговорим в следующем разделе.

Поддержка новых клиентов

Если вы недавно покупали что-то из продукции Apple, то знаете, как запоминается распаковка их товаров. И это не случайно. В течение многих лет у Apple существует специальная команда, занимающаяся только улучшением впечатления от распаковки.

Адам Лашински в книге «Внутри Apple» (*Inside Apple*, издательство Business Plus) пишет: «Несколько месяцев дизайнер упаковки сидел в своей комнате и занимался самым прозаическим делом — открывал коробки» (http://bit.ly/2CBm2Vt). Он продолжает: «Apple всегда стремится использовать коробки, вызывающие идеальный эмоциональный отклик при открытии... Одну за другой дизайнер создавал и тестировал бесконечные серии стрелочек, цветов и пленочек для крохотной этикетки, показывающей покупателю, в какую сторону тянуть невидимую наклейку без рамок на верхней части коробки нового iPod. Этот дизайнер был просто одержим поиском идеального варианта».

И такое внимание к деталям распространяется не только на открытие коробки и извлечение оттуда устройства. В Apple убеждаются: батарея заряжена полностью, так что клиенты смогут начать работу через несколько секунд и общее впечатление будет приятным и ничем не омраченным. Команды Apple по разработке продукта хотят, чтобы клиенты *полюбили* этот продукт с самого начала. Стефан Томке и Барбара Фейнберг в статье «Дизайн-мышление и инновации в Apple» (http://bit.ly/2q6xeRC) пишут: «Стремление вызвать у людей любовь к нашим устройствам и их использованию всегда вдохновляло — и продолжает по сей день — разработку продуктов Apple».

КОГДА API — ВАШ ЕДИНСТВЕННЫЙ ПРОДУКТ

Stripe — это успешная платежная система на основе прекрасного API, который высоко ценят разработчики. Недавно этот стартап, в котором работают менее 700 сотрудников, был оценен в 10 млрд долларов (https://bloom.bg/2ydEsrz). Вся бизнес-стратегия его основателей заключалась в осуществлении платежных услуг с помощью API. Поэтому они решили с самого начала инвестировать в дизайн-мышление и подход «API как продукт». Для Stripe API является *единственным* продуктом. Отношение к API как к продукту помогло им достичь как технических, так и бизнес-целей.

Такое внимание к первому опыту использования продукта применимо и к API. Кажется, что почти невозможно добиться того, чтобы разработчики их *полюбили*, но это вызовет далеко идущие последствия. Если ваш API сложен для понимания, разработчикам будет тяжело с ним работать, а если для того, чтобы начать его применять, требуется слишком много времени, они просто рассердятся и бросят. В мире API время до того момента, как все заработает, часто называют TTFHW — «время до первого “Hello, World”» (Time to First Hello, World). В сфере онлайн-приложений его иногда называют «время до “Вау!”» (Time to Wow!) — TTW.

Время до «Вау!»

В статье «Ускорение роста: создание “вау-момента”» (http://bit.ly/2CXl1r7) Дэвид Скок, сотрудник фирмы Matrix Partners, занимающейся долевыми инвестициями,

описывает важность «вау-момента», которого надо добиться в любых отношениях с клиентом: «“Вау!” — это момент... когда ваш покупатель вдруг видит выгоду, которую получает от использования продукта, и говорит себе: “Вау! Это круто!”». И, хотя Скок обращается напрямую к людям, разрабатывающим и продающим приложения и онлайн-сервисы для клиентов, теми же принципами должны руководствоваться и те, кто разрабатывает и развертывает API.

Ключевой элемент подхода TTW — это понимание не только того, в чем заключается проблема, требующая решения, но и того, сколько времени и работы необходимо, чтобы добиться «Вау!». Усилия, затраченные, чтобы добраться до момента, когда пользователь API поймет, как его применять, и узнает, что тот может решить его важные проблемы, — вот препятствие, которое должен преодолеть каждый API, чтобы завоевать клиента. Подход Скока заключается в планировании шагов, необходимых, чтобы испытать «вау-момент», и в уменьшении препятствий по пути.

Для примера рассмотрим процесс использования API, который присылает список перспективных клиентов для главного продукта вашей компании, `WidgetA`. Типичный ход процесса выглядит примерно так.

1. Отправить запрос `login`, чтобы получить `access_token`.
2. Получить `access_token` и сохранить его.
3. Составить и отправить заявку на `product_list` с применением `access_token`.
4. В полученном списке найти пункт с `name="WidgetA"` и получить `sales_lead_url` этой записи.
5. Использовать `sales_lead_url`, чтобы отправить запрос на все контакты по продажам, где `status="hot"`, с применением `access_token`.
6. Теперь у вас есть список перспективных контактов по продукту `WidgetA`.

Здесь много этапов, но мы видели рабочие потоки с куда большим их количеством. И на каждом этапе пользователь может совершить ошибку (например, отправить неправильно составленный запрос), провайдер API также может выдать ошибку (например, сообщение о том, что время ожидания запроса закончилось). Здесь могут появиться три проблемы с ответом на запрос (`login`, `product_list` и `sales_leads`). TTW будет ограничено временем, которое потребуется новому разработчику, чтобы понять принцип работы этого API и запустить его. Чем больше времени это займет, тем меньше вероятность того, что он испытает «вау-момент» и продолжит пользоваться этим API.

Есть несколько возможностей улучшить TTW в этом примере. Во-первых, можно изменить дизайн, предложив прямой запрос для списка перспективных контактов (например, `GET /hotleads-by-product-name?name=WidgetA`). Можно также потратить время и написать документацию по сценарию, демонстрирующую новым пользователям, как решить эту конкретную проблему. Мы даже можем предложить

пробную среду для тестирования таких примеров, чтобы пользователи могли пропустить этап идентификации при обучении.

Базовые принципы API

Дизайн, документирование и тестирование — это так называемые базовые принципы API. Мы рассмотрим эти и другие принципы подробнее в главе 4.

Любые меры по уменьшению времени до «Вау!» улучшат мнение пользователя API о вашем продукте и увеличат вероятность того, что им будут пользоваться множество разработчиков как в вашей организации, так и за ее пределами.

Поддержка новых клиентов ваших API

Так же как компания Apple тратит время на создание впечатления от распаковки их продукта, компании, успешно пользующиеся подходом AaaP, тратят время на то, чтобы первый опыт применения их API был максимально простым и приносил удовлетворение. И как батарея нового мобильного телефона, плеера, планшета и т. д. от Apple уже заряжена, когда вы его открываете, API тоже могут быть полностью «заряжены» при первом использовании, чтобы разработчикам было просто начать работать и создать что-то уже в первые минуты знакомства с новым API.

Ранее, работая над управлением API, мы говорили клиентам, что они должны провести нового пользователя от первого взгляда на стартовую страницу API до работающего примера примерно за 30 минут. Если процесс затягивался, возникал риск потерять потенциального пользователя, а вместе с ним — все время и деньги, вложенные в разработку и развертывание данного API. Однако после того, как один из нас закончил презентацию по поддержке новых клиентов API, представитель Twilio, компании по API для SMS, подошел к нам и сказал, что они стремятся к сокращению времени на первый опыт до 15 минут или даже меньше.

Сфера, в которой работает Twillo (API для SMS), — невероятно сложная и запутанная. Представьте, каково это — пытаться создать один API, который будет работать с десятками различных шлюзов и компаний для SMS и при этом его будет легко задействовать и понимать. Это не так-то просто. Одним из ключей к достижению их цели — 15 минут на первый опыт — является частое использование измерений и параметров в обучающих курсах, чтобы найти узкие места — моменты, где пользователи API «выпадают», — и определить, сколько времени им требуется на выполнение заданий. «Мы одержимы измерениями, постоянно отслеживаем рост на разных <этапах> адаптации пользователей и измеряем индекс потребительской лояльности для каждого действия», — сказала Элиза Беллагамба, когда руководила продвижением голосовых сервисов Twilio (http://bit.ly/2CzFKA7).

МОМЕНТ НЕО ДЛЯ TWILIO

В 2011 году евангелист API из компании Twilio Роб Спектр опубликовал запись в блоге по поводу обучения использованию API Twilio для SMS. Он рассказал историю о том, как помогал разработчику применять этот API впервые: «За 15 минут мы проработали руководство Twilio по исходящим звонкам для быстрого старта, и после того, как мы обошли несколько препятствий, код был исполнен и его трубка Nokia засветилась. Он посмотрел на меня, перевел взгляд на экран, ответил на звонок и услышал, как его код говорит: “Hello, World”.

— Вау, чувак, — ошарашенно сказал он, — я это сделал.

И в этом — чистая магия».

Спектр называет это моментом Нео (отсылка к персонажу Нео из фильмов «Матрица») и утверждает, что он может быть «мощным вдохновляющим моментом» для разработчиков.

В Twilio неустанно работают над API и первым опытом их использования, чтобы максимально увеличить количество таких вдохновляющих моментов.

Таким образом, прекрасный первый опыт — это не просто результат удачной разработки. Он обусловлен проработанными обучающими курсами, посвященными тому, как начать работу, и тщательным отслеживанием того, как пользователи API *используют* эти курсы. Сбор данных помогает получить информацию, необходимую для улучшения этого опыта. Над первым опытом применения API следует работать так же, как над самим API. И для его улучшения нужно получать обратную связь (лично и автоматически) от пользователей.

Однако подход AaaP не ограничивается первым опытом. Допустим, вы завоевали сообщество пользователей-энтузиастов, которые остались с вами надолго. Теперь необходимо сосредоточиться на общем *опыте разработчика* ваших API.

Опыт разработчика

Обычно клиент еще долго взаимодействует с продуктом после того, как распаковал его. Да, очень важно убедиться, что продукт, извлеченный из коробки, работает, но так же важно помнить о том, что пользователь (как мы надеемся) станет использовать его и дальше. А с течением времени ожидания пользователей меняются. Им хочется чего-то новенького. И надоедает то, что нравилось поначалу. Они начинают исследовать возможности и даже изобретают уникальные способы применения продукта и его особенностей для решения новых проблем, изначально не заложенных в реализации. Эти непрерывные отношения пользователя с продуктом обычно называются опытом пользователя (user experience, UX).

Компания Apple обращает особое внимание на эти продолжительные отношения. Тай Тран, генеральный директор и основатель социального приложения Blue (http://tryblueapp.com/home) и бывший сотрудник Apple, сформулировал это так: «Как только возникает вопрос, делать нам что-то или нет, мы всегда возвращаемся к другому вопросу: "Как это повлияет на опыт пользователя?"» (https://read.bi/2JbmgDb). Как и любая другая успешная компания-производитель, Apple утверждает, что клиенты превыше всего и сотрудники должны обращать особое внимание на их взаимодействие с продуктами Apple. И им несложно вносить множество изменений, если они значительно улучшают продукт. Например, с 1992 по 1997 год Apple создала более 70 моделей компьютера Performa (некоторые из них так и не были выпущены) и при создании каждой из них использовала информацию, полученную из отзывов пользователей предыдущих версий.

Но, наверное, лучшим примером управления опытом пользователя их продуктов служит подход Apple к технической поддержке — Genius Bar. По словам Вана Бейкера из компании Gartner Research, «Genius Bar — это отличительная черта магазинов Apple, и тот факт, что он бесплатный, выделяет их среди всех прочих предложений в этой сфере» (https://tek.io/2ykZrJl). Предлагая клиентам место, куда они могут прийти с любыми вопросами и проблемами, Apple подчеркивает важность непрерывных отношений между пользователем и продуктом.

Все эти элементы опыта пользователя — подтверждение длительных отношений, стремление вносить небольшие улучшения и обеспечение возможности быстро получить техподдержку — очень важны для создания успешных продуктов с API.

Познакомьтесь с целевой аудиторией

Для создания успешного API как продукта необходимо убедиться, что вы выбрали правильную аудиторию. Для этого нужно знать, *кто* использует ваш API и *какие* проблемы они пытаются решить. Мы говорили об этом в разделе «Дизайн-мышление», это важно и для длительных отношений с разработчиками. Сосредоточившись на пользователях и их проблемах, вы начинаете не только понимать, что важно для создания API, но и более творчески подходить к тому, какие задачи API должен выполнять, чтобы помочь вашей ЦА решить ее проблемы.

Ранее в этой главе мы говорили о необходимости соответствовать требованиям потребителей при разработке продукта. Эта работа должна продолжаться и после реализации API. Сбор отзывов, ознакомление с историями пользователей и в целом внимание к тому, как используются (или не используются) API, — это все входит в непрерывный опыт разработчика. В нем есть три важных элемента:

- обнаружение (discovery) API;
- отчеты об ошибках;
- отслеживание применения API.

Эти и другие элементы будут подробно освещены в главе 4, поэтому сейчас мы рассмотрим лишь некоторые из их аспектов, важные для общего опыта разработчика (developer experience, DX) в вашей стратегии AaaP.

Обнаружение API

Обнаружение API — это процесс, когда вы ставите себя на место разработчика — делаете API таким, чтобы его можно было быстро найти в нужное время. Одна из сложностей с программами с API в больших компаниях заключается в том, что даже при наличии подходящих API разработчики в итоге создают собственные, порой не один раз в разных отделах организации. Иногда в этом видят некий внутренний бунт («Они не используют те API, которые мы им предоставляем!»), однако чаще всего возникновение множества дублирующих функционалов свидетельствует о том, что разработчики не могут найти подходящий API в нужный момент.

Обнаружение API

Мы расскажем о роли этого важного базового элемента поддержки API в подразделе «Обнаружение и продвижение» раздела «Знакомство со столпами» главы 4 (с. 101), а о роли изменений в нем в больших системах API — в подразделе «Обнаружение» раздела «Аспекты системы и столпы жизненного цикла API» главы 10 (с. 260).

В решении этой проблемы может помочь центральный реестр API. Можно также создать поисковый хаб или портал для API с доступом к документации, примерам и другой важной информации. Так будет легче обнаружить существующие API.

ПОИСК API

На момент издания этой книги не существует единого популярного общедоступного поискового сервиса для API. Одна из причин этого — сервисы в Интернете сложно проиндексировать, так как большинство из них не имеют открытых ссылок и редко включают в себя ссылки на связанные с ними сервисы. Другая причина состоит в том, что большая часть используемых сегодня API закрыты частными шлюзами и системами защиты доступа и невидимы для любых общедоступных поисковых систем.

Существуют проекты и форматы с открытым доступом, работающие над созданием поисковиков для API, например, {API}Search (https://apis.io/) в формате описания API (http://apisjson.org/) и сервис ALPS (семантика профиля на уровне приложений) в формате описания сервиса (http://alps.io/). Эти и другие проекты предлагают возможность создания в будущем общедоступной поисковой системы API. Пока этого не произошло, отдельные организации могут применять эти стандарты для внутреннего пользования, чтобы начать процесс создания системы API, доступной для поиска.

Как минимум одна компания из тех, со специалистами которых мы говорили, сделала обязательной публикацию API в центральном реестре, где его можно найти. Таким образом, разработчики API не могли запустить его в производство, не добавив в реестр API компании и не убедившись, что все важные API этой организации могут быть найдены в одном месте. Это большой шаг к увеличению возможности обнаружить их программы с API.

Отчеты об ошибках

Ошибки происходят постоянно. Это часть системы API. От всех ошибок нельзя избавиться, даже используя удачный дизайн, чтобы уменьшить число ошибок пользователей, и тестируя, чтобы убрать баги разработки из кода. Вместо того чтобы пытаться сотворить невозможное (удалить все ошибки), лучше внимательно следить за своими API и вести записи и отчеты о возникающих ошибках. Это позволит получить представление о том, как целевая аудитория задействует ваши API, и тем самым обогатить опыт разработчиков.

Мониторинг API

Отчеты об ошибках и отслеживание использования API (будет рассмотрено далее) относятся к базовому принципу мониторинга. Мы обсудим этот навык на уровне API в подразделе «Мониторинг» на с. 99. А изменения отслеживания при росте программы рассмотрим в подразделе «Мониторинг» на с. 257.

Создавая и производя материальные продукты, например одежду, мебель, канцтовары и т. д., сложно предположить, какие ошибки будут возникать при их применении. Если вы не стоите рядом с человеком, пользующимся вашей продукцией, то, скорее всего, упустите детали и отзывы о ней. Поэтому большинство компаний-производителей выпускают множество прототипов и лично контролируют тесты. Но плюс эры электроники и виртуальных продуктов, например мобильных приложений, в том, что можно встроить в них систему отчетов об ошибках и получать важную информацию, даже когда продукт вам уже неподконтролен, а находится в руках пользователей.

В важные контактные точки API можно встроить следующие отчеты об ошибках.

- *Отчеты пользователя.* Можно добавить в приложение функцию отчетов об ошибках. Она запрашивает у пользователя разрешение на отправку детализированной информации при ошибке. Таким образом вы можете зафиксировать неожиданные условия, возникшие на стороне пользователя.
- *Отчеты шлюза.* Можно добавить систему отчетов на маршрутизатор или шлюз API. Это позволит узнать о состоянии запроса, когда он впервые попадает к вам, и поможет найти плохо сформулированные запросы API или другие проблемы, связанные с сетью.

- *Сервисные отчеты.* Можно добавить систему отчетов в сервисы, к которым обращается ваш API. Это поможет найти ошибки в коде сервисов и неполадки на уровне компонентов, например проблемы с взаимозависимостями или внутренние проблемы, связанные с изменениями экосистемы организации.

Отчеты об ошибках помогают получить важную информацию об использовании API и моментах появления проблем. Но это только половина процесса контроля. Важно также отслеживать успешное *применение* API.

Отслеживание использования API

Этот процесс API относится не только к ошибкам. Следует отслеживать все запросы и затем анализировать эту информацию, чтобы найти полезные шаблоны. Как упоминалось на с. 61, API создаются и применяются во многом ради поддержки ваших бизнес-стратегий. Известный специалист по API Ким Лейн сказал: «Понимание того, как API помогут (или не помогут) вашей организации лучше взаимодействовать с целевой аудиторией, — это и есть суть API» (http://bit.ly/2PdTXK1).

Данные, которые требуются, чтобы определить, помогает ли ваш API организации лучше взаимодействовать с целевой аудиторией, обычно называются OKR (objectives and key results — цели и ключевые результаты) и KPI (key performance indicators — ключевые показатели эффективности). Мы рассмотрим их подробнее в подразделе «OKR и KPI» раздела «Измерения и ключевые моменты» главы 6 (с. 123), а сейчас важно понять, что для достижения своих целей вам нужно знать, как ведут себя API в данный момент. Для этого необходимо отслеживать не только возникающие ошибки, как говорилось в предыдущем разделе, но и успешные моменты.

Допустим, вы хотите собрать данные по тому, какие приложения запрашивают те или иные API и соответствуют ли эти приложения требованиям своих пользователей. У отслеживания есть дополнительное преимущество — оно помогает видеть шаблоны, по которым действует большое количество людей. Отдельные пользователи могут не видеть этих шаблонов. Вы можете обнаружить, что приложения все время отправляют одни и те же запросы API, например:

```
GET http://api.mycompany.org/customers/?last-order=90days
GET http://api.mycompany.org/promotions/?flyer=90daypromo
POST http://api.mycompany.org/mailing/ customerid=1&content="It has been
  more than 90 days since…"
POST http://api.mycompany.org/mailing/ customerid=2&content="It has been
  more than 90 days since…"
…
POST http://api.mycompany.org/mailing/ customerid=99&content="It has been
  more than 90 days since…"
```

Этот повторяющийся шаблон может обозначать потребность в новом, более эффективном способе, каким ваша ЦА может отправлять рассылки ключевым группам

клиентов, — и это один запрос от приложения, сочетающий в себе целевую группу клиентов с выбранным продвигаемым контентом, например:

```
POST http://api.mycompany.org/bulk-mailing customer-filter=last-order-
90days&content-flyer=90daypromo
```

Такой запрос создает меньше трафика между клиентом и сервером, уменьшает количество возможных неполадок сети и более прост для использования клиентами API. И он предложен не пользователем, а вами благодаря вниманию к информации по отслеживанию применения API.

ПИТЬ СОБСТВЕННОЕ ШАМПАНСКОЕ

Три года назад одна европейская национальная железнодорожная компания решила организовать форумы для сообществ своих разработчиков, один для внешних, другой — для внутренних.

Мероприятие для внешних разработчиков координировало руководство отделов коммуникаций и управления продукцией. Они заказали IT-отделу сбор статических данных, доступных для внешнего использования, и помогли IT-командам разработать набор простых, сфокусированных на задании API для получения сведений о расположении станций, расписания поездов и т. д. Они были легко реализованы, и IT-отдел счел их менее мощными, чем внутренние API с полным функционалом. Мероприятие прошло довольно успешно.

Через полгода IT-отдел организовал собственный форум, задействуя официальные внутренние API. Вскоре организаторы поняли, что команды разработчиков перешли с внутренних API с полным функционалом на более простые и сфокусированные на задании внешние API. И стали действовать эффективнее и продуктивнее.

Отсюда можно сделать несколько выводов. Во-первых, все разработчики предпочли API, сфокусированные на задании. Во-вторых, создание этих более простых API не потребовало много времени или ресурсов. И в-третьих, IT-отделам всегда следует обращать внимание на то, какие API более популярны и чаще применяются. А последний можно выразить распространенной фразой: «Пить собственное шампанское» (использовать свою продукцию). Как и в любом другом случае, в области API всегда лучше, чтобы внутренние и внешние команды задействовали один и тот же продукт.

Это подводит нас к еще одной важной области опыта разработчика (DX): как сделать работу разработчиков с вашим API проще и безопаснее.

Проще и безопаснее

Кроме поддержки простого обнаружения API, тщательного отслеживания ошибок и общего использования API, важно предоставлять вашим разработчикам быстрый и постоянный доступ к техподдержке и обучению. Ведь для долгих и позитивных

отношений необходимы именно впечатления от того, что происходит *после* успешного ознакомления разработчиков с программой. Мы уже видели пример такого внимания в этом разделе, когда упоминали Genius Bar компании Apple, служащий для поддержки пользователей. Вашим API тоже нужен Genius Bar.

Другой важный аспект поддержки разработчиков — обеспечение *безопасности использования* вашего продукта. Другими словами, должно быть сложно применить его неправильно, причиняя какой-то ущерб. Например, нужно затруднить удаление важных данных или единственного аккаунта администратора и т. д. Если вы обращаете внимание на то, как именно пользователи вашего API (то есть разработчики) задействуют его, это поможет определить зоны, где требуется усилить безопасность.

Чтобы создать мощную и непрерывную связь с разработчиками вашего API, необходимо сочетание простоты и безопасности.

Как сделать API безопасным для использования

Существует ряд элементов API, которые, с точки зрения разработчика, несут в себе *риск*. Иногда определенные запросы к API могут сделать что-то опасное: удалить все записи пользователя, изменить все цены на услугу на нулевые и т. п. Иногда риск заключается даже в *подключении* к серверу API. Например, настройка команды подключения к API с данными может рассекретить логины и пароли в URL или незашифрованные сообщения. Мы сталкивались с множеством таких проблем с безопасностью в наших обзорах API.

Часто можно избавиться от подобных рисков с помощью *дизайна*. То есть изменить его таким образом, что будет сложнее столкнуться с конкретным риском. Например, для API, удаляющего важные данные, можно разработать запрос Отмена. Таким образом, если кто-то удалит эти данные по ошибке, он сможет активировать запрос Отмена и восстановить их. Или, например, можно требовать повышенных прав доступа для исполнения определенных операций, например добавить поле данных (пароль), которое нужно заполнить, если запрос обновляет важную информацию.

Однако иногда уменьшить риск с помощью дизайна слишком сложно. Некоторые запросы к API в принципе рискованны. Например, любой запрос, удаляющий данный, чреват последствиями, независимо от количества изменений в дизайне. Некоторые запросы к API всегда будут долго исполняться, скорее всего потребляя множество ресурсов со стороны сервера. Другие API работают быстро и присылают очень много данных. Например, фильтрующий запрос потенциально может прислать сотни тысяч записей.

Когда запросы к API несут в себе неизбежный риск, негативные последствия можно уменьшить, если добавить в документацию API предупреждения. Таким образом, пользователям будет проще заранее опознать потенциальные опасности и, возможно, избежать критических ошибок. Существует много способов отформатировать документацию так, чтобы указать на возможные риски. Например, выделить текст,

сообщающий пользователю о проблеме («Предупреждение: в зависимости от настроек фильтра этот API может прислать более миллиона записей»). С этой целью можно также использовать ярлыки с символами. Таким образом, не понадобится добавлять в документацию большой объем текста, вместо этого читатели просто узнают ярлык с предупреждением.

Материальные товары зачастую маркируют информационными и предупреждающими символами (рис. 3.1).

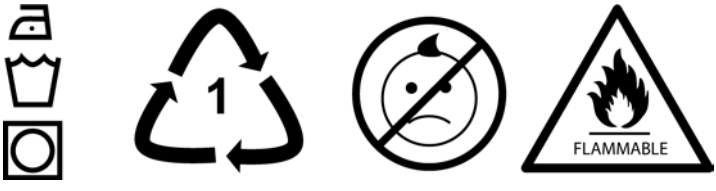


Рис. 3.1. Примеры ярлыков на товарах для дома

Можете использовать такой подход для своих API (рис. 3.2).

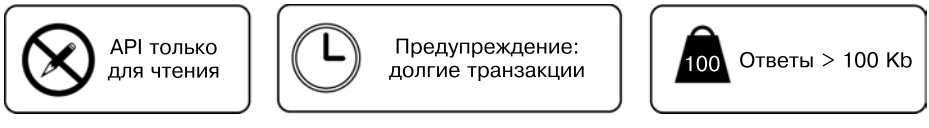


Рис. 3.2. Примеры ярлыков API

Простые предупреждающие символы в сочетании с изменениями дизайна уменьшают вероятность того, что пользователи совершат серьезные ошибки. Это значительно увеличит безопасность продукта с API.

Как сделать API простым в использовании

Сделать API относительно простым в применении — еще одна важная задача. Если выполнение задачи состоит из слишком большого числа этапов, если названия и номера функций сложные или запутанные, если названия запросов к API кажутся пользователям бессмысленными, ваш API столкнется с проблемами. Разработчики не только будут недовольны его использованием, но и могут начать чаще ошибаться.

Вы можете упростить применение API с помощью дизайна. Например, задействуя шаблоны названий, подходящие к задачам, которые выполняют ваши разработчики. Это отсылает нас к пониманию того, кто составляет целевую аудиторию (см. подраздел «Соответствие требованиям потребителей» раздела «Дизайн-мышление» главы 3), и способам решения ее проблем (см. подраздел «Конкурентная бизнес-стратегия» того же раздела). Но даже если вы так и сделаете, если ваш API объемный

(то есть содержит много URL или действий) или сложный (в нем много опций), не всегда можно положиться на дизайн. Вместо этого следует упростить его для пользователей, задав им нужные вопросы и получив подходящие ответы. Вашему API нужно подобие Genius Bar для разработчиков.

Наверное, простейший способ предоставить им Genius Bar для API — это составление документации. Если поместить в нее не только простую справочную информацию (название API, методы, функции, возвращаемые значения), можно поднять ее до уровня «гения». Например, можно добавить раздел «Часто задаваемые вопросы» и привести в нем ответы (и подсказки) на самые распространенные вопросы пользователей. Расширить его, добавив раздел «Как это делается?», в котором поместить короткие пошаговые примеры выполнения распространенных заданий. Можно даже дать функционирующие примеры, которые разработчики смогут использовать в качестве стартового материала для собственных проектов.

Документация

Мы подробнее поговорим о навыке составления документации для API в подразделе «Документация» раздела «Знакомство со столпами» главы 4 (с. 85) и обсудим, как изменятся ваши потребности в этой сфере с ростом системы API, в разделе «Документация» подраздела «Аспекты системы и столпы жизненного цикла API» главы 10 (с. 238).

Следующий шаг после улучшения документации — активная форма или чат для онлайн-поддержки. Форумы поддержки предоставляют пространство для обсуждения, где разработчики могут задавать вопросы сообществу пользователей и делиться своим решением проблем. Если сообщество API велико, эти форумы могут стать источником важных исправлений багов и запросов на новые функции. А еще важным хранилищем знаний, накопившихся за долгое время, особенно если там есть отлаженный поисковый механизм.

Еще быстрее обеспечить поддержку клиентов API наподобие Genius Bar можно с помощью чатов. Они обычно ведутся в реальном времени и добавляют новый уровень персонализации опыту ваших разработчиков. Там также можно использовать общие знания о вашем продукте с API и повышать их уровень.

Наконец, для крупных сообществ API и/или больших организаций имеет смысл обеспечить личную поддержку вашего продукта евангелистами API, наставниками, специалистами по устранению неполадок. Ваша компания может организовать семинары или конференции, где пользователи API будут собираться для работы над проектами или испытания новых функций. Это работает и для внутреннего сообщества API, то есть сотрудников компании, и для внешнего — сотрудников или пользователей общедоступных API. Чем более индивидуальной будет ваша связь с разработчиками, тем большему они смогут вас научить и тем сильнее упростят использование API.

То, что вы потратите время на обеспечение безопасности и облегчение применения API, поможет вам установить добрые отношения с пользователями и в целом улучшить опыт разработчиков.

Выводы

В этой главе мы познакомили вас с подходом «API как продукт» и тем, как его можно задействовать для улучшения дизайна, развертывания API и управления им. Чтобы применять этот подход, необходимо знакомиться с целевой аудиторией, понимать и решать ее проблемы и обрабатывать отзывы от пользователей API.

Мы рассмотрели три ключевых понятия в сфере AaaP:

- дизайн-мышление, которое необходимо, чтобы убедиться, что вы знаете свою целевую аудиторию и понимаете ее проблемы;
- поддержку новых клиентов как способ быстро показать им, как успешно пользоваться вашим продуктом;
- опыт разработчика, используемый для управления жизненным циклом вашего продукта после его выпуска и выработки идей для будущих модификаций.

Попутно мы узнали, что приверженность принципам AaaP помогла таким компаниям, как Apple, Amazon, Twilio и др., не только создать успешные продукты, но и привлечь преданных клиентов. Независимо от того, нацелена ли ваша программа с API только на внутреннее пользование или как на внутренних, так и на внешних разработчиков, сообщество преданных клиентов очень важно для обеспечения ее работоспособности и успеха в перспективе.

Теперь, когда вы получили представление о фундаментальных принципах подхода AaaP, мы можем перейти к распространенному набору навыков, необходимых для взращивания любой успешной программы с API. Мы называем их базовыми принципами API, и именно им посвящена следующая глава.

4 Десять столпов API

> Когда что-то получилось удачно, кажется, что это было легко. Никто не понимает, насколько это на самом деле сложно. Думая о фильме, большинство представляет двухчасовую законченную идеальную версию. Но ради этой двухчасовой версии сотни и тысячи людей работали целыми днями в течение многих месяцев.
>
> *Джордж Кеннеди*

В предыдущей главе мы рассказали об отношении к API как к продукту. Теперь подробнее рассмотрим работу, необходимую для создания и поддержки вашего продукта. Чтобы разработать качественный API, приходится много и тяжело работать. Из главы 1 вы узнали, что у API есть три части: *интерфейс*, *реализация* и готовый *экземпляр*. Чтобы создать API, необходимо затратить время и силы на управление каждой из этих трех составляющих. Кроме того, нужно следить за своевременным обновлением, когда ваш продукт начнет постоянно развиваться и изменяться. Чтобы помочь вам разобраться в этих сложностях, мы разделили всю работу на десять частей — столпов.

Мы называем их *столпами*, потому что они поддерживают ваш продукт. Если вы их не укрепляете, он обречен на падение и крах. Но это не значит, что для создания успешного API во все столпы надо вкладываться по максимуму. Положительная сторона наличия десяти столпов в том, что они не должны поддерживать одинаковый вес. Какие-то из них могут быть крепче других. Вы даже можете счесть, что в некоторые вообще почти не стоит вкладываться. Важно то, что общая сила этих столпов поддерживает ваш API, даже когда с течением времени он эволюционирует и меняется.

Знакомство со столпами

Каждый из столпов API очерчивает собственное поле деятельности. Или, другими словами, каждый очерчивает ряд *решений*, связанных с API. На самом деле работу

по созданию API нельзя четко разделить на части, и некоторые из столпов будут перекрывать друг друга, но это нормально. Мы стремимся не устанавливать непреложные истины, относящиеся к работе с API, а, напротив, разработать полезную модель для исследования и обсуждения работы по созданию продуктов с API. Мы будем надстраивать эти основные понятия в течение всей книги, рассматривая более сложные модели организации команды, развития продукта и аспектов системы в следующих главах.

Вот десять определенных нами столпов работы с API.

1. Стратегия.
2. Дизайн.
3. Документация.
4. Разработка.
5. Тестирование.
6. Развертывание.
7. Безопасность.
8. Мониторинг.
9. Обнаружение.
10. Управление изменениями.

В этой главе мы познакомим вас с каждым из этих столпов и исследуем их особенности и важность для успеха продукта с API. Опишем также пространство решений для каждого из них и дадим общие указания, как его укрепить. В этой главе не будет отдельных советов по применению этих столпов — полное обсуждение каждой из сфер деятельности тянет на отдельную книгу, а нам нужно затронуть еще множество разделов управления. Однако назовем некоторые из важнейших решений в каждой сфере с точки зрения руководства. Начнем с первого столпа продукта с API — стратегии.

Стратегия

Выдающиеся продукты начинаются с разработки выдающейся стратегии, и продукты с API не исключение. Столп «Стратегия» включает в себя две ключевые области решений: причина, по которой вы хотите создать этот API (цель), и то, как он поможет вам достичь этой цели (тактика). Важно понимать, что стратегическая цель для API не может существовать в вакууме. Любая цель, которую вы выбираете для продукта с API, должна приносить пользу и всей организации. Разумеется, это означает, что прежде всего вы должны понимать стратегию или бизнес-модель своей организации. Если вы не в курсе ее целей, уточните их, прежде чем создавать новые API.

Усиление вашего бизнеса с помощью API

Влияние вашего продукта с API на стратегию организации во многом зависит от направленности бизнеса. Если основной источник дохода компании — продажа доступа к API сторонним разработчикам (например, API для связи от Twilio или API для платежей от Stripe), тогда стратегия вашего продукта будет крепко связана со стратегией всей компании. Если API работает эффективно, компания получает доход, а если не работает, ее ждет крах. Продукт с API становится основным каналом дохода вашей организации, и его архитектура и бизнес-модель войдут в число целей компании.

Однако у большинства организаций уже есть традиционный бизнес, который API просто должны поддерживать. В этом случае API не станет новым основным источником дохода, если компания не поменяет кардинально свою стратегию. Например, банк, работающий уже сотни лет, может открыть API для внешних разработчиков, чтобы поддержать инициативу банковского обслуживания в открытом формате. Если фокусироваться только на бизнес-модели API, можно использовать схему получения дохода от него, согласно которой разработчики платят за доступ к функциям банковского обслуживания, основанным на API. Но если держать в голове общую картину работы банка, эта стратегия API может быть вредной, потому что создает барьер для пользователей. Вместо этого можно предложить банковский API бесплатно (в убыток себе) в надежде, что улучшение удаленного доступа к банку приведет к росту продаж самих банковских услуг.

Стратегия API задействуется не только для продуктов, предлагаемых для внешнего пользования, это важный столп и для внутренних API. Для них тоже потребуется определить стратегию. Важное различие между стратегиями внутреннего и внешнего API только в том, для каких пользователей они предназначены. Но в обоих случаях необходимо понять, зачем нужен этот API и как он может помочь в решении проблемы. Неважно, какова причина появления вашего API, в любом случае стоит разработать для него стратегическую цель, выгодную для организации.

Когда стратегическая цель продукта определена, необходимо разработать тактику, которая поможет вам ее достичь.

Определение тактики

Достичь стратегической цели API поможет план, который следует составить для вашего продукта. Необходимо разработать тактику работы, которая улучшит вероятность успеха. Фактически ваша стратегия будет направлять всю предстоящую работу, основанную на решениях. Трудность состоит в том, чтобы понять, как связаны между собой поле деятельности по каждому решению и цель.

Рассмотрим несколько примеров.

- *Цель — увеличить возможности для бизнеса на вашей платформе.* Если цель заключается в создании расширенного набора API для бизнеса, тактика должна

включать в себя прежде всего изменения в разработке и создании API. Например, стоит привлечь к работе влиятельных участников и партнеров вашего бизнеса на ранней стадии создания API, чтобы убедиться, что ваш интерфейс предоставляет им нужные возможности. Следует довериться им и тогда, когда речь пойдет об определении первостепенных пользователей и ограничений для API.

- *Цель — монетизировать внутренние активы.* Монетизация требует тактики, помогающей представить продукт сообществу пользователей, которое сочтет его полезным. Обычно это означает также, что вам предстоит работа на конкурентном рынке, поэтому большое значение приобретает опыт разработчика (DX) в использовании вашего API. Тактически это означает увеличение вложений в такие столпы работы, как дизайн, документация и обнаружение. Вам понадобятся и маркетинговые исследования для определения нужной целевой аудитории API. Нужно будет убедиться в том, что ваш продукт ей подходит.
- *Цель — собрать идеи для бизнеса.* Если ваша цель — найти инновационные идеи вне своей компании, понадобится разработать тактику, поощряющую применение API инновационными способами. Таким образом, необходимо выставить его на внешний рынок и сделать привлекательным для сообщества пользователей, которое принесет наибольшую пользу с точки зрения потенциальных инноваций. Стоит приложить максимум усилий к области обнаружения API, чтобы добиться максимального уровня его использования для сбора максимального количества идей. Вам также понадобится четкая тактика, чтобы определить лучшие идеи и пути их дальнейшего развития.

Как видите, чтобы разработать сильную тактику для API, вам понадобится:

- определить, какие столпы необходимы для успеха;
- выяснить, какое сообщество пользователей приведет вас к успеху;
- собрать данные об общей обстановке, чтобы управлять работой по принятию решений.

Адаптация стратегии

Когда вы начинаете создавать API, важно разработать правильную тактику, однако не менее важно, чтобы стратегия оставалась гибкой и готовой к изменениям. Устанавливать стратегическую цель и тактический план раз и навсегда — неудачное решение. Необходимо корректировать стратегию, основываясь на результатах, получаемых от продукта. Если вы никуда не продвигаетесь, необходимо поменять тактику, возможно, слегка изменить цель или даже стереть этот API и начать заново.

Для отслеживания прогресса нужен способ измерения результатов работы API. Крайне важно иметь набор параметров для стратегических целей, иначе вы не поймете, насколько успешно продвигаетесь вперед. Мы подробнее обсудим цели и измерения API в главе 6, когда познакомим вас с OKR и KPI для API. Измерения также зависят от способа сбора данных об API, поэтому понадобится серьезно отнестись к мониторингу.

Вы должны быть готовы изменить стратегию, если меняется то, что окружает API, например, если организация ставит иную стратегическую цель, или на рынке появляется новый конкурент, или правительство вводит новые ограничения. В каждом из этих случаев скорость адаптации может сильно повлиять на ценность вашего API. Но изменения стратегии ограничены изменяемостью API, поэтому важно вкладывать силы и время в управление изменениями (мы обсудим это позже в данной главе).

Ключевые решения по стратегии

- *В чем заключаются цель и тактический план API?* Определение цели и плана ее достижения — важнейшая часть стратегической работы. Необходимо тщательно продумать, где должно приниматься это решение. Вы можете позволить отдельным командам определить собственные цели и тактику, чтобы получить преимущество локальной оптимизации, или централизовать планирование целей, чтобы улучшить оптимизацию системы. Если вы решите децентрализовать работу со стратегией API, понадобится создать рычаги контроля и способы поощрения, чтобы ни один API не нанес непоправимого вреда. Например, централизация этапа одобрения решения о постановке цели может замедлить процесс, но предотвратит внезапные проблемы.
- *Как измеряется влияние стратегии?* Определение цели API — локальная задача, но необходимо будет убедиться, что ваша цель совпадает с интересами компании. Это измерение может быть децентрализовано и передано командам, разрабатывающим API, а может быть и централизованным, и стандартизированным. Например, для отчетов по API можно ввести непрерывный процесс со стандартизированными параметрами, которого команды должны придерживаться. Преимущество этого подхода в том, что у вас будет постоянный объем данных для анализа на системном уровне.
- *Когда менять стратегию?* Иногда цели приходится менять, но кто имеет право принимать такое решение? Проблема изменения цели API заключается в том, что оно может очень сильно повлиять как на сам API, так и на зависящих от него людей. Вы можете предоставить командам возможность самостоятельно определять цель нового API, но при этом необходимо больше контролировать изменения цели, особенно когда API уже активно используется.

Дизайн

Принятие решений о том, как создаваемый продукт будет выглядеть, ощущаться и применяться, — это и есть дизайн. Все, что вы создаете или изменяете, требует решений по дизайну. В принципе, принятие всех решений, описанное в разных разделах этой главы, можно считать дизайном. Но столп «Дизайн API», который мы описываем далее, заключается не только в этом. Речь пойдет о конкретном типе дизайна — решениях, принимаемых при разработке *интерфейса* API.

Мы выделили дизайн интерфейса в отдельный столп, потому что он очень сильно влияет на API. Интерфейс — это всего лишь одна из составляющих, но пользователи, работающие с вашим продуктом, видят только его. Для них интерфейс — это и есть API. Поэтому любой дизайн интерфейса, над которым вы работаете, значительно воздействует на решения, которые вы принимаете во всех случаях. Например, решение о том, что API должен иметь событийно управляемый интерфейс, кардинально меняет работу по его реализации, развертыванию, мониторингу и документированию.

При дизайне интерфейса придется принимать множество решений. Вот пример нескольких важных вопросов, которые нужно будет рассмотреть.

- *Терминология*. Какие слова и термины следует понимать вашим пользователям? О каких специальных символах они должны знать?
- *Стили*. Какие протоколы, шаблоны сообщений и стили интерфейса будут применяться? Например, будет ли ваш API задействовать шаблон «Создать, прочесть, обновить, удалить» (CRUD)? Или событийно-ориентированный стиль? Или стиль, похожий на запросы GraphQL?
- *Взаимодействие*. Как API будет помогать пользователям решать проблемы? Например, какие запросы они должны сделать, чтобы достичь цели? Как им будут сообщать о статусе запроса? Какие настройки, фильтры и подсказки по использованию вы им предоставите?
- *Безопасность*. Какие особенности дизайна помогут пользователям избежать ошибок? Как вы будете сообщать им об ошибках и проблемах?
- *Единообразие*. Насколько пользователям будет знаком интерфейс? Будут ли термины этого API совпадать с терминами других API в вашей организации или сфере? Будет ли дизайн интерфейса похож на API, которые они использовали раньше, или удивит их новизной? Будет ли дизайн отвечать стандартам, принятым в вашей сфере?

Это не исчерпывающий список, но, как видите, вам необходимо будет принять очень много решений. Разработка хорошего интерфейса — это сложная работа. Но давайте уточним нашу цель. Что такое хороший дизайн интерфейса API и как улучшить решения в этой сфере?

Что такое хороший дизайн

Если вы определили стратегию своего API, значит, уже установили для него стратегическую цель. Надо понять, как дизайн интерфейса может помочь вам приблизиться к ней. Как говорилось в разделе «Стратегия», у вас может быть множество различных целей и тактик. Но в целом все они сводятся к выбору между двумя основными целями.

1. Привлечь больше пользователей API.
2. Уменьшить затраты на разработку API.

На практике для обеспечения эффективности вам потребуется гораздо более подробная стратегия. Но, обобщая цели таким образом, мы можем сделать важное наблюдение: хороший дизайн интерфейса поможет вам достичь обеих этих целей. Если общее впечатление от использования API приятное, то работать с ним захочет больше пользователей. Дизайн интерфейса играет важную роль в получении опыта разработчика, поэтому хороший дизайн привлечет больше пользователей. Хороший интерфейс также уменьшает вероятность допустить ошибку и упрощает действия, с помощью которых решается проблема пользователя. Это упрощает разработку ПО, которое вы пишете, применяя API с хорошим дизайном.

Поэтому стоит прилагать усилия к тому, чтобы создать хороший дизайн интерфейса. Но конкретного набора решений, которые делают его хорошим, не существует. Качество интерфейса зависит исключительно от целей его пользователей. Невозможно сделать его удобным в применении, если вы не знаете, для кого его разрабатываете. К счастью, если вы уже поняли, *зачем* создаете этот API, то выяснить, *для кого* это делаете, — несложное упражнение. Сделайте это сообщество пользователей своей целевой аудиторией и принимайте решения по дизайну, которые улучшат их впечатления от работы с вашим API.

ОПЫТ РАЗРАБОТЧИКА

В этой главе и на протяжении всей книги мы будем упоминать об *опыте разработчика* (developer experience, DX). DX — это опыт пользователя вашего API, с той поправкой, что имеется в виду конкретный тип пользователей — разработчики программного обеспечения. DX — это сумма всех взаимодействий разработчика с вашим API. Дизайн интерфейса играет в получении этого опыта значимую роль, важны также документация, маркетинг и техподдержка. В итоге DX является отражением того, насколько довольны или недовольны ваши пользователи.

Используйте готовые методы дизайна

Чтобы дизайн интерфейса приносил максимальные результаты, лучше всего делать ставку на готовые методы или процессы. Для создания дизайна важно строить предположения о том, что может сработать. С чего-то надо начинать, поэтому вначале можно скопировать понравившийся интерфейс API или последовать указаниям из какой-нибудь записи в блоге. Это совершенно нормально. Но если вы хотите максимально увеличить доход от создаваемого интерфейса, нужно будет тестировать эти предположения и убеждаться в том, что вы принимаете оптимальные решения, то есть лучшие из возможных.

Например, вы можете выбрать упрощенный процесс.

1. Найти прототип интерфейса.
2. Написать собственное приложение-клиент, использующее прототип.

3. Дополнить прототип, основываясь на полученной информации, и попробовать еще раз.

Более сложный процесс может выглядеть так.

1. Устроить совещание, посвященное дизайну, со всеми заинтересованными лицами — пользователями, специалистами службы технической поддержки, разработчиками.
2. Совместно разработать терминологию для интерфейса.
3. Провести опросы в сообществе пользователей.
4. Создать прототип.
5. Протестировать прототип на целевых сообществах пользователей.
6. Утвердить прототип с помощью разработчиков.
7. При необходимости повторить.

Важное различие между этими двумя примерами заключается в количестве средств, которые вы инвестируете, и информации, которую получаете. Решение о том, насколько сильно нужно вложиться в процесс дизайна, стратегическое. Например, интерфейс внешнего API, которое будет продаваться на конкурентном рынке, вы наверняка проработаете тщательнее, чем интерфейс внутреннего API для команды разработчиков. Но помните, что даже упрощенный процесс дизайна может принести крупные дивиденды.

ФОРМАТЫ ОПИСАНИЯ API

Вы можете облегчить себе работу по созданию дизайна, используя машинно-распознаваемое описание интерфейса. Не у всех стилей API есть установленный стандартный формат, но у самых распространенных он существует. Например, если вы разрабатываете API на SOAP, можно использовать формат WADL (http://bit.ly/2yt4ebG), для API на HTTP со стилем CRUD применяется спецификация OpenAPI (http://bit.ly/2Pn1Z3m), а для разработки API на gPRC можно взять буферы протоколов (http://bit.ly/2PdNxdR). Все эти форматы описания упрощают создание прототипов с помощью инструментальных средств и могут передавать описание интерфейса в виде файла.

Ключевые решения по дизайну

- *Каковы ограничения по дизайну?* Команда по API, которую не ограничивают в области дизайна, может создать модель интерфейса, максимально облегчающую использование и опыт пользователя. Это удобно для конкретного пользователя, но при этом мы жертвуем гибкостью интерфейса для пользователей в целом. Подобная локальная оптимизация влияет на всю систему. Если вы производите несколько API и пользователям нужно задействовать

больше одного, придется ввести некоторые ограничения на решения по дизайну. Таким образом, нужно будет централизовать либо дизайн, либо выбор вариантов, предоставляемых дизайнерам. Некоторые централизованные команды публикуют «гид по стилям», чтобы документировать подобные ограничения. Он может включать в себя официально разрешенные термины, стили и взаимодействия.

- *Как можно поделиться моделью интерфейса?* Иными словами, как поддерживать непрерывную работу по дизайну API. Например, если вы полностью централизуете это решение, то можете решить, что все команды по API должны создавать дизайн в формате описания OpenAPI. Это ограничивает варианты дизайна опциями, предлагаемыми спецификацией OpenAPI, зато упрощает разделение работы между командами и использование одних и тех же инструментальных средств и автоматизации в системе.

Документация

Даже если вы создадите прекрасный дизайн интерфейса для API, пользователи не смогут начать применять его без небольших подсказок. Например, может потребоваться показать пользователям, где именно API расположен в сети, научить их терминологии сообщений и интерфейса и сообщить, в каком порядке следует создавать запросы. Столп «Документация по API» заключает в себе работу по созданию материалов для такого обучения. Мы называем его «Документация по API», а не «Обучение API», потому что чаще всего обучение работе с API и происходит в форме чтения человеком документации.

Предоставление качественной документации полезно по той же причине, что и разработка качественного интерфейса: положительный опыт разработчика увеличивает стратегическую ценность API. Если документация неудачная, понимать API будет сложнее. Если научиться его использовать слишком трудно, количество пользователей уменьшится. Если они будут вынуждены с ним работать, то станут дольше разрабатывать программное обеспечение и в нем будет больше проблем. Так разработка качественной документации оказывается важной частью создания качественного опыта разработчика.

Методы создания документации

Документацию по API можно предоставлять по-разному: в виде отсылок к ресурсам вашего API, похожим на энциклопедию, в виде обучающих курсов и содержательных материалов, даже в форме четко задокументированных сложных образцов приложения, которые пользователи могут скопировать и использовать для обучения. Существует невероятно богатое разнообразие стилей, форматов и стратегий технической документации. Для упрощения разделим документацию по API по соответствию двум фундаментальным принципам: «обучай, но не рассказывай»

и «рассказывай, но не обучай». Наш опыт свидетельствует: для самого качественного обучения пользователей пригодятся оба подхода.

Подход «рассказывай, но не обучай» состоит в том, чтобы сообщать пользователям полезные факты о вашем API. Примерами такого подхода, основанного на фактах, может служить составление перечня кодов ошибок API или списка схем тела сообщения. Такой тип документации предоставляет пользователям справочник по интерфейсу. Поскольку он состоит в основном из фактов, его довольно просто разработать. Обычно такую документацию можно быстро создавать с помощью инструментальных средств, особенно если дизайн интерфейса упорядочен в машинно-распознаваемом формате (см. врезку «Форматы описания API» ранее в данной главе). В целом такой тип фактического описания деталей и поведения интерфейса требует от команды меньше усилий по разработке дизайна и принятию решений. Ключевые решения связаны с выбором того, какие части API надо документировать, а не с тем, как лучше всего передать эту информацию.

В противоположность ему подход к документации «обучай, но не рассказывай» сосредоточен на разработке обучения. Вместо того чтобы просто сообщить факты для изучения, этот подход предоставляет пользователям персонализированное обучение. Его предназначение — помочь им достичь целей применения API в процессе его изучения, используя целенаправленный подход. Например, если вы создали API для навигации, можете написать обучающий курс из шести этапов, показывающий пользователям, как получить из адреса информацию по GPS. Таким образом вы поможете им выполнить типичное задание, прилагая минимум усилий.

Однако документация не должна быть пассивной. Читать справочники, руководства и обучающие материалы полезно, но вы можете также добавить инструменты, которые помогут пользователям изучить API более интерактивно. Например, предоставить им инструмент для исследования API с веб-интерфейсом, позволяющий отсылать запросы к API в реальном времени. Хороший инструмент для исследования API — это не просто тупая программа, отправляющая запросы в Сеть. Он направляет обучение, предоставляя пользователю список действий, терминов, предложений и исправлений. Большой плюс интерактивных инструментов в том, что они сокращают для пользователей время получения обратной связи, — это цикл изучения, попыток применить полученную информацию на практике и обучения на результатах. Без таких инструментов пользователям придется тратить время на написание кода или поиск внешних инструментов, что увеличит время обратной связи.

Портал разработчиков

В мире API портал разработчиков — это место (обычно сайт), где размещаются все ресурсы для поддержки API. Создавать его не обязательно, но он может сильно помочь в улучшении опыта разработчика для API, давая пользователям удобный способ изучения вашего продукта и взаимодействия с ним.

Вложения в документацию

Если вы предоставляете только один тип документации по API, возможно, этого недостаточно для вашего сообщества. У разных пользователей разные потребности, и если вам важно их обучение, нужно удовлетворить все потребности. Например, для новых пользователей может быть удобен подход «обучай, но не рассказывай», потому что он директивный и ему легко следовать. Но опытным пользователям API понравится документация в стиле «рассказывай, но не обучай», потому что они могут быстро сориентироваться в нужных фактах. Интерактивные инструменты, вероятно, придутся по душе тем, кому нравится практический опыт, но они не подходят для тех, кто любит разбираться и планировать — или просто предпочитает читать документацию.

На практике создание всей этой документации может оказаться затратным. В конце концов, кому-то придется ее разрабатывать и писать — и не один раз, а на протяжении всего существования API. Одной из сложностей работы с документацией по API является ее синхронизация с изменениями в интерфейсе и реализации. Если документы некорректны, пользователи быстро начнут проявлять недовольство. Поэтому нужно будет принять разумные решения о том, какое количество документации вы сможете поддерживать.

Ключевым фактором в принятии решения о размере вложений в документацию должна быть выгода для организации от улучшения обучения API. Например, хорошая документация может помочь выделить внешний продукт с API среди конкурентов. Она также может сильно помочь разработчикам, использующим внутренний API, незнакомым с системой, бизнесом или сферой деятельности владельца этого API. Уровень вложений в документацию для внешнего API, работающего на конкурентный рынок, обычно выше, чем для внутреннего, и это нормально. В конце концов, вам придется решить, сколько документации нужно вашему API. К тому же эти вложения всегда можно увеличить, если API будет расти.

Ключевые решения по документации

- *Как разработать процесс обучения?* Решения по разработке процесса обучения обычно принимают отдельно от решений по дизайну, реализации и развертыванию API. Если у вас много API, пользователи оценят наличие единого процесса обучения по всем ним. Однако централизация этого решения обычно ведет к потерям: инноваций становится меньше, появляется больше ограничений для команд по API, также в связи с централизацией технического описания возможно возникновение узкого места. Нужно будет сбалансировать необходимость единообразия, с одной стороны, и то разнообразие и инновации, которые следует поддерживать, — с другой. Одним из способов сделать это является применение гибридной модели, где для большей части API документация централизована, но команды также имеют право создавать процессы обучения, если они пробуют что-то новое.

- *Когда нужно писать документацию?* Как ни удивительно, существуют разные подходы к тому, когда команда должна начинать создание документации. Если писать ее заранее, то получится дороже из-за большой вероятности изменений в дизайне и реализации, но окупится тем, что можно на ранних этапах обнаружить проблемы в использовании API. Необходимо будет решить, можно ли безопасно децентрализовать это решение или ему нужно более централизованное управление. Например, нужна ли каждому API независимо от его предназначения документация, написанная заранее? Или лучше каждой команде принимать это решение самостоятельно, обдумав все за и против?

Разработка

Столп «Разработка API» включает в себя все решения, принимаемые в ходе воплощения API. Сложно разработать *реализацию* API, полностью соответствующую дизайну его *интерфейса*. У столпа «Разработка» невероятно большое пространство решений. Вам понадобится решать, какие технологические продукты и фреймворки использовать для реализации API, как будет выглядеть архитектура этой реализации, какие языки программирования и конфигурации применять и как API должен вести себя во время запуска. Другими словами, вам нужно будет разработать программное обеспечение API.

Пользователям API неважно, как он реализован. Для них не имеют значения ваши решения по языкам программирования, инструментам, базам данным и дизайну ПО. Им важен только итоговый продукт. Если API делает то, что от него требуется, и так, как требуется, пользователи будут довольны. А то, что вы задействовали какую-то конкретную базу данных или фреймворк, для них просто полезная информация.

То, что пользователям неважно, что вы выберете, не значит, что ваши решения по разработке не имеют значения. Они очень значимы, в особенности для тех людей, которые будут создавать, поддерживать и изменять API в течение всего времени его работы. Если вы выберете сложные в использовании технологии или малораспространенные языки, не понятные никому в компании, поддерживать API будет гораздо сложнее. А если применить слишком неудобные инструментальные средства или языки, на которых невероятно тяжело программировать, будет сложно вносить изменения в API.

Думая о разработке API, люди часто фокусируются только на решениях, которые надо принимать для создания первой версии продукта. Они действительно значимы, но это лишь небольшая часть работы. Более важная цель — принимать решения по разработке, которые улучшат качество, расширяемость, изменяемость и удобство в сопровождении вашего API в течение всего срока его работы. Для разработки ПО с учетом этого нужны опыт и навыки, поэтому необходимы инвестиции в специалистов и инструменты. В конце концов, написать простую программу может любой человек после нескольких занятий по программированию. Но для того, чтобы

создать программы, работающие при увеличении масштаба или одновременном использовании множеством потребителей и успешно справляющиеся со всеми возникающими сложностями, но при этом остающиеся доступными для поддержки и внесения изменений, нужен настоящий профессионал.

Нет никаких жестких правил по разработке и архитектуре ПО для API, так же как их нет для разработки ПО в целом. Разумеется, существует много указаний, принципов и советов, как *следует* разрабатывать ПО. В целом в сфере разработки API можно применять любые качественные методы разработки ПО, реализуемого на сервере. Особенностью API является жизнеспособная система фреймворков, специфичных для API, и инструментальных средств для разработки программ. Рассмотрим возможности, открывающиеся при задействовании этих полезных средств.

Применение фреймворков и инструментов

В любом типичном процессе разработки используется большое количество инструментов, но в случае разработки API нас интересует особая категория — инструменты, помогающие уменьшить усилия по принятию решений и разработке новой реализации API. В эту категорию входят фреймворки, облегчающие работу по написанию кода для API, а также отдельные инструменты, предлагающие реализацию без кода или с небольшим количеством кода.

Одним из самых популярных инструментов в сфере управления API является шлюз для API. Он реализуется на сервере и разработан для уменьшения затрат на развертывание API в сети. Обычно шлюзы разрабатываются очень расширяемыми, надежными и безопасными — как правило, основным мотивом для их применения в системной архитектуре является именно укрепление безопасности реализации API. Это полезный инструмент, потому что он заметно снижает затраты на разработку программы с API.

Затраты на разработку уменьшаются при использовании таких инструментов, как шлюз, потому что они предназначены для решения большинства ваших проблем и зачастую не требуют много сил на программирование. Например, обычный шлюз для API на HTTP должен быть готов принять запрос с порта HTTPS, проанализировать тела запросов JSON, начать взаимодействовать с базой данных сразу после начала работы и потребовать небольших изменений конфигурации перед запуском. Разумеется, все это происходит не по волшебству — кто-то должен запрограммировать этот инструмент на исполнение данных задач. В итоге вы передаете затраты по разработке API внешней компании.

Когда инструменты работают качественно, это подарок небес. Но проблема с такими инструментами, как шлюзы для API, заключается в том, что они могут исполнять только то, для чего разработаны. Как и при использовании любой автоматики, в этом случае гибкость ограничена функциями техники. Поэтому выбор правильного инструмента очень важен в сфере разработки. Если ваша стратегия

API и дизайн интерфейса приводят к необходимости совершать нестандартные поступки, вам придется заниматься разработкой самостоятельно.

Отношения между интерфейсом и реализацией

Первоначальной целью вашей разработки является поддержка стратегии и дизайна интерфейса. Неважно, насколько у вас прекрасная архитектура и удобный для поддержки код, если API не выполняет того, что должен по замыслу разработчика интерфейса, это провал. Из этого утверждения можно сделать два вывода.

1. Соответствие *опубликованному* дизайну интерфейса — важный параметр качества для реализации.
2. Необходимо обновлять реализацию каждый раз, когда вы изменяете интерфейс.

Это означает, что без дизайна интерфейса нельзя разрабатывать API. А еще что вашим разработчикам нужен надежный способ узнавать, как выглядит интерфейс и когда он изменяется. Если команда по дизайну выбирает непрактичный интерфейс, который невозможно адекватно реализовать, это ставит продукт с API под удар. Если интерфейс и реализацию разрабатывает один человек или разные, но работающие в одной команде, это не очень страшно, но в ином случае следует убедиться в том, что в ваш метод разработки API входит перепроверка интерфейса командой по реализации.

ИСПОЛЬЗОВАНИЕ ОПИСАНИЙ API, БЛИЗКИХ К КОДУ

Одним из способов улучшить вероятность синхронизации реализации и интерфейса является интеграция описания интерфейса в саму реализацию. Например, если ваш формат описания API передает дизайн интерфейса, можно сохранить этот файл в репозитории исходного кода или даже разработать автоматизированный тест, который будет подтверждать соответствие интерфейсу. Развивая эту идею, можно даже создать скелет программы, основываясь на формате описания, хотя на самом деле это эффективно только для первой версии.

Можно применить и противоположный подход: вместо получения описания интерфейса и использования его вместе с кодом внедрить описание интерфейса вручную прямо в код. Например, некоторые фреймворки позволяют задействовать аннотации, описывающие интерфейс API. Сочетание кода и аннотации способно стать источником истины для интерфейса, и его можно даже применять для создания документации по API. Любой из этих подходов поможет вам убедиться в том, что реализация API не нарушает обещаний, данных дизайном интерфейса.

Ключевые решения по разработке

Что можно использовать для реализации? Это центральный вопрос в рамках руководства реализацией. Он включает в себя много решений. Из каких баз данных,

языков программирования и компонентов системы можно выбирать? Какие библиотеки, фреймворки и инструменты стоит задействовать для поддержки работы? С какими поставщиками надо сотрудничать? Можно ли использовать код из открытых источников?

На момент написания книги многие интересовались децентрализацией таких решений, чтобы улучшить локальную оптимизацию. Многие компании сообщали нам, что их API стали эффективнее, их стало проще создавать и поддерживать после того, как командам дали больше свободы при разработке реализации. Но, как всегда, у децентрализации есть своя цена: становится меньше единообразия и возможностей провести оптимизацию на системном уровне. На практике это означает, что сотрудникам станет сложнее переходить из одной команды в другую, к тому же будет меньше возможностей экономить при увеличении масштаба API и контролировать всю реализацию в целом.

Наш опыт свидетельствует: стоит дать командам больше свободы в разработке реализации, но необходимо обдумать, какой уровень свободы в принятии решений сможет выдержать ваша система. Одним из способов упростить поддержку децентрализованных решений по реализации является централизация элемента выбора — вы централизуете поиск вариантов технологий, но децентрализуете выбор каждой команды.

Тестирование

Если для вас важно качество API, необходимо будет потратить силы на тестирование. Столп «Тестирование API» включает в себя решения о том, *что* нужно тестировать и *как* это делать. В целом тестирование API не сильно отличается от обычного тестирования любого проекта программного обеспечения. Вам нужно будет применять методики проверки качества ПО к *реализации*, *интерфейсу* и готовому *экземпляру* API так же, как к традиционному приложению ПО. Но, как и разработка, тестирование в сфере API проходит несколько иначе, чем везде, благодаря системе инструментов, библиотек и вспомогательных приложений.

Что требует тестирования

Главная цель тестирования API — убедиться, что он может достичь стратегической цели, которую вы определили в процессе его создания. Но как мы уже видели на протяжении главы, эта стратегическая цель появляется благодаря работе, основанной на решениях, во всех десяти столпах. Таким образом, вторая цель тестирования API — убедиться, что вся проделанная работа была достаточно качественной для того, чтобы поддержать вашу стратегию. Например, если API сложен для использования и изучения, это может повлиять на стратегическую цель «привлечь больше пользователей API». Это также означает, что необходимо

определить конкретные тесты, чтобы добиться высокого качества интерфейса. Нужно также протестировать проделанную работу на внутреннее единообразие. Например, может понадобиться проверить, что разработанная вами реализация совпадает с интерфейсом.

Вот список типичных категорий тестов, применяемых владельцами API.

- *Тестирование удобства использования и опыта пользователя*. Поиск проблем с применением в интерфейсе, документации и на этапе обнаружения. Например, *предоставить разработчикам документацию по API и наблюдать через плечо, как они пишут по ней клиентский код*.
- *Тестирование отдельных единиц*. Поиск ошибок в реализации на уровне деталей. Например, *запустить тест JUnit на каждом уровне кода реализации API, написанной на Java*.
- *Тестирование интеграции*. Поиск ошибок в реализации и интерфейсе с помощью запросов к готовому экземпляру API. Например, *запустить тестовый скрипт, отправляющий запросы API к запущенному экземпляру API в среде разработки*.
- *Тестирование работы и загрузки программы*. Поиск нефункциональных ошибок в уже опубликованных экземплярах и их окружении. Например, *запустить скрипт, тестирующий работу программы и симулирующий загрузку на уровне производства в запущенном экземпляре API в тестовой среде, которая имитирует условия производства*.
- *Тестирование безопасности*. Поиск уязвимых мест в интерфейсе, реализации и готовом экземпляре API. Например, *нанять ударную группу для поиска уязвимых мест в запущенном экземпляре API в безопасной тестовой среде*.
- *Тестирование производства*. Поиск ошибок в использовании, функциональности и работе программы после того, как продукт с API был опубликован в производственной среде. Например, *выполнить многомерный тест с применением документации по API, в которой разным пользователям предоставляются несколько разные версии содержимого, и улучшить документацию, основываясь на результатах*.

Это, разумеется, не исчерпывающий список, существует множество других тестов. И даже описанные здесь тесты можно разбить на много подкатегорий. Главное стратегическое решение в области тестирования — определить, сколько тестов будет достаточно. К счастью, ваша стратегия по API может помочь принять это решение. Если вы выбрали в качестве приоритетов качество и единообразие, то может оказаться, что нужно потратить много времени и денег на тестирование API перед публикацией. Но если у вас экспериментальный API, можете рискнуть и выполнить минимальное количество тестов. Например, совершенно нормально, если у API для платежей солидного банка и API новой социальной сети будут совершенно разные по масштабам подходы к тестированию.

Инструменты для тестирования API

Тестирование может оказаться дорогим, поэтому полезно найти инструменты, упрощающие улучшение качества продукта. Например, некоторые организации успешно применяют метод опоры на тест, когда тесты пишутся *до* создания реализации или интерфейса. Цель такого подхода состоит в том, чтобы изменить производственную культуру команды на такую, в которой тесты являются центральным элементом, чтобы все решения по дизайну отражались в реализации, сильнее адаптированной под тесты. Его результатом становится API более высокого качества, потому что в него заранее встроено соответствие всем тестам.

Кроме этого, для уменьшения затрат на тестирование можно использовать инструментальные средства и автоматизацию. Самые полезные инструменты в арсенале для тестирования API — это симуляторы и дублеры или имитации тестов. Поскольку взаимосвязанная природа программного обеспечения API затрудняет тестирование отдельных частей, вам понадобится как-то имитировать другие компоненты. В частности, могут понадобиться инструменты для имитации каждого из этих компонентов.

- *Клиент*. В ходе тестирования API вам понадобится имитация запросов, которые будут приходить от клиентов. Существует множество инструментов, способных это сделать. Лучшие из них можно конфигурировать так, чтобы приблизиться к типам сообщений и количеству трафика, которые вы получите в реальности.
- *Серверная часть*. С большой вероятностью у вашего API будут свои взаимосвязи. Например, набор внутренних API, база данных или какой-либо внешний ресурс. Для проведения тестов на интеграцию, безопасность и проверки работы программы вам наверняка понадобится как-то симулировать эти взаимосвязи.
- *Среда*. Понадобится также симулировать производственную среду, когда вы будете проводить тесты перед запуском API. Много лет назад это потребовало бы поддержки регламентированной среды и управления ею исключительно для этих целей. Но сейчас многие организации используют инструменты виртуализации для снижения расходов на воссоздание среды для тестирования.
- *API*. Иногда вам может понадобиться даже симулировать собственный API. Например, когда нужно протестировать один из поддерживающих компонентов — скажем, программу для просмотра API на портале разработки, отправляющую запросы. К тому же симуляция вашего API — это ценный ресурс, который можно предоставить пользователям. Такую симуляцию часто называют *песочницей*, скорее всего, потому, что это место, где разработчики могут поиграть с API и данными без каких-либо последствий. Она требует некоторых вложений, но может значительно улучшить опыт разработчика для вашего API.

Как приблизить песочницу к условиям производства

Выпуская песочницу для пользователей API, убедитесь, что она максимально точно воспроизводит среду производства. Пользователи окажутся гораздо довольнее, если единственное изменение, которое от них потребуется, — это перенести написанный код в экземпляр, запущенный в производство. Потратить много времени и сил на выявление неполадок в интеграции API и обнаружить, что готовый экземпляр выглядит и ведет себя по-другому, для них будет крайне удручающе.

Ключевые решения по тестированию

- *Где должно проходить тестирование?* В последние годы процесс тестирования становится все более и более децентрализованным. Важное решение — определить, насколько вы хотите централизовать каждый из описанных нами тестов для API. Централизация процесса тестирования позволит вам лучше контролировать его, но появится риск создания узкого места. Частично его можно уменьшить с помощью автоматизации, но тогда нужно будет решить, кто займется конфигурацией и поддержкой системы автоматики. Большинство организаций применяют обе системы — и централизованную, и децентрализованную. Мы советуем децентрализовать ранние стадии тестирования для ускорения процесса и централизовать поздние из соображений безопасности.
- *Сколько тестов вам необходимо?* Даже если вы решили, что отдельные команды могут создавать собственные тесты, можете централизованно принять решение по поводу минимально необходимого уровня тестирования. Например, некоторые компании используют инструменты покрытия кода, предоставляющие отчеты о том, какой объем кода был протестирован. С точки зрения параметров и качества покрытие кода неидеально, но это количественно измеримая величина и она позволяет установить минимальный порог, который должен присутствовать в работе всех команд. Если у вас есть подходящие сотрудники, можете децентрализовать и это решение и позволить отдельным командам по API решать, что будет правильно для их API.

Развертывание

Реализация API — это то, что воплощает *интерфейс*, но ее нужно правильно развернуть, чтобы она приносила пользу. Столп «Развертывание API» включает в себя всю работу по перемещению реализации API в среду, где ее сможет использовать ваша целевая аудитория. Развернутый API — это работающий *экземпляр*, и вам может понадобиться управлять несколькими его копиями, чтобы API работал как следует. Сложность работы по развертыванию API состоит в том, чтобы убедиться, что все экземпляры ведут себя единообразно, доступны пользователям и их можно максимально легко изменить.

Развертывание программного обеспечения сейчас гораздо сложнее, чем в прошлом, в основном из-за того, что архитектура ПО становится все сложнее и связанных между собой взаимозависимостей стало больше, чем когда-либо. Кроме того, компании сейчас довольно многого ожидают от систем в смысле доступности и надежности — они хотят, чтобы все работало постоянно и при каждом запросе. И не забудьте: они требуют, чтобы изменения выполнялись незамедлительно. Чтобы соответствовать этим ожиданиям, вам придется принимать качественные решения по поводу системы развертывания API.

Как разобраться с неопределенностью

Улучшение качества развертывания API подразумевает, что все ваши экземпляры API ведут себя так, как пользователи от них ожидают. Очевидно, большая часть работы над этим происходит вне сферы развертывания API. Необходимо написать качественный, четкий код реализации и тщательно его протестировать, если вы хотите уменьшить количество проблем при производстве. Но иногда даже после принятия всех этих профилактических мер проблемы все равно появляются. Это происходит потому, что после публикации API приходится разбираться с неопределенностью и непредсказуемостью.

Например, что произойдет в случае внезапного резкого роста спроса на ваш продукт с API? Справятся ли программы с возросшей загрузкой? Что случится, если оператор случайно опубликует старую версию программы или внешний сервис, от которого зависит API, внезапно станет недоступен? Неопределенность может проявиться во многих случаях: из-за пользователей, человеческой ошибки, аппаратной составляющей, внешних взаимозависимостей. Для усиления безопасности развертывания вам нужно одновременно использовать два противоположных друг другу подхода: сокращать количество неопределенности и в то же время принимать ее.

Распространенный метод сокращения неопределенности при развертывании API — применение принципа *неизменности*. Неизменность — это невозможность измениться, другими словами, состояние «только для чтения». Ее можно задействовать по-разному. Например, если вы не разрешаете операторам менять серверные переменные среды или устанавливать программное обеспечение вручную, говорите, что у вас неизменная инфраструктура. Можно также создать неизменные программные пакеты для развертывания API, их нельзя редактировать — только удалять. Принцип неизменности укрепляет безопасность, потому что благодаря ему уменьшается неопределенность, вызванная действиями человека.

Однако у вас никогда не получится полностью избавиться от неопределенности. Невозможно предсказать все случайности и проверить все возможности. Поэтому важной частью вашей работы по принятию решений будут попытки

понять, как сохранить систему в безопасности, даже если происходит что-то неожиданное. Часть этой работы происходит на уровне реализации API (например, написание защитного кода), часть — на уровне системы (например, разработка устойчивой системной архитектуры), но довольно многое необходимо проделать на уровне развертывания и работы с программой. Например, если постоянно наблюдать за состоянием готовых экземпляров API и системных ресурсов, можно найти проблемы и исправить их до того, как они повлияют на ваших пользователей.

Разработка устойчивого программного обеспечения

Один из наших любимых источников информации по укреплению безопасности развернутого API — книга Майкла Найгарда «Реализуй это!» (Release It!, издательство Pragmatic Bookshelf). Если вы ее еще не читали, обязательно прочтите. Это просто сокровищница схем реализации и развертывания, необходимых для укрепления безопасности и устойчивости вашего продукта с API.

Одна из неопределенностей, которую вам придется принять, — это изменения API. Хотя было бы проще заморозить все изменения, лишь только API начнет работать как следует, изменения — это неизбежность, к которой вы должны быть готовы. Целью работы по развертыванию должно быть максимально быстрое внесение изменений.

Автоматизация развертывания

Существует только два способа ускорить развертывание: выполнять меньше работы или делать ее быстрее. Иногда этого можно добиться, изменив процесс работы, например выбрав другой подход к ней или другую установившуюся практику. Это сложно, но действительно может помочь. Мы углубимся подробнее в эту тему позже, когда будем говорить о сотрудниках и командах в главе 7.

Другой способ ускориться — заменить задания по развертыванию, выполняемые людьми, на выполняемые автоматически. Например, если автоматизировать процесс тестирования, создания и развертывания кода API, можно будет выпускать программы одним нажатием кнопки.

Инструменты и автоматизация развертывания могут показаться быстрым и выигрышным путем, но помните о долгосрочных затратах. Вводить автоматизацию в рабочий процесс — это как проводить механизацию на фабрике: эффективность улучшается, но процесс становится менее гибким. Автоматизация также приносит с собой расходы на установку и поддержку. Вряд ли она начнет работать «сразу после распаковки» и сама приспособится к меняющимся требованиям и окружению. Поэтому, улучшая систему с помощью автоматизации, будьте готовы к затратам на поддержку автоматики и ее удаление.

APIOps: DevOps для API

Многое из описанного в этом разделе совпадает с принципами культуры DevOps. Существует даже новый термин для применения методов DevOps конкретно к API — APIOps. Нам кажется, что DevOps прекрасно подходит к сфере API и у этого подхода есть чему поучиться, как бы вы его ни называли.

Ключевые решения по развертыванию

- *Кто решает, когда пора публиковать API?* Вопрос о том, кто будет публиковать API, — центральный в руководстве развертыванием. Если у вас есть высококвалифицированные сотрудники, которым вы доверяете, устойчивая к сбоям архитектура и в вашей сфере бизнеса простительна случайная неудача, можете полностью децентрализовать это решение. В ином случае надо будет определить, какие его части следует централизовать. Распределение этого решения обычно очень детальное. Например, вы можете доверить «кнопку публикации» участникам команд, которым доверяете, или публиковать программу в тестовой среде, где централизованная команда сможет определить, подходит или не подходит решение. Распределяйте решение так, чтобы оно вписывалось в ваши ограничения и позволяло сделать это максимально быстро, но соблюдая должный уровень безопасности.
- *С помощью чего происходит развертывание?* В последние годы приобрел особую важность вопрос, каким образом программное обеспечение развертывается и доставляется. Оказывается, что это решение может постепенно развернуть всю систему в другом направлении. Например, растущая популярность контейнеризированного развертывания удешевила и упростила создание неизменных, готовых к развертыванию в облаке копий программ. Необходимо обдумать, кто будет принимать это важное для вашей организации решение. Если оно будет децентрализовано и оптимизировано локально, тот, кто его принимает, может не до конца понять, каким образом оно повлияет на безопасность, совместимость и масштаб. Но если его полностью централизовать, то при этом может не найтись решения, подходящего для всех реализаций и всего программного обеспечения, которые нужно разместить. Как обычно, в этом случае пригодятся ограничение вариантов и распределение элементов решения.

Кроме вопроса о централизации, нужно будет также обдумать, какая команда лучше всего подходит для принятия самых качественных решений. Должны ли варианты формата развертывания определять команды по операциям и межплатформенному ПО? Или команда по архитектуре? А может, команда по реализации? Ключевым фактором здесь является распределение кадров: участники каких команд смогут лучше оценить ситуацию?

Безопасность

API упрощают связь между программным обеспечением, но в то же время приносят новые проблемы. Открытость API делает их потенциальной мишенью и создает новое пространство для атаки. Появилось больше дверей, которые можно открыть, и за ними больше сокровищ! Поэтому вам нужно потратить какое-то время на укрепление безопасности своего API. Столп «Безопасность API» включает в себя решения, которые нужно принять, чтобы:

- защитить систему, клиентов API и конечных пользователей от возможных угроз;
- поддержать законное использование API;
- защитить личные данные и ресурсы.

Эти три простые цели скрывают невероятно сложную предметную область. Одна из главных ошибок владельцев продуктов с API — считать, что безопасность API держится на нескольких технологических решениях. Мы не хотим сказать, что технологические решения в сфере безопасности не имеют значения, — разумеется, это не так! Но если вы действительно хотите укрепить столп «Безопасность API», необходимо расширить рабочее окружение решений по поводу безопасности.

Комплексный подход

Чтобы заметно укрепить безопасность API, необходимо сделать ее частью процесса принятия решений по всем описанным в этой главе столпам. Для этого важно применять характеристики безопасности во время выполнения программы. Вначале нужно найти идентификаторы, распознать клиентов и конечных пользователей, авторизовать использование и ограничить скорость. Многое из этого можно выполнить самостоятельно. Можно также задействовать более безопасный и быстрый подход — применить инструменты и библиотеки, которые сделают все это за вас.

Безопасность API включает в себя также множество действий за пределами взаимодействия клиента и программного обеспечения API. Ее могут укрепить изменения в производственной культуре, например поощрение подхода «безопасность прежде всего» у инженеров и дизайнеров. Можно ввести процессы, которые предотвращают внесение небезопасных изменений в готовую программу. Пересмотреть документацию, чтобы убедиться, что информация не просачивается за ее пределы. Можно обучить сотрудников отдела продаж и техподдержки не допускать случайного обнародования личных данных и не поддаваться психологическим атакам.

Конечно, все сказанное не ограничивается традиционной сферой безопасности API. Но ведь она не может существовать самостоятельно. Это часть взаимосвязанной системы, и ее следует считать одним из элементов целостного подхода к безопасности компании. Бессмысленно делать вид, что она не связана с общей стратегией безопасности. Те, кто захочет воспользоваться вашей системой в своих целях, явно так считать не будут.

Таким образом, вам понадобится принимать решения о том, как интегрировать работу с API в общую стратегию безопасности внутри компании. Иногда сделать это довольно легко, иногда — сложнее. Одна из больших сложностей API — найти баланс между желаниями сделать его открытым и простым в использовании и закрыть к нему доступ. Как далеко вы зайдете в каждом из этих стремлений, должно зависеть от вашей организационной стратегии и стратегических целей API.

Ключевые решения по безопасности

- *Какие решения требуют одобрения?* Все решения, принимаемые в вашей организации, могут сделать систему уязвимой. Но тщательно проверять их все невозможно. Необходимо определить, какие из них делают API наиболее безопасным, и убедиться, что они самые качественные. Это во многом будет зависеть от общей обстановки. Есть ли зоны в вашей архитектуре, которым вы доверяете, но которым нужны безопасные границы? Дизайнеры и разработчики уже имеют опыт работы с системами безопасности? Происходит ли вся работа «дома», или вы сотрудничаете с внешними разработчиками? Все эти факторы могут изменить выбор решений, требующих одобрения.
- *Сколько безопасности нужно API?* Бóльшая часть аппарата безопасности — это стандартизация рабочих решений для защиты системы и ее пользователей. Например, у вас могут существовать правила по поводу того, где могут храниться файлы и какие алгоритмы шифрования применять. Усиление стандартизации уменьшает возможность использования инноваций. В контексте безопасности это обычно оправдывается последствиями одного неудачного решения, но не все API требуют одинаковой тщательности проверки и одного и того же уровня защиты. Например, API, который внешние разработчики используют для финансовых операций, потребует больше вложений в безопасность, чем API для сохранения данных о работе приложения. Но кто должен принимать это решение?

Как обычно, основные критерии — общая обстановка, талант и стратегия. Централизация этого решения позволяет команде по безопасности примерно оценить ситуацию, основываясь на собственном понимании системного окружения. Однако иногда из-за таких общих правил что-то может просочиться в щели, особенно когда команды по API применяют инновации, которые централизованная команда могла не учесть. Если принятие этого решения — прерогатива команд, они сами могут оценить ситуацию, но это требует от них высокого уровня знаний по безопасности. И наконец, если вы работаете в сфере, где безопасность и снижение рисков являются основными приоритетами, в итоге может оказаться, что у вас везде высший уровень безопасности вне зависимости от локального окружения и влияния на скорость выполнения работы.

Мониторинг

Важно, чтобы ваш продукт с API обладал таким качеством, как наблюдаемость состояния. Нельзя успешно управлять API, не имея точной и актуальной информации

о том, как он работает и как его используют. Столп «Мониторинг API» посвящен тому, как сделать эту информацию доступной и полезной. Со временем при расширении системы API мониторинг становится для управления API таким же важным, как и дизайн интерфейса или разработка реализации, ведь, если бродить в темноте, можно споткнуться и упасть.

В готовом экземпляре API нужно отслеживать много моментов:

- проблемы (например, ошибки, системные сбои, предупреждения, неполадки);
- состояние системы (например, ЦП, память, ввод-вывод, состояние жесткого диска);
- состояние API (например, время с начала работы, общее состояние, количество выполненных заданий);
- архив сообщений (например, тела сообщений с запросами и ответами, заголовки сообщений, метаданные);
- данные об использовании (например, общее количество запросов, применение устройства/ресурсов, количество запросов на одного пользователя).

Как узнать больше о мониторинге

За исключением пунктов об API и отслеживании использования, описанные нами параметры применяются не только для компонентов программного обеспечения с API. Если вам нужно качественное руководство по мониторингу сетевых компонентов ПО, советуем прочитать Site Reliability Engineering[1] от Google (O'Reilly). Эта книга знакомит с разработкой систем ПО и включает в себя очень понятный список пунктов, которые требуют контроля. Еще один хороший ресурс — «Метод RED» от компании Weaveworks (http://bit.ly/2Rd4fHF). Он определяет три категории параметров микросервиса: оценку (rate), количество ошибок (errors) и продолжительность (duration).

Каждая из этих групп параметров по-своему поможет вашему API. Данные о состоянии и проблемах помогут вам найти ошибки и разобраться с ними, что может уменьшить последствия возникающих проблем. Данные об обработке сообщений могут помочь проверить работу API на наличие неполадок на системном уровне. Данные об использовании помогают улучшить продукт, показывая, как именно пользователи работают с API. Но прежде всего нужно будет обеспечить доступ к этим данным. И конечно, следует убедиться, что ваш метод сбора и представления данных надежен.

Чем больше данных вы соберете, тем больше получите возможностей улучшить продукт. Поэтому в идеале нужно получать непрекращающийся поток данных по

[1] *Бейер Б., Джоунс К., Петофф Дж., Мёрфи Н. Р.* Site Reliability Engineering. Надежность и безотказность как в Google. — СПб.: Питер, 2019.

API. Затраты на производство, сбор, хранение и анализ этих данных неизбежны, но иногда они становятся просто неподъемными. Например, если время полного цикла обработки запросов API удваивается из-за того, что ему нужно скопировать данные, нужно либо сократить количество журналов, либо найти лучший способ делать это. Одно из важнейших решений — что вы можете позволить себе отслеживать, зная, каковы будут затраты на мониторинг.

Еще одно важное решение — насколько совместимым должен быть мониторинг API. Если API представляет данные таким образом, что они совместимы с другими инструментами и со стандартами в вашей сфере деятельности или организации, задействовать их будет гораздо легче. Разработка системы мониторинга во многом похожа на разработку интерфейса. Если интерфейс для мониторинга уникален, будет сложнее научиться использовать его и собирать данные. Это не страшно, если у вас один API и вы самостоятельно его отслеживаете, но если API десятки, а то и сотни, придется пожертвовать уникальностью. Мы подробнее рассмотрим эту идею в главе 6.

Ключевые решения по мониторингу

- *Что нужно отслеживать?* Это важное решение. Поначалу можете оставить его на усмотрение отдельных команд, но при расширении API выгоднее будет единообразие. Совместимые данные дадут больше возможностей наблюдать влияние на систему и ее поведение. Таким образом, вам понадобится централизовать какие-то элементы данного решения.
- *Как собирать, анализировать и передавать данные?* Централизация решения по осуществлению мониторинга облегчит работу с данными по API, но может ограничить свободу отдельных команд. Необходимо решить, какие из элементов этого решения централизовать, а какие — распределить между командами и децентрализовать. И еще важнее оно становится, когда речь идет о данных, требующих защиты.

Обнаружение и продвижение

API имеет ценность только тогда, когда им пользуются, но для этого его сначала надо найти. Столп «Обнаружение API» посвящен тому, как облегчить вашей целевой аудитории обнаружение API. Для этого нужно помочь пользователям быстро понять, что делает API, чем он может быть им полезен, как начать работу и как в нем разобраться. Обнаружение — важный фактор для опыта разработчика. Чтобы ваш продукт было действительно легко найти, понадобятся качественные решения по дизайну, документированию и реализации, а также дополнительная работа именно по обнаружению.

В сфере API есть два основных типа обнаружения: в период разработки и в период работы приложения. Обнаружение в период разработки направлено на то, чтобы пользователям было проще узнать о продукте с API. Оно включает в себя

информирование о его существовании, функциональности и проблемах, которые он может решить. А обнаружение в период работы приложения происходит уже после того, как API был развернут. Оно помогает программному обеспечению обнаруживать местонахождение API в сети, основываясь на каких-либо фильтрах или параметрах. Обнаружение в период разработки направлено на людей, и для него в основном требуется работа по продвижению и маркетингу. Обнаружение в период работы приложения направлено на автоматические клиенты и основывается на инструментальных средствах и автоматизации. Подробнее рассмотрим каждый из вариантов.

Обнаружение в период работы приложения

Это способ улучшения изменяемости вашей системы API. Если у вас хорошая система обнаружения в период работы, то вы можете менять местонахождение экземпляров API, и это почти не повлияет на клиентов. Это особенно полезно, если запущено много экземпляров одновременно, — например, архитектура на основе микросервисов часто поддерживает обнаружение в период работы, чтобы сервисы было проще находить. Для этого требуется работа в основном в сферах разработки и развертывания API.

Обнаружение в период работы — это полезный механизм, и его стоит ввести, если вы управляете сложной системой API. У нас недостаточно времени, чтобы углубляться в детали его реализации, но для этого нужны инвестиции в технологии на уровне системы, API и пользователей. Когда в этой книге упоминается столп «Обнаружение API», то имеется в виду в основном обнаружение в ходе разработки.

Обнаружение в период разработки

Чтобы помочь людям узнать о своем API в период разработки, нужно убедиться, что им доступен нужный тип документации. Пользователи должны быстро получать доступ к документации о том, что делает API и какие проблемы решает. Такая копия маркетинга продукта является важной, но не единственной частью обнаружения. Главная часть этого столпа — помощь пользователям, которые ищут информацию. Для достижения успеха вам нужно сотрудничать с сообществом пользователей и рекламировать им свой продукт. Как вы это делаете, зависит от окружения вашей пользовательской базы.

- *Внешние API.* Если API предназначен в основном для людей, не работающих в вашей группе или организации, понадобятся инвестиции, чтобы донести до них свой посыл. Нужно будет представить им продукт с API примерно так же, как вы представили бы программное обеспечение: используя оптимизацию поисковых механизмов, спонсирование мероприятий, взаимодействие с сообществом и рекламу. Ваша цель — убедиться, что все потенциальные пользователи API понимают, как этот продукт поможет им в решении их проблем. Конечно,

конкретные маркетинговые приемы, которые вы применяете, будут во многом зависеть от контекста, характера API и целевой аудитории.

Например, если вы разрабатываете продукт с API для SMS и конкурируете за внимание пользователей-разработчиков, то нужно рекламировать его там, где могут оказаться потенциальные клиенты: на конференциях веб-разработчиков, в блогах на тему двухфакторной аутентификации или на конференциях по телекоммуникациям. Если ЦА — независимые разработчики, можно положиться на рекламу в Сети, но если вы хотите привлечь крупные предприятия, потребуется вложить деньги в команду менеджеров по продажам, создающих сеть из людей, с которыми они общаются. Если конкуренция на вашем рынке велика, придется потрудиться над созданием у API отличительных черт, а если продукт уникален, то нужно лишь немного поколдовать в области оптимизации поисковых систем, и к вам придут клиенты. В области маркетинга сопутствующие факторы играют главную роль.

- *Внутренние API.* Если API пользуются ваши собственные разработчики, то аудитория вам обеспечена. Но это не значит, что можно не обращать внимания на вероятность обнаружения вашего API. Внутренний API тоже надо найти, чтобы применить, и если постепенно это становится все сложнее, возникнут проблемы. Если внутренние разработчики о нем не знают, они не будут им пользоваться и даже могут потратить время на создание другого API, полностью дублирующего его функции. Появление внутреннего конкурентного рынка API обычно является хорошим знаком, и возможность повторного использования часто переоценивают. Но если ваши сотрудники дублируют качественный API только потому, что не знают о нем, — это пустая трата ресурсов.

  Внутренние API можно рекламировать примерно так же, как и внешние. Отличается только система маркетинга. Если внешний API направлен, например, на поисковик Google, то внутреннему может оказаться достаточно просто попасть в корпоративную электронную таблицу, где перечисляются все API компании. Чтобы прорекламировать внешний API, эффективно будет спонсировать конференцию разработчиков, а для рекламы внутреннего стоит просто пообщаться со всеми командами разработчиков в организации и научить им пользоваться.

Большой проблемой в рекламе внутреннего API бывает отсутствие стандартизации внутри компании. Может оказаться даже, что в большой организации существует больше одного списка API. Чтобы выполнить свою работу качественно, вам нужно убедиться, что API числится во всех важных списках и реестрах. Сложно может быть узнать обо всех командах разработчиков в вашей компании и уделить всем время, но если для вас важно, чтобы API использовался, то стоит сделать это.

Ключевые решения по обнаружению

- *Каким будет опыт обнаружения?* Необходимо обеспечить качественный опыт обнаружения API для ваших пользователей. Для этого нужно выбрать инструменты обнаружения, опыт пользователя и целевую аудиторию. С ростом вашего

API понадобится также решить, насколько единообразным должен быть этот опыт, и если нужна высокая степень совпадений, понадобится централизовать эти решения.

- *Где и как будут рекламироваться API?* Продвижение API требует затрат сил и времени, поэтому нужно выбрать того, кто решит, когда начинать его рекламировать. Это решение можно оставить на усмотрение отдельных команд по API или централизовать. Если вы передаете решение командам, следует убедиться, что никакие централизованные инструменты и процессы для обнаружения не мешают им достичь своих целей.
- *Как поддерживается качество опыта обнаружения?* С течением времени API меняются и информация в любой системе обнаружения становится менее точной. Кто должен проверять эту информацию? Кто должен следить, не ухудшился ли опыт пользователя с момента публикации API?

Управление изменениями

Если бы не нужно было изменять API, управлять ими стало бы гораздо проще. Но API нужно исправлять, обновлять и улучшать. Вы должны быть всегда готовы вносить изменения. Столп «Управление изменениями» включает в себя все решения по планированию и управлению, необходимые при изменениях. Это жизненно важная и сложная сфера — на самом деле вся эта книга посвящена именно управлению изменениями.

В целом у управления изменениями есть три цели:

- выбор лучших вариантов изменений;
- максимально быстрая реализация этих изменений;
- внесение изменений таким образом, чтобы ничего не нарушить в работе программы.

Выбор лучших вариантов вызывает изменения, которые позволяют вам достичь стратегических целей API. Но ради этого нужно научиться расставлять приоритеты и планировать изменения, основываясь на затратах, внешних факторах и ограничениях. Это и есть работа по управлению продуктом, и это одна из причин, по которым мы ввели в главе 3 концепцию «API как продукт». Если вы поставите четкие стратегические цели и определите целевое сообщество пользователей, то сможете принимать более удачные решения по поводу того, какие изменения самые полезные. Узнав больше о том, что нужно делать в каждом из остальных девяти столпов, вы сможете лучше оценивать затраты. Вооружившись информацией о затратах и полезности, сможете принимать разумные решения относительно вашего продукта с API.

Балансировать между безопасностью и скоростью изменений непросто, но необходимо. Каждое из решений, которые вы принимаете в столпах продукта с API,

на что-то влияет. Фокус в том, чтобы стараться принимать решения, максимально увеличивающие одно из этих качеств с минимальными потерями для другого. В главах 5–7 мы исследуем эту мысль подробнее с точки зрения затрат, изменений с течением времени и влияния на изменения организации и производственной культуры. А в последних главах этой книги введем дополнительную сложность — масштаб. Главы 8–10 относятся к управлению изменениями не в одном API, а в их системе.

Реализация изменений — это только половина работы по управлению изменениями. Другая половина — информирование пользователей о том, что у них прибавилось работы, потому что вы что-то изменили. Например, если меняете модель интерфейса, то, наверное, следует сообщить командам по дизайну, разработке и операциям, что у них появляется новое задание. В зависимости от характера изменений вам, скорее всего, нужно будет сообщить и пользователям, что им придется обновить клиентское ПО. Обычно это делается с помощью контроля версий API. Как создавать версии? Это зависит от стиля API и используемых протоколов, форматов и технологий. Мы подробнее поговорим об этом в пункте «Контроль версий» подраздела «Дизайн» раздела «Аспекты системы и столпы жизненного цикла API» главы 10 (с. 237–238).

Ключевые решения по управлению изменениями. *Какие изменения надо вносить быстро, а какие — безопасно?* Решение о том, как обращаться с разными типами изменений, очень важное. Если его централизовать, можно создать общее правило, предусматривающее разные процессы внесения изменений в зависимости от их влияния на систему. Некоторые называют этот подход бимодальным или двухскоростным, но нам кажется, что в сложной организации больше двух скоростей. Вы можете также децентрализовать это решение и позволить отдельным командам самостоятельно оценивать влияние. Опасность заключается в том, что команды могут оценить его неточно, поэтому следует убедиться в надежности своей системной архитектуры.

Выводы

В этой главе мы рассмотрели десять столпов, поддерживающих работу по созданию продукта с API. Каждый столп содержит набор решений, которые повлияют на конечный продукт. Не все они требуют одинакового количества усилий, и вам нужно будет решить, насколько важен каждый из столпов, основываясь на общей обстановке и целях API. При росте системы API также нужно будет обдумать, как распределить решения по каждому из этих столпов. Мы подробнее расскажем об этом в главе 10.

Но прежде, чем мы до этого доберемся, необходимо более детально исследовать десятый столп — управление изменениями. Какова цена изменений в API? Подробно рассмотрим это в следующей главе.

5 Непрерывное улучшение API

> Нет необходимости меняться. Выживание необязательно.
>
> *У. Эдвардс Деминг*

В предыдущей главе мы ознакомились с жизненным циклом API и определили десять столпов работы, на которых следует сосредоточиться. Жизненный цикл и эти столпы определяют, что именно необходимо сделать для выпуска вашего первого API. Кроме того, столпы важны для всех изменений, которые вы будете вносить в период жизненного цикла уже опубликованного API. Управление изменениями в API — это важнейшая часть стратегии управления им в целом.

Изменения в API могут серьезно повлиять на ваше программное обеспечение, саму программу и опыт пользователей. Внесение в код изменений, которые повредят API, может даже создать катастрофическую цепную реакцию, влияющую на все компоненты, использующие этот код. Изменения, которые не вредят API, также способны вызвать большие затруднения, если изменяют интерфейс неожиданным образом. Продукты с API создают сложные группы взаимозависимостей. Поэтому управление изменениями — важный раздел в управлении API.

Если бы в опубликованные API не надо было вносить изменения, управлять ими было бы довольно легко. Но, разумеется, это неотъемлемая часть активно применяемых API. В какой-то момент нужно будет исправить ошибку, улучшить опыт разработчика или оптимизировать код реализации. Эти задачи требуют вмешательства в используемый API.

Управление изменениями API усложняется большим объемом работы. Продукт с API — это не только интерфейс. Он состоит из многих частей: интерфейсов, кода, данных, документации, инструментов и процессов. Все эти части продукта API можно изменять и управлять ими надо осторожно.

Управлять изменениями API непросто, но это необходимо. К тому же возможность внесения изменений дает больше свободы. Если бы опубликованный API

нельзя было изменять, было бы гораздо сложнее выпускать изначальный релиз. Приходилось бы применять к разработке API правила конструирования и запуска космических ракет. Понадобилось бы потратить очень много времени и сил на предварительное планирование и сделать серьезные вложения в разработку, чтобы убедиться, что этот API сможет проработать долго. Пришлось бы сначала учесть все возможные неполадки и предотвратить их.

К счастью, вам не обязательно работать по такой схеме. Конечно, если вы добавите в свой API возможность вносить изменения, это может принести большую выгоду. Более дешевые и простые изменения означают, что их можно делать чаще. Это позволяет вам больше рисковать, потому что появляется возможность быстрее исправлять неполадки. Таким образом, вы можете чаще улучшать API.

В этой главе мы представим философию непрерывного улучшения для API с возможностью внесения изменений. Вы узнаете, как непрерывная серия небольших поэтапных изменений может стать наилучшим способом улучшения вашего продукта с API, а также почему API так сложно изменять и как можно увеличить их способность к изменениям. Но прежде, чем углубиться в эти темы, необходимо лучше разобраться с тем, что означает выражение «изменения в API».

Изменения в API

В главе 1 мы обозначили разницу между составляющими продукта с API: интерфейсом, реализацией и готовым экземпляром. После публикации API необходимо будет управлять изменениями всех этих составляющих. Иногда придется изменять все вместе, но может оказаться, что вы будете изменять какие-то из этих элементов API независимо от других. В данном разделе мы рассмотрим влияние изменений, происходящих в каждой из этих частей. И даже добавим новый тип элементов API под названием «ресурсы поддержки», которые включают в себя части API, используемые только для расширения опыта разработчика.

Эти четыре типа изменений в API будут взаимно влиять друг на друга. Они формируют комплекс взаимозависимых изменений: изменения в модели интерфейса будут иметь далеко идущие последствия, в то же время ресурсы поддержки можно легко изменять отдельно от остальных элементов. Когда мы начнем исследовать каждый из типов изменений, вы станете лучше понимать причины существования этих взаимозависимостей.

Жизненный цикл релиза API

Программное обеспечение становится другим, когда в него вносятся изменения. Действия, предпринимаемые для того, чтобы сделать правильные изменения в кратчайшее время и с наилучшим качеством, — это процесс релиза. Как и у программного обеспечения, у API тоже есть процесс релиза — ряд действий для внесе-

ния изменений. Мы называем этот процесс жизненным циклом из-за циклического характера изменений: пока одно осуществляется, следующее уже готово. Понимать, как протекает жизненный цикл релиза, очень важно, потому что он сильно влияет на возможность изменения вашего API.

Каждое изменение, которое вы вносите в API, должно быть реализовано. Жизненный цикл релиза — это набор действий, которые приводят к его реализации. Он определяет, как изменение из идеи превращается в осуществленную и поддерживаемую часть системы. Жизненный цикл релиза объединяет все столпы, описанные нами в главе 4, в последовательность согласованной работы.

Если жизненный цикл релиза медленный, количество изменений API уменьшится. Если в нем не гарантируется качество, изменять API будет более рискованно. Если он отклоняется от требований к изменениям, последние будут менее полезными. Важно понимать, как протекает жизненный цикл релиза. Плюс в том, что у API он не отличается от жизненного цикла программного обеспечения или поставки компонентов системы. Таким образом, вы можете применять существующие руководства для релизов программного обеспечения к компонентам, из которых состоит API. Рассмотрим самые популярные из них.

Один из самых известных жизненных циклов программного обеспечения — это традиционный *жизненный цикл разработки систем* (system development lifecycle, SDLC). В той или иной форме он существует с 1960-х годов. Он определяет набор этапов для разработки и выпуска системы программного обеспечения. Число и название используемых этапов могут меняться, но обычно этот набор выглядит следующим образом: подготовка, анализ, разработка, конструирование, тестирование, реализация и техническая поддержка.

Если идти по ступеням SDLC по порядку, это будет разработка программного обеспечения по *каскадной* модели. На самом деле Уинстон Ройс изобрел не каскадную модель, но теперь данный тип жизненного цикла называют именно так. Это означает, что каждая фаза SDLC должна быть закончена до начала следующего этапа. Таким образом, изменение проходит с верхней ступени последовательно через каждую следующую ступень.

Один из недостатков каскадной модели состоит в том, что нужен высокий уровень уверенности в требованиях и вообще в этой области, потому что непросто иметь дело с большим количеством изменений в технических заданиях. Если у вас с этим проблемы, можно использовать циклический процесс разработки программного обеспечения. *Циклический* SDLC позволяет команде разработки производить несколько циклов релизов для одного множества требований. Каждый цикл выполняет подмножество требований, чтобы в результате последовательности всех циклов все требования были выполнены.

Если развивать идею циклов, мы придем к *спиральному* SDLC. В этом типе цикла программное обеспечение разрабатывается, конструируется и тестируется в цикличных стадиях, и каждый цикл потенциально может выполнить изначальные

требования. В спиральном SDLC воплощается дух гибкой методологии разработки и метода SCRUM.

Мы рассмотрели три распространенные формы жизненного цикла программного обеспечения. У каждой из них есть свои плюсы и минусы, и вам нужно будет выбрать подходящий жизненный цикл релиза. В этой книге мы пытаемся дать вам возможность использовать любой стиль. Говоря об изменениях, будем упоминать ваш жизненный цикл релиза, но не станем навязывать порядок, в котором должны производиться базовые действия, или применяемый жизненный цикл программного обеспечения. Вместо этого сосредоточимся на улучшении продукта, доступном благодаря жизненному циклу релиза. Но прежде, чем перейти к этой теме, подробнее поговорим о типах изменений в API, которые придется поддерживать жизненному циклу вашего релиза.

Изменение модели интерфейса

У каждого API есть модель интерфейса. Это информация, описывающая поведение API с точки зрения *пользователя*. Она рассматривает набор абстрактных понятий, определяющих, как API будет работать, и включает в себя подробности протоколов связи, сообщений и терминологии. Отличительной чертой модели API является то, что она не воплощена в жизнь, — это абстракция и ее невозможно использовать в компьютерной системе.

Хотя модель интерфейса нельзя задействовать с помощью программного обеспечения, ею можно поделиться с пользователями. Для этого модель должна быть сохранена или *выражена*. Например, можно выразить модель интерфейса, нарисовав квадратики и линии на маркерной доске. Такую нарисованную модель нельзя *задействовать*, но как абстракция она поможет вашей команде работать над дизайном API.

Конечно, модели интерфейса — это не только рисунки на доске и наброски на салфетках. Их также можно выражать с помощью языков на основе моделей или даже с помощью кода приложения. Например, Open-API Specification — популярный стандартизированный язык для описания моделей интерфейса. Использование стандартизированного языка для моделирования дает вам бонус — систему инструментов, которая поможет уменьшить затраты на реализацию вашей модели.

Рисовать или составлять модель можно как угодно — не существует никаких правил, регламентирующих уровень детализации или формат передачи модели. Но имейте в виду, что метод, который вы выберете для выражения модели, сильно повлияет на уровень детализации и описания. Маркерные доски и техника свободного рисования предоставляют максимальную свободу мысли, но ограничены физическими размерами и возможностью реализовать модели. Языки описания API обеспечивают более быстрый путь к реализации, но ограничивают свободу жестким синтаксисом и терминологией.

Столп «Дизайн» жизненного цикла нашего API фокусируется на создании и изменении модели интерфейса, поэтому большая часть описанной нами работы идеально ему подходит. Но она связана не только с дизайном. На самом деле большая часть столпов жизненного цикла зависят от модели интерфейса. Это происходит потому, что они тоже являются выражениями этой модели.

Допустим, вы выразили модель интерфейса в виде рисунка на доске или с помощью языка Open-API — и так же будете выражать ее с помощью кода приложения, документации по API и модели данных. Когда интерфейс уже опубликован и разработчики начинают писать использующий его код, они своей реализацией тоже выражают вашу модель. Все эти выражения модели подразумевают отношения взаимозависимости. Вот почему изменения в модели так важны.

Предметно-ориентированное проектирование

Идея программного обеспечения, ориентированного на модель, где реализация является выражением этой модели, пришла к нам из подхода к разработке ПО, созданного Эриком Эвансом, — предметно-ориентированного проектирования (domain-driven design, DDD). Если вы еще не читали его книгу Domain-Driven Design: Tackling Complexity in the Heart of Software[1] (Addison-Wesley), стоит сделать это!

Лучшие продукты с API имеют единообразные модели интерфейса. Поэтому разработчикам не приходится разрешать конфликты между документацией и уже опубликованными API из-за того, что выражаемые ими модели как-то различаются. Это стремление к единообразию усложняет внесение изменений в модели интерфейса, потому что их необходимо синхронизировать во всем продукте.

Использование единообразной модели не означает, что код реализации и внутренняя база данных должны задействовать ту же модель, что и интерфейс вашего API. На самом деле обычно применять одну и ту же модель для интерфейса, кода и данных — неудачная идея: то, что идеально подходит пользователям интерфейса, не обязательно идеально работает внутри вашей собственной реализации. Единообразная модель означает, что внутренние части вашей реализации API нужно будет перевести в эту модель интерфейса, прежде чем они «достигнут поверхности» API.

Изменения в модели интерфейса имеют огромное влияние на систему, но они неизбежны для любого активно используемого продукта с API. Может понадобиться добавить техподдержку новой функции, изменить что-то, чтобы упростить применение API, или, возможно, убрать устаревшую часть интерфейса, потому что ваша бизнес-модель кардинально изменилась. Из-за всех взаимозависимостей

[1] *Эванс Э.* Предметно-ориентированное проектирование (DDD): структуризация сложных программных систем. — М.: Вильямс, 2011.

изменения модели интерфейса всегда потенциально могут повлиять на код в приложениях, задействующих этот API.

Потенциальное влияние изменений модели интерфейса на пользователей API во многом связано с уровнем связанности между их кодом и вашим интерфейсом. Если особенностью API, которые вы создаете и реализуете, является слабая связанность, то даже при крупных изменениях эффект окажется меньшим. Например, использование API с событийно-ориентированным стилем или стилем гипермедиа имеет преимущество — более слабую связанность между клиентским кодом и API. В случае событийно-ориентированной системы вы можете менять алгоритм соответствия шаблону, не внося при этом никаких изменений в компонент, отправляющий события. API с гипермедиа может позволить вам совершать манипуляции с обязательными свойствами для инициирования работы, не меняя клиентский код, совершающий запрос.

Выбор подходящего стиля интерфейса может помочь уменьшить затраты на изменения его модели. Но такая увеличившаяся изменяемость API тоже не бесплатна. Чтобы они работали, придется создать подходящую инфраструктуру и реализации на обеих сторонах — клиентской и серверной. Зачастую ограничения и окружение, в котором вы работаете, сужают ваш выбор: например, разработчики, которые пишут клиентское ПО для вашего API, могут не иметь опыта написания приложений в стиле гипермедиа. В таких случаях придется просто смириться с тем, что изменения модели интерфейса будут дорого стоить.

Лучший способ уменьшить внешнее влияние изменений модели — вносить их *до того*, как интерфейс будет опубликован. Джошуа Блох, разработчик Java Collections API, сказал: «Общедоступные API, как бриллианты, — вечны» (https://www.infoq.com/articles/API-Design-Joshua-Bloch). Как только вы открываете интерфейс для пользователей, внесение изменений значительно усложняется. Мудрый владелец продукта с API по максимуму загружает изменения в модель интерфейса на стадии дизайна, чтобы не платить за них после публикации API.

Изменения в реализации

Реализация API — это модель интерфейса, выраженная с помощью компонентов, которые воплотят ее в жизнь. Она позволяет другим компонентам программного обеспечения использовать этот интерфейс. Реализация API включает в себя код, конфигурацию, данные, инфраструктуру и даже выбор протокола. Эти компоненты обычно являются *скрытыми* частями продукта — они заставляют API работать, но мы не обязаны делиться их деталями с пользователями.

Невозможно опубликовать API без реализации, и ее придется изменять в течение всего времени существования API. Поскольку реализация воплощает модель интерфейса, ее потребуется менять при каждом изменении модели. Но иногда у вас

будет возможность изменить реализацию независимо от модели. Например, может понадобиться исправить ошибку в коде реализации, уменьшить время ожидания медленного API или даже полностью переписать код, потому что он просто вам больше не нравится.

В тех случаях, когда изменения в реализации не привязаны к модели, их эффект скрыт за интерфейсом API. Таким образом, пользователям не придется ничего менять, чтобы воспользоваться этими обновлениями. Это не означает, что на них это никак не *повлияет*, — например, оптимизация работы приложения может сильно повлиять на восприятие этой работы конечным пользователем. Но так вы можете избежать управления изменениями в клиентском ПО, которое зависит от вашего API. В общем, изменения в реализации могут быть сделаны и после публикации API, не требуя изменений в модели интерфейса.

Риск внесения независимых изменений в реализацию состоит в том, что они могут ухудшить надежность, единообразие и доступность продукта. Например, если изменения в коде портят работу экземпляра API или реализация начинает отличаться от своей документации, ваши клиентские приложения пострадают. Изменения в реализации могут повлиять на готовый экземпляр и ресурсы поддержки API, поэтому каждый из этих элементов надо согласованно обновлять, тестировать и одобрять.

Изменение экземпляра

Как мы уже рассказали, реализация API выражает модель в виде интерфейса, который можно использовать. Но это возможно, только когда она запущена на устройстве в сети и доступна для пользовательских приложений. *Готовый экземпляр* API — это управляемое работающее выражение модели интерфейса, доступное для применения вашей целевой аудиторией.

Любые изменения в модели интерфейса или реализации потребуют соответствующих изменений в готовых экземплярах. Изменения в API нельзя считать внесенными, пока не обновлены экземпляры, используемые пользовательскими приложениями. Однако экземпляр API можно менять независимо, не затрагивая при этом модель или реализацию. Это может быть простой случай изменения конфигурационного значения или что-то более сложное, например дублирование и удаление работающего экземпляра. Такие изменения влияют только на свойства системы во время действия программы, прежде всего на доступность, наблюдаемость, надежность и качество работы.

Чтобы уменьшить влияние на систему независимых изменений в готовом экземпляре, необходимо заранее продумать дизайн системной архитектуры. Мы обсудим свойства системы и самые важные факторы позже, когда начнем говорить о системе API.

Изменения в ресурсах поддержки

Если API — это продукт, он должен быть не просто запущенным на сервере кодом, выражающим модель. Из главы 1 мы узнали, что поддержка разработчиков, которые должны использовать наши API, — это важная часть философии «API как продукт». Создание позитивного опыта разработчика почти всегда требует каких-либо ресурсов поддержки за пределами реализации интерфейса. Например, в эти ресурсы могут входить документация по API, регистрация разработчиков, инструменты для выявления неполадок и раздачи важных материалов и даже сотрудники техподдержки, которые будут помогать разработчикам решать проблемы.

В течение всего времени работы API вам нужно будет обновлять и улучшать материалы, процессы и сотрудников, которые поддерживают продукт. Зачастую это окажется результатом каскадных изменений модели интерфейса, реализации или экземпляра API. Ресурсы поддержки, находящиеся ниже по течению, будут затронуты при изменениях любой части API. Таким образом, затраты на изменения API вырастут с ростом количества ресурсов поддержки опыта разработчика.

Независимые изменения можно вносить и в ресурсы поддержки, например изменить внешний вид страницы с документацией в процессе модернизации. Такие изменения сильно влияют на опыт разработчика вашего продукта с API, но не на модель интерфейса, реализацию или готовый экземпляр — точнее, могут влиять, но не напрямую, а в результате того, что программой начинают чаще пользоваться и больше интересоваться.

Изменения в ресурсах поддержки вызывают наименьший каскадный эффект, но также могут привести к наивысшим затратам, потому что больше всех зависят от других элементов API. Снижение затрат на эти изменения может принести большие дивиденды с точки зрения общих затрат на изменения в продукте с API. Поэтому имеет смысл вкладываться в дизайн, инструменты и автоматизацию, чтобы тратить меньше сил на изменение этих ресурсов.

Непрерывное управление изменениями

Мы описали четыре типа изменений, с которыми вы столкнетесь в работе с API: изменения модели интерфейса, реализации, экземпляра и ресурсов. Все они поразному влияют на API и пользовательские приложения, поэтому необходимо внимательно управлять изменениями, чтобы точно не снизить качество своего продукта.

Если применять подход AaaP, эти изменения можно считать попытками улучшить продукт с API, а не просто изменениями ради изменений. Таким образом, все время, потраченное на изменения в интерфейсе, реализации, экземпляре или ресурсах, можно оправдать улучшением опыта разработчика или снижением затрат спонсоров на поддержку продукта.

Не каждое отдельное изменение сразу же улучшит ваш продукт. Например, вы можете улучшить масштабируемость экземпляров API, чтобы они соответствовали спросу в будущем, — это изменение не окупится, пока не возрастет объем их использования. Такое изменение не приведет к немедленному измеримому улучшению опыта разработчика, но может предотвратить ухудшение этого опыта в будущем. Суть в том, что каждое изменение надо рассматривать с точки зрения возможности улучшить продукт, даже если оно окупится не сразу.

Постепенное улучшение

Если изменения — это путь к улучшению продукта, тогда разумной целью управления будет максимально облегчить изменение API. Лучшая версия вашего API появится благодаря непрерывному циклу изменений или улучшений. Некоторые из них практически не будут улучшать продукт немедленно — некоторые ваши попытки могут даже временно ухудшить опыт разработчика вашего API. Если это произойдет, необходимо применить другое улучшение, чтобы снизить последствия провального эксперимента. Со временем эти постоянные попытки постепенно улучшить API выгодно повлияют на ваш продукт и опыт разработчика.

Постепенное улучшение означает, что вы представляете, куда хотите двигаться, но решили на пути к цели делать маленькие шаги, а не выпускать изменения типа «большой взрыв» в попытке соответствовать всем будущим требованиям. Серия небольших изменений дает команде по API возможность отреагировать на результаты каждого из них, эффективно проводя серию небольших экспериментов, чтобы найти лучший путь к цели, которая продолжает меняться.

Концепция непрерывного внесения небольших изменений — устоявшаяся схема, корнями уходящая в промышленность. В 1980-е годы первопроходец в сфере контроля качества У. Эдвард Деминг высказал версию этой идеи в виде подхода, который он назвал системой углубленного знания. Система Деминга работает со сложной природой больших организаций и применяет научный подход для улучшения способа производства продукции. Один из краеугольных камней его подхода — это цикл «план — действие — исследование — обновление» (Plan — Do — Study — Act (PDSA)), который определяет постепенный и основанный на экспериментах подход к улучшению процесса (рис. 5.1).

Процесс PDSA определяет четыре этапа применения улучшений. Сначала вы составляете *план* — теорию о том, как можно улучшить систему в соответствии с целью и какие изменения необходимо внести, чтобы испытать эту теорию. Затем вы *действуете* — реализуете намеченные в плане изменения. После этого *исследуете* — отслеживаете и измеряете эффект от этих изменений и сравниваете результаты с планом. И наконец, вооружившись новой информацией, можете *обновить* цель, теорию или действия, реализующие изменения, чтобы улучшить систему.

Рис. 5.1. Цикл PDSA Деминга

Например, если вы хотели улучшить опыт разработчика вашего API, можете начать с цели — уменьшить время, которое требуется разработчикам, чтобы понять, как работает интерфейс. План может состоять в обновлении документации, чтобы она стала более понятной. Затем воплощаете план — обновляете ресурсы и документацию, потом исследуете количество ошибок, сделанных разработчиками, которые просмотрели новую документацию. С помощью этих измерений вы можете пересмотреть необходимые изменения в документации или даже решить внести более эффективное изменение в саму модель интерфейса.

Колесо PDSA описывает постепенный, основанный на экспериментах процесс улучшения системы: вы вносите небольшие изменения, измеряете их эффект и используете полученные знания, чтобы продолжить улучшать систему. Это очень эффективный способ работы со сложными системами — такими, в которых тяжело предугадать результаты небольших изменений.

Идеи Деминга и его колесо PDSA изначально были созданы для улучшения контроля качества на фабриках и сборочных производствах, но эта схема оказалась полезной в любых случаях, когда нужно улучшать сложные системы, в том числе системы программного обеспечения. Такие методики производства ПО, как «Бережливая разработка» (Lean), «Непрерывное улучшение» (Kaizen) или «Гибкая методология» (Agile)[1], объединяет один и тот же принцип непрерывного улучшения определенного объекта. Иногда объект улучшения — это процесс, иногда —

[1] Советуем почитать: *Скотчер Э., Коул. Р.* Блистательный Agile. Гибкое управление проектами с помощью Agile, Scrum и Kanban. — СПб.: Питер, 2019. — *Примеч. ред.*

продукт, но в любом случае постоянные изменения, ориентированные на цель, обеспечивают гибкость и успех.

В этой книге мы не будем вас учить, как использовать конкретные методики непрерывного изменения, но попытаемся с максимальной пользой задействовать концепцию постепенного улучшения. Применение концепции постоянного и постепенного улучшения к вашему продукту с API означает целенаправленную разработку процесса для всех четырех типов изменения API. Это означает разработку такой системы, в которой изменения эффективны и несут незначительный риск, затраты на отдельные изменения снижаются, а вместо этого появляется возможность вносить их чаще. Мы предоставляем вам самим проработать такой механизм, но применение непрерывного цикла улучшений API — ключевое требование для продукта, который сможет постоянно оставаться высококачественным.

Скорость изменения API

Если вы хотите вносить много небольших изменений в продукт с API, необходимо убедиться, что это можно делать быстро. Иначе стоимость непрерывных изменений станет заметной проблемой. Чем быстрее вы применяете улучшения, тем короче будет путь спонсоров вашего API к инновациям и конкурентным преимуществам на рынке. Но эти изменения должны быть разумного качества, иначе надежность и качество продукта окажутся подорваны.

Повышение скорости изменений и улучшение их качества важно для любого типа API — внешнего, внутреннего или партнерского. Если вы не будете успевать быстро вносить качественные изменения в интерфейс, это негативно скажется на улучшении опыта пользователя, возможностях для бизнеса, а также на запуске новых продуктов. Оптимизация скорости и безопасности жизненного цикла изменений API вносит вклад в общую скорость изменений в вашей организации.

Но, учитывая, что у вас ограниченное количество сотрудников, денег и времени, как вносить изменения в API так, чтобы *оптимизировать* эти ресурсы? Когда предложенное вами изменение проходит через все стадии жизненного цикла релиза API, как убедиться, что вы движетесь с максимальной скоростью?

Есть три основных способа увеличить скорость жизненного цикла API, не жертвуя качеством: с помощью инструментов, планирования организационного процесса и снижения усилий.

Инструменты и автоматизация

Одним из решений для увеличения скорости и безопасности изменений является введение инструментов и автоматизации вместо выполнения работы людьми. Инструменты — это привлекательный вариант, потому что он снижает вероятность человеческой ошибки и уменьшает время на выполнение задания. Например,

инструменты «непрерывная интеграция и непрерывная доставка» (CI/CD) могут автоматизировать тестирование и выпуск реализации API и значительно снизить затраты на внесение изменений.

Однако польза инструмента зависит от его качества и времени, которое вы готовы потратить на то, чтобы его установить и настроить. Всегда будут существовать изначальные затраты и риски, связанные с новыми инструментами, поэтому, если продукт с API уже стабилен и активно используется (это фаза API, которую мы в дальнейшем будем называть осуществлением), то их нужно вводить осторожно, на экспериментальной основе.

Все типы изменений API можно автоматизировать с помощью инструментальных средств. В момент написания этой книги рынок инструментов для безопасности, документации, развертывания и конфигурирования API процветал. Эти инструменты обеспечивают более быстрые и надежные процессы изменений.

Планирование организационного процесса и корпоративная культура

Труд по внесению изменений в API может считаться умственным — работой, которая требует скоординированного процесса принятия решений. Если вы создаете один API силами небольшой команды, усилия по координации обычно тоже невелики, но в масштабе большой организации с множеством API и компонентов программного обеспечения большие затраты на координированное принятие решений уменьшают возможность отдельной команды изменить продукт.

Эта зависящая от человека часть процесса изменений обычно создает самое узкое место на пути к высокой скорости в основном потому, что его сложнее всего понять и изменить. Невозможно купить организационное планирование или корпоративную культуру так же, как документацию для API или инструмент CI/CD.

В главе 7 мы уделим больше времени организационным аспектам управления API, включая возможности создания платформы для принятия решений, которая облегчает быстрые и качественные изменения.

Сокращение напрасных затрат ресурсов

Еще один способ увеличить скорость и качество улучшений — тратить на них меньше ресурсов. Избавившись от той работы над API, которая хуже всего окупается с точки зрения достижения целей продукта, вы сможете значительно увеличить скорость изменений. Сокращение напрасных трат ресурсов также уменьшает вероятность того, что что-то пойдет не так, и в итоге делает процесс изменений более надежным.

Например, API, который создан и применяется одной и той же командой разработчиков, скорее всего, не требует такого же уровня вложений, как внешний

API, используемый сотнями разработчиков сторонних компаний. Но здесь нужно рассмотреть множество перестановок и переменных. В главе 6, когда речь пойдет о жизненном цикле продукта с API, мы введем систему переменных, которая даст вам отправную точку для размышлений о том, какие инвестиции вы хотите сделать.

Улучшение изменяемости API

Мы привели несколько веских причин для принятия подхода непрерывного улучшения жизненного цикла API. Показали, что быстрое внесение небольших изменений — это идеальный способ улучшить продукт с API, и подробнее рассмотрели типы изменений и улучшений, которые для этого необходимы. Но на практике сложно применить этот подход к API, потому что, когда интерфейс усложняется и его начинают использовать другие команды, затраты на изменения возрастают.

Есть три основных типа затрат, связанных с изменением API, которые могут уменьшить степень изменяемости: трата ресурсов на выполнение работы, стоимость упущенной возможности и издержки, связанные с изменением зависимых компонентов. Если минимизировать эти три типа затрат, вы сможете чаще изменять API. Больше изменений — больше возможностей постепенно улучшать свой продукт.

Затраты ресурсов

Самые очевидные затраты на изменение модели интерфейса, реализации, экземпляра или ресурсов поддержки вашего API — это время, энергия и деньги, необходимые, чтобы провести изменение через весь жизненный цикл API. Если уменьшить их, вы значительно увеличите возможность добавить новые улучшения в продукт с API.

Ранее мы говорили о необходимости быстрых изменений и определили, что сокращение трат ресурсов, инструментальные средства и организационные изменения помогут снизить некоторые затраты на эту работу. Но на самом деле увеличение скорости изменений — это сложная проблема.

Количество ресурсов, необходимых для изменения API, зависит как минимум от следующих факторов: сложности проблемы, опыта и квалификации сотрудников, выполняющих эту работу, разработки процесса изменений, сложности и качества реализации. Это длинный и не исчерпывающий список. К счастью, уменьшение затрат на работу — это основная цель профессиональной разработки программного обеспечения, и вам в помощь выпущены горы исследований, советов и экспертных мнений, а то, что полезно для изменений в ПО, обычно полезно и для продуктов с API.

Определение конкретных методик изменений, процессов контроля качества, архитектур, реализаций и инструментов для автоматизации, которые можно

использовать для сокращения трат ресурсов, выходит за рамки этой книги. Мы попытались представить несколько основных стратегий, схем и методик, которые лучше всего помогут добиться высокой скорости изменений, но трудную работу по превращению общих советов в конкретные методы для своей организации вам придется выполнить самим.

Стоимость упущенной возможности

Другой вид затрат, который может замедлить перемены, — это желание отказаться от изменения API, чтобы сначала собрать больше информации. Потеря возможности ради сбора информации становится затратой с точки зрения изменения API. Том и Мэри Поппендик, создатели «Бережливого подхода к разработке программного обеспечения», описывают это действие как откладывание важного решения до *позднейшего ответственного момента*.

Что еще сложнее, вам нужно также принимать во внимание затраты на то, что не внесены изменения и из-за этого упущены возможности улучшить продукт и собрать информацию после изменения. Во многих ситуациях лучше игнорировать принцип позднейшего ответственного момента, чтобы не отвлекаться на переживания о том, что нужно еще подождать и получить больше информации. Небольшие изменения кода в опубликованном компоненте ПО — пример момента, когда решение можно считать недостаточно важным для того, чтобы беспокоиться об упущенных возможностях. К тому же, если вы сразу же узнаете об ошибке и у вас будет мало времени на восстановление после проблемы, беспокоиться точно не стоит.

Многие типичные изменения в продуктах с API подходят под эти характеристики: они не очень важные и после них несложно восстановиться. Например, изменение внешнего вида документации по API, предназначенной для чтения человеком: вы быстро узнаете о его успешной реализации и его довольно просто откатить в случае возникновения проблем. Однако после реализации некоторых типов изменений в API сложно восстановиться, и они требуют осторожного обращения — например, изменение модели интерфейса вашего API, которое может иметь далеко идущие последствия. Необходимо правильно управлять такими изменениями и всегда учитывать потери при отсутствии достаточного количества информации.

Один из способов снижения стоимости упущенной возможности при изменениях — лучше собирать информацию. В главе 8 мы представим такое качество системы, как *видимость*, которое может значительно снизить эту стоимость.

Затраты из-за связанности

Когда речь идет об API, в особенности о *модели интерфейса*, самым сложным препятствием на пути к простым изменениям является *связанность*, которую мы создаем между API и его пользователями. Существует множество стилей API, но

какой бы вы ни выбрали, в итоге все равно появится зависимость или связанность между отправителями и получателями сообщений. Такая связанность сильно влияет на то, что можно изменить в API и когда это сделать.

API — это всего лишь канал для связи и общения между модулями программного обеспечения. При общении между людьми используется общее понимание слов, жестов и знаков, облегчающее осмысленный разговор. Компонентам ПО тоже нужно общее знание, чтобы общаться. Например, общее знание терминологии сообщений, сигнатур интерфейса и структур данных полезно при обеспечении осмысленного взаимодействия между двумя компонентами. Важный фактор изменяемости API состоит в том, сколько таких правил общения жестко запрограммировано в коде опубликованного компонента. Когда семантика API определяется во время разработки, растут затраты на изменение интерфейса.

Связанность неизбежна и может существовать в любом месте во множестве форм. На самом деле, когда вы слышите о том, что конкретный API сильно или слабо связан, нужно практически детективное расследование, чтобы точно понять, что имелось в виду. То, что адрес API в сети где-то жестко запрограммирован? Или речь идет об изменяемости семантики и терминологии сообщений? Или, возможно, о том, насколько легко можно создавать новые экземпляры API, не влияя на пользователей?

Например, событийно-ориентированную архитектуру часто описывают как предлагающую слабую связанность между отправителями и получателями событий. Но зачастую при ближайшем рассмотрении оказывается, что эта слабая связанность относится только к известным отправителю сообщения сведениям о том, какие компоненты получают его сообщения. На самом деле структура, формат или терминология сообщений о событиях могут создать проблемы для многих получателей.

Некоторые стили API особенно директивны в том, что определяется во время разработки. Если вы создаете интерфейс в стиле RPC, то практически наверняка используете какой-либо *язык определения интерфейса*, очень четко документирующий модель интерфейса. Плюс четко определенной модели интерфейса в том, что становится проще писать код, — API со стилем RPC обычно имеют множество инструментов, максимально облегчающих начало работы.

Проблема четко определенной модели интерфейса проявляется, когда требуется внести в нее изменения. Если вы пользуетесь подходом непрерывного улучшения, то можете понять, что есть много возможностей для небольших улучшений вашего интерфейса. Но поскольку семантика API жестко запрограммирована в опубликованном коде, изменения в модели интерфейса потребуют соответствующих изменений в нем.

Обычно мы стремимся избежать появления проблем в приложениях-клиентах, зависимых от наших API. Но на практике может оказаться, что надежность одних клиентов вас заботит меньше, чем надежность других. Например, изменение в API, которое повредит редко используемому приложению от сторонней компании, более

оправданно, чем изменение, которое повредит мобильному приложению вашей организации, которым пользуются покупатели.

Нельзя однозначно ответить на вопросы, какой уровень связанности нужен вашему API и когда следует вносить изменения. Если бы слабая связанность ничего не стоила, ее бы применяли все, но долгосрочная полезность невозможна без краткосрочных расходов, а создание API, хорошо справляющихся с изменениями, требует затрат на начальном этапе. Придется принимать решения о стоимости изменения и о том, какой тип API вам нужен, на довольно раннем этапе.

Имейте в виду, что низкий уровень изменяемости в сочетании с высокой стоимостью изменений в коде означает, что непрерывное улучшение модели API — нереалистичная стратегия. В лучшем случае ваши непрерывные улучшения будут ограничены изменениями модели интерфейса, которые не вредят приложениям-клиентам. В таком случае лучше начать с подхода «идеальный дизайн на начальном этапе», прежде чем модель интерфейса начнет активно использоваться.

Разве идеальный дизайн на начальном этапе — не антисхема?

Если вы знакомы с манифестом «Гибкая методология разработки», то можете подумать, что описанное в этом разделе — это пример антисхемы идеального дизайна на начальном этапе (BDUF), которой специалисты по гибкой методологии стараются избегать. Отказ от объемной фазы дизайна имеет смысл для разработки ПО, так как означает, что можно применять короткие этапы изменений и непрерывно работать над дизайном, основываясь на растущей реализации. Такой постепенный подход к разработке продукта дает гораздо больше пространства для того, чтобы подстроиться, поэтому отказ от BDUF полезен.

К сожалению, когда речь идет об API, такие постепенные изменения может быть сложно вносить, потому что изменения интерфейса создают эффект домино в отношении кода приложений, которые его используют. Мы советуем заранее разрабатывать все детали API: как архитектуру здания или мраморную статую, так как API обычно сложно изменить после того, как они созданы и опубликованы.

Выводы

В этой главе мы выделили четыре типа изменений в API: изменения модели интерфейса, реализации, экземпляра и ресурсов поддержки. А также познакомили вас с подходом непрерывного улучшения и проанализировали, почему он полезен для API. Мы подчеркнули важность скорости изменений и прошлись по основным препятствиям изменяемости API, включая связанность между клиентским кодом и API.

В следующей главе познакомим вас с моделью развития, которая поможет вписать непрерывные изменения в среду постоянно эволюционирующего продукта с API.

6 Жизненный цикл продукта с API

Старение — обязательно, взросление — нет.

Приписывается Чили Дэвису

В деле управления API жизненно важно понимать, какой эффект производят изменения. Как говорилось в предыдущей главе, существуют различные типы затрат, связанных с изменениями API: траты ресурсов, стоимость упущенной возможности, затраты из-за связанности. Общая стоимость изменения зависит от того, в какую часть API оно вносится.

Кроме того, стоимость изменений в API динамична — с изменением внешнего окружения API меняются и затраты на его изменение. Например, затраты из-за связанности для неиспользуемого API практически равны нулю, но, когда тот же API соединен с сотнями клиентских приложений, эти затраты сильно возрастают.

В реальности управление API еще сложнее, чем в этом примере. Что, если у API есть только одно клиентское приложение, которым владеет основной партнер вашего бизнеса? А если зарегистрированных разработчиков сотни, но никто из них не приносит доход вашим основным продуктам? Что, если API приносит доход, но больше не подходит под вашу бизнес-модель? В каждом из этих случаев затраты на изменение кардинально различаются. Существуют сотни вариаций внешнего окружения, которые нужно принимать во внимание. Из-за этого трудно дать общую оценку развития API для всех продуктов.

Несмотря на сложность этого проекта, было бы неплохо создать универсальную модель развития API. Во-первых, мы получили бы общий способ измерения успеха API. Во-вторых, появился бы фреймворк для управления API на каждом этапе его существования, особенно с точки зрения затрат на изменения. Поэтому мы попытаемся создать такую модель, которая подойдет всем.

В этой главе мы познакомим вас с жизненным циклом продукта с API и представим модель развития API. Опишем пять стадий развития, которые можно отнести

ко всем API. Чтобы они соответствовали внешнему окружению, мы представим способ определения ключевых моментов, которые будут подходить вашим бизнес-стратегиям. Наконец, изучим, как каждая стадия жизненного цикла продукта влияет на столпы работы с API. Но прежде, чем углубиться в тему жизненного цикла продукта, разработаем способ измерения продуктов с API.

Измерения и ключевые моменты

Жизненный цикл продукта с API, с которым мы обещали вас познакомить в этой главе, состоит из пяти стадий, разграниченных ключевыми моментами. Ключевой момент стадии жизненного цикла определяет входной критерий для API. Пока API будет развиваться от стадии создания через появление дохода до удаления, он будет проходить через «ворота» ключевых моментов. Чтобы определить ключевые моменты для продукта, вам нужен способ измерения и отслеживания API.

В главе 4 мы описали столп управления API «Мониторинг». Создание системы сбора данных — важный первый шаг к измерению прогресса продукта с API. Нельзя отследить, насколько вы продвинулись, если не знаете, где сейчас находитесь. Сбор данных — техническая задача, которую можно решить с помощью качественных дизайна и инструментов, но определение нужных данных для измерений жизненного цикла продукта требует другого подхода.

Вам понадобится определить ключевые моменты, подходящие для своих API, стратегии и бизнеса. Если вы делаете это сами, мы можем создать общий набор стадий жизненного цикла, которые можно применять к вашему уникальному внешнему окружению. Чтобы создать эти ключевые моменты, следует определить цели и параметры, подходящие для продукта. В этом разделе мы познакомим вас с двумя инструментами, которые помогут определиться, — OKR и KPI.

OKR и KPI

На протяжении всей книги мы будем использовать термин *«ключевой показатель эффективности»* (key performance indicator, KPI), говоря об измерении стоимости или качества чего-либо. KPI — это не магия, это просто красивый термин для описания особого способа сбора данных. KPI описывает, насколько хорошо работает измеряемый объект. Сложность здесь в том, чтобы найти наименьшее количество параметров, которые обеспечат максимальное понимание ситуации. Поэтому эти параметры и называются *ключевыми* показателями эффективности.

KPI полезны, поскольку показывают целенаправленные измерения. В отличие от общего сбора данных данные KPI тщательно отбираются. KPI должны помочь команде по управлению понять, как обстоят дела у другой команды или продукта. Они проясняют, как работает то, что они измеряют, и это помогает при оптимизации. Например, два KPI для кол-центра — это количество пропущенных звонков и среднее

время ожидания звонящих. Если часто оценивать эти данные и одновременно стремиться их улучшить, это может сильно повлиять на решения по управлению.

Если показатели эффективности сильно влияют на решения по управлению, то крайне важно тщательно отбирать данные. Неточные измерения приводят к неудачным решениям, поэтому необходимо выбрать правильные KPI. Кто-то должен определить главные факторы успеха организации и в соответствии с ними разработать параметры. Но как это происходит?

Некоторые компании используют *OKR*, чтобы определить свои *цели* (objectives) и *ключевые результаты* (key results), необходимые для их достижения. OKR заставляют команды по управлению отвечать на вопросы «Куда мы хотим двигаться?» и «Что потребуется, чтобы достичь этого?». В зависимости от того, кого вы слушаете, OKR либо сильно связаны с KPI, либо должны полностью их заменить. В любом случае OKR тоже полезны, потому что представляют собой целенаправленную попытку связать многоуровневые цели в организации с необходимыми для прогресса результатами.

OKR в LinkedIn

Некоторые организации находят OKR невероятно полезными для достижения успеха. Например, генеральный директор LinkedIn Джефф Вайнер считает OKR важным инструментом совмещения командных и индивидуальных стратегий с целями организации (http://bit.ly/2qNvZHm). Он уверен, что OKR — это «то, что вы хотите выполнить в конкретный период времени, и это скорее не четкий план, а завышенная цель. Это то, что нужно срочно донести до клиентов». Для Вайнера OKR полезны только тогда, когда цели продуманы и непрерывно транслируются, располагаются по нисходящей и закрепляются.

Используя эти термины в книге, мы не стремимся убедить вас в необходимости OKR или KPI. Для успешного управления API вам не нужен консультант по OKR или KPI. Это полезные инструменты, но основное значение имеют корпоративная культура и перспектива постановки целей и измерения показателей продукта. Мы решили применять эти конкретные термины, потому что знаем, что они являются ключами к огромному количеству информации, советов и инструментов для тех из вас, кто захочет углубиться в тему. Но самое главное — это иметь четкие цели и измеримые данные, чтобы отследить прогресс вашего продукта.

Дополнительное чтение

Тем, кто хочет больше узнать о KPI и OKR, мы советуем начать с книги Энди Гроува «Высокоэффективный менеджмент» (High Output Management, издательство Vintage), с которой началось движение OKR. Если вам нужно больше инструкций, обратите внимание на книгу «Цели и ключевые результаты» (Objectives and Key Results, издательство Wiley) Бена Ламорта и Пола Р. Нивена.

Определение цели API

Цель, которую вы ставите для отдельного API, должна отражать стратегические цели ваших команды и организации. Цель API не обязана полностью совпадать с общей целью организации, но должна быть с ней связана. Таким образом, определение цели API поможет компании продвинуться к своей цели. Если ваш API приносит ожидаемый доход, это выгодно организации. Отношения между целью API и задачами организации должны быть ясными и понятными.

Для создания такой связи между целями вам необходимо понимать кое-что о стратегии организации. Надеемся, что это так. Если же нет, это должно стать вашим первым шагом. В мире OKR цели можно располагать по нисходящей, и каждая часть компании определяет свои цели, связанные с общей. Например, команда генерального директора ставит стратегическую задачу и определяет ключевые результаты, что позволяет подразделению компании выработать цели, которые внесут свой вклад в достижение этих результатов. У отделов внутри подразделения тоже могут быть свои цели, связанные с определенными результатами, и т. д. Таким образом OKR могут спускаться через разные команды и различных сотрудников компании.

OKR не единственный путь достижения такой связи между целями. Например, сбалансированная система показателей Роберта Каплана и Дэвида Нортона (http://bit.ly/1vt3X2Q) включает в себя похожий метод нисходящего расположения целей по показателям. Эта и другие похожие системы (http://bit.ly/2TfD0h8) используются как минимум с 1960-х годов. Мы предоставляем вам самостоятельно определить, как связать свои цели по API с задачами организации. Важнее всего, чтобы цели существовали и приносили доход вашей компании и спонсорам.

Правил относительно того, что может, а что не может являться целью для API, не существует, но в табл. 6.1 приведены примеры некоторых распространенных типов целей.

Таблица 6.1. Примеры целей для API

Тип цели	Описание
Использование API	Достичь определенного количества обращений за определенный период
Регистрация API	Достичь определенного количества новых регистраций или общего количества регистраций
Тип клиентов	Привлечь конкретный тип клиентов, например банк
Влияние	Позитивно повлиять на бизнес с помощью API, например увеличить процент заказов продукции
Идеи	Собрать определенное количество новых идей/моделей для бизнеса от сторонних пользователей API

Их можно смешивать и сочетать, например задать цели и по использованию, и по типу клиентов, но имейте в виду, что чем больше целей, тем сложнее оптимизировать

дизайн под конкретную цель. Цель API движет вперед всю работу, которую вы для нее делаете, но это не означает, что она никогда не изменится. Вам придется переоценивать цели, если поменяются установки всей организации или окажется, что ваша цель не приносит никакого дохода.

Определение измеримых результатов

Цель имеет смысл, только если степень ее выполнения можно точно измерить. Иначе это не цель, а в лучшем случае стремление. Управление API требует постановки четкого набора измеримых целей и подстройки своей стратегии таким образом, чтобы продвигаться в их направлении. Для этого нужна продуманная разработка параметров или KPI вашего API.

Качественные измерения позволяют добиться качественных целей. Таким образом, измеримые результаты позволят нам достичь уже определенных целей. Постановка четких и измеримых целей сильно упрощает определение ключевых результатов или ключевых показателей прогресса. Но прежде всего необходимо определить параметры этих измерений.

Если вы в принципе заинтересованы в определении качественных параметров данных, советуем вам прочесть книгу Дугласа Хаббарда «Как измерить что угодно» (*How to Measure Anything*, издательство Wiley). Она послужит прекрасной отправной точкой для понимания того, почему и как надо измерять. Хаббард рассказывает, что цель измерения — помогать принимать решения в сферах, в которых вы не уверены. Именно это нам и нужно — мы можем знать свою цель, но сомневаться по поводу того, насколько продвинулись к ней или как измерять желаемые результаты.

В своей книге Хаббард приводит вопросы, которые вы можете задать себе, чтобы понять, какой инструмент для измерения вам нужен. Применим эти вопросы к сфере измерения API.

- *Из каких частей состоит то, насчет чего мы не уверены?* Большинство вещей можно разделить на части. Хаббард рассказывает, что разделять объекты измерения очень полезно. Когда у объекта измерения высокий уровень неопределенности, поищите способы разделить его на меньшие части, которые измерить проще. Например, вы хотите измерить степень удовлетворенности разработчиков вашим API. Здесь неопределенность очень велика, но как разбить ее на меньшие, лучше измеримые части? Мы думаем, что это возможно: запросы в техподдержку, рекомендации и рейтинги продукции — это измеримые параметры, которые можно использовать для определения уровня удовлетворенности пользователей.
- *Как его (или его) отдельные части измеряли другие?* Изучайте измерения, выполненные другими людьми, когда это возможно. Если вам повезет и вы найдете измерения в той же предметной области, можно будет их скопировать.

Но даже если такой возможности не представится, изучение чужих измерений очень информативно и поможет вам с вашими собственными. Начать можно с лекции специалиста по стратегии API Джона Массера «KPI для API» (https://www.slideshare.net/jmusser/kpis-for-apis), но, к сожалению, в публичном доступе не так уж много примеров измерений API.

Впрочем, многие из измерений, применяемых к API, имеют эквиваленты в других сферах. Любые измерения опыта разработчика можно выполнять, опираясь на общую сферу измерения опыта пользователя. Измерения влияния на бизнес можно выводить в целом из измерений OKR и KPI. Измерения регистрации, использования и активности имеют параллели в сфере управления продукцией. Поэтому поиск чужого опыта по решению похожих проблем не должен представлять особой сложности.

- *Как определенные наблюдаемые объекты поддаются измерению?* После разделения и определения других источников у вас должно сформироваться более ясное представление о том, что вы хотите измерить. Чтобы ответить на этот вопрос, надо определить, как это измерить. Например, для измерения количества запросов к техподдержке потребуется отследить все каналы, по которым они приходят: электронную почту, социальные сети, телефон и личное взаимодействие. На этом этапе вы начинаете разрабатывать систему сбора данных.
- *Насколько нам действительно нужно это измерить?* Будь у вас неограниченный бюджет, ваша система сбора данных была бы идеальной. Но нужно ли это? Хаббард предлагает обдумать, насколько для принятия решений по нашему продукту с API нужна идеальная информация. Здесь важнее всего обстановка, в которой он существует. Насколько важен этот API для вашего бизнеса? Как решения по управлению повлияют на организацию? Например, если вы разрабатываете API только для собственного пользования, вам может быть практически не нужно внимательно им управлять и вы будете мало вкладывать в точные измерения.
- *Где есть риск допустить ошибку?* Здесь нужно обдумать, насколько недостоверными могут быть предполагаемые измерения. Есть ли погрешности и противоречия? Влияет ли на результаты метод наблюдения? Цель — определить, в чем могут заключаться проблемы и каково их решение. В сфере API проблемы могут возникать из-за технических сложностей (передают ли инструменты точные данные?), отсутствия каких-то данных (отслеживаем ли мы все запросы к техподдержке?) или неправильного разделения (точно ли это правильная оценка степени удовлетворенности разработчиков?).
- *Какой инструмент выбрать?* Под инструментом Хаббард подразумевает процесс или систему для непрерывного сбора данных измерений. В сфере API это должен быть тот вид KPI, который надо измерять параллельно с отслеживанием разработанной для него реализации.

Вооружившись ответами на эти вопросы и примерами из других источников, вы сможете определить нужные измерения. Сделав это и выполнив то, что мы описывали в подразделе «Мониторинг» на с. 99, вы будете готовы разрабатывать KPI для жизненного цикла продукта.

Жизненный цикл продукта с API

Общая модель развития продукта уже существует. Она называется *жизненным циклом продукта* и определяет четыре стадии — разработку, рост, зрелость и упадок, через которые проходят все продукты с точки зрения рыночного спроса. Мы взяли концепцию *жизненного цикла продукта* и применили ее к API, чтобы создать *жизненный цикл продукта с API*. Он состоит из пяти стадий: *создания*, *публикации*, *окупаемости*, *поддержки* и *удаления* (рис. 6.1).

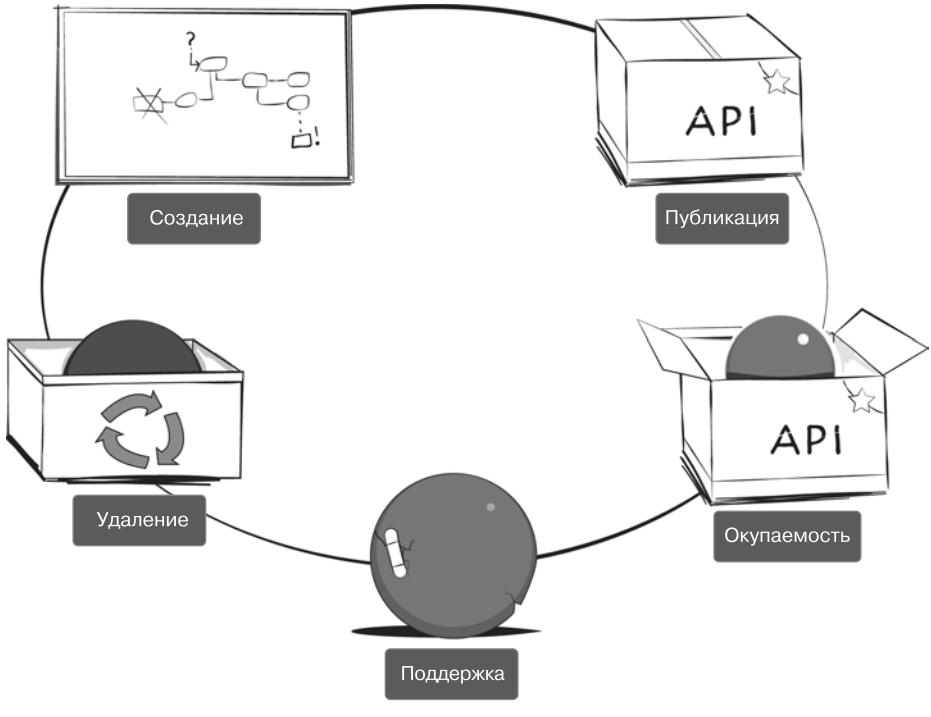


Рис. 6.1. Жизненный цикл продукта с API

Как мы уже упоминали в начале главы, жизненный цикл продукта с API — это модель, которая поможет вам отслеживать прогресс API и менять стиль управления по мере его развития.

В главе 5 мы описали жизненный цикл релиза API. Жизненный цикл продукта включает в себя все эти релизы. Каждая стадия жизненного цикла продукта с API может содержать много отдельных релизов. Релизы и изменения напрямую не помогают продукту достичь следующей стадии развития, но могут сделать это косвенно.

В следующих разделах подробно разберем каждую из стадий жизненного цикла продукта. Мы расскажем о том, что происходит на каждой из них и какие ключевые моменты необходимо определить, чтобы их достичь.

Стадия 1: создание

API на стадии *создания* обладает следующими характеристиками.

- Это новый API или заменяющий уже несуществующий API.
- Он не развернут в работающей среде.
- Он недоступен для надежного использования.

У каждого API есть исходная точка — кто-то где-то в организации решает, что надо опубликовать API, а самого API в тот момент еще не существует. Есть много причин для разработки API, и на данной стадии необходимо точно определить эти движущие факторы. Вы хотите продавать доступ к этому API? Он ускорит разработку приложений? Или это просто канал для доступа к данным? Понимать, для чего компании нужен этот конкретный API, крайне важно для определения целей, ценности и целевой аудитории.

API, находящиеся на исходной стадии, легко изменяются. Как мы узнали из главы 5, модель интерфейса труднее менять, когда она активно используется приложениями. На стадии создания API есть возможность вносить важные изменения, не волнуясь по поводу затрат из-за связанности. Расход ресурсов в это время также будет минимальным, потому что появление ошибок или дефектов мало на что влияет.

Однако скрытые затраты в стадии создания включают в себя растущую стоимость упущенной возможности, пока вы не доведете продукт с API до следующей стадии развития. Это может произойти, если вы не публикуете API для пользователей, потому что хотите больше времени потратить на различные аспекты дизайна, пока можно безопасно вносить изменения. Но если от вашего API зависит работа других команд, организаций или сотрудников, его отсутствие может стать проблемой. Часто оказывается, что публикация хорошего API сегодня полезнее для бизнеса, чем публикация отличного — завтра.

Продолжительность пребывания API на стадии создания становится важным решением по управлению продуктом. Необходимо сравнить важность свободы при создании дизайна и растущую стоимость упущенной возможности с затратами из-за связанности и расходом ресурсов, которые возрастут на последующих стадиях.

Проверенное правило — сначала разбираться с теми частями API, которые труднее всего изменять. Например, если вы создаете API на HTTP в стиле CRUD, нужно по максимуму разработать, протестировать и улучшить модель интерфейса на стадии создания, потому что позже затраты из-за связанности резко возрастают.

На стадии создания продукта с API необходимо собрать команду, которая станет его развивать. Всегда можно добавить в нее сотрудников, когда сложность продукта вырастет, но создание изначальной команды продукта — это важный шаг для API. Как будет сказано в главе 7, размер, качество и культура производства в ней сильно повлияют на создаваемый продукт. Важно максимально точно подобрать эти качества на ранней стадии его существования.

Ключевые моменты стадии создания. Жизнь любого API начинается с его создания, но нужно решить, когда конкретно наступит этот момент. Как определить начало пути продукта с API? Конечно, все может произойти само собой. Децентрализовать решение о создании и предоставить отдельным командам возможность разрабатывать конкурирующие продукты — совершенно нормально. Но вы можете определить какой-то минимальный уровень осмотрительности перед началом работы с продуктом с API.

Например, вы можете решить, что перед началом разработки следует определить стратегическую цель для каждого продукта. Это потребует централизовать какую-то часть решений — скорее всего, одобрение. Или можно ввести правило, по которому любой сотрудник организации может создать продукт с API, но работу можно начинать, только если он найдет еще троих человек, которые захотят потратить на него три месяца работы.

Определение ключевого момента создания во многом зависит от общей обстановки, но важно иметь общее понимание того, что требуется для запуска жизненного цикла продукта с API. Это поможет избежать напрасных вложений в продукты, которые не стоят ваших усилий.

Стадия 2: публикация

API на стадии *публикации* обладает следующими характеристиками.

- Экземпляр API развернут в рабочей среде.
- Она доступна одному или нескольким сообществам разработчиков.
- Стратегическая ценность API еще не окупилась.

Публикация API — это важный ключевой момент для продукта, который представляет собой точку входа во вторую стадию развития. API считается опубликованным, когда экземпляр становится доступен пользователям. Это момент, когда вы официально «открываете двери» в API и приветствуете всех заинтересованных.

Публикация невозможна без развертывания — реализации API в одном или большем количестве экземпляров, — но само по себе развертывание автоматически не считается публикацией. Например, можно разместить прототип дизайна API и на стадии создания, не объявляя, что он готов. Публикация происходит, когда вы сообщаете пользователям, что API открыт для бизнеса и готов к использованию.

Если вы создаете открытый API для сторонних разработчиков, то на этой стадии убеждаетесь, что разработчики, потребностям которых он отвечает, могут его найти и применять. Если ваш API — внутренний, это момент, когда его добавляют в каталог предприятия и доступ к нему открывается для команд других проектов. В случае API, поддерживающего одно приложение, именно на этой стадии вы сообщаете команде по разработке, что он стабилен и готов к использованию.

Открытие доступа для клиентских приложений — первый шаг к получению дохода от API. Но на стадии публикации этот доход потенциальный. Если сравнивать с магазином, то вы уже открыли двери, но еще ничего не продали. Нельзя получить доход от API без публикации, но публикация еще не гарантирует, что вы его получите. Создание API еще не значит, что целевая аудитория к вам придет.

Промежуток между публикацией API и получением дохода зависит от стратегии API. Если цель его создания нереалистична, вы застрянете на стадии публикации. Если цель проста, она будет достигнута при первом же применении. Окружение тоже играет важную роль. Если вы разрабатываете API для собственного приложения, то лучше контролируете его судьбу, а открытый API, созданный для сторонних организаций, потребует терпения и инвестиций. Но в любом случае целью должно быть как можно скорее перевести API в стадию *окупаемости*.

Возражения против скорейшего перехода к окупаемости может вызвать то, что изменения воздействуют на опубликованный продукт с API. Хотя на этой стадии у вас есть возможность влиять на зависимые от него клиентские приложения, эти клиенты еще не возмещают его стоимость вашему бизнесу. Таким образом, если вы вводите изменение, которое может иметь сильный эффект, краткосрочной потери дохода не произойдет. Некоторые организации видят в этом возможность ставить больше экспериментов, собирать больше данных и сильнее рисковать.

Однако нужно осторожно вносить изменения на стадии публикации, зная о возможных долгосрочных эффектах. Существующие клиенты могут потенциально принести доход, если продолжат использовать API, но избыток изменений способен оттолкнуть их еще до этого момента. Если продукт с API существует на конкурентном рынке, это тоже может негативно повлиять на возможность привлечь самые нужные типы клиентов для реализации вашей стратегии.

Обычно в этот момент API легко изменить, потому что главные пользователи еще не активизировались. Но имейте в виду, что изменение качества API на этой стадии может иметь неожиданные последствия и не позволить вам получить от целевой базы пользователей желаемые инвестиции. Экземпляр API, который

часто не работает или меняет модель интерфейса, вредя клиентским приложениям, подает явные (и не положительные) сигналы потенциальной базе пользователей.

Уровень изменений в API на стадии публикации должен зависеть от самих API, их изменяемости, объема доступа (общий, частный или партнерский) и типов пользователей, для которых они предназначены.

Ключевые моменты стадии публикации. Ключевые моменты, которые вы выберете для стадии публикации, должны определять, когда API готов к активному использованию. Нужно будет решить, какое переключающее событие будет обозначать эту готовность. Вот несколько примеров:

- API загружен в рабочую среду;
- сайт API заработал;
- API зарегистрирован в корпоративном реестре;
- открытие доступа к API анонсировано по электронной почте.

К тому же стоит выбрать параметры, определяющие, что API уже используется. Это поможет выяснить, как изменения, которые вы хотите внести на стадии публикации, повлияют на пользователей. Например, полезными параметрами могут быть реестр пользователей, количество обращений и просмотров документации.

Стадия 3: окупаемость

API на стадии *окупаемости* обладает следующими характеристиками.

- Опубликованный экземпляр API создан и доступен.
- Ее использование выполняет поставленную перед ней основную задачу.
- Доход, получаемый с помощью нее, имеет тенденцию возрастать.

Подход к API как к продукту — это постоянное улучшение, призванное поддержать цели бизнеса. До этого момента ваш продукт с API обладал *потенциальной* ценностью. Но когда целевая аудитория начинает использовать его таким образом, который соответствует вашим стратегическим целям, вы наконец можете считать его ценность *реализованной*.

Окупаемость API — это финальная цель. Максимально быстрый переход к стадии окупаемости и максимально долгое получение дохода — это отличительные признаки API с высокой ценностью. Задача владельца API — решить, что именно является окупаемостью. Для данной стадии сложно определить параметры, потому что для этого владелец продукта должен прекрасно разбираться в целях API.

Правильное определение целей позже становится важным шагом к окупаемости — или как минимум к умению измерить свою способность создавать ценные API

и управлять ею. Видимость и прозрачность ваших продуктов крайне важны, если вы управляете группой API одновременно, поэтому измерения окупаемости имеют большое значение.

Например, целью окупаемости API для платежей, который рекламировали сторонним разработчикам как платный продукт, могут быть 10 000 платных платежей за месяц. По этому параметру владелец продукта может четко понять, что, даже если 6000 разработчиков зарегистрировались, чтобы использовать API, 5000 запросов на платежи в месяц означают, что API не окупился.

А у API, применяемого только внутри одной организации, будет совершенно другая цель окупаемости. Например, API для платежей, задействуемый внутри банковской архитектуры ПО, может иметь такую цель, как проведение платежей онлайн-банкинга. В этом примере, как только система онлайн-банкинга начинает использовать этот API для платежей, он считается окупившимся вне зависимости от количества запросов.

Картину усложняет то, что цель окупаемости должна не только отражать общую обстановку, в которой действует API, но и постоянно пересматриваться и проверяться с изменением этой обстановки. Например, API для платежей, который вы с выгодой для себя передаете сторонним разработчикам, потребует изменения цели окупаемости при изменении бизнес-стратегии, лежащей в его основе, например, если вы решите, что долгосрочная стабильность требует от вас нацеленности прежде всего на рынок конкретного предприятия. Тогда соответствующая ключевая цель окупаемости может измениться примерно на такую: «Провести 500 запросов на платежи в компании Fortune 500».

Ключевые моменты стадии окупаемости. Чтобы создать KPI, определяющий, достигли ли вы этой стадии, понадобится очень хорошо представлять, для кого вы создаете API. Целевую аудиторию API будет довольно просто определить, если вы смогли поставить цели с разумным уровнем четкости. Это не значит, что необходимый вам пользователь API должен подходить под конкретные параметры образа пользователя, — многие API максимально подстраиваются, чтобы подходить всем и каждому. Важно быть уверенными в том, какой тип доступа пользователей означает, что этот API окупился.

В этот момент полезно также измерить уровень заинтересованности пользователей в вашем API. На самом деле основная цель API на этой стадии — обеспечить уровень заинтересованности, ведущий к использованию продукта на законном основании, что бы это ни означало в вашем случае.

API вошел в стадию окупаемости, но ваша работа на этом не закончена. Успех — это то, что вы продолжите получать доход от продукта. Для этого нужен набор параметров, который поможет отслеживать прогресс и в соответствии с полученными данными принимать решения по управлению продуктом. OKR и KPI, которые мы обсуждали ранее в этой главе, лучше всего применять к API именно на этой стадии.

Стадия 4: поддержка

API на стадии *поддержки* обладает следующими характеристиками.

- Он активно используется одним или несколькими клиентскими приложениями.
- Его окупаемость находится в стагнации или стремится к уменьшению.
- Его уже активно не улучшают.

Пока API приносит доход, он остается на стадии окупаемости. Но однажды скорость роста замедлится и API перейдет в стабильную фазу или начнется уменьшение применения, приносящего ценность. Когда это случается, API переходит в стадию *поддержки*.

Сейчас API все еще нужно изменять, но цель изменений на стадии поддержки слегка отличается от цели на стадии окупаемости. Теперь изменения вносят, чтобы максимально долго удерживать API в стабильном состоянии. Это могут быть исправления ошибок, модернизация и изменения ради соответствия каким-то требованиям, но на привлечение новых пользователей будет направлена очень небольшая их часть.

Внесение изменений на стадии поддержки требует особой осторожности: необходимо убедиться в том, что пользователям API, которые все еще приносят доход, эти изменения не повредят. На этой стадии лучше не рисковать. Если требуется значительное изменение, вам может понадобиться перенести API обратно на стадию публикации и попробовать снова получить потерянный доход (иногда для этого выпускают новую версию API).

Ключевые моменты стадии поддержки. Ключевые моменты на стадии поддержки будут зависеть от ключевых моментов стадии осуществления и в основном основываться на тенденциях. Например, если у вас уже есть параметры прироста количества пользователей для стадии окупаемости, то на стадии поддержки вам помогут соответствующие измерения прироста за последние полгода. Если количество пользователей не растет или падает, это может быть признаком того, что API вошел в стадию поддержки. Нужно будет определить, какие параметры станут ключевыми показателями, за какой период и в какой момент начинается стагнация.

Стадия 5: удаление

API на стадии *удаления* обладает следующими характеристиками.

- Опубликованный экземпляр API создан и доступен.
- Доход от нее уже не оправдывает продолжение поддержки.
- Принято решение об окончании ее эксплуатации.

Все когда-нибудь заканчивается, и, возможно, ваш продукт с API тоже однажды устареет. API может войти в эту стадию по многим причинам, включая потерю спроса, изменение затрат на поддержку, появление новых, более подходящих альтернатив и смену целей вашего бизнеса. Все эти сценарии можно посчитать причинами невозможности поддерживать окупаемость или фундаментальными изменениями цели продукта с API.

Когда в цикле развития API входит в стадию удаления, это признак того, что его *нужно удалить*, а не того, что он уже исчез. Команда этого продукта должна спланировать и выполнить удаление API из списка работающих и доступных. Команда может решить, в чем на самом деле заключается удаление API, но обычно целью является максимальное снижение связанных с ним затрат. Иногда для этого нужно удалить все экземпляры данного API с серверов производства, а в других случаях — просто отметить его как неактивный и перестать вносить дальнейшие изменения или предоставлять для него техническую поддержку.

Решение о том, в чем заключается удаление, обычно зависит от затрат на этой стадии, включающих в себя то, что вы забираете что-то полезное у пользователей продукта с API. Решение удалить API, от которого зависят другие, может быть сложным для выполнения. Владелец API, используемых внутри технической архитектуры организации, может не иметь прав на удаление экземпляра, так как последствия могут оказаться неожиданными. Если API открытый, организация может беспокоиться за репутацию своей торговой марки и доверие сообщества пользователей, которое можно утратить, удалив существующую функцию.

С точки зрения продукта удаление API не нужно считать провалом или ошибкой. Это естественная часть цикла непрерывного улучшения всей системы API.

Ключевые моменты стадии удаления. Как и на других стадиях развития API, важно, чтобы команда продукта определила ключевые моменты, которые покажут, что API находится на стадии удаления. Они могут быть связаны с работой API (например, количество сообщений, обработанных за какой-то период времени) или с затратами (например, затраты на улучшение API для достижения целей бизнеса).

Компания Google известна тем, что удаляет продукты и проекты, не соответствующие измеримым параметрам за определенный период времени. В Google таким параметром может быть количество активных пользователей (в сотнях тысяч), а ожидаемый прирост пользователей — довольно большим. Подобные параметры имеют смысл для стратегии, требующей значительного прироста пользователей, но не подойдут внутреннему API для идентификации пользователей.

Ключевые моменты для стадии удаления могут показывать нижнюю или верхнюю границу параметра. Например, вы можете установить минимальное количество запросов, которые должен обработать API на стадии поддержки, или максимальный уровень затрат, достигнув которого продукт войдет в стадию удаления. Затраты на удаление продукта сильно варьируются в зависимости от типов приложений, которые

он поддерживает, и количества пользователей-разработчиков. Поэтому нужно будет самостоятельно установить эти границы, основываясь на уникальной среде своего API.

Применение жизненного цикла продукта к столпам

Жизненный цикл продукта с API, который мы только что описали, помогает понять, каковы пути развития вашего продукта. Это поможет при планировании затрат на изменения в API на каждой стадии. Жизненный цикл продукта также поможет управлять работой, которую нужно проделать для вашего API. В данном разделе используем десять столпов разработки продукта с API, которые описали в главе 4, чтобы показать, как будет меняться эта работа в зависимости от стадии жизненного цикла продукта.

Столпы работы по управлению API важны на каждой стадии этого цикла. У вас не получится проигнорировать ни один из них. Однако на определенных стадиях некоторые столпы важнее других (табл. 6.2). На этих столпах стоит сосредоточиться и, возможно, больше в них инвестировать в течение определенных стадий жизненного цикла продукта с API.

Таблица 6.2. Влияние столпов в зависимости от стадии жизненного цикла

Столп	Создание	Публикация	Окупаемость	Поддержка	Удаление
Стратегия	✓				✓
Дизайн	✓	✓			
Разработка	✓	✓			
Развертывание		✓	✓		
Документация			✓		
Тестирование	✓		✓		
Безопасность	✓				
Мониторинг		✓		✓	
Обнаружение		✓	✓		
Управление изменениями			✓		✓

Работа со столпами

Столпы, которые мы выделяем в этих подразделах, — не единственные аспекты, над которыми нужно работать. Вы будете вносить изменения и улучшения в API в течение всего жизненного цикла. Возможно, придется работать над всеми столпами на всех стадиях существования API. Наша цель — показать, какие столпы сильнее влияют на разных стадиях, чтобы вы смогли соответственно спланировать вложение в них времени и сил.

Создание

На стадии создания вы сосредотачиваетесь на разработке лучшей модели API, прежде чем привлечь активных пользователей. Здесь понадобится особое внимание к стратегии, дизайну, разработке, тестированию и безопасности.

Стратегия

На стадии создания необходимо прежде всего развивать стратегию. Когда она будет разработана, вы получите еще очень мало отзывов об использовании продукта с API, потому что бóльшая часть работы в это время посвящена дизайну и реализации. Из-за недостатка данных о стратегии она практически не будет меняться на этой стадии. Иначе будет, только если затраты на реализацию стратегии окажутся слишком высокими. Например, вы можете понять, что создавать дизайн и реализацию, соответствующие вашей стратегической цели, будет непрактично. В таком случае придется вносить изменения.

На стадии создания вы:

- разрабатываете изначальную стратегию;
- проверяете, практично ли создавать по ней дизайн и реализацию;
- обновляете свои цели и тактику, основываясь на их выполнимости.

Дизайн

В главе 5 мы описали, как становится все сложнее менять модели интерфейса API после начала его активного использования. Именно поэтому дизайн модели интерфейса так важен на ранних стадиях жизненного цикла продукта с API. Если вы разработаете качественный дизайн во время его создания, у вас будет больше возможностей для доработки, улучшения и инноваций на ранних стадиях.

Одной из сложностей разработки дизайна на стадии создания является большое количество допущений. Вы предполагаете, что решения по дизайну модели интерфейса, которые вы приняли, будут понятны разработчикам. А также думаете, что этот дизайн будет практичен в реализации. К сожалению, предположения часто оказываются ложными.

Чтобы улучшить дизайн интерфейса на такой ранней стадии, вам нужно будет оценить модель. Понадобятся отзывы команды по реализации о том, что созданный вами дизайн выполним, — в идеале эта оценка включает в себя разработку прототипов, к которым можно делать запросы. Вам также понадобятся отзывы от разработчиков, представляющих целевую аудиторию.

На стадии создания вы:

- разрабатываете изначальную модель интерфейса;
- тестируете дизайн с точки зрения пользователя;
- оцениваете реализуемость этой модели.

Разработка

На стадии создания разработка заключается в реализации модели интерфейса. Как мы уже сказали, она может включать в себя также разработку прототипов для тестирования дизайна. Основная цель разработки на первой стадии существования продукта — создать работающую реализацию, которая предоставляет все функции, описанные в модели интерфейса. Но, чтобы получить долгосрочную ценность, реализация должна также сокращать затраты на поддержку кода, данных и инфраструктуры и изменения в них.

На стадии создания вы:

- получаете прототипы;
- тестируете дизайн интерфейса с точки зрения реализации;
- разрабатываете первоначальную реализацию API.

Тестирование

На стадии создания вам нужно будет тестировать дизайн интерфейса и первоначальную реализацию. Это позволяет обнаружить проблемы, возникающие при задействовании API, и улучшить дизайн на ранней стадии. Затраты на тестирование удобства использования могут варьироваться, как и при любой проверке качества. Дорогостоящая версия может включать в себя лабораторные тесты удобства применения, фокус-группы, исследования и интервью. Более дешевая версия может предусматривать просто написание кода для API.

Нужный уровень инвестиций должен определяться ценностью, которую вы получите, улучшая качество. Если вы работаете на высококонкурентном рынке API и у ЦА много вариантов выбора, то имеет смысл инвестировать в удобство использования. Если вы разрабатываете API только для себя, стоит провести ровно столько тестов, сколько нужно, чтобы подтвердить свои предположения по поводу дизайна. Но в любом случае тестировать эти предположения необходимо, чтобы избежать роста затрат на изменения модели интерфейса.

Вы можете решить протестировать и реализацию, но на стадии создания ее качество еще не очень важно. API пока не опубликован для пользователей, поэтому можно отложить тестирование реализации на более позднее время. Это не означает, что тестировать код на стадии создания не нужно. Использование таких методов, как

разработка на основе тестирования, может улучшить качество реализации в долгосрочной перспективе. Мы имеем в виду, что это решение нужно принимать, основываясь на собственном рабочем окружении.

На стадии создания вы:

- определяете стратегию тестирования модели интерфейса и осуществляете ее;
- определяете стратегию тестирования реализации.

Безопасность

О безопасности стоит сказать следующее: разумнее всего инвестировать в нее в течение всего существования API. Количество необходимых для этого усилий будет зависеть от ограничений, наложенных на вас сферой вашей деятельности, правительством и конкурентным рынком, но трудно представить сценарий, при реализации которого не потребуется вообще ничего предпринимать для обеспечения безопасности. Вам в любом случае придется работать над ней, чтобы защитить себя, свою систему и пользователей.

Многое нужно сделать до публикации API. Открыть двери в свою программу и только потом подумать о безопасности — неудачное решение. Поэтому мы выделили стадию создания как самую важную для столпа «Безопасность API». Это может показаться нелогичным, но мы считаем, что работа по обеспечению безопасности на этой стадии имеет наивысшее значение и дает наилучшие шансы на успех. Безопасность больше всего важна для работающего экземпляра API, но ее фундамент закладывается при разработке дизайна и реализации.

На стадии создания работа по безопасности должна быть сосредоточена на применении политики безопасности к предложенному вами дизайну. Если в вашей сфере или организации нет четких требований, нужно определить их самостоятельно. Сейчас важно сделать безопасность вопросом первостепенной важности при создании дизайна и реализации.

Работа по реализации на стадии создания должна включать в себя разработку безопасной инфраструктуры для вашего API. Сюда входят функции контроля доступа, а также контроля чрезмерного использования, из-за которого ваша услуга может стать недоступной для законных пользователей. Ни один API не может быть слишком незаметным или неважным, чтобы подвергать его рискам уязвимости. Компоненты, которые посчитали незначительными и не стоящими вложений в безопасность, позже становятся прекрасными мишенями для применения в незаконных целях.

На стадии создания вы:

- определяете требования к безопасности;
- тестируете модель интерфейса на соответствие требованиям безопасности;
- определяете стратегию защиты первоначальной реализации и экземпляров.

Публикация

Стадия *публикации* — это момент открытия дверей в ваш продукт с API. Начинается его официальное использование. На этой стадии другие люди начинают зависеть от вашего API и писать код, основываясь на той модели интерфейса, которую вы им рекламировали. Сейчас важнее всего столпы дизайна, разработки, развертывания, документации, мониторинга и обнаружения.

Дизайн

Хотя большая часть работы над дизайном происходит на стадии создания, она важна и на стадии публикации, потому что позволяет улучшить дизайн интерфейса, основываясь на реальном применении. Опубликовав API, вы поймете, правильными ли были ваши предположения. Что-то обнаружится уже при тестировании на стадии создания, но вы узнаете много нового, когда реальные пользователи возьмут ваш API в руки.

Разумеется, вы будете вносить изменения в интерфейс на протяжении всего существования продукта. Каждый раз, когда потребуется добавить новую характеристику, улучшить существующую операцию или упростить использование, вы станете менять модель интерфейса. Но такие изменения проще вносить на стадиях создания и публикации. Стадия публикации — это последняя возможность внести важные изменения в дизайн, нанося минимальный вред или хотя бы не влияя на пользователей, которые приносят доход.

На стадии публикации вы:

- анализируете простоту использования интерфейса;
- тестируете предположения по дизайну, сделанные на стадии создания;
- улучшаете модель интерфейса, основываясь на том, что узнали.

Разработка

Если вы меняете интерфейс, в итоге придется менять и реализацию. Но это не самая интересная часть столпа «Разработка» на стадии публикации. Мы выделяем его, потому что на этой стадии проще всего оптимизировать реализацию независимо от модели интерфейса. У вас есть возможность улучшить работу реализации и упростить изменения и увеличение масштаба.

Разумеется, можно сделать это и на стадии создания, но публикация дает преимущество — вы можете основать оптимизацию на реальном использовании программы. В отличие от модели интерфейса, реализацию можно менять понемногу и постепенно. Таким образом вы избежите большого объема работы по предварительному написанию кода. Вместо этого можно оптимизировать его небольшими

фрагментами по мере того, как вы больше узнаете о том, что нужно улучшать. Конечно, вы станете оптимизировать реализацию в течение всего жизненного цикла API, но на стадии публикации можно сделать максимум с минимальным риском.

На стадии публикации вы:

- оптимизируете реализацию с точки зрения масштабирования и работы программы;
- оптимизируете реализацию с точки зрения изменений;
- делаете это, основываясь на наблюдениях за использованием.

Развертывание

API нельзя считать опубликованным, если экземпляр не был развернут. Таким образом, это ключевой столп стадии публикации. Как минимум вам нужно убедиться, что экземпляр доступен пользователям, а также стоит начать создание инфраструктуры для развертывания, которая будет поддерживать дальнейший рост. Это особенно важно, если стратегическая цель API включает в себя увеличение объема использования. Например, достижение какой-то цели в области дохода или инноваций наверняка потребует архитектуры разработки, которая выдержит повышенный спрос.

Один из аспектов разработки — создание процесса, который позволит вносить в API изменения (помните о том, что важно поддерживать высокую скорость изменений). Разработка и создание этого процесса должны в идеале начаться на стадии создания продукта, но на стадии публикации он требуется уже более срочно.

Другой аспект разработки — ввод экземпляров API в эксплуатацию. Под ним подразумеваются создание и поддержка системы, соответствующей требованиям по масштабу, доступности и изменяемости вашего продукта. В хорошей системе эксплуатации API будут доступны и в рабочем состоянии даже при росте количества запросов к системным ресурсам. Поддержка рабочего состояния экземпляров — важнейшая часть формирования качественного опыта разработчика. API, который часто недоступен или необъяснимо медленно работает, вряд ли сможет добраться до стадии окупаемости.

На стадии публикации вы:

- развертываете экземпляр API;
- фокусируетесь на открытии доступа к API;
- планируете и разрабатываете развертывание с учетом будущего спроса.

Документация

Работать с документацией придется в течение всего жизненного цикла API, но особенно важно это на стадиях публикации и окупаемости. На стадии публикации

вам нужно будет увеличивать доход от API, привлекая нужных пользователей. Это дает возможность поэкспериментировать с дизайном документации и придумать, что поможет привлечь этих пользователей.

Таким образом, вы начинаете с низкого уровня развития документации и создаете ее в процессе изучения пользователей вашего API. Например, вначале можете предложить только техническую справку, но, основываясь на наблюдении за использованием, добавить обучающие курсы и примеры. В частности, это позволяет разобрать проблемные места или точки развития API в документации. Вы можете найти их, если на стадии развития инвестируете в пользовательское тестирование или на стадии публикации изучите вопросы пользователей.

На стадии публикации вы:

- публикуете документацию;
- улучшаете ее, основываясь на реальном использовании API.

Мониторинг

Отзывы о продукте очень важны на стадиях публикации и окупаемости жизненного цикла API. На стадии публикации вам нужны качественные параметры, чтобы определить, достигли ли вы ключевого момента осуществления. На стадии окупаемости нужны данные, позволяющие убедиться в том, что спрос на API и доходы от него все еще растут. Мониторинг полезен на протяжении всего жизненного цикла продукта, но на этих конкретных стадиях он жизненно важен. Обычно на обеих стадиях работают с одними и теми же параметрами, поэтому, если здесь вы вложитесь в качественный мониторинг, то позже сможете применить это решение еще раз.

На стадии публикации вы:

- разрабатываете и реализуете стратегические параметры для API;
- разрабатываете и реализуете систему мониторинга для API;
- создаете систему мониторинга, которую можно использовать и на стадии окупаемости.

Обнаружение

Обнаружение — самый зависимый от ситуации из всех десяти столпов. Работа по обнаружению — это то, что вы предпринимаете для продвижения продукта с API, создания отношений с разработчиками и общего укрепления связи API с целевой аудиторией. Если вы разрабатываете API для собственной команды, для его обнаружения вам нужно просто отправить e-mail. Если создаете его для большого предприятия, надо будет пройти процесс поступления и регистрации новых сервисов.

Если ваша аудитория еще более обширна, может понадобиться нанять команду из десяти человек для создания и реализации маркетинговой стратегии. Таким образом, вложения сил и средств могут очень сильно варьироваться.

Но в любом случае, независимо от потраченных сил, обнаружение важнее всего на стадии *публикации*. Именно тогда вам нужно максимально заинтересовать пользователей своим API, потому что у вас уже есть доступные экземпляры и правильное использование может помочь создать реализованную ценность продукта. Но как происходит обнаружение и сколько вы в него инвестируете, полностью зависит от вашей ситуации.

На стадии публикации вы инвестируете в маркетинг, отношения с пользователями и поисковую доступность API.

Окупаемость

Дойти до стадии *окупаемости* — цель каждого продукта с API. Теперь главная задача — увеличить доход, получаемый от API, при этом не влияя на пользователей, которые больше всего вам в этом помогают. Самые важные столпы на этой стадии — развертывание, документация, тестирование, обнаружение и управление изменениями.

Развертывание

Когда ваш API уже окупился, необходимо, чтобы система была доступной и всегда в рабочем состоянии. Поэтому очень важной становится архитектура развертывания. На стадии публикации вы создавали изначальный дизайн развертывания, а сейчас фокусируетесь на его поддержке и улучшении. Вы должны поддерживать рабочее состояние своего сервиса, даже если спрос неожиданно меняется. Такие изменения могут потребовать в том числе создания новой реализации. Это совершенно нормально, если вы можете защитить самых полезных пользователей от негативных последствий.

На стадии окупаемости вы:

- убеждаетесь, что экземпляры API все время доступны;
- постоянно улучшаете и оптимизируете архитектуру развертывания;
- при необходимости улучшаете реализацию.

Документация

Стадия окупаемости — это возможность продолжить улучшать опыт разработчика для своего продукта, в частности улучшая документацию и обучение. Изменение модели интерфейса в этот момент становится сложнее, но изменение документации

гораздо меньше влияет на пользователей. Люди куда лучше приспосабливаются к изменениям, чем программное обеспечение, поэтому у вас есть возможность экспериментировать с новыми форматами, стилями, инструментами и презентацией. Цель — сохранить уровень применения программы, уменьшая пробелы в знаниях новых пользователей.

На стадии окупаемости вы:

- продолжаете улучшать документацию;
- экспериментируете с дополнительными ресурсами поддержки, например, программами для анализа API, клиентскими библиотеками, книгами, видео;
- привлекаете новых пользователей, уменьшая пробелы в их знаниях.

Тестирование

На стадии окупаемости тестирование предотвращает негативные последствия для пользователей от изменений любой части API. Здесь использование API напрямую приносит вам доход. Изменения необходимы, но при этом нужно снизить риск нежелательных последствий. Размер вложений в такое тестирование должен быть основан на уровне негативных последствий.

В идеале тесты, которые вы проводите на стадии окупаемости, должны быть разработаны еще на стадиях публикации и создания API. Но, когда ваш продукт приближается к стадии окупаемости и входит в нее, нужно оценить стратегию тестирования, чтобы убедиться, что она обеспечивает оптимальный уровень снижения риска. Когда API достигнет стадии поддержки и удаления, необходимость в тестировании снизится. На этих стадиях вы сможете полагаться на уже созданные ресурсы.

На стадии окупаемости вы:

- применяете стратегию тестирования к изменениям в интерфейсе, реализации и экземпляре;
- постоянно улучшаете способы тестирования;
- создаете решения по тестированию, которые можно использовать на следующих стадиях.

Обнаружение

Обнаружение на стадии окупаемости похоже на обнаружение на стадии публикации. Отличие только в том, что здесь работа должна быть более точной. Вы станете лучше понимать, какие сообщества пользователей приносят больший доход, поэтому можете больше инвестировать в отношения именно с ними.

На стадии окупаемости вы:

- продолжаете инвестировать в маркетинг, отношения с пользователями и поисковую доступность;
- больше инвестируете в ценные сообщества пользователей.

Управление изменениями

Главная часть жизненного цикла API — растущее влияние изменений на него. На самом деле на протяжении всего раздела мы описывали управление изменениями в каждом из остальных столпов продукта с API. Но в целом, когда речь идет о самом столпе «Управление изменениями», он становится максимально важным на стадии окупаемости.

В главе 5 мы описали четыре типа изменений, которыми вам нужно управлять: изменения модели интерфейса, реализации, экземпляра и ресурсов поддержки. В рамках каждого столпа вы вносите изменения в некоторые из этих частей API, зачастую одновременно. Ими нужно управлять, чтобы снизить негативный эффект, и это важнее всего, когда продукт активно используется и приносит доход. Здесь проявляется вся ценность системы управления изменениями и стратегии контроля версий.

На стадии окупаемости вы:

- разрабатываете и применяете систему управления изменениями;
- сообщаете об изменениях пользователям, службе техподдержки и спонсорам;
- поддерживаете изменения с целью уменьшения влияния на доход.

Поддержка

На стадии поддержки вы уже не получаете нового дохода, но не должны вредить существующим пользователям. Ваша цель в этот момент — поддерживать систему в рабочем состоянии. Для этого нужна большая работа, но самая важная задача — это мониторинг.

Мониторинг. Если API находится на стадии поддержки, единственной задачей является поддержание статус-кво. Меньше внимания уделяется дизайну, разработке и изменениям и больше — поддержке и доступности. В этот момент вам не обязательно улучшать мониторинг, потому что бóльшая часть этой работы выполнена на стадиях публикации и окупаемости. Тем не менее это самый важный столп для стадии поддержки, поэтому необходимо затратить время и приложить силы, чтобы убедиться, что вы получаете нужные данные на уровне системы и продукта.

Одна из задач — добиться того, чтобы система сообщала вам, когда случается что-то необычное. Это значит, что нужно что-то сделать. Другая цель мониторинга на стадии поддержки — следить за ценностью, которую предоставляет API. Когда она становится слишком низкой, возможно, пора его удалять.

На стадии поддержки вы:

- убеждаетесь, что система мониторинга работает;
- определяете схемы, которые потребуют особого внимания;
- наблюдаете за параметрами, которые могут показать, что пора принимать решение об удалении.

Удаление

Хоть это и последняя стадия жизненного цикла, помните: API еще существует. Сейчас вы решаете, что продукт с API должен быть удален. Самыми важными столпами на этой стадии являются стратегия и управление изменениями.

Стратегия

Когда приходит время удалить API, необходимо решить несколько конкретных стратегических вопросов. Как поддержать существующих пользователей, как компенсировать им потери и успокоить их? Существует ли новый API, который могут применять эти пользователи? Когда и каким образом будет удален API? Как сообщить базе пользователей о приближающемся удалении? Независимо от масштаба, рабочего окружения и ограничений API нужно будет создать стратегию удаления, хотя бы минимальную и неофициальную.

Для этого следует определить новые цели, тактику и план действий. Изначальная цель API, поставленная на стадии создания, уже не является вашей задачей. Вместо этого нужна цель удаления продукта. Например, ею может стать минимизирование количества потерянных пользователей, если вы хотите перевести их на новый API. Или скорейшее уменьшение затрат на поддержку продукта. Каждая из этих совершенно разных целей требует тактического плана и действий для их достижения.

На стадии удаления вы:

- определяете стратегию удаления (или перехода);
- определяете новую цель, тактический план и действия, необходимые для ее достижения;
- измеряете прогресс на пути к этой цели.

Управление изменениями

Управление изменениями на стадии удаления означает управление эффектом от удаления продукта. Сейчас не время улучшать API, поэтому мы фокусируемся не на контроле версий или выпуске нового варианта. Вместо этого требуется уменьшить влияние грядущего удаления на пользователей, ваши торговую марку и организацию и эффективно управлять этим изменением. Эта работа должна быть согласована со стратегией удаления.

На стадии удаления вы:

- снижаете эффект от удаления API;
- разрабатываете и реализуете план сообщения пользователям и деактивации экземпляра;
- управляете изменениями в реализации и готовой программе, необходимыми для деактивации.

Выводы

В этой главе мы описали жизненный цикл продукта с API, состоящий из пяти стадий работы успешного продукта с API. Рассказали также, что для определения уровня развития API нужны проработанные цели и параметры измерения. Наконец, описали, как управление одним продуктом API меняется на разных стадиях жизненного цикла. В следующей главе мы рассмотрим жизненный цикл API с точки зрения сотрудников и команд, которые с ним работают.

7 Команды по API

> Великие достижения в бизнесе не совершаются одним человеком. Они совершаются командой.
>
> *Стив Джобс*

Вы могли заметить, что мы откладывали на потом обсуждение создания и набора команд для работы над вашей программой с API и управления ими. Эта тема очень важна, но собрать и обобщить общую информацию о таком индивидуальном для каждой организации предмете весьма непросто. Каждая компания по-своему управляет сотрудниками, устанавливает собственные границы внутри организации (подразделений, отделов, продуктов, услуг, команд и т. д.) и создает собственную иерархию для сотрудников. Такое количество различий усложняет описание единого набора рекомендуемых методик построения успешных команд по API.

Однако, поговорив с представителями нескольких компаний, мы смогли определить общие схемы и методики, которыми можем поделиться с вами. По нашим наблюдениям, все организации описывают необходимую работу и сотрудников, ответственных за ее выполнение, с помощью названий каких-либо команд, должностей и рабочих обязанностей. Единой системы в названиях *должностей* сотрудников в разных компаниях мы не нашли, зато обнаружили довольно схожий набор *обязанностей* в командах. Другими словами, не столь важно, как называется должность, важно, что выполнять надо примерно одинаковую работу.

То, что нужно сосредоточиться на обязанностях, а не на должностях, говорил разработчик программного обеспечения, писатель и преподаватель Саймон Браун. В частности, он сказал: «Невозможно просто стать разработчиком ПО за одну ночь или после повышения. Это должностная обязанность, а не звание» (http://bit.ly/2K7okNi).

Наш опыт свидетельствует: это относится ко всем обязанностям в команде по API. Поэтому начнем эту главу с так называемого списка распространенных обязанностей в команде, работающей над API. Примерно так же, как мы представили

столпы API (см. главу 4), рассмотрим эти обязанности через призму распространенных задач, которые в вашей организации кто-то должен решать. Поэтому также обсудим, как столпы API соотносятся с этими обязанностями.

Мы также обнаружили, что иногда состав команды по API может варьироваться в зависимости от стадии развития API, над которым она работает. Например, на ранней стадии создания вам не нужно уделять внимание тестированию или DevOps, а на стадии поддержки обычно не так уж много работы для создателей клиентских и серверных приложений. Поэтому кратко рассмотрим, сотрудники с какими обязанностями могут понадобиться в течение всего жизненного цикла каждого из ваших API.

Еще один важный аспект — взаимодействие команд между собой. В большинстве компаний, с которыми мы работаем, есть некий отдел координации (или «команда по командам») — он помогает всем командам, отделам и подразделениям вне зависимости от стадии развития их API обмениваться информацией, управляет их взаимодействием и поощряет совместную работу. Дополнительный процесс «работы с работниками» будет подробно рассмотрен в следующих главах, когда мы познакомим вас с нашим пониманием системы API.

Наконец, для обеспечения продуктивного сотрудничества между командами необходимо то, что называется корпоративной культурой. Мы обнаружили, что успешные компании вкладывают много времени и ресурсов в управление этой сферой. Как говорилось в главе 2, расширить руководство API можно с помощью распределения элементов принятия решений. Один из способов создать единообразие в области принятия решений в таком случае — обращать много внимания на корпоративную культуру и, если требуется, умело направлять ее в нужное русло. В последнем разделе этой главы мы опишем несколько ключевых понятий, которые используют организации, чтобы определить корпоративную культуру, наблюдать за ней и влиять на нее для улучшения общей эффективности своих программ с API.

Но для начала определим набор распространенных обязанностей, которые мы видим в большинстве команд по API, и их применение для того, чтобы выполнить всю работу по столпам API, которые обсуждались в главе 4.

Обязанности в команде по API

Так же как мы продемонстрировали распространенные *навыки* для работы с вышеупомянутыми столпами API, мы создали и список распространенных *обязанностей* тех, кто работает с API. В этой главе данные обязанности представлены в виде списка должностей. Однако, как свидетельствует наш опыт, должности в командах по API не очень стандартизированы в рамках компаний. Программный менеджер API в одной организации может называться владельцем API в другой, а архитектор API в компании B будет архитектором продукта в компании Z и т. д.

Поэтому названия должностей, используемые здесь, могут не совпадать с названиями должностей в вашей компании. Однако мы практически уверены, что сами обязанности в вашей компании существуют или как минимум *должны* существовать, потому что как столпы API включают в себя все навыки, необходимые для создания успешной программы с API, так и описанные здесь обязанности *необходимо* выполнять какому-то участнику команды.

Таким образом, читая список обязанностей в команде по API и должностей, которые мы с ними соотнесли, вы можете сопоставить их с аналогичными внутри своей организации. Кстати, это очень хорошее упражнение. Если вы просмотрите свой список должностей и их описаний и обнаружите, что в нем не представлена одна или более описанных нами обязанностей, это явный признак того, что у вас есть возможность улучшить описание должностей в организации так, чтобы все эти обязанности были охвачены.

Объем ответственности

Помните, что каждая обязанность связана с определенным объемом ответственности. Когда кто-то выполняет обязанность, он берет на себя ответственность за все решаемые ею задачи. А большая часть задач включает в себя принятие решений с определенным набором навыков (дизайн, разработка, развертывание и т. д.).

Оставив это объяснение в качестве фона, пройдемся по списку обязанностей, чтобы вы в целом поняли, кто за что отвечает при создании работоспособной программы с API. Вы заметите, что мы разделили список на две части:

- деловые обязанности;
- технические обязанности.

Это разделение может показаться предвзятым и не соответствовать распределению должностей и ответственности внутри вашей организации. Но мы считаем, что это может помочь разобраться с тем, сотрудники с какими обязанностями занимаются достижением целей бизнеса (OKR), а с какими — выполнением технических задач (KPI). Мы обсуждали, как эти два понятия соотносятся друг с другом и используются в управлении API, в подразделе «OKR и KPI» раздела «Измерения и ключевые моменты» главы 6 (с. 123).

Напоминание по поводу обязанностей и должностей

Помните, что должности, которые мы перечисляем в книге, были придуманы, чтобы подчеркнуть связь между обязанностями в команде по API и *столпами API* из главы 4. Обязанности и должности внутри вашей организации, скорее всего, будут отличаться от них, однако столпы API включают в себя все навыки и сферы ответственности по API, которые необходимо охватить вашей компании. Как вы соотнесете столпы с должностями и обязанностями — вам решать, читатель.

Деловые обязанности

Первая группа обязанностей, о которой мы поговорим, — это так называемые деловые обязанности. Исполняющие их сотрудники в основном сосредоточены на деловой стороне API. Обычно их сфера ответственности включает в себя защиту интересов клиента, связь продукта с четкими стратегическими целями (например, продвижение новых продуктов, улучшение сквозных продаж и т. д.) и соотнесение вашего API с OKR компании. Иногда такие сотрудники работают в отделах, связанных с развитием бизнеса или с разработкой продукта. Иногда они занимают должности, связанные с IT. Важная разница между этими обязанностями и техническими, которые мы опишем далее, в том, что деловые обязанности связаны в первую очередь с целями бизнеса.

Мы определили пять деловых обязанностей по принятию решений, которые обычно существуют в командах жизнеспособных программ с API.

- *Менеджер продукта с API.* Менеджер продукта (МП) — иногда его называют владельцем продукта — основное контактное лицо API. Это связано с подходом «API как продукт» (AaaP), описанным в главе 3. Этот сотрудник несет ответственность за наличие у API четких OKR и KPI и за то, что остальные члены команды могут поддержать необходимые столпы API (глава 4). МП также ответственен за мониторинг API и его успешное проведение по всему жизненному циклу (глава 6). Обязанности менеджера продукта с API заключаются в том, чтобы дать ответ на вопрос «Что делать?» (то есть описать работу, которую нужно сделать) и разъяснить его остальным членам команды. За ответ на вопрос «Как это сделать?» будут отвечать участники команды с техническими обязанностями. Менеджер продукта также несет ответственность за то, чтобы ожидаемый опыт разработчика (дизайн продукта, знакомство с ним и последующие отношения) соответствовал требованиям пользователей API. Обязанность МП — убедиться, что все части совпадают, как должны.
- *Дизайнер API.* Дизайнер API ответственен за все аспекты дизайна. Он должен убедиться, что интерфейс функционирует, его можно использовать, опыт разработчиков в результате работы с ним будет положительным. Кроме того, дизайнер должен убедиться, что API помогает команде достичь определенных OKR бизнеса. Иногда дизайнер сотрудничает с техническими специалистами, чтобы удостовериться, что дизайн помогает команде достичь и технических KPI. Зачастую он также является контактным лицом для клиентов и может стать «голосом клиента», помогая команде принимать решения о внешнем виде API. Наконец, дизайнера могут попросить убедиться, что общий дизайн совпадает с указаниями по стилям, выпущенным внутри компании.
- *Технический писатель для API.* Технический писатель для API несет ответственность за создание документации по API для всех заинтересованных лиц, связанных с продуктом. В их число входят не только пользователи API (например, разработчики, использующие конечный продукт), но и участники внутренней

команды, и другие заинтересованные лица из организации (например, директор по информационным технологиям, технический директор и т. д.). У большинства технических писателей есть техническое образование и опыт программирования, но не у всех, и это не всегда требуется. Для технического писателя важно эффективно взаимодействовать с другими людьми, а также результативно искать информацию и брать интервью, так как им нужно понимать точку зрения и производителей, и пользователей API. Поэтому часто технические писатели плотно сотрудничают с дизайнером и менеджером продукта, чтобы убедиться, что документация точна и соответствует указаниям по стилям и дизайну внутри компании.

- *Евангелист API*. Евангелист API несет ответственность за продвижение и поддержку использования API и его культуры внутри компании. Это особенно необходимо в больших организациях, где внутренние пользователи не могут быстро получить доступ к команде, которая создала продукт. Евангелисты должны удостовериться, что все внутренние разработчики, задействующие API, понимают его и могут с помощью него достичь своих целей. Они также отвечают за сбор отзывов от пользователей API и их передачу остальным членам команды продукта. В некоторых случаях евангелисты могут нести ответственность за создание образцов, демоверсий, материалов для обучения и других ресурсов поддержки, предназначенных для улучшения опыта разработчика.
- *Специалист по отношениям с разработчиками*. Специалист по отношениям с разработчиками, иногда также называемый адвокатом разработчиков или специалистом по DR, обычно сосредоточен на внешнем использовании API (то есть вне создавшей его компании). Как и евангелисты API, специалисты по DR отвечают за создание образцов, демоверсий, материалов для обучения и других ресурсов для продвижения продукта. И опять же, как и евангелисты, зачастую именно специалисты по DR ответственны за получение отзывов от пользователей и преобразование этих отзывов в исправление ошибок или появление новых характеристик. Однако, в отличие от внутренних евангелистов, эти специалисты часто получают задание «продать» продукт с API более широкой аудитории, поэтому они выезжают в организации клиентов, проводят предпродажные мероприятия и осуществляют непрерывную поддержку продукта для важных клиентов. Их дополнительные обязанности могут включать в себя выступления на публичных мероприятиях, написание записей в блог или статей о применении продукта и другие действия для оповещения о вашей торговой марке, с помощью которых команда может достичь установленных бизнес-целей.

Хотя в основном эти пять должностей связаны с бизнес-целями и стратегиями, как вы видите из описаний, большая их часть требует также некоторого уровня технических знаний и навыков для достижения этих целей. Следующий список обязанностей, которые мы рассмотрим, сосредоточен только на технических аспектах создания, развертывания и поддержки продукта с API.

Технические обязанности

Второй список обязанностей, которые мы определили, — это так называемые технические обязанности. Они относятся в основном к техническим деталям реализации дизайна API, его тестирования, развертывания и поддержки в рабочем состоянии все время существования. Обычно сотрудники, выполняющие эти обязанности, ответственны за озвучивание решений IT-отдела, включая борьбу за безопасную и масштабируемую реализацию, которую можно будет должным образом поддерживать в течение длительного времени. Зачастую технические специалисты отвечают за достижение важных KPI, а также за помощь специалистам со стороны бизнеса в достижении их OKR.

Несмотря на то что мы разделили список ролей на две группы, между деловыми и техническими обязанностями существуют некоторые параллели. Например, у должности менеджера продукта есть параллель в виде должности ведущего инженера API с технической стороны. И у обеих групп обязанностей в области API есть главная цель — создание и развертывание технически стабильного и экономически оправданного продукта с API.

Мы определили шесть технических должностей, которые принимают основные решения в области реализации, развертывания и поддержки успешных API.

- *Ведущий инженер API.* Ведущий инженер API — это основное контактное лицо по всем вопросам, связанным с разработкой, тестированием, мониторингом и развертыванием продукта с API. Эта должность — эквивалент должности менеджера продукта, только с технической стороны. Так же как менеджер продукта ответственен за ответ на вопрос «Что делать?» — за цели по дизайну и цели организации, ведущий инженер API отвечает за ответ на вопрос «Как это сделать?» — что нужно, чтобы создать, разместить и поддерживать API. Ведущий инженер несет ответственность за координацию действий других технических специалистов внутри команды.
- *Архитектор API.* Архитектор API отвечает за архитектурные детали дизайна продукта и за то, чтобы API мог без труда взаимодействовать с нужными ресурсами системы, включая API других команд. Он также отвечает за всю архитектуру программного обеспечения и систем всей организации, в том числе ограничения по безопасности, параметры стабильности и надежности, выбор протоколов и форматов и другие так называемые нефункциональные элементы, установленные для систем ПО вашей компании.
- *Разработчик клиентской части приложения (фронтенд-разработчик).* Разработчик клиентской части приложения с API отвечает за то, чтобы оно обеспечивало получение качественного опыта пользователя. Для этого необходимо внести его в реестр API компании, на портал для пользователей, а также совершать другие действия, связанные с клиентской или пользовательской частью API. Так же как и дизайнер, разработчик клиентской части должен защищать интересы пользователей API, но с технической точки зрения.

- *Разработчик серверной части приложения (бэкенд-разработчик)*. Разработчик серверной части приложения отвечает за детали реализации интерфейса API, соединяя его с любыми другими сервисами, необходимыми для выполнения его работы, и в целом преданно реализуя идеи менеджера продукта и разработанное дизайнером описание того, что и как должен делать API. Он несет ответственность за то, чтобы API, запущенный в производство, был надежен, стабилен и единообразен.
- *Специалист по тестированию/ОК*. Специалист по тестированию/обеспечению качества (ОК) несет ответственность за все связанное с одобрением дизайна API и тестированием его функциональности, безопасности и стабильности. Обычно в обязанности этого специалиста входят написание самих тестов (или помощь в этом разработчику клиентской/серверной части) и обеспечение их эффективного запуска. Зачастую это не только лабораторные испытания и тестирование поведения, но и тесты на межоперационную совместимость, масштабируемость, безопасность и мощность. Обычно для них используются фреймворки и инструменты, отобранные отделом тестирования/ОК для всей компании.
- *Специалист по DevOps*. Специалист по DevOps ответственен за все аспекты создания и развертывания API, в том числе отслеживание работы API, чтобы убедиться, что он соответствует установленным техническим KPI и вносит свой вклад в OKR компании. Обычно для этого необходимы разработка инструментов для поставки ПО, написание скриптов сборки (или обучение этому других), управление расписанием релиза, архивация файлов, созданных во время сборки кода, и при необходимости поддержка откатов неудачных релизов. Специалист по DevOps отвечает также за поддержку сводной панели, показывающей данные отслеживания в реальном времени, а также за хранение и при необходимости поиск файлов регистрации API офлайн, чтобы помочь изучить, продиагностировать и решить любые проблемы, которые могут возникнуть, когда API запущен в производство. В зависимости от вариантов развертывания продукции вашей компании специалисты по DevOps должны будут поддерживать несколько режимов, включая рабочий режим, режим разработки, тестирования, перемещения данных и производства, как в локальной среде, так и в облачной системе.

В этом разделе мы представили работу, которую нужно выполнить в течение жизненного цикла API, в виде списка обязанностей или сфер ответственности. Чтобы было проще обсуждать, мы составили два списка обязанностей (деловых и технических) и дали им названия, похожие на распространенные реальные должности.

Как сказано в начале этого раздела, обязанности разграничивают области экспертного знания, которые должны быть охвачены при создании программ с API. Далее мы рассмотрим создание команды.

Команды по API

В предыдущем разделе мы определили 11 обязанностей, представляющих разные сферы ответственности. Командам нужны сотрудники для выполнения этих обязанностей, чтобы охватить все важные аспекты управления API в течение всего его жизненного цикла. Однако описанные нами обязанности отличаются от ролей реальных специалистов в команде. Нет необходимости в том, чтобы каждая обязанность в команде относилась к отдельному сотруднику. Некоторые могут исполнять более одной обязанности. Например, во многих организациях евангелист API и специалист по отношениям с разработчиками — зачастую один сотрудник. Еще один пример — в некоторых небольших командах одному человеку могут поручить обязанности по тестированию/QA и по DevOps.

Люди могут относиться к нескольким командам

В этом разделе мы опишем несколько конкретных команд, однако вам самим предстоит решить, как распределить сотрудников. Совершенно нормально укомплектовать каждую команду по API сотрудниками, которые будут работать в ней полный день и занимать только эту должность, но также нормально разрешить им работать в нескольких командах одновременно. Позже в этой главе мы расскажем вам о модели «отряды, племена, отделения и гильдии», принятой в компании Spotify, в которой используется матричный подход к участию в командах.

Может оказаться также, что команде не нужно охватывать все роли на всех стадиях развития API (см. главу 6). Например, в фазе поддержки жизненного цикла API вам обычно не нужна помощь разработчиков клиентской и серверной частей приложения. А в некоторых организациях часть обязанностей исполняют не участники конкретной команды, а «плавающие» сотрудники компании. Например, обязанности дизайнера может взять на себя один из сотрудников отдела дизайна продукции, работающий по запросу с любой командой по API, где требуется разработка дизайна.

Команды и развитие API

На каждой стадии жизненного цикла продукта с API некоторые обязанности первостепенны, а некоторые играют второстепенную роль. Первостепенны обязанности сотрудников, чьи решения могут иметь более ощутимый эффект. Например, на стадии создания почти все участники команды несут большую ответственность, но решения дизайнера по поводу дизайна интерфейса очень сильно влияют на всю работу.

Сотрудники с первостепенными обязанностями ответственны также за работу, которую нужно сделать обязательно. Например, на стадии публикации невозможно

разместить API, если никто не исполняет обязанности специалиста по DevOps и не создает архитектуру развертывания.

Как видите, формирование команд сильно зависит от развития API и от того, какие обязанности важны в конкретный момент. Учитывая это, пройдем по стадиям жизненного цикла API (см. главу 6) и определим на каждой из них первостепенные и второстепенные обязанности, а также действия, за которые будут ответственны выполняющие их сотрудники.

Стадия 1: создание

- *Первостепенные обязанности*: менеджер продукта, дизайнер, ведущий инженер.
- *Второстепенные обязанности*: евангелист API, DevOps, архитектор API, бэкенд-разработчик.

Стадия создания — это возможность разработать основную стратегию и качественный дизайн интерфейса, не влияя на реальных пользователей. Чтобы разработать лучшую стратегию API, вам нужен человек, хорошо понимающий общее состояние организации и сферу продукта с API и способный выбрать правильный курс действий. Обычно это менеджер продукта. Хороший менеджер продукта с API обладает достаточным опытом, чтобы определить цель API, которая поможет спонсирующей организации, а также разработать тактический план ее достижения.

Должность дизайнера нужна для разработки дизайна интерфейса. Хороший дизайнер API сможет принимать высококачественные решения по поводу дизайна модели интерфейса, основываясь на собственном опыте, то есть решать, как должна выглядеть модель и как нужно тестировать и одобрять предположения по поводу дизайна. Что самое важное, хороший дизайнер чувствует, сколько нужно инвестиций в дизайн, основываясь на общей ситуации.

Наряду с работой по дизайну интерфейса кто-то должен создать и первую реализацию API (табл. 7.1 и 7.2). Часть этой работы будет по своей природе экспериментальной и исследовательской. Например, дизайнер может решить, что для испытания дизайна необходимо разработать прототип. А еще создать первую реализацию API. Именно она в итоге будет опубликована на следующей стадии. Эта разработка обычно требует командных действий, а управляют ими архитектор и ведущий инженер API.

Таблица 7.1. Первостепенные действия на стадии создания

Действие	**Сотрудник (-и)**
Разработка стратегии	Менеджер продукта
Разработка модели интерфейса	Дизайнер
Разработка изначальной реализации	Архитектор API, ведущий инженер, бэкенд-разработчик

Таблица 7.2. Дополнительные действия на стадии создания

Действие	Сотрудник (-и)
Разработка прототипов	Ведущий инженер, бэкенд-разработчик
Тестирование реализуемости дизайна	Архитектор API, ведущий инженер, бэкенд-разработчик, технический писатель
Тестирование безопасности дизайна и реализации	Архитектор API, специалист по тестированию/ОК
Тестирование реализуемости дизайна на рынке	Евангелист API, специалист по ОР
Тестирование простоты использования дизайна	Дизайнер
Планирование и реализация стратегии тестирования реализации	Ведущий инженер, специалист по тестированию/ОК

Стадия 2: публикация

- *Первостепенные обязанности*: менеджер продукта, технический писатель для API, DevOps.
- *Второстепенные обязанности*: разработчик клиентской части, дизайнер, бэкенд-разработчик, евангелист API, специалист по ОР.

Если вы достигли стадии публикации, значит, готовы предоставить пользователям доступ к своему продукту. Для этого вам нужны люди, разбирающиеся в развертывании, документации и действиях, гарантирующих обнаружение API. Есть также множество дополнительных действий, которые стоит совершить, если это ценный API и у вас есть возможность их выполнить.

Важная часть работы на этой стадии — публикация изначального набора документации, поэтому нужен сотрудник, который будет исполнять обязанности технического писателя — создавать и публиковать документы. Технический писатель — ключевая обязанность на стадии публикации. Хороший специалист упростит начало работы с программой потенциальным пользователям и поможет существующим ускорить работу. На этой стадии вам определенно нужен этот результат, потому что так вы быстрее дойдете до стадии окупаемости.

Для публикации API необходимо разместить готовые экземпляры API. Обычно это обязанность специалиста по DevOps. На этой стадии он отвечает за разработку процесса развертывания, контроль выполнения и архитектуру развертывания экземпляра API (табл. 7.3).

Таблица 7.3. Первостепенные действия на стадии публикации

Действие	Сотрудник (-и)
Создание и публикация документации	Технический писатель
Разработка архитектуры развертывания и развертывание экземпляров	DevOps
Публикация API (то есть официальное открытие доступа)	Менеджер продукта

Наконец, менеджер продукта должен запустить процесс публикации. В зависимости от вашего API и общей обстановки началом запуска могут быть разные события, например регистрация API во внутреннем каталоге сервисов, отправка e-mail потенциальным пользователям или что-то другое. В любом случае именно менеджер продукта отвечает за то, чтобы это произошло.

Кроме этих первостепенных действий, существуют дополнительные (табл. 7.4), направленные на улучшение качества продукта с API. Документация и другие ресурсы поддержки должны где-то размещаться, поэтому многие организации на этой стадии создают портал для разработчиков. Когда API уже будет активно использоваться, вы сможете улучшить дизайн и реализацию, основываясь на данных об их применении (столп «Мониторинг API»). Полезно также продолжать поощрять использование с помощью работы по обнаружению API и маркетинга.

Таблица 7.4. Дополнительные действия на стадии публикации

Действие	**Сотрудник (-и)**
Разработка и реализация портала	Разработчик клиентской части
Продвижение API на рынке	Евангелист API, специалист по OP
Сбор отзывов пользователей о дизайне	Евангелист API, специалист по OP
Улучшение дизайна интерфейса	Дизайнер
Сбор информации об использовании развернутых экземпляров	Ведущий инженер, DevOps
Улучшение и оптимизация реализации	Ведущий инженер, бэкенд-разработчик
Тестирование безопасности реализации и развертывания	Архитектор API, специалист по тестированию/OK

Стадия 3: окупаемость

- *Первостепенные обязанности*: DevOps, менеджер продукта.
- *Второстепенные обязанности*: дизайнер, специалист по тестированию/QA, архитектор API, ведущий инженер, бэкенд-разработчик, разработчик клиентской части, технический писатель, специалист по OP, евангелист API.

Когда API начинает приносить доход, ставки вырастают. Теперь нужно, чтобы сотрудники следили за постоянной доступностью API для важных пользователей. Поэтому первостепенными действиями становятся управление изменениями и улучшение архитектуры развертывания.

Несмотря на то что API уже окупается, вам предстоит внести еще много изменений в интерфейс, реализацию и готовые экземпляры. Хороший менеджер продукта должен уметь управлять всеми этими изменениями без потери дохода и отрицательного влияния на пользователей. Как конкретно это нужно делать, во многом зависит от сотрудников, стратегических приоритетов и культуры организации.

В то время как менеджер продукта управляет изменениями, специалист по DevOps сосредоточен на улучшении устойчивости к внешним воздействиям, наблюдаемости, масштабируемости и работе архитектуры развертывания (табл. 7.5). Хороший специалист по DevOps сможет применить правильные инструментальные средства и методы, основываясь на ситуационных аспектах API. Целью является предотвращение ухудшения качества для постоянных, приносящих доход пользователей.

Таблица 7.5. Первостепенные действия на стадии окупаемости

Действие	**Сотрудник (-и)**
Улучшение и оптимизация архитектуры развертывания	DevOps
Управление изменениями и распределение приоритетов в этой сфере	Менеджер продукта

Чтобы продолжить увеличивать доход, имеет смысл продолжить улучшать и продвигать свое предложение на рынке. Поэтому на данной стадии нужны такие же дополнительные действия по анализу, реализации и обнаружению, как и на предыдущей (табл. 7.6). Это не обязательно, но без постоянного улучшения ваш API может быстро перейти в стадию поддержки, не успев окупить вложения в него.

Таблица 7.6. Дополнительные действия на стадии окупаемости

Действие	**Сотрудник (-и)**
Улучшение дизайна интерфейса	Дизайнер
Улучшение и оптимизация тестов	Специалист по тестированию/ОК
Улучшение и оптимизация реализации	Архитектор API, ведущий инженер, бэкенд-разработчик
Тестирование безопасности реализации и развертывания	Архитектор API, специалист по тестированию/ОК
Улучшение и оптимизация начала работы с программой и обучения пользователей	Разработчик клиентской части, технический писатель, DevOps
Продвижение API на рынке	Евангелист API, специалист по ОР

Стадия 4: поддержка

- *Первостепенные обязанности:* специалист по ОР, DevOps, архитектор API.
- *Второстепенные обязанности:* менеджер продукта, ведущий инженер API, бэкенд-разработчик.

На стадии поддержки ваша цель — сохранять API в рабочем состоянии. Ключевая обязанность здесь — специалист по DevOps, который должен контролировать и поддерживать развернутые экземпляры. Кроме выполнения основной работы, важно следить за возможными изменениями системы, которые могут создать новые задачи для команды по API. Хороший архитектор API будет настроен на

изменения, которые могут потенциально повлиять на работу API, и сможет определить, какие изменения нужны API, чтобы внести их, не мешая работе продукта.

Вам также понадобится определенный уровень отношений с пользователями и их поддержки, даже если API уже активно не продается. Лучше всего эту поддержку сможет предоставить специалист по OP. Он проследит, чтобы продукт был полезен новым и уже существующим пользователям, даже если уровень дохода от него находится в стагнации (табл. 7.7).

Таблица 7.7. Первостепенные действия на стадии поддержки

Действие	Сотрудник (-и)
Улучшение и оптимизация системы мониторинга	DevOps
Поддержка существующих пользователей	Специалист по OP
Определение системных изменений, которые ухудшат качество API	Архитектор API

Наконец, чтобы укрепить эту поддержку, менеджер продукта и технические специалисты должны быть готовы внести любые нужные изменения (табл. 7.8). Хотя количество улучшений к тому моменту значительно уменьшается, их все равно придется вносить, чтобы поддержать то, над чем работают архитектор API и специалист по OP.

Таблица 7.8. Дополнительные действия на стадии поддержки

Действие	Сотрудник (-и)
Планирование изменений в реализации	Менеджер продукта
Внесение нужных изменений в реализацию	Ведущий инженер, бэкенд-разработчик
Внесение нужных изменений в развертывание	DevOps, бэкенд-разработчик

Стадия 5: удаление

- *Первостепенная обязанность*: менеджер продукта.
- *Второстепенные обязанности*: специалист по OP, евангелист API, архитектор API, DevOps, ведущий инженер.

Основная работа на стадии удаления — работа со стратегией, поэтому ключевую роль играет менеджер продукта (табл. 7.9). Хороший менеджер продукта сможет определить оптимальную стратегию деактивации для данных обстоятельств. У него должно быть достаточно опыта для разработки тактического плана не только для создания нового API, но и для удаления старого.

Таблица 7.9. Первостепенные действия на стадии удаления

Действие	Сотрудник (-и)
Разработка стратегии удаления	Менеджер продукта

Следование этой стратегии требует удаления развернутого экземпляра из архитектуры развертывания и поддержки пользователей во время перехода на другую программу. Специалист по DevOps отвечает за деактивацию API в сфере развертывания, а специалист по OP — за его деактивацию для пользователей.

Вам также может понадобиться технический план для выполнения стратегического плана, составленного менеджером продукта. Например, может быть логичным отправлять ответные сообщения с информацией о приближающейся деактивации или выбирать специальные заголовки для ответов, показывающие, что API находится на стадии удаления. Этот план должен разработать сотрудник, обладающий техническими знаниями, поэтому обычно это делает архитектор API или ведущий инженер (табл. 7.10).

Таблица 7.10. Дополнительные действия на стадии удаления

Действие	**Сотрудник (-и)**
Оповещение пользователей о плане удаления и помощь с переходом на новый API	Специалист по OP, евангелист API, технический писатель
Разработка технической стратегии удаления	Архитектор API, ведущий инженер
Обновление архитектуры развертывания и осторожное удаление экземпляров	DevOps, ведущий инженер

В этом разделе мы обсудили, как стадия жизненного цикла отдельного продукта с API влияет на состав команды по его разработке и на первостепенные и второстепенные обязанности участников этой команды. И узнали, что с изменением API меняется комплектация его команды.

Другой важный аспект деятельности команд по API — масштабирование нескольких из них. В большинстве компаний с жизнеспособными программами с API работает больше одной команды по API. Как они трудятся вместе? Какую тактику нужно использовать, чтобы их цели не пересекались и не противоречили друг другу? И как убедиться в единообразии действий множества команд? Эти вопросы — последнее, что мы рассмотрим в этом разделе.

Масштабирование команд

Понимание того, что обязанности — это необходимые структурные единицы команд, а комплектование команд зависит от стадии развития, на которой находится продукт, — это только начало сложностей с руководством командами по API. Другой важный элемент — это работа с многими командами. Зачастую команда есть у каждого API, а API больше одного. Работа с множеством команд *(командой команд)* переносит вас на новый уровень сложности.

Стоит относиться к каждой команде по API как к независимой группе. Таким образом они смогут решать свои проблемы, почти не завися от других команд.

Но реальность не совпадает с теорией. Теоретически команды не нужны друг другу. Но на самом деле они не могут друг без друга работать! Как же с этим разобраться? Существует постоянный баланс между поддержанием независимости и плодотворным сотрудничеством. Важно разработать больше одной стратегии для команд, а также хорошо разбираться в том, как разные части (команды) соединяются в одно целое.

В своей книге «Команда команд: правила вовлеченности в сложном мире» (*Team of Teams: New Rules of Engagement for a Complex World*, издательство Portfolio) генерал Стэнли Маккристал говорит о новом способе мышления, помогающем большим организациям добиться успеха: «Сейчас, когда мир развивается быстрее и все становится более взаимозависимым, нам нужно найти способы масштабировать текучесть команд внутри целых организаций». Это означает — понять, как поощрить совместную работу команд, не заставляя их быть зависимыми друг от друга.

Одна из организаций, известных тем, что умеет масштабировать систему своих команд, — компания потокового аудио Spotify. На опубликованную ее специалистами работу, посвященную этой теме, часто ссылаются, говоря о способах улучшения эффективности и отдельных команд, и коммуникации между ними. Несмотря на то что она несколько устарела (шесть лет — очень долгий срок для Интернета!), мы обнаружили, что многие организации используют подходы, похожие на описанные в ней. Это происходит часто, поэтому мы считаем, что стоит понять основные из изложенных там мыслей и исследовать, как вы можете применить их в своей компании.

Команды и обязанности в Spotify

В 2012 году преподаватели гибкой методологии разработки Хенрик Книберг и Андерс Иварссон опубликовали работу «Гибкое масштабирование в Spotify» (http://bit.ly/2Tb7twZ). Ее первое предложение таково: «Работа с несколькими командами в организации, разрабатывающей продукцию, — это всегда сложная задача!» Затем Книберг и Иварссон объясняют, как в Spotify разработали модель управления командами, помогающую делиться информацией по максимуму, не ставя под удар независимость команд. Такую модель или ее вариации мы сегодня наблюдаем во многих компаниях.

В командной модели Spotify есть четыре ключевых элемента, или способа группирования:

- отряд;
- племя;
- отдел;
- гильдия.

Отряды — это небольшие самодостаточные команды из 5–7 сотрудников, похожие на Scrum-команды. Это базовая рабочая единица Spotify. У членов отряда есть все необходимые навыки для выполнения порученной им работы, от дизайна до развертывания, как и у участников команд, о которых мы здесь говорили. В Spotify у каждого отряда есть миссия или задача *внутри* большой группы одного продукта. Например, для разработки проигрывателя на Android один отряд может заниматься воспроизведением, другой — поиском и т. д. Отряды выполняют отдельные задачи.

Племя появляется в Spotify с увеличением масштаба деятельности — например, уже упомянутый проигрыватель на Android, или сайт, или серверный сервис по хранению информации, используемый всеми остальными клиентскими программами. В данном случае племя — это объединение отрядов. В Spotify стараются удержать общее количество сотрудников в племени на уровне 100 человек. Оно считается довольно большим, чтобы в группе было нужное разнообразие специалистов для выполнения задачи, но недостаточно большим, чтобы стало сложно поддерживать нормальные отношения.

Числа Данбара

Максимальный размер отряда (семь человек) и племени (100) основаны на работах британского социоантрополога Робина Данбара. Мы поговорим о нем подробнее в подразделе «Разумное использование чисел Данбара» раздела «Культура и команды» далее.

Применяя отряды и племена, Spotify может разработать эффективную стратегию создания и поддержки своих продуктов и сервисов. Однако это только половина задачи. Важно также достичь определенного уровня производительности внутри сообщества. Для этого нужно взаимодействие между отрядами и командами, чтобы делиться знаниями и обеспечивать единообразие в командах и продуктах, — и здесь в игру вступают отделы и гильдии Spotify.

Поскольку все команды самодостаточны, скорее всего, в каждой есть дизайнер, бэкенд-разработчик приложения, менеджер продукта и т. д. Любой сотрудник, исполняющий эти обязанности, решает стоящие перед ним задачи и обладает собственным опытом обучения. Часто его опыт близок к опыту других сотрудников, исполняющих те же обязанности в других командах. Например, чтобы быть хорошим менеджером продукта в племени инфраструктуры, нужен тот же набор навыков, который есть у всех менеджеров продукта, даже если их конкретные подходы не совпадают. Поэтому логично, что все менеджеры продукта внутри одного племени периодически собираются и делятся опытом и знаниями друг с другом. В Spotify это называется *отделом* — когда сотрудники с одинаковыми обязанностями в одном племени (то есть в группе одного и того же продукта) собираются вместе и обмениваются опытом.

В свою очередь, *гильдии* в Spotify — это способ поделиться знаниями с группами других продуктов. Например, если несколько менеджеров продуктов из всех сфер деятельности компании (от продуктов, предназначенных для клиентов, до внутренних систем) собираются вместе, появляется дополнительный уровень обмена знаниями. В вашей компании гильдия может быть собранием руководителей команд со всего мира, которые встречаются раз в год, чтобы обсудить, над чем они работают в своих отделениях.

В Spotify модель, состоящая из отрядов, племен, отделов и гильдий, обеспечивает необходимое сочетание самодостаточных команд и отсутствия изолированных групп сотрудников, не общающихся между собой. Так поддерживается баланс между независимостью и сотрудничеством.

Масштабирование команд на бумаге

Подход Spotify к масштабированию команд представляет децентрализованную точку зрения. Масштабирование встроено в саму рабочую модель. Другой способ, который мы видели в некоторых компаниях, — создание центральной команды, которая собирает информацию у других команд и делится ею через брошюры, документы, относящиеся к стандартам, и семинары, посвященные лучшим методикам[1].

Таким образом, можно расширить обмен знаниями, не устраивая личных встреч или внутренних конференций. Этот метод особенно эффективен для очень больших организаций, где команды разбросаны по всему миру и находятся в разных часовых поясах, что затрудняет проведение даже виртуальных конференций.

Однако полагаться *только* на общую документацию и записи презентаций для обмена опытом и знаниями внутри компании — неудачная мысль. Обучение, обсуждение и сотрудничество в реальной жизни всегда эффективнее и часто продуктивнее, чем чтение статей или просмотр видео.

И это подводит нас к последнему разделу главы, посвященному корпоративной культуре. То, как участники команды работают вместе и как команды сотрудничают друг с другом, сильно зависит от культуры и ценностей, существующих внутри организации. Поэтому важно тратить время на изучение и создание собственной корпоративной культуры.

Культура и команды

Корпоративная культура может служить скрытой формой руководства в организации. В главе 2 мы рассказали о централизованных и децентрализованных решениях. Вкратце повторим: когда решения централизованы, вы должны использовать власть, чтобы люди правильно с ними работали. Но когда они децентрализованы,

[1] Мы подробнее поговорим об этой команде в разделе «Центр подготовки» главы 9.

применение власти как инструмента контроля и одобрения не работает. Здесь и нужна культура. Она как невидимая рука, формирующая принятие решений внутри команд по всей компании, которое не требует избытка властных механизмов, таких как процессы, стандарты и общие инструменты. С правильными культурой и сотрудниками вы можете спокойно децентрализовать бóльшую часть решений, не утрачивая единообразия результатов. Поэтому вложения в создание корпоративной культуры могут окупиться стократ. Неизменная культура гарантирует единообразные результаты, даже если вы передаете командам право на принятие решений и увеличиваете их ответственность.

Для принятия правильных решений нужно не только знать, что следует изменить и как распределить ответственность за эти изменения. Корпоративная культура тоже является важным его элементом. Некоторые специалисты в сфере IT не готовы обсуждать это, но культура организации заслуживает внимания.

Понятие «корпоративная культура» впервые появилось на страницах книги «Меняющаяся культура предприятия» (*The Changing Culture of a Factory*, издательство Psychology Press). Доктор Эллиот Джекс определил ее следующим образом: «Культура предприятия — это общепринятый и традиционный образ мышления и действий, который разделяют в той или иной степени все его работники и которому должны научиться новоприбывшие».

Признание того, что у организации в принципе есть культура, приводит к возможности повлиять на существующую культуру, направить ее в определенную сторону. Так формируется представление о том, как понять, какая именно культура в вашей организации и как ее можно изменить.

В 1970–1980-е годы появляется много книг и теорий по выявлению и классификации корпоративной культуры, а также управлению ею. Одно из важнейших изданий того периода — «Образы организаций» Гарета Моргана (*Images of Organization*, издательство Sage). Морган выступил с идеей о том, что корпоративную культуру можно охарактеризовать, используя простые метафоры — машину, организм, мозг и т. д. Эти метафоры помогают понять, как работает культура компании и как определить способы, которыми ее можно изменить.

Здесь мы не будем пытаться пересказать произошедшее за 70 лет изучения корпоративной культуры, но отметим, что многие компании, с которыми мы сотрудничаем, активно работают над тем, чтобы разобраться в культуре своей организации и в том, как ее можно значительно улучшить и перенаправить. Исходя из этого, назовем три темы, которые часто затрагиваются, когда мы посещаем компании, разрабатывающие API и управляющие ими, а также оказывающие услуги в сфере IT.

- Наблюдения Мэла Конвея за тем, как *взаимодействие* групп влияет на результат.
- Теории Робина Данбара о том, как *размер* команды влияет на коммуникацию.
- Наблюдения Кристофера Александера за тем, как *разнообразие* влияет на продуктивность.

Мы также затронем роль экспериментирования в корпоративной культуре и то, как оно влияет на команды.

Как работает закон Конвея

В последние несколько лет работа Мэла Конвея «Как комитеты создают новое?» (http://bit.ly/2RSyMKS), опубликованная в 1967 году, стала почти *обязательной* темой в презентациях, посвященных микросервисам и API в целом. В этой работе представлено то, что Фред Брукс в 1975 году в книге *Mythical Man Month* (Addison-Wesley) назвал законом Конвея. Закон Конвея утверждает следующее: «Организация, которая разрабатывает системы... неизбежно создает модели, являющиеся копиями коммуникативной структуры этой организации».

Этот закон представляет собой наблюдение, свидетельствующее, что организация группы влияет на результат ее работы. Эрик С. Реймонд, автор эссе «Собор и базар» (*The Cathedral & the Bazaar*, издательство O'Reilly) (http://bit.ly/cathedral-and-bazaar), сделал дополнительное наблюдение, которое часто цитируют: «Если у вас над компилятором работают четыре группы, вы получите четырехпроходный компилятор». Если формулировать кратко, закон Конвея говорит, что в области производства программного обеспечения организационные границы определяют, какие приложения у вас получатся. Это и хорошо, и плохо одновременно.

Как сказано в начале главы, ПО, которое мы пишем, — тупое: оно делает только (и в точности) то, что ему приказывают люди. Конвей напоминает: то, как мы группируем сотрудников, где проводим границы между группами, определяет полученные результаты. Поэтому некоторые IT-консультанты говорят о применении обратного закона Конвея. Они советуют сначала сформировать команды и очертить границы, чтобы получить желаемые результаты. В какой-то степени это может сработать, но тут появляются свои проблемы. В той же самой работе 1967 года Конвей предупреждает, что не стоит работать организационным «скальпелем» слишком агрессивно: «[Закон Конвея] вызывает проблемы, потому что необходимость взаимодействия в любой момент зависит от системы, работающей в этот момент. Поскольку первый предложенный дизайн почти никогда не оказывается лучшим из возможных, вам может понадобиться изменить эту систему. Таким образом, для эффективного дизайна важна гибкость организации».

Очевидно, невозможно перехитрить закон Конвея! Здесь нужен компромисс. Важно обратить внимание на то, что знание структуры организации — это ключ к пониманию корпоративной культуры. Компании, которые явно хорошо управляют своей культурой, имеют как минимум две общие черты: они работают над тем, чтобы сделать границы одновременно четкими и гибкими.

Установить четкие границы между командами на ранней стадии проекта — нормально. Это поможет распределить ответственность внутри команд и разделить интерфейсы между командами. Важно помнить и о предупреждении Конвея: «Первый

предложенный дизайн почти никогда не оказывается лучшим из возможных». Поэтому, работая над API и составляющими его проектами, важно корректировать границы, основываясь на новых обстоятельствах, появившихся в реальной жизни. Это тоже нормальная часть работы.

Как и многие другие аспекты в сфере управления API, управление культурой непрерывно. Конвей подсказывает, как можно повлиять на изменения (например, сфокусироваться на границах между командами), и предупреждает, что первые попытки редко становятся лучшими вариантами из возможных («вам может понадобиться изменить эту систему»). Поэтому появляются вопросы: как команды и их размер могут повлиять на корпоративную культуру. Они приводят нас к работам Робина Данбара.

Разумное использование чисел Данбара

Конвей рассказывает о том, как команды и границы между ними влияют на результат любого дела. Поэтому появляется логический вопрос: как формируется команда и, более конкретно, каков ее оптимальный размер? Многие наши клиенты обращаются за ответом к исследованию доктора Робина Данбара, проведенному в 1990 году (http://bit.ly/2B6fXhL). В статьях по популярной социологии его теории о том, как мозг влияет на размер группы, известны благодаря так называемому числу Данбара, которое устанавливает, что мы можем успешно отслеживать и поддерживать полезные отношения не более чем со 150 людьми. Любая группа, превышающая это количество, перегружает мозг большинства людей, они оказываются не способны руководить и управлять ею. Поэтому, как только группа перерастает это количество, координация и сотрудничество между ее членами становятся гораздо сложнее.

Существует много подтверждений силы числа 150 при взаимодействиях с группами. Уильям «Билл» Гор, основатель и председатель компании W. L. Gore с 1970 по 1986 год, установил правило (http://bit.ly/2z9p5kk), что как только на отдельном предприятии появлялось более 150 сотрудников, эту группу разделяли и строили новое здание, иногда рядом со старым. Пэтти Маккорд из Netflix называет это количество числом «встать на стул» (http://bit.ly/2Pu35uW): как только подчиненных становится больше 150, руководитель уже не может просто встать на стул, чтобы обратиться ко всей группе.

Хотя 150, по Данбару, — важное число, но наш опыт подтверждает, что исследование, стоящее за ним, еще более значимо. Теория Данбара заключается в том, что мы должны тратить время и энергию для обеспечения успешного взаимодействия в группах, а размер группы влияет на количество ресурсов, необходимых для поддержания успешных связей. Его раннее исследование показало, что командам из 150 сотрудников «требуется 42 % рабочего времени на “социальные ухаживания”». Другими словами, при росте команд всем нужно все больше времени посвящать усилиям по их сплочению. В современных офисных условиях «социальные ухаживания» принимают форму собраний, переписки по e-mail и в мессенджерах, звонков, ежедневных пятиминуток, планерок и т. д. Координация больших команд дорого стоит.

Хорошая новость — у Данбара было больше одного числа. Он определил серию чисел, начиная с пяти — 15, 50, 150 и т. д. — до превышающих 1000. На нижнем конце шкалы (5 и 15) стоимость координации — «социальных ухаживаний» — невысока. В группе из пяти человек все хорошо знакомы друг с другом, каждый знает свою задачу и почти всегда все знают, кто не выполняет свою часть работы (если такой человек есть). Время на «социальные ухаживания» почти минимально. Даже при увеличении группы до 15 человек затраты на коммуникацию относительно невысоки. Вы могли заметить, что рекомендованные в этой книге размеры групп приближаются к отметке пять человек (± 2–3).

Такая маленькая (от пяти до семи человек) команда — это так называемая команда первого уровня по Данбару. Этот размер очень распространен в стартапах на ранней стадии. Команды второго уровня по Данбару, размером до 15 человек, часто можно встретить в новых компаниях, прошедших первый этап, когда их спонсировали меценаты, и активно строящих свой бизнес. Многие IT-компании, с которыми мы общались, стараются сохранять размеры своих команд на первом и втором уровнях по Данбару, чтобы снизить затраты на коммуникацию и максимально увеличить их общую эффективность. Например, Spotify стремится к созданию отрядов в 5–7 человек (см. подраздел «Команды и обязанности в Spotify» в предыдущем разделе).

Данбар показывает, что размер команды имеет значение, а взаимодействие в небольших командах может быть более продуктивным, чем в больших. Конвей напоминает, что взаимодействие между командами определяет итоговый результат. Таким образом, сложность формирования успешной культуры управления API состоит в руководстве множеством команд внутри большой организации. Как в работе с системой API вы сталкиваетесь с задачами, отличающимися от задач, которые решаются в ходе работы с одним или несколькими API, так и, работая с системой команд, вы встречаетесь с уникальными аспектами. С этой проблемой команды команд могут помочь работы архитектора Кристофера Александера.

Работа с культурной мозаикой по Александеру

Руководство одной командой и/или ее поддержка — уже непростая задача. Мотивировать группу работать, помогать им найти опору и свой стиль и научиться вносить вклад в миссию компании — это тяжелый, но благодарный труд. Люди, которые часто этим занимаются, знают, что любая команда уникальна. Каждой команде приходится проходить один и тот же путь по-своему. Различия между командами и станут ключом к появлению разнообразия и силы в вашей компании. Хотя может показаться, что проще требовать от всех команд вести себя и выглядеть одинаково, это не признак здоровой экосистемы.

Такая система команд обладает теми же сложностями и возможностями, что и система API (см. главу 8), и многие аспекты системы, которые мы очерчиваем в этой главе, применимы и к системе команд. Во время роста предприятия вы столкнетесь с появлением разнообразия, объема, непредсказуемости и других элементов здоро-

вой экосистемы. На самом деле системы людей (то есть команды) обычно становятся лучше с возникновением разнообразия. Многие из нас бывали в ситуации, когда появление в команде человека со стороны привело к ее усилению. По этому поводу существует много афоризмов, включая «То, что нас не убивает, делает нас сильнее», — о том, что неожиданные сложности помогают нам расти над собой. Книга Нассима Талеба Antifragile[1] (Random House), выпущенная в 2012 году, основана именно на этой идее.

Другую точку зрения на силу команды команд можно найти в работах архитектора и мыслителя Кристофера Александера. Его книга «Язык шаблонов» (A Pattern Language, издательство Oxford University Press), изданная в 1977 году, считается толчком к появлению шаблонов в программном обеспечении. Она описывает понятие «мозаика субкультур» как способ организации здоровых и стабильных сообществ.

ВЛИЯНИЕ АЛЕКСАНДЕРА НА ПРОГРАММНОЕ ОБЕСПЕЧЕНИЕ

Хотя Кристофер Александер и строит здания, его работы и идеи сильно повлияли на архитектуру программного обеспечения. Его книга «Язык шаблонов» представила идею шаблонного мышления при создании больших систем, и ее часто цитируют и считают катализатором появления шаблонов в ПО. Читать книгу про шаблоны тяжело, и только один из нашей четверки заявляет, что прочел ее целиком. Но у Александера есть книга меньше и доступнее — «Вневременной способ строительства» (*The Timeless Way of Building*, издательство Oxford University Press). Мы часто рекомендуем работы Александера архитекторам ПО, работающим с очень большими системами.

Идея шаблона мозаики субкультур по Александеру (http://bit.ly/2Psdveh) описывает три основных способа работы с большими собраниями людей и меньшими группами, появляющимися внутри целого. Работы Александера посвящены городам, но наш опыт свидетельствует: у них есть важные параллели с руководством IT, работающим с организациями на глобальном уровне и на уровне предприятия.

Александер выделяет три подхода к появлению подгрупп внутри больших сообществ, каким, с его точки зрения, является город.

- *Гетерогенный*. Люди смешиваются между собой вне зависимости от образа жизни или культуры. Их образ жизни сводится к общему знаменателю, который оказывается однообразным и скучным.
- *Гетто*. Люди группируются по самым основным и банальным отличительным признакам — расе и/или экономическому положению. Создаются изолированные группы, также однообразные внутри каждого гетто.
- *Мозаика*. Несколько небольших областей с четкими разделяющими их границами, между которыми можно свободно перемещаться, чтобы познакомиться с интересующими и вдохновляющими видами образа жизни и культурами.

[1] *Талеб Н.* Антихрупкость. Как извлечь выгоду из хаоса. — М.: КоЛибри, 2016.

Может оказаться сложновато разобраться в примерах из области планирования города, которые приводит Александер, но мы наблюдали, что принципы гомогенности («Мы все используем одинаковые инструменты и процессы для всей компании»), гетто («Мы здесь все специалисты по базам данных, отдел обеспечения качества — в другом здании») и мозаики («Я присоединился к этой группе, потому что хотел поработать над нашим мобильным приложением») преобладают и в IT-компаниях. Внутри каждой компании появляются общие культурные элементы и субкультуры. Знание об этих подэлементах — первый шаг к работе с ними и в большинстве случаев — к их использованию с целью создания здоровой и стабильной культуры управления API.

Мы сообщили вам, что недостаточно просто понимать динамику взаимодействия внутри одной команды (как утверждает Данбар). А также подчеркнули важность связей между командами, описанных Мэлом Конвеем. Наконец, мы познакомили вас с понятием системы команд и с тем, как важно обращать внимание на способ формирования команд (по Александеру) и систему, в которой они работают. Но как это все окупится? Зачем тратить время на эти элементы культуры, особенно в сфере управления API?

Корпоративная культура определяет уровень инноваций, экспериментов и креативности ваших команд и сотрудников. Это ключ к успеху для компании.

Этот последний аспект культуры мы сейчас и обсудим.

Поддержка экспериментов

Важный повод тратить время на выстраивание корпоративной культуры — поддержка инноваций и преобразования повседневной работы организации. Важная причина этого выражена цитатой, приписываемой гуру управления бизнесом Питеру Друкеру: «Культура ест стратегию на завтрак».

Действительно, при любой стратегии компанию ведет вперед преобладающая в ней *культура*. Таким образом, если вы хотите сменить направление своей команды, группы по разработке продукта или даже целой организации, необходимо сосредоточиться на культуре. Этому нас учит и Конвей: именно организация и ее границы влияют на результат работы группы.

Ключ к поддержке инновации — создания новых продуктов, методов и идей — лежит в возможности безопасно проводить эксперименты. Эксперимент — это не запуск в производство непродуманной идеи. Как и многое, что мы обсудили в этой книге, эксперимент начинается с небольшой идеи, например возникшей внутри одной команды, и снова и снова проходит повторяющиеся стадии разработки, чтобы определить связанные с ним идеи, научиться на них, отбросить лишнее и найти что-то ценное, полезное и нужное, что будет стоить потраченных на его реализацию ценного времени и ресурсов.

В книге «Руководство от Делла» (*Direct from Dell*, издательство HarperCollins), выпущенной в 2006 году, бизнесмен и филантроп Майкл Делл сформулировал это так: «Чтобы сотрудники применяли больше инноваций, нужно обеспечить им безопасность в случае провала». Главная мысль состоит в том, что провалить задачу должно быть не просто легко, но и безопасно. Нужно обеспечить атмосферу, которая позволяет командам свободно экспериментировать, но не дает совершать ошибки, которые дорого вам обойдутся и могут нарушить важные операции компании. Один из способов создания подобной системы — использование элементов решения, выделенных ранее в этой книге. Когда команды знают границы своей деятельности, они четче видят, какие эксперименты могут ставить, чтобы понять, как улучшить свою работу.

Важно также понимать, что гораздо лучше, когда эксперименты проводят многие команды, чем когда немногие или всего одна. В разделе «Центр подготовки» главы 9 мы обсудим пользу наличия специализированной команды, которая может помочь определить указания и ограничения для всего предприятия. Однако это не то место, где происходят все эксперименты. Как и в других аспектах IT, чрезмерно сильная централизация и концентрация может привести к повышению уязвимости и непредсказуемости. В то же время распределение действий среди множества команд и групп по разработке продуктов повышает вероятность появления ценных идей и снижает опасность того, что эти эксперименты серьезно навредят компании.

Последний пункт может показаться нелогичным некоторым руководителям в сфере IT. Можно подумать, что большое количество экспериментов усиливает непредсказуемость, но так бывает, только если все они проводятся в одном месте, например в центре обучения или другом центре экспериментов. Такую концентрацию рисков Нассим Талеб обсуждает в книге *Skin in the Game*[1] (Random House). Автор «Черного лебедя», «Антихрупкости» и других бестселлеров, Талеб напоминает читателям: «Вероятность успеха для собрания людей неприменима к [одному человеку]». Иначе говоря, сообщество из 100 команд, проводящих эксперименты с новыми API, — это не то же самое, что одна команда, проводящая 100 экспериментов подряд. Вы можете использовать знание элементов решения, чтобы снизить риск при увеличении количества экспериментов.

А увеличение количества экспериментов увеличивает частоту применения инноваций, что ведет к модели непрерывного управления API (см. раздел «Жизненный цикл продукта с API» главы 6), которую проще поддерживать с течением времени.

Чтобы вывести всю эту работу на одновременно стабильный и экономичный (но не обязательно *производительный*) уровень, вам нужно многообразное сообщество команд, работающих над проектами, которыми они увлечены. Здесь-то в игру и вступает мозаика, описанная Александером.

---

[1] *Талеб Н.* Рискуя собственной шкурой. Скрытая асимметрия повседневной жизни. — М.: КоЛибри, 2018.

Выводы

В этой главе мы затронули много тем. Во-первых, определили набор обязанностей, охватывающий принятие решений и ответственность, необходимую для того, чтобы разработать, создать и поддерживать API. Обсудили также, как эти обязанности можно использовать, чтобы собрать команду людей, которые работают над самим API. А еще увидели, что один человек может выполнять несколько обязанностей в одной или нескольких командах.

Кроме того, мы коснулись того, как разные стадии жизненного цикла API могут повлиять на необходимость в определенных обязанностях в команде по API. Оказывается, команды динамичны, а обязанности отражают количество вовлеченных в работу людей и стадию развития API. К тому же мы исследовали, как компания Spotify разработала командную модель, учитывающую взаимодействие между командами на различных уровнях внутри компании. Отметили, что можно применить централизованный или децентрализованный подход к обмену опытом и сотрудничеству между командами внутри компании.

Наконец, мы исследовали значение корпоративной культуры для работы команд. Такие факторы, как размер команды, могут влиять на качество взаимодействия и точность результатов деятельности, а если вы не настроите коммуникацию между командами, внутри организации могут появиться «технические гетто», которые способны затормозить внедрение инноваций и ухудшить творческий подход к работе.

Последний раздел, посвященный значению культуры и улучшению коммуникации между командами, подводит нас к важному моменту. До сих пор мы говорили об управлении одним API и о том, как убедиться, что он соответствует требованиям пользователей: о типичных навыках, необходимых для создания и поддержки API, о правильном управлении изменениями на протяжении его жизненного цикла, об обязанностях и командах, которые нужны, чтобы все это заработало.

Однако, как уже говорилось, существует и другой аспект этой темы — сотрудничество между командами и продуктами. Во всех компаниях, которые мы посетили, существует больше одного API, больше одной команды и больше одного способа совместной работы. Мы называем этот мир множества API и множества команд системой API вашей компании. А управление системой сильно отличается от управления одним объектом или одним API.

Когда объем вашей ответственности перерастает один API или продукт, необходимо сменить свой взгляд на задачи и поиск решений этих задач. Процитируем (снова) Стэнли Маккристала: «Порыв начать, подобно гроссмейстеру, контролировать каждый ход организации должен уступить место подходу садовника — поощрению вместо указаний. Подход садовника к руководству ни в коем случае не пассивен. Руководитель действует как посредник с открытыми глазами, но связанными руками, который создает и поддерживает систему, в которой работает организация».

О том, что нужно, чтобы вырастить систему API в вашей компании, мы и расскажем в следующих главах.

8 Системы API

> Теория эволюции путем нарастающего естественного отбора — единственная известная нам теория, которая в принципе способна объяснить существование организованной сложности.
>
> *Ричард Докинз*

С ростом функциональности API становится все важнее управлять развивающимися программами так, чтобы польза и ценность *всей совокупности API организации* максимально выросли. Необходимо помнить об этом балансе, потому что лучший (или относительно хороший) способ предоставления индивидуального сервиса с помощью API может быть совсем не таким полезным, если смотреть на него с точки зрения легкости использования этого сервиса как части общей системы.

ОПРЕДЕЛЕНИЕ СИСТЕМЫ API

Система API — это совокупность всех API, опубликованных организацией (рис. 8.1). API в системе могут находиться на разных стадиях развития (создание, публикация, окупаемость, поддержка, удаление) и быть нацеленными на разные аудитории (внутренние, партнерские, внешние). Они могут иметь и другие различия, например стиль или способ реализации.

Цель системы API — создание среды, которая поможет улучшить эффективность разработки дизайна, реализации и работы с API. В итоге система API может помочь организации достичь бизнес-целей, таких как ускорение циклов изготовления продукта, упрощение тестирования и изменения продукта и создание среды, в которой идеи и инициативы для бизнеса максимально быстро отражаются в IT.

Современные системы API непрерывно растут в смысле общего количества API, а также количества API, используемых новыми сервисами. Учитывая это увеличение взаимозависимостей, мы понимаем, что полезно было бы, если бы разработчикам

новых сервисов не пришлось разбираться в совершенно разных дизайнах API и применять их. Различия могут быть кардинальными — например, задействуют ли API стиль REST или событийно-ориентированный стиль (см. подраздел «Дизайн» раздела «Знакомство со столпами» главы 4), — но даже если стили совпадают, могут обнаружиться такие различия, как использование формата JSON или XML.

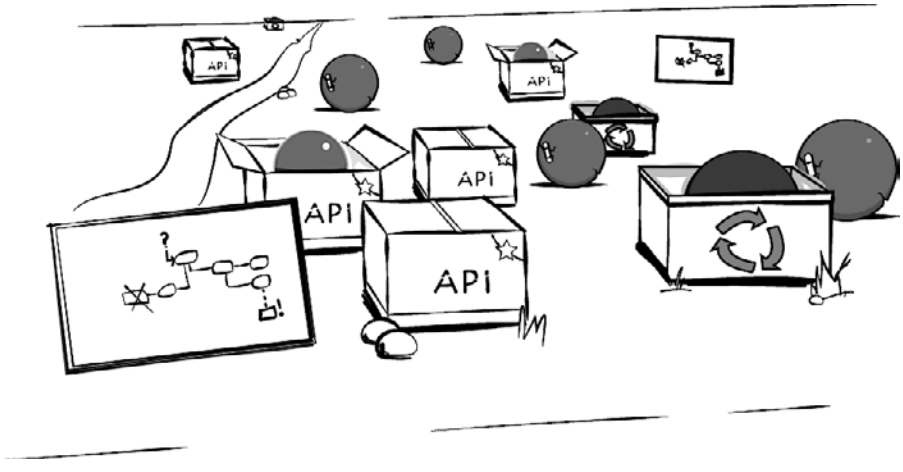


Рис. 8.1. Система продуктов с API

С точки зрения пользователя API, полезно было бы также иметь общую терминологию. Например, при задействовании нескольких API, предоставляющих пользовательские данные в какой-либо форме, будет проще, если у них у всех будет одна основная пользовательская модель (даже если она представлена слегка по-разному, тем не менее полезно иметь общую модель для разных сервисов).

Мысль о пользе стандартизации со всей очевидностью свидетельствует о том, что чем ее больше, тем лучше. В какой-то мере так и есть, но в то же время известно, что на стандартизацию нужны время и ресурсы, она обычно не сводится к «единственно верной и лучшей модели», а просто создает модель, с которой все могут как-то смириться, и поэтому в целом эта инвестиция имеет как риски, так и преимущества.

Возможно, использование уникального формата для каждого API будет не лучшим решением, разумнее выбрать из существующих, например, JSON или XML. В этом случае плюсы применения существующих стандартов перевешивают вероятные выгоды от уникальных форматов. Однако может быть очень дорого стандартизировать какие-то элементы разных сервисов, например уже упомянутую пользовательскую модель. В этом случае логично не идти на такие неоправданные расходы ради поиска единственно верной пользовательской модели и просто остановиться на модели предметной области.

Говоря в общем, в идеале для каждого сервиса мы хотим не изобретать велосипед заново, когда это не нужно, а использовать повторно те элементы дизайна, которые сокращают затраты ресурсов на создание дизайна, его понимание и реализацию. Если мы сможем достичь такого идеального уровня повторного применения или хотя бы приблизиться к нему, это позволит создателям сервисов сосредоточиться на тех аспектах дизайна, на которых нужно сосредоточиться, не отвлекаясь на уже решенные проблемы.

Мы видим, что все больше организаций поступают именно так. Но самое важное в этом — понимать и удостовериться, что руководства для дизайнеров постоянно обновляются: оцениваются и утверждаются новые методики, старые методики выходят из обращения, а главной движущей силой этих изменений является постоянно эволюционирующий набор методик API в организации.

Важно понимать, что система API — это подвижная и постоянно меняющаяся среда, и чтобы она продолжала работать, архитектура должна следовать по такому же пути непрерывного развития. Тогда она становится похожа на очень масштабную систему, например Интернет, который, с одной стороны, работает все время, а с другой — непрерывно меняется и в нем постоянно возникают новые стандарты и технологии.

Археология API

Хотя сейчас мы наблюдаем большое количество организаций, которые только начинают создавать программы с API, важно помнить, что в любой организации, в которой использовались информационные технологии, практически наверняка уже существуют какие-то API, применяющиеся уже долгое время.

АРХЕОЛОГИЯ API

Археологи занимаются тем, что выкапывают артефакты и пытаются определить время и место их происхождения. Эта концепция может быть применена и к API. *Археология API* — это поиск средств интеграции, исследование того, зачем и как они были созданы, и их документации, чтобы лучше понимать историю и структуру сложных IT-систем. Заниматься археологией API в организациях может быть очень полезно с точки зрения информации о существующих методах взаимодействия IT-компонентов (в любой форме).

Если исходить из определения, то API — это любой интерфейс, позволяющий двум программным компонентам взаимодействовать. Если сузить определение до современного понимания сетевого API, то это *любой* интерфейс, позволяющий двум программным компонентам взаимодействовать через сеть.

Во многих организациях такие интерфейсы не называются API, и они не созданы для повторного применения (вспомните историю о знаменитом уставе API

Джеффа Безоса, которую мы рассказывали в подразделе «Устав Безоса» раздела «Дизайн-мышление» главы 3). Но чаще всего эти интерфейсы существуют, даже если они были созданы и использовались для интеграции только для разового применения (подрывая этим одно из главных полезных свойств API — возможность повторного использования).

Поиск и применение таких прото-API может быть полезен, потому что он показывает, где появлялась необходимость в интеграции (даже если она создавалась с помощью методов, не соответствующих целям API). Не все эти прото-API стоит заменять обычными API, но просто разобравшись в истории, можно получить идеи по поводу того, как наблюдалась необходимость в интеграции, что предпринималось в этой области и где может появиться необходимость в дополнительной интеграции.

ПРОТО-API

Необходимость во взаимодействии компонентов существует во всех сложных системах, состоящих из отдельных частей. API — один из способов осуществления этого, но есть и множество других. С точки зрения API любой механизм, применяемый для взаимодействия компонентов и не являющийся API, может считаться прото-API. В идеальной системе с помощью API выполняются все взаимодействия компонентов без исключения. Если помнить об этом идеальном образе, любое взаимодействие без помощи API становится кандидатом на модернизацию — на то, чтобы его заменили на API. Поэтому любой механизм взаимодействия компонентов без помощи API можно считать прото-API.

В целом археология API может помочь вам лучше понять систему API, даже если сейчас она состоит в основном из прото-API. Она обеспечивает стартовую точку для понимания необходимости интеграции, возникавшей в прошлом, и того, с помощью каких инвестиций в API можно лучше всего распутать грозящую возникновением проблем сеть из множества пользовательских интеграций. С опытом и со временем заменять интеграции, созданные до появления API, современными моделями становится все проще.

Управление API в больших масштабах

Управление API в больших масштабах — это балансирование между введением общего дизайна на уровне системы и максимальным увеличением свободы выбора дизайна на уровне отдельных API. Выбор между централизованной интеграцией ради согласованности и оптимизации и децентрализацией ради гибкости и возможности развития — это обычная проблема сложной системы.

- *Централизованная* интеграция принесла нам типичную IT-архитектуру предприятия прошлого. Главной движущей силой была стандартизация *исполнения* функций, чтобы предоставлять их оптимизированно и с минимальными затратами. Высокий уровень интеграции действительно упрощает оптимизацию, но

в то же время влияет на возможность внесения изменений и развития итоговой системы.

- *Децентрализация* — это противоположный подход. Самым доступным и масштабным его примером является Интернет. Главная движущая сила здесь — стандартизация *доступа* к функциям, чтобы можно было предоставлять их множеством разных способов, постоянно развивающимся. При этом функции остаются доступными, так как доступ основан на общих соглашениях о взаимодействиях функций. Главная цель децентрализации — *ослабление связанности* (http://bit.ly/2FpcUpf), то есть облегчение изменений в отдельных частях общей системы без необходимости менять другие части.

ДЕЦЕНТРАЛИЗАЦИЯ И ИСПОЛНЕНИЕ

Если мы что-то и узнали из не до конца исполненных обещаний времен сервис-ориентированной архитектуры, основанной на SOAP, так это то, что *тщательно управляемое исполнение* — ключевой аспект реализации перспектив ориентированности на сервисы. SOAP исполнял обещание предоставить доступ к функциям, но не справлялся с такой же важной задачей *правильного управления исполнением функций*. Таким образом, хотя SOAP и был полезен (в качестве сервисов появились функции, прежде недоступные), он не соответствовал потребностям в более гибкой и развивающейся среде.

Сложность управления системой API заключается в том, чтобы запомнить эту проблему и избежать ловушки, в которую попал SOAP. В SOAP говорилось, что важна только доступность сервисов. Это был важный первый шаг, но он не позволил справиться со *слабой связанностью функций*. API и, если особо фокусироваться на техниках реализации и развертывания, микросервисы[1] позволяют нам еще раз обдумать, что важно для больших систем сервисов и как создавать системы, не попадающие в ловушку SOAP.

Принцип платформы

Многие говорят о платформах, обсуждая API и основные цели бизнеса. Однако при этом могут иметь в виду совершенно разные вещи. Важно помнить: если на уровне бизнеса кажется удачной идеей создавать что-то как платформу, это не значит, что именно так это надо разрабатывать на техническом уровне.

---

[1] Для более глубокого исследования стиля микросервисов советуем прочитать книгу Иракли Надарейшвили, Ронни Митры, Мэтта Макларти и Майка Амундсена «Архитектура микросервисов: Соединение принципов, методик и культуры» (Microservice Architecture, издательство O'Reilly, https://oreil.ly/2DDqCDi) и книгу Криса Ричардсона «Микросервисы. Паттерны разработки и рефакторинга» (Питер, 2019).

На уровне бизнеса платформа предоставляет то, на чем можно что-то построить, и глубже этой довольно размытой формулировки мы зайти не сможем. Часто на привлекательность понятия «платформа» влияют два основных фактора.

- *Какой у платформы охват?* То есть сколько пользователей можно охватить, создавая что-то на этой платформе? Это обычно определяется по числу ее пользователей или подписчиков. Зачастую это самый важный параметр, вычисляемый по общему количеству или с помощью качественных факторов, определяющих нужных пользователей, которых можно охватить с помощью этой платформы.
- *Какие функции у этой платформы?* Если что-то создавать на ней, как она помогает вам или ограничивает вас с точки зрения дохода? А также легко ли можно менять платформу, чтобы добавлять новые функции, в идеале не вредя ее пользователям?

Эти параметры очень важны для бизнеса, однако существует фактор, о котором часто забывают: платформы всегда заставляют пользователей придерживаться определенных ограничений, но делают это по-разному. Вот примеры.

- *Веб-приложения* могут использоваться кем угодно и чем угодно, что соответствует базовым стандартам сети. В самом простом случае это современный браузер с поддержкой скриптов. Кто угодно может создавать их и открывать к ним доступ, и любой человек может их задействовать. Нет центрального элемента, контролирующего работу *веб-платформы*[1].
- *Приложения в App Store* компании Apple внешне похожи на веб-приложения, но предоставляются и используются совершенно иначе. Их можно загрузить только из App Store, таким образом, у Apple есть эксклюзивное право решать, что могут устанавливать пользователи. К тому же их можно запускать только на устройствах с iOS, то есть Apple обладает монополией на продажу устройств, которые способны применять данные приложения. Приложения в App Store созданы специально для среды iOS, то есть вложения в их создание ограничены только этой платформой. Чтобы использовать приложение на любой другой платформе, включая Интернет, его нужно воспроизвести в другой среде разработки и даже на другом языке программирования, что означает: клиентскую сторону приложения необходимо воссоздать практически с нуля.

Можно применить этот шаблон и к системам API и идее создания платформы API для приложений.

---

[1] На самом деле это не совсем так. Существуют некоторые явно централизованные аспекты, например управление названиями, то есть тем, у кого есть возможность или право владеть ценной «недвижимостью» в форме названий DNS.

Иногда под платформой API подразумевается конкретная среда, предоставляющая доступ к API. Вскоре это начинает слегка напоминать традиционную шину служб предприятия, где подобная платформа должна предоставлять инфраструктуру, а API становятся доступными благодаря тому, что могут ее задействовать.

В других случаях, говоря о платформе API, имеют в виду общий набор принципов, используемый и предоставляемый сервисами. В таком случае, если сервис становится частью платформы, это никак не связано с тем, где и как открывается к нему доступ. Если сервисы придерживаются одних и тех же принципов, протоколов и шаблонов, они предоставляют API на этой платформе и, таким образом, становятся частью системы API.

Второй тип платформы более абстрактный, но в то же время у него больше возможностей. Разделяя то, что выполняют функции, и то, как они это делают, он облегчает вклад пользователей в платформу. А также открывает много путей для инноваций, позволяя приложениям экспериментировать с методами реализации, не ставя под удар их возможности вносить вклад в систему API.

И снова возьмем в качестве примера Интернет. Если смотреть только с точки зрения API, он позволяет многому меняться со временем. Например, сеть передачи данных (Content Delivery Network, CDN) не встроена в сам Интернет. Ее существование стало возможным из-за сложности содержимого Интернета и гибкости браузера, позволяющего генерировать веб-страницы, основываясь на нескольких источниках, взятых, возможно, из разных мест. На это можно возразить, что потенциал для создания CDN уже существовал в принципах и протоколах самых первых веб-страниц, но шаблон CDN появился только тогда, когда в нем возникла необходимость.

Именно эта способность подстраиваться под новые задачи требуется и в системах API. Мы проектируем систему, основанную на открытых и расширяемых принципах и протоколах, но можем и хотим при необходимости менять ее. Мы также создаем шаблоны поддержки, помогающие приложениям более эффективно решать проблемы, и хотим развивать со временем и эти шаблоны.

Принципы, протоколы и шаблоны

Главный вывод из предыдущего раздела: платформа не должна требовать одного конкретного способа что-то делать (ответа на вопрос «Как?») или одного конкретного места (ответа на вопрос «Где?»). Хорошая платформа создана для *непрерывной разработки архитектуры*, то есть архитектура платформы (а не только архитектура продукта) постоянно развивается, а не разрабатывается сразу вся, чтобы потом пребывать неизменной все время своего существования.

НЕПРЕРЫВНАЯ РАЗРАБОТКА АРХИТЕКТУРЫ

Непрерывная разработка архитектуры — это методика постоянного развития основ архитектуры продукта. Отдельные продукты создаются с использованием существующих основ архитектуры, и по отчетам о применении этих основ можно улучшить архитектуру продукта, делая ее более эффективной.

Гибкая методология разработки и DevOps посвящены улучшению разработки отдельных продуктов. Они не рассказывают о том, как улучшать основы этого процесса, чтобы более эффективная архитектура системы была полезна для архитектуры продукта. Непрерывная разработка происходит, когда отдельные продукты создаются в рабочем окружении уже существующей системы. Система облегчает создание новых продуктов, а они помогают системе найти области, требующие улучшения, таким образом создавая цикл обратной связи.

Непрерывная разработка архитектуры фокусируется на развитии основы, не вредящем уже существующим продуктам и позволяющем создавать комбинации продуктов. В идеале основы архитектуры все время развиваются так, чтобы быть совместимыми с предыдущими версиями, поэтому работа системы почти не нарушается.

Чтобы непрерывная разработка архитектуры стала возможной, платформа должна быть создана на принципах, протоколах и шаблонах. Мы можем проиллюстрировать эту мысль с помощью веб-платформы. Давно известно, что Интернет потрясающе устойчив и в то же время гибок. За прошедшие почти 30 лет[1] его основная архитектура не менялась, но, разумеется, заметно эволюционировала. Как это явное противоречие стало возможным? Ведь другие системы сталкиваются с проблемами гораздо быстрее, двигаясь по менее радикальным траекториям.

Одна из главных причин заключается в том, что ничто в Интернете не показывает, как именно реализуется или задействуется определенный сервис. Сервисы могут быть реализованы на любом языке, и, разумеется, с годами они меняются, поскольку меняются и языки программирования. Они предоставляются с помощью любой работающей среды — сначала это были серверы, располагающиеся в подвалах, теперь размещенные в самом Интернете, а в последнее время появляются и облачные хранилища. Сервисы могут быть использованы любой клиентской программой — они тоже кардинально изменились от простых браузеров на основе командной строки до сложных графических браузеров на современных мобильных телефонах. Фокусируясь исключительно на интерфейсе, который определяет, как происходят распознавание информации, обмен ею и демонстрация, архитектура Интернета оказалась лучше приспособлена к естественному росту, чем любая сложная IT-система, которую мы видели. Причины этого удивительно просты.

[1] Первый вариант проекта Всемирной сети (WWW) был предложен Тимом Бернерсом-Ли в 1989 году.

- *Принципы* — это фундаментальные понятия, построенные на самой основе платформы. В случае Интернета один из таких принципов — то, что ресурсы распознаются по идентификаторам единообразного ресурса (uniform resource identifier, URI), а затем распознающий URI протокол дает разрешение на взаимодействие с этими ресурсами. Таким образом, хотя мы и можем (как минимум в теории) перейти к Интернету без HTTP (и в каком-то смысле уже переходим, потому что везде протоколы меняются на HTTPS), трудно представить, что возможен переход к Интернету без ресурсов. Принципы отражаются в стилях API, потому что стили опираются на разные фундаментальные понятия.
- *Протоколы* определяют конкретные механизмы взаимодействий, основанные на фундаментальных принципах. Хотя большая часть взаимодействий в Интернете сейчас построена на протоколе передачи гипертекста (Hypertext Transfer Protocol (HTTP)), небольшая доля трафика еще относится к протоколу передачи файлов (FTP) и более специализированным протоколам, например WebSockets и WebRTC. Соглашение по поводу протокола делает взаимодействие возможным, а тщательная разработка платформы позволяет системе протоколов также развиваться, что мы сейчас и наблюдаем на примере HTTP/2[1].
- *Шаблоны* — это конструкции более высокого уровня. Они объясняют, как сочетаются взаимодействия по, возможно, нескольким протоколам для достижения целей приложения. Один из примеров — распространенный механизм Oauth, который представляет собой сложный шаблон на основе HTTP, предназначенный для достижения конкретной цели — трехступенчатой идентификации. Шаблоны — это способы решать распространенные проблемы. Они могут быть собственно протоколами, как Oauth, или методиками, как приведенный ранее пример с CDN. Но, как и протоколы, со временем шаблоны развиваются, добавляются новые, а старые деактивируются и уходят в историю. Шаблоны — это общие методики решения проблем в пространстве решений, установленном принципами и протоколами.

Зачастую шаблоны развиваются со временем, чтобы соответствовать меняющимся требованиям. Например, идентификация на основе браузера была относительно широко распространена на заре Интернета, потому что ее можно было легко контролировать с помощью конфигурации сервера и она хорошо работала для относительно простых сценариев того времени. Но с ростом Сети ограничения такого подхода стали очевидными[2], поддержка идентификации стала стандартной

---

[1] HTTP/2 — хороший пример того, как платформа Интернета может переходить от одной технологии к другой, но она была специально разработана почти без семантических отличий от HTTP/1.1, а бóльшая часть изменений и улучшений направлена на более продуктивное взаимодействие.

[2] Идентификация через браузер могла создавать неудобства для пользователя, например, было непросто разлогиниться из какого-либо сервиса.

функцией во всех популярных фреймворках для веб-программирования, и этот более гибкий подход заменил прежний метод идентификации через браузер.

Важно помнить, что эта петля обратной связи и ответственна за успех Интернета. Архитектура платформы начинается с простых решений. Появляются приложения, и какие-то из них расширяют границы того, что поддерживает платформа. При наличии спроса к платформе добавляются новые характеристики и функции, и это облегчает создание новых приложений, использующих данные функции. В обязанности архитекторов платформы входит наблюдать, где именно приложения расширяют границы, помогать разработчикам переходить их и развивать платформу, чтобы она лучше отвечала потребностям разработчиков.

В системах API происходит такая же эволюция методов. И в ней надо видеть не *проблему*, а *особенность*, потому что по мере того, как команды обучаются и появляются новые шаблоны, а иногда даже протоколы, методы можно подстраивать и улучшать. Секрет успешных программ с API — подходить к ним как к непрерывно развивающимся, разрабатывать их и управлять ими так, чтобы эта эволюция продолжалась.

Системы API как языковые системы

Каждый API представляет собой особый язык. Он заключается во взаимодействии тех, кто предоставляет сервис и кто им пользуется, когда появляется определенная функция. Но важно запомнить, что в данном разделе термин «язык» относится к взаимодействиям с API, то есть к его дизайну, а не к внутренней работе API, то есть его реализации.

Некоторые аспекты языка отдельного API решаются на фундаментальном уровне.

- *Стиль* API определяет базовые шаблоны диалогов (например, синхронный запрос/ответ или асинхронный на основе событий) и основные ограничения. Например, в API на основе RPC ключевое абстрактное понятие диалога — запрос функции, а в API на основе REST диалог строится на понятии ресурсов.
- Затем *протокол* API определяет базовые механизмы языка. Например, в API на основе HTTP при управлении основами диалога точно будут важны поля заголовка HTTP.
- Внутри протокола API часто существует множество технологических подъязыков в форме расширений основной технологии. Например, сейчас существует около 200 полей заголовка HTTP (http://webconcepts.info/concepts/http-header/), хотя основной стандарт определяет лишь небольшой их набор. API могут выбирать, какой из этих подъязыков они будут поддерживать, основываясь на потребностях своего диалога.
- Определенные аспекты API могут относиться к разным областям, и их можно использовать повторно в нескольких API (более подробно мы рассмотрим

это в подразделе «Словарь» на с. 188). В качестве примера: эти повторно применяемые части API можно назвать типами данных, затем ссылаться на них и задействовать в разных API, чтобы не изобретать велосипед.

Главный вывод из сказанного состоит в том, что *управление языками* — важная часть *управления системой*. Как и во всех системах, управление языками — сложная задача. Если их слишком сильно унифицировать, пользователи системы будут чувствовать себя скованно и не смогут самовыражаться так, как им хочется. Если же не пытаться поощрять возникновение общих языков, в системах появится множество решений одной и той же проблемы и они станут очень сложными[1].

Один из самых распространенных шаблонов управления системой API — продвигать повторное использование языка с помощью пряника, а не кнута.

- *Метод кнута* характеризуется небольшой командой руководителей, решающих, какие языки должны задействоваться, а затем объявляющих, что другие языки запрещены. Обычно это решение приходит сверху, и становится сложно экспериментировать с новыми решениями и применять новые методики.
- *Метод пряника* позволяет предлагать для повторного использования любой язык, если с ним связаны инструменты и библиотеки, которые способны облегчить жизнь пользователям. Таким образом, язык должен доказать свою полезность. Это также означает, что можно добавить язык в список, продемонстрировав его пользу.

С применением метода пряника набор предложенных языков со временем будет и должен развиваться. Если возникают новые языки, должны появляться и новые способы доказать их пользу, и если это получится, их тоже добавляют в список.

В итоге языки могут «попасть в опалу» из-за появления более успешных конкурентов или из-за того, что пользователи просто начали пользоваться другим методом работы. Это давно происходит в сфере XML/JSON. Хотя существует много XML-сервисов, сейчас выбором по умолчанию для API является JSON (а через несколько лет, возможно, мы увидим, как он постепенно замещается другой технологией).

Выражать API через API

Масштабирование API предполагает, что, когда приходит время расширяться, у вас уже есть план автоматизации растущего количества задач для отдельных API и для системы. Автоматизация требует точного определения того, как открывается доступ к информации и предоставляется возможность собирать и использовать

[1] Это хороший пример различия между комплексностью и сложностью. Комплексность системы API определяется характеристиками различных API и их отражением в API. Сложность появляется, когда одна и та же проблема решается по-разному в разных API, создавая разнообразие языков там, где оно не нужно с точки зрения функциональности.

ее. Если задуматься, то задача открытия доступа к информации и есть основное предназначение API! Это приводит нас к главной мантре выражения API через API: «Все, что вы хотите сказать об API, нужно выразить с помощью этого API».

Таким образом, мы знакомимся с идеей о том, что важная часть масштабируемого управления системой API вращается вокруг идеи применения так называемого API для инфраструктуры (или, точнее, компонента инфраструктуры в существующих API). Очень простым примером такого API для инфраструктуры может послужить способ демонстрации информации о состоянии API. Если каждый API соответствует *стандартному шаблону* (подробнее об этом — в подразделе «Словарь» раздела «Восемь аспектов систем API» далее), автоматизировать задачу сбора информации о состоянии всех API очень легко. Если упростить, это выглядит примерно так.

- Начните с инвентаризации активных в данный момент экземпляров сервисов, проверяйте каждый из них раз в 10 минут.
- Начните со стартовых страниц сервисов, перейдите по ссылке со словом status, чтобы найти информацию об их состоянии.
- Соберите информацию о состоянии с каждого сервиса и обработайте/визуализируйте/заархивируйте ее.

По этому сценарию легко написать скрипт, регулярно собирающий эту информацию, и, основываясь на ней, создать инструменты и новые идеи. Это стало возможно, потому что *компонентом API* является *стандартный способ доступа к определенным аспектам API*.

Теперь мы понимаем, что управление системой API и ее развитие частично зависят от развития способов использования API для подобной автоматизации, когда это необходимо. Задействуя принципы непрерывной разработки архитектуры, можно со временем добавлять эту информацию и модифицировать при необходимости уже существующие сервисы.

В этом примере открытие информации о состоянии стало новым шаблоном и появилась методика применения этой информации. Новая методика может из экспериментальной стать реализуемой, если в системе API используются такие категории в указаниях по дизайну. В дальнейшем она может перейти в состояние увядающей и устаревшей, когда ее будут задействовать только старые сервисы, если в какой-то момент система перейдет на другой способ демонстрации состояния API.

В последнем абзаце мы использовали слова «экспериментальная», «реализуемая», «увядающая» и «устаревшая» в качестве возможных состояний процесса руководства. Мы не предлагаем вам пользоваться именно этим набором, но важно понимать, что любое руководство также развивается со временем. То, что когда-то было удачным способом решения проблем, может быть заменено более быстрым и устойчивым методом. Руководство должно помогать командам

решать, как им разбираться с проблемами. Если команды будут отслеживать состояние методик, им станет проще понимать, как они развиваются, поэтому разумно следить за тем, какие решения удачны сейчас, какие могут стать удачными в будущем и какие считались удачными в прошлом. В разделах «Структурирование руководства в системе API» и «Жизненный цикл руководства в системе API» главы 9 мы обсудим конкретные способы структурирования и развития руководства подробнее.

Решение проблемы методом, который становится элементом дизайна API, упрощает управление большими системами API, потому что определенные элементы дизайна повторяются среди API и их можно использовать для автоматизации.

Понимание системы

Системы API не отличаются от других систем продуктов или функций, цель которых — развиваться легко и почти беспрепятственно и служить прочным фундаментом для создания новых функций, внутренних или внешних. Во всех этих случаях достигают компромисса между оптимизацией ради одной известной цели и оптимизацией ради облегчения изменений. Оптимизация ради облегчения изменений всегда требует компромиссов при наличии фиксированных целей. Ключевые факторы изменяемости — открытость системы для эволюции и управление ею таким образом, который позволяет вносить новые идеи в текущее состояние системы и в ее динамику с течением времени.

В этой картине важное место занимает мысль, которую мы обсудили в предыдущем разделе: все, что должно быть сказано об API, должно быть выражено с помощью самого API. Это может быть так же просто, как предоставление информации о состоянии, упомянутое ранее, или куда более сложно, как, например, требование, согласно которому вся документация по API должна быть частью самого API, или управление безопасностью API через сам API. При таком подходе API переходят на самообслуживание, открывая доступ к информации, которая нужна для их понимания и использования.

Иногда такой подход может стоить дорого. Если брать крайний случай, когда вы хотите создать API, открытые для доступа миллионам разработчиков, тогда имеет смысл сделать их максимально высокотехнологичными, чтобы разработчики могли максимально легко понять и применить их. В этом случае появляется продукт, разработанный для масс-маркета, и поэтому он оптимизирован именно для такого использования.

В большинстве систем API появятся сотни или тысячи API, и невозможно, да и не обязательно, доводить их до такого же идеала, как продукты для масс-маркета. Но и небольшая стандартизация может потребовать большой работы, например: убедиться, что команда по API легко находит контактную информацию,

предоставить минимальную документацию, распознаваемое машиной описание и, возможно, создать несколько примеров для начала.

А когда кажется, что API нужно еще слегка отполировать, может помочь эволюционная модель системы: команды по API начинают применять методы улучшения опыта разработчика, и эти методы становятся установленными и получают поддержку. Повторим еще раз: важно наблюдать, где появляется необходимость изменения, какие решения используются в API, и поддерживать методики, которые вы хотите закрепить на уровне системы.

Восемь аспектов систем API

Управление системами API может пугать своей сложностью. Для него требуется установить баланс между скоростью и независимостью продукта, с одной стороны, и проблемами согласованности и устойчивости — с другой. Но прежде, чем подробно обсудить развитие системы API (мы сделаем это в главе 9), хотим представить вам фреймворк, который позволит оценить, насколько ваша система готова к долгосрочному развитию.

В модели восьми аспектов систем API мы предположили, что API создаются и разрабатываются множеством разных способов и идут по множеству разных путей в течение своих жизненных циклов, как говорилось в главе 6. Эти восемь аспектов представляют собой как бы кнопки или рычаги управления системой руководства API. Вам нужно будет понаблюдать за ними и настроить их, чтобы получить наилучший результат.

Предположение состоит в том, что разработка и реализация стратегии системы похожи на модель, которую мы обсудили в разделе «Принцип платформы» данной главы, где платформы являются открытыми, а добавление на них API должно соответствовать принципам, протоколам и шаблонам этих платформ.

Если иметь в виду такую открытую модель системы API, то важно рассмотреть следующие восемь аспектов, которые в той или иной форме взаимодействуют с *дизайном* и *реализацией* отдельных API и *организацией* всей системы API. Они также помогут наблюдать за системой и понять, как она постоянно эволюционирует.

В следующих разделах мы представим и опишем восемь важных аспектов, о которых следует помнить при управлении системой API. Будем применять их и в главе 9, для модели развития системы, которая использует риски, возможности и потенциальные вложения во все эти области как способ управлять развитием системы. Задействуем также их в главе 10, чтобы объяснить, как управление жизненным циклом на уровне системы может помочь с жизненным циклом отдельного API.

Мы определили восемь важных аспектов: разнообразие (variety), словарь (vocabulary), объем (volume), скорость (velocity), уязвимость (vulnerability), видимость (visibility),

контроль версий (versioning) и изменяемость (volatility), чтобы направлять и фокусировать управление системами API. Далее обсудим каждый из этих аспектов подробно.

Разнообразие

Под *разнообразием* подразумевается то, что в системах API часто имеются программы, созданные и разработанные разными командами на разных технологических платформах и для разных пользователей. Цель API — позволить этому разнообразию существовать и обеспечить командам бóльшую автономию.

Например, может иметь смысл наличие одного руководства по дизайну, предлагающего API на основе REST в качестве выбора по умолчанию для сервисов на основной платформе, потому что они максимально просты в применении и с ними может работать максимально возможное число пользователей. Однако конкретно для серверной части мобильных приложений может быть разумно предложить API, основанный на запросах и использующий такую технологию, как GraphQL[1], потому что так мобильные приложения смогут получать именно те данные, которые им нужны, за одно взаимодействие.

Системы API должны устанавливать баланс разнообразия. Одной из целей управления системой API является управление разнообразием и его ограничение, чтобы разработчикам не нужно было учиться взаимодействовать с огромным количеством разных стилей API. В то же время ограничивать разнообразие одним вариантом дизайна может быть неразумно, если существуют группы предпочтительных вариантов, прекрасно подходящих для разных сценариев, и, таким образом, наилучшие продукты предоставляются большему количеству пользователей.

Управление разнообразием в системах API — это баланс между некоторым ограничением числа вариантов, позволяющим избежать непродуктивного множества «оттенков» API, и в то же время открытостью для тех вариантов, которые делают систему API более ценной, позволяя продуктивно разрабатывать большее количество приложений на этой платформе.

Самый фундаментальный аспект таков: относитесь к управлению разнообразием как к долгосрочному руководству — не создавайте в своей системе ничего, что помешает развивать такое понимание разнообразия, которое вы со временем захотите поддержать. Идеальные системы API через несколько лет точно не будут выглядеть так, как сейчас, поэтому не перекрывайте пути работы с развитием разнообразия, чтобы не загнать себя в угол.

---

[1] Советуем почитать: *Бэнкс А., Парселло Е.* GraphQL: язык запросов для современных веб-приложений. — СПб.: Питер, 2019. — *Примеч. ред.*

ПРЕДПОЧТЕНИЯ В API С ТЕЧЕНИЕМ ВРЕМЕНИ

У вас могут быть определенные предпочтения в области дизайна API, и вы можете их использовать при руководстве разработкой. Можете поощрять разработчиков придерживаться этих предпочтений, потому что так вы получите оптимальное соотношение доходов и расходов с точки зрения системы..

Но не стоит ориентироваться только на этот набор предпочтений. Со временем может появиться что-то лучшее, и вы передумаете. Или пользователям потребуются определенные API, и вы захотите им угодить.

Например, можно рассмотреть GraphQL: неважно, что вы думаете об этой конкретной технологии, но если работаете над партнерскими или внешними API, то можете узнать, что определенное количество пользователей предпочитает именно GraphQL. Важно иметь возможность поддержать эти группы предпочтений, потому что со временем они эволюционируют и будут показывать, в каком направлении развивается ваша система.

Никогда не думайте, что нашли единственно верный способ создавать API: любое ваше действие зависит от технологий и предпочтений пользователей в данный момент и со временем может измениться.

Поддерживать и контролировать разнообразие — это длительный процесс. Его поддержка должна быть встроена в систему с самого начала. Ограничивать его с помощью принципов, протоколов и шаблонов — это значит находить баланс между пониманием того, как используются API и какой доход они приносят, и выбором ради увеличения дохода. С развитием системы API (см. главу 9) вы может начать лучше понимать ее состояние, пути развития и применения. Тогда можно будет контролировать разнообразие с помощью баланса между затратами на его увеличение (что снижает согласованность внутри системы) и более специфичным дизайном API (что улучшает ценность API в подгруппе внутри системы).

Словарь

Каждый API — это язык, как сказано в подразделе «Системы API как языковые системы» раздела «Системы API как языковые системы» этой главы. Он определяет взаимодействие разработчиков с сервисом через шаблоны взаимодействий, скрытые протоколы и обмен формами представления. Стандартизация структурных элементов API с помощью общей терминологии — действенный способ увеличить согласованность внутри системы.

Для некоторых аспектов этого языка не обязательно каждый раз заново изобретать велосипед. Простой пример — проблема сообщений об ошибках. Многие API на REST на основе HTTP составляют собственные сообщения об ошибках, потому что хотят выйти за границы использования только стандартизированных кодов

статусов HTTP. Такой формат можно определить индивидуально, но в формате «деталей проблемы» от RFC 7808 (https://tools.ietf.org/html/rfc7807) уже имеется стандартная форма (если API задействует JSON или XML). Применяя терминологию отчетов об ошибках, вы получаете два преимущества.

- Командам, *разрабатывающим* API, не придется изобретать, составлять и документировать новую терминологию для сообщений об ошибках. Они могут адаптировать существующую и, возможно, расширить ее, чтобы отразить специфические аспекты своих сообщений.
- Командам, *использующим* API, не нужно изучать авторский формат, они поймут эту часть языка API, после того как впервые столкнутся с данной терминологией. Таким образом, разработчикам будет проще понять аспекты API, не специфичные конкретно для этой программы.

Следующий пример взят из RFC 7807 и показывает, как такой формат может даже сочетать стандартную и специфическую терминологию. В этом примере части `type`, `title`, `detail` и `instance` (тип, название, детали и программа) определены RFC 7807, а данные в `balance` и `accounts` (баланс, аккаунты) — специфические для конкретного API. Возможно, в своей системе API вы всегда будете использовать отчеты об ошибках от RFC 7807, но данные со временем эволюционируют, когда API начинают показывать конкретные детали проблемы:

```
{
  "type": "https://example.com/probs/out-of-credit",
  "title": "You do not have enough credit.",
  "detail": "Your current balance is 30, but that costs 50.",
  "instance": "/account/12345/msgs/abc",
  "balance": 30,
  "accounts": ["/account/12345", "/account/67890"]
}
```

Во многих случаях такое повторное использование терминологии достигается с помощью стандартов — неважно, технических стандартов на уровне Интернета или стандартов внутри одной системы API. Важно максимально избегать изобретения велосипеда.

На самом деле способность одинаково работать с официальными стандартами и стандартами своей системы очень важна для того, чтобы иметь возможность решить, когда имеет смысл переключиться на один из стандартов ради какого-то аспекта языка API.

EIM И API: ИДЕАЛ ПРОТИВ ПРАГМАТИЗМА

Хотя использование официальных стандартов является довольно прямолинейным способом избежать повторного составления словаря, иногда организации идут дальше. Самый крайний случай — это идея *модели информации предприятия* (enterprise information model, EIM), целью которой является создание полной и согласованной

модели всего, что должно быть представлено в организации. Во многих случаях (чаще в больших организациях) эта цель недостижима: на документирование всего словаря сложной организации затрачивается очень много сил, а когда оно закончено, оказывается, что реальность и системы уже изменились, и EIM становится слепком прошлого.

EIM должна развиваться вместе с организацией, но синхронизировать их очень сложно. Например, у организации может быть определенная модель пользователя и связанной с ним информации. Скорее всего, эта информация все время дополняется, а различные продукты расширяют/дополняют модель пользователя так, как им удобно. Попытки убедиться, что эти расширения и дополнения всегда согласованы и скоординированы, замедлят дизайн и исполнение сервиса. На практике это означает, что зачастую приходится выбирать между EIM, отражающей статическую модель организации, и увеличением способности организации к изменению при необходимости, при этом отказываясь от идеально разработанной и гармоничной модели всего.

Более реалистичный подход состоит в том, чтобы предположить, что эффективная EIM — это перечень всех функций, доступных с помощью API. Но при этом необходимость решить, насколько вам нужна стандартизация словаря, лежит на руководстве системой API. Для некоторых вопросов, касающихся нескольких областей (например, вышеупомянутых сообщений об ошибках), предпочесть стандартную терминологию довольно легко.

Понятия с более четкой предметной областью будут выражены с помощью дизайна API, и тогда именно он станет EIM в этой области. Недостаток такого подхода заключается в том, что у вас не будет одной согласованной и единообразной модели всего, какой часто стремится стать EIM. Но его преимуществом является то, что модель предметной области теперь полностью управляема (с помощью API), а как сказано в определении этого подхода, то, что не выражено и/или не управляется с помощью API, — это не часть EIM.

Управление словарем при расширении API получается лучше всего, если оно сосредоточено в основном на том, чтобы его было проще найти и использовать повторно, а не на создании единственной верной модели своей концепции. Если словарь легко найти и повторно применить, разработчики будут заинтересованы в том, чтобы пользоваться им, пока он отвечает их целям, потому что так им не придется разрабатывать свой.

Как *определиться* с терминологией? Сложная тема. Не будем углубляться в дебри UML и XML, определения, документации и составления словарей, но напомним, что целью API является *не открыть всем* модель реализации, а создать для нее *интерфейс*, что часто отличается от внутренней модели сервиса или предметной области (как описано в главе 3, одна из стратегий предполагает *начать* с этой модели интерфейса, даже не задумываясь поначалу о ее реализации).

Словарями можно управлять по-разному. Каждый способ имеет свои преимущества и ограничения.

- Когда словари используются для *полного отражения взаимодействий API*, они становятся точными моделями значения и идентификации различных концепций своей предметной области. Типичные примеры — схемы в XML или типы документов в JSON. Различается и *распознавание* этих словарей: в некоторых случаях, например в интернет-API, применяются типы данных, в других — идентификаторы схем, которые затем в явной или скрытой форме ассоциируются с их использованием в API.
- Терминология тоже часто задействуется в качестве *структурных элементов внутри представления*, что позволяет API поддерживать те представления, части которых применяют эту конкретную терминологию. У XML есть для этого довольно сложный механизм с пространствами для имен, а у JSON нет формального способа определения того, что какая-то часть представления использует стандартную терминологию. Ранее мы рассмотрели пример из RFC 7808, в котором есть встроенные стандартные термины, к тому же он позволяет API добавлять собственные значения в формат сообщения об ошибке.
- Словарь может быть и *общим типом данных*. В этом случае часто существует шаблон определения и поддержки развивающегося набора значения для типа данных с помощью *реестра*. Реестры позволяют пользователям делиться развивающимся набором известных значений для определенных типов данных, это распространенный шаблон для фундаментальных технологий Интернета. В качестве примера можно привести связи ссылок гипермедиа: существует реестр, из которого разработчики могут узнать об уже существующих связях и куда при необходимости добавить новые.

Важно выбрать правильный способ утверждения терминологии и управления ею. Он определит, будет ли командам по API проще (частично) собирать свои программы из уже существующих структурных элементов, а не создавать их с нуля. Но утверждать терминологию стоит, только если существует четкая модель, по которой ее можно легко найти и применить повторно. Лучше всего использовать терминологию с помощью инструментальных средств, чтобы у дизайнеров был определенный набор вариантов. Например, при создании API на REST у HTTP есть набор концепций, которые являются открытыми и развивающимися словарями.

Как показано на рис. 8.2, у HTTP есть немало связанных с ним словарей (рисунок представляет собой скриншот «Веб-концепций» — открытого хранилища, предоставляющего доступ к стандартным и распространенным значениям для этих словарей). Вряд ли в системе API дизайнерам API на HTTP понадобятся все 200 существующих полей заголовков HTTP, но на одной из подгрупп этих значений можно основывать инструменты для дизайна и реализации API, тем самым устанавливая общую методику выбора поля заголовка внутри организации.

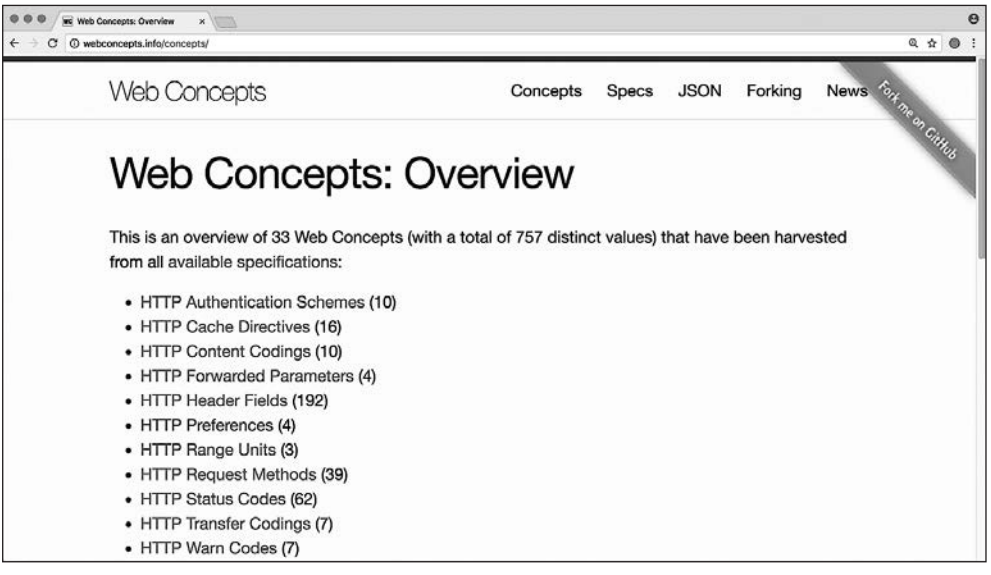


Рис. 8.2. Веб-концепции: терминология HTTP

Терминология HTTP служит примером скорее технической информации, где важно поделиться методами использования конкретной технологии. В случае конкретной области те же принципы можно применить к концепциям этой области, например, типам клиентов. К примеру, у компании может существовать словарь по пяти различным типам клиентов, который со временем может вырасти и охватить дополнительные типы. Создать реестр терминов этой сферы будет полезно, чтобы общие значения были доступны разработчикам и инструментам и могли развиваться со временем.

Объем

Как только организация начинает серьезно относиться к своей стратегии цифровой трансформации, *объем* существующих или доступных благодаря API сервисов может резко вырасти. Одной из причин этого становится то, что все бóльшая часть организации отражается в технологиях, появляется «цифровой отпечаток», поэтому количество API в ней увеличивается. Оно легко может вырасти до сотен или тысяч, если мы говорим об организациях, превышающих определенный размер и уже разрабатывавших API. Системы API должны с легкостью работать с таким масштабом, поэтому размер системы становится прежде всего решением в интересах организации.

Вторая причина — в том, что при работе со стратегией «API как продукт» (см. главу 3) все, что делается в организации, задумывается по принципу «первым делом API», потому что только так оно может стать частью сетевого эффекта, при котором

все больше дел в организации выполняется с помощью API. Это также может означать, что многие такие инициативы при AaaP далеко не пройдут, потому что API должны не только облегчать и ускорять для организации процесс сочетания услуг, но и помогать экономить на этом, чтобы продукты можно было быстро создавать и оценивать (и, возможно, отклонять).

Существуют разные подходы к работе с объемом API в системе. Иногда подход состоит в том, чтобы уменьшить объем насколько возможно (что довольно относительное понятие в сложных организациях) и попытаться внимательно управлять системой API. В других случаях фокусируются на облегчении работы с этим объемом и на поиске новых идей для системы API, поэтому объем становится в основном проблемой управления и с ним можно работать посредством постепенных улучшений.

В любом случае рост объема естественен, так как система API развивается и в ней появляются новые сервисы. Поэтому так же естественно, что с ростом системы работа с растущим объемом не должна становиться проблемой.

Скорость

Одна из важнейших характеристик цифровой трансформации — ускорение разработки, реализации, тестирования и изменения продукта. Это происходит благодаря тому, что управление отдельными API в организации развивается и система API позволяет работать быстрее, чем до появления этих навыков. Организация также становится гораздо более гибкой сетью отдельных, но при этом взаимозависимых сервисов: если до этого существовало много структурных элементов, которые долго находились в стабильном состоянии и новые элементы добавлялись относительно редко, то теперь все меняется и добавляется гораздо быстрее.

Дополнительная *скорость* является одним из ключевых отличительных аспектов, приносящих выгоду организациям на рынке, однако важно не забывать о *безопасности*. Иначе растущая скорость станет угрожать устойчивости операций, а это недопустимо.

Для многих организаций скорость становится одним из главных мотивов для начала цифровой трансформации: рынок сейчас меняется быстрее и появление конкуренции ускоряется, поэтому им жизненно важно иметь возможность действовать или хотя бы реагировать как можно быстрее. Любые факторы, замедляющие скорость превращения идей в продукцию или препятствующие управлению портфолио постоянно растущего и развивающегося продукта, вредят и конкурентоспособности организации.

В системах API скорость значительно увеличивается, если дать командам больше свободы в разработке, чтобы они могли работать в соответствии со своими предпочтениями, выбором и графиком. Это необходимо для сокращения до минимума всех частей процесса, которые могут замедлить выпуск API. Как упоминалось

ранее, важный урок, извлеченный из прежних подходов к API, состоит в том, что *разъединение* (то есть возможность изменять и размещать индивидуальные компоненты независимо от других) необходимо для сокращения времени на создание API[1]. Но если вы позволяете добавлять, изменять и исполнять отдельные функции независимо друг от друга, необходимо изменить и традиционные методы тестирования API и работы с ним.

Как мы говорили ранее, в подразделе «Принцип платформы» раздела «Управление API в больших масштабах», один из распространенных подходов к тому, чтобы избежать связанности, приносящей потери в скорости, заключается в том, чтобы отказаться от интеграции. Использовать платформу так, как мы описываем, означает уйти от идеи интеграции и принять идею децентрализации. Это выгодно, так как слабое связывание обеспечивает высокую скорость в отдельных частях, потому что уменьшаются затраты на координацию при внесении изменений. Недостаток этого подхода в том, что придется подстроить выпуск API и операции под новую систему и убедиться, что она соответствует стандарту устойчивости, необходимой организации.

Если взять крайности, можно в очередной раз привести в пример Интернет — это интересное упражнение. Он быстро меняется, потому что могут появиться новые сервисы, измениться старые, и все это может повлиять или не повлиять на пользователей. Можно возразить, что в некоторой степени Интернет никогда не работает: всегда что-то сломано, какой-нибудь сервис недоступен или на пользователя влияют изменения сервиса. Но итоговая скорость всей системы сполна искупает присущую ей хрупкость, и с помощью качественного управления изменениями и правильных методов развертывания и тестирования возможно найти в подобной системе баланс между скоростью и ценностью.

Уязвимость

Если в организации не применяются информационные технологии, она неуязвима (по крайней мере напрямую) для информационных атак. Но, поскольку появляется тенденция к расширению технологий, бизнес связан с ними все теснее, а их функции открываются посредством API, появляется масса уязвимых мест и с ними надо разбираться. Вы должны убедиться, что опасность увеличения мишени для атаки в системе API более чем компенсируется преимуществами от возросших гибкости и скорости.

Экономический подход API выгоден, потому что организации могут быстрее реагировать и реструктурироваться, когда используют внутренние API, а также передавать некоторые функции внешним работникам с помощью API. Однако у всего этого есть и обратная сторона — появляются взаимозависимости. Например, в июне 2018 года Twitter приобрел компанию Smyte, предоставляющую технологии для борьбы с оскорблениями. Многие компании пользовались услугами

[1] Ранее мы сравнивали это с технологией SOAP, которая фокусировалась только на API и не указывала, как управлять растущей и меняющейся системой сервисов SOAP.

Smyte с помощью API, предлагавших инструменты для предотвращения оскорблений, сексуальных домогательств и рассылки спама в Сети. У этих компаний даже были заключены договоры со Smyte. Сразу же после приобретения Smyte без предупреждения закрыла свои API, создав проблемы для компаний, которые полагались на них.

Этот и подобные ему случаи учат нас всегда относиться к *внешним взаимозависимостям* как к уязвимым местам, обязательно встраивать в сервисы устойчивость, чтобы противостоять возможным разрывам связей между ними, и сделать это фундаментальным методом разработки. Можно пойти дальше и распространить это правило на любую взаимозависимость, не только внешнюю, потому что при увеличении скорости возрастает и вероятность появления проблем с уже запущенными зависимостями и любое неустойчивое использование сервиса предсказуемо может стать проблемой.

Зачастую реализовать эту устойчивость не так-то просто. В некоторых ситуациях взаимозависимости очень важны и практически невозможно как-то компенсировать их недоступность. Но даже в таких случаях важно ответственно подойти к этой ситуации: сервис не должен ломаться, зависать или уходить в неопределенное операционное состояние, он должен четко сообщить о ситуации, чтобы ее можно было проанализировать и исправить.

Кроме технической уязвимости, существует также тот факт, что с увеличением количества доступных сервисов увеличивается и количество API, которые могут использовать как мишень для атаки. Это перерастает в проблему, когда речь идет о попытках злоумышленников получить доступ к системе или просто нарушить ее работу. Проблемы также возникают, когда API обнародует информацию или функции, которые по юридическим, нормативным или конкурентным причинам раскрывать нельзя. Это может серьезно повлиять на отношение к организации, поэтому таким случаям необходимо уделять не меньше внимания, чем прямым угрозам.

В целом, с уязвимостью нужно справляться с помощью безопасной работы с API. С точки зрения безопасности лучше относиться ко всем взаимозависимостям, как к брешам в защите, и никогда не зависеть от доступности или конкретного поведения другого API, а также всегда следить за тем, чтобы злоумышленники не могли получить доступ к API в системе или нарушить их работу.

Видимость

Видимость и большой объем — практически естественные враги. Если у вас мало программ и все, кто разрабатывает и использует их, а также управляет ими, входят в одну относительно небольшую команду, тогда большая часть действий видна или их очень просто обнаружить. Нужно спросить окружающих, и они тотчас расскажут, где найти искомое и как оно работает. Если вам нужен обзор всех

составляющих, можете их просто просмотреть, чтобы разобраться. Но все это, разумеется, не относится к большим системам API.

В больших и децентрализованных системах обеспечить видимость гораздо сложнее. Типичные схемы видимости в больших масштабах как в жизни, так и в IT-системах обычно сочетают в себе два аспекта (и тут мы снова вспомним Интернет, потому что это самый большой пример видимости, существующий на данный момент).

- *Публикация чего-либо* должна происходить так, чтобы это можно было *обнаружить*. В Интернете для этого нужны рабочий веб-сервер и HTML для публикации, который используется для индексирования содержимого. Существуют механизмы Sitemaps и Schema.org, предназначенные для облегчения обнаружения, но они были изобретены гораздо позже начала работы поисковых механизмов.
- *Поиск* — обычно больше зависящее от внешнего окружения задание, чем просто обнаружение. Поиск чаще всего связан с контекстом и тем, насколько «полезные» или «бесполезные» результаты поиска были получены. Google произвел революцию своим алгоритмом PageRank (http://ilpubs.stanford.edu:8090/422/1/1999-66.pdf), рассчитывающим релевантность по популярности. Во многих случаях к поиску относятся как к дополнительному сервису, ставя его на второе место после изначальной задачи по обнаружению и сбору информации.

Как мы обсудили в разделе «Выражать API через API» ранее в данной главе, предпосылка для видимости API и ее использования в системе заключается в том, чтобы раскрывать информацию об API через эти же API. Именно этот гениальный ход заставил функционировать Интернет: вся информация о веб-странице находится на веб-странице, и существует единый способ получить доступ ко всем веб-страницам. Таким образом, видимость означает отслеживание того, как можно помочь пользователям, сделав API лучше видимыми, размышления о том, заключается ли проблема в возможности их найти (можно ли их вообще локализовать?), в представлении (доступна ли необходимая информация через API?) или в поиске (применяют ли поисковые механизмы открытую информацию?), и доработку тех факторов системы, которые нужно доработать, чтобы улучшить видимость.

Важна видимость не только самого API, но и его элементов. Например, такие аспекты, как стандартный формат деталей ошибки, который мы обсуждали в разделе «Словарь», помогает улучшить видимость внутри API, потому что теперь станет возможно использовать инструменты, понимающие детали ошибки в разных API. В принципе, существует прочная связь между терминологией и видимостью: чем больше общей терминологии у разных API, тем проще пользоваться этим их общим аспектом. Для модели выражения API через API видимость, полученная благодаря общей терминологии, очень полезна: если вы хотите сказать что-то об API, выразите это через него и в идеале сделайте этот способ общим для нескольких API.

Контроль версий

Одна из сложностей, возникающих при переходе от интеграции к децентрализации, — то, что изменения также происходят децентрализованно. Плюс здесь в том, что увеличивается скорость — одно из важных преимуществ систем API. Но, чтобы разумно справляться с изменениями в системе API, к *контролю версий* нельзя подходить так, как в интегрированных системах.

Распространенный подход при этом — *избегать появления версий* как можно дольше или как минимум избегать описания того, как выпускаются разные версии, и различий между ними для пользователей. Мы можем снова обратиться за примером к Интернету: немногие сайты выпускают новые версии, с которыми пользователям нужно учиться работать заново. Вместо этого они стремятся вносить улучшения таким образом, чтобы пользователи поняли, как использовать новые функции не в ущерб установившемуся порядку действий, выполняемых для применения сервисов.

Главная цель создания версий в системах API должна быть похожа на цель создания версий сайтов: не навредить существующим пользователям и изначально разработать все API с функцией расширения, чтобы между версией, ожидаемой пользователями, и версией, предоставляемой сервисом, существовала слабая связь[1].

В подобной модели не нужен жесткий контроль версий: к изменениям, совместимым с предыдущими версиями, относятся как к *улучшенной/расширенной версии API*, которую не обязательно выбирать, если вы не хотите узнать о новых функциях и использовать их. Изменения, несовместимые с предыдущими версиями, вредят API. Необходимо выпускать новую версию, и пользователи должны переходить на нее со старой. В таком случае это новый API, а не новая версия старого API.

Можно сказать, что тема версий для API нерелевантна. Однако API развивается с течением времени, поэтому полезно иметь возможность говорить о слепке функций API в какой-то определенный момент и понимать, что с тех пор изменилось. Поэтому все-таки номера версий — полезная концепция, которая может пригодиться для определения функций API и поиска по его истории.

Семантический контроль версий. Это простая схема контроля (https://semver.org/), основанная на использовании номеров версий, имеющих структуру и значение. Семантические номера версий структурированы по схеме `СТАРШАЯ.МЛАДШАЯ.ПАТЧ`. Эти части состоят из цифр и имеют следующее значение.

- Версии-патчи просто исправляют ошибки, не влияя на конкретные интерфейсы. Они лишь корректируют неправильное поведение или вносят другие изменения, влияющие только на реализацию.

[1] Придерживаться шаблонов для устойчивого расширения (http://bit.ly/2DDGmWR) можно, используя значимую основную семантику, пользуясь правильно определенной моделью расширения и правильно определенной моделью обработки взаимодействий с API и возможной работы с расширениями.

- Младшие версии — это изменения интерфейса, совместимые с предыдущими версиями. Клиенты могут продолжать задействовать их, не подстраиваясь под эти изменения.
- Старшие версии вносят изменения, требующие обновления API. Клиенты не могут просто перейти с одной старшей версии на другую.

При использовании семантического контроля версий продуктов с API номера версий становятся частью документации, потому что несут в себе информацию о том, что изменилось между версиями. Обычно полезно также документировать конкретные изменения, но семантический контроль версий представляет собой стартовую точку, в которой клиенты могут решить, хотят они исследовать обновления API или нет.

Изменяемость

Как мы говорили, когда речь зашла о *скорости* и *контроле версий*, динамика систем API отличается от интеграционных подходов, поэтому необходимо о ней помнить. *Изменяемость* неизбежна в больших децентрализованных системах: сервисы меняются (при этом стоит избегать изменений, которые вредят системе), перестают работать (при децентрализованной разработке и операциях создается менее централизованная модель доступа), могут даже исчезнуть (они не вечны). При использовании ответственных методов разработки в системе API необходимо помнить об этом, чтобы позаботиться обо всех взаимозависимостях.

Изменяемость можно считать результатом децентрализации. Необходимо принять тот факт, что она вызывает появление более сложного набора сценариев ошибок, чем сценарий «работает/не работает», существующий в интегрированных монолитных системах. Это неизбежный побочный эффект радикального разделения компонентов, и, разумеется, возникают затраты на работу с этой дополнительной сложностью (один из аспектов знаменитого «Устава API» Джеффа Безоса и его последствий, о чем мы рассказывали в подразделе «Устав Безоса» раздела «Дизайн-мышление» главы 3).

Для разработчиков переход от принципа «программирование как часть системы» к принципу «разработка как часть системы» может быть непростым. Традиционные предположения об устойчивости и доступности становятся неверными, а переход к модели, где любая внешняя взаимозависимость приложения может привести к провалу, требует дисциплины при разработке.

В то же время прекрасно известны техники постепенного сокращения возможностей. Снова возьмем для примера Интернет: хорошо разработанные веб-приложения часто и устойчиво задействуют постепенное сокращение возможностей. Этот принцип применяется к существующим взаимозависимостям от других сервисов, а также зависимостям от среды, в которой выполняется программа (браузера). Веб-приложения работают в среде, которую контролировать

гораздо сложнее, чем обычную среду для запуска. Поэтому им нужна встроенная устойчивость, иначе они не смогут работать в большом количестве браузеров или большом количестве сред.

Похожий подход к разработке нужен и для приложений в системах API. Чем более защищенным вы делаете приложение при программировании, тем более вероятна его устойчивость к вариациям в рабочей среде. Главная цель приложений в системах API — максимально устойчивая работа и отсутствие предположений, зависящих от доступности других компонентов.

Выводы

В этой главе мы ближе рассмотрели системы API. Узнали о том, как уже существующие программы для интеграции могут быть рассмотрены в качестве прото-API, как с расширением появляются новые сложности в работе с API и как должна выглядеть идеальная платформа для API.

Самое важное, что надо запомнить, — это то, что для получения дохода от систем API необходимо относиться к ним как к постоянно меняющемуся окружению, где изменения вызываются наблюдением за работой отдельных API, а роль системы — создать на основе этих изменений принципы, протоколы и шаблоны, которые помогут командам по API стать продуктивнее.

В этой главе мы ввели восемь V-систем API — набор аспектов, о которых полезно помнить, работая с конкретными задачами системы. Поскольку управление системой API — это всегда постепенный эволюционный процесс, мы будем использовать эти аспекты в следующей главе, где обсудим, как они могут помочь сообщать о вложениях в систему API и как вложения ускоряют развитие во всех этих аспектах.

9 Путь системы API

> Настоящая проблема в том, что программисты слишком долго волновались о производительности не там и не тогда, когда нужно. Преждевременная оптимизация — корень всех зол (или по крайней мере большей их части) в программировании.
>
> *Дональд Кнут*

В главе 8 мы подробно рассмотрели системы API, сосредоточившись на их основах и главнейших аспектах. Теперь перейдем к теме развития этих систем. Как и прежде, рассмотрим его как путь, а не как конечную точку: система API никогда не разовьется полностью, поскольку она всегда продолжает развиваться, следуя за эволюцией технологий и путями развития отдельных продуктов с API (это мы обсудили в главе 6).

В рамках этой аналогии такой взгляд на системы API похож на постоянную эволюцию Интернета. Он тоже никогда не разовьется полностью: появление новых технологий, новые сценарии и схемы использования постоянно подпитывают его эволюцию. Это может показаться пугающим, но именно непрерывное развитие — причина успеха Интернета. Без развития он в какой-то момент стал бы нерелевантным и победил бы какой-либо другой подход.

Системы API должны, как и Интернет, постоянно развиваться. *Непрерывная разработка архитектуры* закладывает фундамент для этого развития, чтобы архитектура могла эволюционировать в соответствии с изменениями потребностей и развитием принципов, протоколов и шаблонов.

Но даже при лучшей непрерывной разработке приходится сталкиваться с ограничением ресурсов, отведенных на изменения в архитектуре. Команды по API также могут справиться только с максимальным уровнем изменений. Так что, если существующая система достаточно хорошо работает в качестве платформы для продуктов с API, может быть экономичнее повторно использовать установленные

принципы, протоколы и шаблоны, а не пытаться создать идеальную платформу для каждой отдельной проблемы.

Поэтому, чтобы понять, как протекает развитие на уровне системы, необходимо знать, за чем в системе наблюдать и во что и как инвестировать, чтобы улучшить ее. Для этого мы снова задействуем восемь аспектов, с которыми познакомили вас в разделе «Восемь аспектов систем API» главы 8. Однако сейчас задействуем эти области, чтобы продемонстрировать опасность, которая возникнет, если не улучшать их, и расскажем, как это делать. С этой целью мы определили несколько контрольных точек, которые помогут вам лучше понять, каков уровень развития разных областей и куда направлять инвестиции, чтобы сделать его более единообразным.

Прежде чем обсуждать эти параметры и методы развития, расскажем о некоторых организационных аспектах, сопутствующих информированию и управлению системой API, действующему на опережение.

Структурирование руководства в системе API

Создание руководства и управление им — важная часть управления системой API. Оно сообщает каждому, *зачем* что-то нужно сделать, *что* делается для решения проблемы и *как* реализация может придерживаться руководства. Руководством нужно управлять, как развивающимся документом, который любой может прочесть, прокомментировать и обновить. Так каждый разработчик может почувствовать себя участником создания и развития этого документа.

Для повышения эффективности системы API всегда следует четко разделять ответы на вопросы «Что?» и «Как?» в теме требований к API, а также давать развернутый ответ на вопрос «Зачем?», объясняющий причину действий, и обеспечивать инструменты и поддержку конкретных способов удовлетворения требований.

- *«Зачем?» (мотивация руководства)*. Описывает причину появления данных требования или рекомендации, объясняя, что это не надуманное правило и у него есть объяснение. Если задокументировать причину, то при предложении альтернативных путей проще определить, действительно ли они нацелены на одну и ту же причину.
- *«Что?» (руководство дизайном)*. Объясняет подход, выбранный для достижения цели, чтобы прояснить, что должен выполнить API, чтобы удовлетворить данные требования. Для этого нужно определить четкие требования к самому API, а не к его реализации. Главнейший аспект описания ответа на вопрос «что делать?» — убедиться, что он не смешался с ответом на вопрос «как это делать?», который задается отдельно.
- *«Как?» (руководство по реализации)*. Рассказывает о конкретных способах реализации того, что надо делать. Эти способы могут включать в себя определенные инструменты или технологии. Возможны несколько подходов к тому, как

выполнить одну и ту же задачу. Со временем, когда команды, разрабатывающие API, найдут или создадут новые способы решения проблем, к существующим подходам будут добавлены новые методы, которые постепенно станут частью системы.

Эти указания должны помочь каждому эффективно работать в команде в системе API. Они созданы, чтобы увеличить продуктивность команд по API и изменить культуру и методы разработки API, которые легко отслеживать и которыми просто управлять.

Далее приведем конкретный пример использования этой схемы. Здесь решается распространенная задача — деактивация API, а конкретно — как сообщить пользователям API о приближающейся деактивации. В данном примере есть один пункт «Зачем?», два «Что делать?» и три «Как это сделать?».

- *«Зачем?» (мотивация руководства)*. Пользователям данного сервиса будет полезно узнать о грядущем списании API. Поэтому у него должен быть механизм оповещения о том, что скоро он будет выведен из эксплуатации.
- *«Что?» (руководство по дизайну № 1)*. API могут использовать поле заголовка HTTP «Закат» для оповещения о приближающемся списании. Необходимо конкретизировать, какие ресурсы будут задействовать поле заголовка (чаще всего выбирают домашний ресурс) и когда он будет появляться (обычно это происходит сразу после определения момента списания). Можно также уточнить, что поле заголовка будет появляться в течение какого-то определенного времени перед списанием, чтобы дать пользователям гарантированный период, когда они могут разобраться с последствиями данного решения или перейти на другой API.
- *«Как?» (руководство по реализации № 1 для руководства по дизайну № 1)*. Один из методов реализации процесса состоит в том, чтобы контролировать поле заголовка HTTP «Закат» во время конфигурации. Как только конфигурация заканчивается, поле заголовка перестает демонстрироваться в ответах. Когда становится известно о предстоящем списании, добавляется данная конфигурация и поле заголовка появляется на ресурсах, определенных для этого API (обычно на домашнем ресурсе).
- *«Как?» (руководство по реализации № 2 для руководства по дизайну № 1)*. Еще один метод реализации состоит в том, чтобы добавить поле заголовка HTTP «Закат» с помощью шлюза API. Вместо самой реализации API поле заголовка добавляется шлюзом API, как только методика сконфигурирована и запущена. После этого поле заголовка появляется на ресурсах, определенных для API (обычно на домашнем ресурсе).
- *«Что?» (руководство по дизайну № 2)*. API, у которых есть зарегистрированные пользователи, могут использовать для связи с ними внешние каналы, чтобы объявить о предстоящем списании. Это руководство применимо, только если у вас есть такой список и канал для связи считается достаточно надежным.

- *«Как?» (руководство по реализации № 1 для руководства по дизайну № 2).* Один из методов реализации состоит в том, чтобы связаться со всеми зарегистрированными пользователями API по электронной почте. Она идеальна для ссылок на доступный ресурс (журнал изменений API или часть его документации), содержащий информацию о деактивации. У этого ресурса должен быть стабильный URI, чтобы на него можно было ссылаться при общении с пользователями.

Многие организации, в стратегии которых API играет важную роль, создают руководство по API в какой-либо форме. Некоторые руководства публикуются открыто, что позволяет свободно их просматривать. Например, можно увидеть, что в качестве руководства используют такие организации, как Google, Microsoft, Cisco, Red Hat, PayPal, Adidas или Белый дом.

Гид по стилям API

Арно Лоре, известный как Мастер API, собрал некоторое количество опубликованных руководств в своем «Гиде по стилям API» (http://apistylebook.com/), пользуясь информацией от организаций, в основном крупных, от Microsoft до Белого дома. Это интересный для исследования ресурс с точки зрения того, что именно большие организации пишут в руководствах по API.

Способ публикации этих руководств в открытом доступе многое может рассказать об их создании и управлении ими (и об общей философии, стоящей за руководствами и управлением ими), даже если вы на них еще не взглянули.

- Документы в PDF явно предназначены только для чтения. PDF публикуется из некоего невидимого источника. Это просто способ собрать, отформатировать и опубликовать уже существующее содержимое. В этом случае вы почти не чувствуете вовлеченности в управление этим руководством и его развитие.
- HTML обычно воспринимается чуть лучше, потому что в большинстве случаев опубликованный HTML и является источником, то есть читатели видят сам источник документа, а не отформатированный и оторванный от него продукт, как в случае с PDF. Но управление HTML-источником и его изменение все еще неочевидны, поэтому читатель осознает, что он отделен от стадий создания и редактирования.
- Многие системы контроля версий имеют функцию публикации, поэтому их можно использовать для размещения руководств. Например, у GitHub есть простые встроенные способы форматирования и публикации (в самом простом случае — файлы Markdown), хотя, скорее всего, некоторых важных функций форматирования вы там не найдете[1]. Преимущество GitHub заключается

[1] Содержание в формате Markdown создается напрямую в веб-окне репозитория. Более амбициозные писатели выбирают текстовый процессор Pages, в котором можно создать сайт прямо из репозитория.

в простоте применения функций комментирования, сообщения о проблемах и предложения изменений. Кроме того, большинство разработчиков уже привыкли к GitHub, поэтому им удобно его задействовать и не требуется обучение.

Существует еще одно дополнительное правило для создания руководства, необходимо тестировать разработчиков на его знание (то есть в нем должны быть инструменты, помогающие разработчикам определить, разобрались ли они в руководстве). Это не только делает его более открытым и очевидным для использования, но и означает, что можно тестировать автоматически. Полностью автоматизированное тестирование по всему руководству может не оправдать вложений в него или оказаться невозможным, но к нему как минимум надо стремиться, поэтому мы предлагаем добавить к более типичному шаблону «Зачем? Что? Как?», который описали ранее, четвертый элемент.

«Когда?» Описывает момент, когда можно сказать, что необходимая работа была выполнена. Для этого нужен способ протестировать, действительно ли она выполнена правильно, и желательно автоматизированный тест в процессе развертывания, который будет контролировать выполнение работы и следование руководству.

Как и все в хорошо управляемой системе API, тесты могут со временем улучшаться. Можно начать с простых тестов на достоверность, чтобы обеспечить минимальную уверенность и положительную обратную связь, говорящую о том, что к руководству обращались. Если со временем становится понятно, что это не так полезно, как должно быть, тогда необходимо улучшить тесты, чтобы команды получили более полезную обратную связь и им стало легче проверять соответствие конкретному руководству.

Жизненный цикл руководства в системе API

Поскольку руководство — это развивающийся набор рекомендаций, у него также есть жизненный цикл: сначала появляются предложения, какое-то время они исследуются, затем могут стать рекомендациями по тому, что и как надо делать. Но, как любые элементы системы, со временем их могут заменить новыми и альтернативными способами выполнения задач, тогда эти рекомендации пройдут через фазу заката и в итоге станут устаревшими. Стадии жизненного цикла руководства в системе API можно определить следующим образом.

- *Экспериментальная стадия.* В этой фазе руководство исследуется, то есть используется как минимум в одном продукте с API, чтобы лучше понять, имеет ли оно смысл в качестве руководства на системном уровне. В этот момент оно документируется, но вложений, чтобы было проще ему следовать, не делается.
- *Реализация.* Когда руководство утверждено на системном уровне, его следует поддерживать (чтобы существовал хотя бы один ответ на вопрос «Как это сде-

лать?») и как минимум принимать во внимание, прежде чем отбросить. Некоторые руководства отбрасывать нельзя, поэтому команды обязаны им следовать.

- *Увядание*. Как только появляются новые или более удачные способы выполнения определенной задачи, руководство может войти в период увядания, когда его еще можно придерживаться, но в идеале команды должны рассмотреть вариант следования тому руководству, которое находится в стадии реализации.
- *Устаревание*. В итоге руководство может устареть, и его больше не нужно применять к новым продуктам. Можно также перенастроить существующие продукты на более современные способы выполнения задач. Устаревшее руководство все еще полезно сохранять в исторических целях и для документирования того, как разрабатывались и использовались более старые продукты.

Эти стадии — всего лишь один из методов управления развитием руководства, и вы свободно можете определить для себя другой метод. Кроме того, у вас могут быть способы маркировать его разными уровнями требований, например «опциональное» или «обязательное». Но в ходе работы с подобными уровнями должны создаваться и исключения, чтобы, к примеру, обязательное руководство можно было пропустить, если имеется достаточно доказательств, что следование ему вызовет проблемы.

Важный вывод из этой темы: необходимо принять тот факт, что руководство будет непрерывно меняться, создать способ отслеживания этих изменений и управления ими в своей организации. Это мы обсудим в следующем разделе, где познакомим вас с широко применяемым большими организациями способом решать проблему управления руководствами.

Центр подготовки

Разные организации по-разному называют команды, которые составляют руководства по API и управляют ими, а также в целом помогают другим командам, реализующим стратегию API. Одно из распространенных названий — «центр компетенции», но для многих оно несет в себе отрицательные коннотации, потому что в таком случае предполагается, что все, кто в нем не состоит, некомпетентны. Поэтому нам больше нравится название «центр подготовки» (Center for Enablement, C4E), которое также хорошо отражает меняющуюся роль современных IT-команд.

Хотя управление руководствами может показаться всего лишь технической деталью, на практике оно крайне важно. C4E должны собирать и редактировать информацию, которая будет опубликована, а отдельные команды по API — предоставлять ее. Центр подготовки также несет ответственность за определение того, в какие аспекты руководства необходимо инвестировать для поддержания инфраструктуры, чтобы проблема, которой прежде занимались целые команды, была решена с помощью доступных инструментальных средств.

Другая часть обязанностей C4E — убедиться, что следование руководству по API не создает никаких узких мест. Идеальная картина выглядит так: команды разобрались в руководстве, у них есть модель реализации обновлений, они достаточно компетентны и имеют поддержку от C4E для использования инструментов, поэтому следование руководству по API их работу не замедляет. Все узкие места должны быть найдены, и с ними нужно разобраться, чтобы реализовывать API практически беспрепятственно.

Разумеется, все это зависит от ограничений, действующих в организации. Например, в некоторых сферах существуют внутренние инструкции или юридические нормы, требующие, чтобы организации проверяли и утверждали каждый релиз. Делать это необходимо, и данные процессы нельзя полностью автоматизировать. Но это скорее исключения, чем правила, поэтому большинство руководств действительно нужно воспринимать как то, чему *стоит* следовать, и C4E должен обеспечить, чтобы команды могли успешно это делать.

УПРАВЛЕНИЕ ИНЖЕНЕРАМИ: CHAOS MONKEY

Еще одна интересная обязанность C4E состоит в том, чтобы определять способы, которыми нефункциональные требования могут быть перенесены в общую культуру разработки, принятую в вашей организации. Одним из примеров служит популярный инструмент Chaos Monkey («Обезьяна хаоса») компании Netflix. Его история такова: согласно общей методике разработки сервисы в такой среде, как сложная и взаимозависимая система API Netflix, должны быть максимально устойчивыми, чтобы проблемы отдельных сервисов затрагивали как можно меньше зависимых от них сервисов.

Проблема обязательного «кода устойчивости» заключается в том, что его тяжело протестировать. Netflix решила ее с помощью гениального Chaos Monkey — инструмента, симулирующего изолированные и контролируемые проблемы в инфраструктуре и наблюдающего за поведением других сервисов в этот момент. Это позволяет инженерам контролировать проверку устойчивости. Данная ситуация является примером подхода, который мы называем «управление инженерами»: создав инструмент, определяющий неустойчивый код, менеджеры системы добиваются от инженеров большей дисциплины при разработке кода. Если он не будет устойчивым, тестирование обнаружит проблемы до того, как они станут критическими, поэтому у разработчиков есть дополнительная возможность тестирования при запуске.

С таким подходом C4E проще расширять систему API по мере разработки и развертывания новых API, а отдельным командам — понимать, какие существуют требования, и получать (хотя бы частично) автоматизированные тесты на их знание. Конечно, для части руководства все равно потребуются проверки и обсуждения, но чем легче сфокусироваться на этих аспектах, не подлежащих тестированию, тем проще C4E будет помочь командам, которым это нужно.

В целом C4E должны обслуживать руководства на системном уровне. У них двойная цель: максимально облегчить командам по API *создание новых продуктов*, а пользователям API — *использование API во всей системе*. Поскольку в обязанности C4E входит поддержание баланса между простотой создания и простотой применения, его самые важные задачи — постоянно собирать отзывы создателей и пользователей и понять, как постоянно развивать систему API для блага обеих групп.

Постоянное развитие системы API означает, что необходимо помнить об аспектах системы, описанных в разделе «Восемь аспектов систем API» главы 8. Это также означает, что C4E приходится решать, когда в какие аспекты инвестировать, наблюдая, по какому из них можно не предоставлять высокотехнологичной поддержки, а на какой стоит потратить время и силы. Например, в случае аспекта «Объем» некоторое время можно не тратить слишком много сил на расширение системы до сотен или тысяч API, но, когда все больше и больше команд создают и используют API, расширение объема становится крайне важным и требует более крупных вложений.

Главная цель C4E — помогать командам по API более эффективно участвовать в создании системы API. Мы рассматривали команды в главе 7. Дополняя это обсуждение, в следующем разделе поговорим о командах C4E и о том, как с помощью управления отдельными API и системой API создать команду, максимально полезно поддерживающую команды по API.

После обсуждения команды C4E и того, что ее окружает, перейдем к более тщательному обзору ответственности этой команды. В разделе «Развитие и восемь аспектов» мы последовательно рассмотрим обращение с различными аспектами системы как с частью стратегии API и влияние инвестиций в эти аспекты на управление API в организации. Инвестиции могут представлять собой предоставление инструментов, процессов или стандартов, но во всех этих случаях главный параметр успеха — это то, насколько они улучшают систему API с точки зрения создателей и пользователей API.

Команда C4E и общая обстановка. Команда C4E должна обслуживать руководства всей системы и помогать командам четко исполнять их требования. Взаимодействуя с командами продуктов с API, C4E собирает обратную связь о том, какие новые шаблоны могут появиться, и узнает, как должны развиваться принципы, протоколы и шаблоны, чтобы улучшить систему API.

Для этого C4E должен развиваться вместе с системой. Возможно, изначально это будет даже не отдельная команда, а члены разных команд продуктов с API, которые возьмут на себя описанные здесь обязанности. Однако со временем, скорее всего, в больших организациях, которые делают значительные инвестиции в API, центр подготовки превратится в настоящую команду с собственными сотрудниками. И даже тогда важно помнить, что ее основная обязанность — поддерживать команды продуктов. Кевин Хикки пишет об этом так: «Теперь вы не централизованная группа [по архитектуре предприятия], принимающая решения за команды по

разработке, вы ответственны за влияние и сбор информации. Вы теперь должны не делать выбор, а помогать другим сделать правильный выбор и затем распространить эту информацию» (http://bit.ly/2DBOTJO).

Обязанности, которые мы определили в разделе «Обязанности в команде по API» главы 7, во многих случаях относятся и к C4E или как минимум значительно влияют на их действия на системном уровне. Появляются и новые обязанности, которых обычно нет на уровне команды.

Например, обязанности, связанные с соответствием. Во многих организациях необходимо убедиться, что организация соответствует нормам и законодательству, отслеживать изменения в требованиях и обеспечивать соответствие этим изменениям. В случае системы API это часто относится к существующему руководству, которого обязательно придерживаться, как говорилось ранее в этом разделе. Чтобы избежать появления узких мест, в идеале должна существовать возможность тестирования команд по API на соответствие, поэтому она может стать частью процесса создания. На практике это не всегда возможно, а иногда даже запрещено (может существовать правило о том, что кто-то должен утвердить продукт после ознакомления с ним). Но при любых требованиях организации важно думать о соответствии с точки зрения API, определить области, где следует выпустить руководство по соответствию, и поддерживать команды API, чтобы максимально облегчить им реализацию продуктов, соответствующих всем требованиям.

Еще одна обязанность, обычно существующая только на системном уровне, — предоставление инфраструктуры и инструментов. Как показано в разделе «Структурирование руководства в системе API» ранее в данной главе, типичное руководство в системе API разделено на «Зачем?», «Что?» и «Как?». Мы считаем, что каждому «Зачем?» (мотивация руководства) должны соответствовать как минимум одно «Что?» (руководство по дизайну) и одно «Как?» (руководство по реализации), а также возможная инфраструктура и инструменты для тестирования, которые помогут командам легко проверять соответствие продукта существующему руководству. При каждом тесте реализации должна существовать обязанность на системном уровне помогать командам как можно более эффективно выполнять указания и подтверждать соответствие руководству. Это может означать предоставление инструментов и/или инфраструктуры для этого. Тогда важной обязанностью на системном уровне становятся создание и поддержка этих инструментов или инфраструктуры. Чем лучше помощь и инструменты, которые команды получают на системном уровне, тем больше команд может сосредоточиться на решении задач вашего бизнеса и самого продукта, а не на попытках встроиться в систему. Любые препятствия, с которыми они сталкиваются, должны рассматриваться как важный сигнал о том, что где-то требуются более серьезная помощь и больше инструментов.

В общем, команда C4E очень важна для поддержки команд продуктов с API. Она сообщает им о решениях, необходимых для создания руководства по основным мо-

ментам, требующим принятия решений, и помогает, предоставляя инфраструктуру и инструменты для эффективного решения распространенных задач в сфере API, чтобы бóльшая часть энергии команд была потрачена на решение задач бизнеса. Другими словами, команда C4E несет ответственность за то, чтобы каждая из команд продуктов с API могла эффективно развивать свой API и при этом принимать правильные решения, а также за то, чтобы такая эффективность распространялась на множество API.

Развитие и восемь аспектов

Восемь аспектов систем API, представленные в главе 8, не очень важны, когда системы API находятся на стадии планирования и эволюции. Они также могут служить руководствами в момент определения уровня развития в этих сферах, когда нужно отразить мотивы и преимущества развития и принять решение по поводу возможных инвестиций в данные сферы.

При *непрерывной разработке архитектуры* важно понимать, что инвестиции в эти сферы должны быть эволюционными и соответствовать конкретным потребностям системы API. Если архитектура разработана качественно, инвестиции можно производить по необходимости и постепенно и это не потребует повторной разработки архитектуры системы. Если она соответствует принципам непрерывной разработки, значит, само развитие системы API постоянно эволюционирует, ориентируя улучшения так, как надо, и система постоянно улучшается на основе отзывов разработчиков и пользователей.

Как и эволюция, это постоянное улучшение не ведет к какой-либо определенной или даже предсказуемой цели. Ценность системы определяется тем, насколько хорошо она поддерживает разрабатываемые в ней продукты и насколько они отвечают требованиям пользователей. И методы разработки, и потребности пользователей со временем меняются, поэтому улучшение неизбежно становится непрерывным процессом и никогда не войдет в какую-то идеальную конечную стадию.

Главная цель архитектуры системы — максимально упростить этот непрерывный процесс, позволяя ей адаптироваться под меняющиеся потребности создателей и пользователей. Развитие системы можно измерить объемом поддержки, которую она предоставляет. В случае определенных нами восьми аспектов можно отдельно посмотреть на то, как для каждого из них определяется развитие и выглядит стратегия управления развитием.

Поэтому развитие в данном контексте не означает стабильного конечного состояния. Оно означает, что аспект системы управляется таким образом, что он обеспечивает лучшую поддержку создателям и пользователям API, учитывая их текущие требованиям. Это означает также постоянное улучшение поддержки, основанное на меняющихся требованиях.

Такая идея развивающейся системы несколько отличается от цикла развития продуктов с API, который мы обсуждали в главе 6. Продукты приходят и уходят, и это происходит во время их собственного жизненного цикла, у которого есть начало и конец. Система существует, чтобы поддерживать продукты, и должна это делать с помощью непрерывного развития. Здесь нет одного линейного пути или конечного состояния, следовательно, нет и стадий. И мы рассмотрели это, изучив, как определенные нами восемь аспектов в разные моменты могут служить принципами для руководства при непрерывном улучшении системы, разработке ее стратегии и принятии решений по поводу инвестиций на системном уровне.

Разнообразие

Как сказано в подразделе «Разнообразие» раздела «Восемь аспектов систем API» главы 8, разнообразие в системе зависит от количества ограничений при разработке и реализации API и объема свободы, которой обладают команды при разработке продуктов API, которые они считают качественными решениями поставленных задач.

С разнообразием работать непросто, потому что в системах оно всегда представляет собой баланс между продвижением определенного уровня согласованности и повторного использования и попыткой не слишком ограничивать команды и не заставлять их применять неподходящие решения задачи. Поэтому у разнообразия есть две «плохие» крайности.

- *Отсутствие разнообразия* означает, что выбранный шаблон становится «золотым молотком» из поговорки (http://bit.ly/2B5bEna) — единственным путем решения всех проблем[1] (что часто оказывается неподходящим решением как минимум для нескольких из них).
- *Избыток разнообразия* приводит к эффекту «прекрасных снежинок» (явление, когда каждый считает себя уникальным и хочет выделиться) — команды тратят силы на решение проблем, уже имеющих адекватные решения, и в результате пользователи страдают, пытаясь продуктивно сочетать API, потому что в их дизайне нет согласованности.

Соблюсти баланс между ними непросто, и единственно правильного решения для выбора точки на шкале между «золотым молотком» и «прекрасными снежинками» не существует. Поэтому неправильно характеризовать развитие разнообразия в системе API тем, велико оно или мало. Во многих случаях оно может сложиться случайно из-за отсутствия гибкости (если низкое) или невозможности управлять согласованностью (если слишком высокое).

[1] Известное высказывание американского психолога Абрахама Маслоу гласит: «Думаю, соблазнительно, имея из инструментов только молоток, обращаться со всеми предметами, как с гвоздями».

РАЗВИТИЕ РАЗНООБРАЗИЯ

- *Развитие* разнообразия означает, что им в системе API управляют сознательно. Используемые в данный момент варианты и их обоснование четко документированы.
- Эти варианты при необходимости должны эволюционировать: расширение разнообразия — управляемый процесс, связанный с балансом между повторным применением и созданием новых решений, если существующие не соответствуют требованиям.
- Увеличить разнообразие можно без вреда для системы. Возможно, некоторые инструменты и поддержку системы придется настроить, чтобы они справлялись с новыми схемами дизайна, но вся инфраструктура должна быть разработана так, чтобы можно было постепенно увеличивать разнообразие. Инфраструктура должна стать частью базовой архитектуры.

Разнообразие может относиться ко многим понятиям в системе API в зависимости от ее организации. Например, для систем, основанных на HTTP и использующих API в стиле URI, одним из факторов разнообразия может быть выбор способа присвоения серийных номеров. Поскольку бóльшая часть современных систем API, вероятно, задействуют и позволяют API поддерживать XML и JSON (или HAL, основанный на JSON, как формат более высокого уровня), это самые популярные варианты настоящего и недавнего прошлого.

Вполне вероятно, что благодаря дизайнерам API появляются новые способы присвоения номеров — например, стандарт схемы описания источников (Resource Description Framework, RDF). Вопрос должен заключаться в том, считать ли новый формат ценным дополнением к системе. У вас должна быть возможность начать с нескольких API и проверить, как они будут работать с новым вариантом. Возможно, они не смогут получить доход от существующих инструментов и поддержки, поскольку использование нового формата экспериментально (на системном уровне на этой стадии нет инвестиций).

Когда новый вариант начинает считаться продуктивным, он может вызвать обновления доступных инструментов и поддержки. Высокоразвитые системы способны выполнить эти обновления как постепенные изменения, добавляемые при необходимости. Таким образом, расширение разнообразия — всего лишь способ воспользоваться полезными функциями добавленного варианта и постепенный рост затрат на обновление инструментов и поддержки.

Самое важное последствие такой точки зрения заключается в том, что все инструменты и поддержка должны быть готовы к подобным обновлениям. Те из них, что не способны справиться с растущим разнообразием, ограничивают доход, который обновление может принести системе. Такие инструменты и поддержка считаются способными вызвать проблемы с точки зрения системы API.

Важное последствие стратегии разнообразия — это взгляд на функции инструментов и поддержки API. Как говорилось в подразделе «Выражать API через API» раздела «Управление API в больших масштабах» главы 8, когда все делается с помощью API, включая взаимодействие инструментов и поддержки, расширять разнообразие становится проще. Если новые варианты поддерживают те же API, они все еще могут взаимодействовать с существующей инфраструктурой инструментов и поддержки.

СТРАТЕГИЯ РАЗВИТИЯ РАЗНООБРАЗИЯ

Инвестируя в инструменты и поддержку, всегда помните о том, что произойдет с инвестициями при увеличении разнообразия. Старайтесь избегать инструментов и поддержки, не имеющих четкой стратегии эволюции. В конце концов, инструменты и поддержка *должны подстраиваться под ваш выбор самого продуктивного уровня разнообразия системы*, а не диктовать его.

Словарь

Как говорилось в подразделе «Словарь» раздела «Восемь аспектов систем API» главы 8, многие API используют терминологию, определяющую некоторые аспекты модели API. Терминология может вступить в игру по-разному, и во многих случаях при первоначальном релизе API применяется один словарь, но нужно быть готовыми к тому, что со временем он изменится. В таком случае он становится частью модели расширения этого API, и вопрос состоит в том, как разрабатывается это расширение и как им управляют.

То, что словари, используемые в системе API, действительно эволюционируют, — это результат того, что модели предметных областей API (или охват предметных областей) имеют тенденцию со временем развиваться. И эволюция словаря в API просто является отражением этого процесса. Они часто эволюционируют из-за того, что уточняется понимание предметной области, например, если добавляются ссылки на социальные сети в пользовательскую модель, содержавшую до этого лишь основную личную информацию. Тогда вопрос будет заключаться в том, как работать с данными (существующими пользовательскими записями без ссылок на соцсети) и кодом (приложениями без встроенной поддержки соцсетей), созданными до развития пользовательской модели. Дисциплинированное управление такой эволюцией словаря определяет развитие работы со словарем в системе API.

Концепции терминов

В подразделе «Словарь» на с. 188 обсуждается, какая терминология может использоваться для API, включая независимую от предметной сферы (например, коды языков), специфичную для предметной сферы (часть сферы, отраженную в API) и саму сферу концепций для дизайна API (например, распространенные коды статусов HTTP).

РАЗВИТИЕ СЛОВАРЯ

Базовая стартовая точка для отдельного API — *определение потенциальных словарей, с которыми API может развиваться*. Это напрямую связано с *определением точек расширения API*: если команда по API ожидает, что словарь будет развиваться, она должна определить это в самом API и предоставить пользователям модель обработки.

Когда эволюция словаря становится естественной частью API, приобретает значение ответственное управление ею. С одной стороны, это означает *ответственный контроль версий со стороны API и их документирование*, с другой — помощь клиентам в применении API таким образом, чтобы правильно поддерживать эволюцию. Каким именно, зависит в основном от того, как отдельные API решают реализовать эволюцию словаря.

Управление эволюцией словаря может быть передано отдельным API. Однако существует и альтернативная модель, в которой система API позволяет словарям развиваться независимо от API. Типичная схема включает в себя *использование реестров* (http://bit.ly/2Fo1qCy), а поддержка реестров и управление ими сами по себе могут стать частью системы API.

Последний аспект развития требует дополнительных объяснений. Есть два способа делегирования эволюции словаря (то есть управления ею вне API): с помощью ссылки на внешний орган власти, ответственный за управление словарем, и с помощью управления им внутри системы API, но отделяя API от развивающихся словарей.

- *Внешний орган власти*. Распространенным примером этого подхода служит использование языковых тегов, то есть названий человеческих языков, например «английский» или даже «американский английский». В большинстве случаев, вероятно, не стоит включать в API статический список языков, поскольку они развиваются со временем. Вместо этого имеет смысл дать ссылку на один из списков Международной организации по стандартизации (International Organization for Standardization, ISO), приведенный в стандарте ISO 639 (http://1.usa.gov/14iSOkR). Реализуя эту схему, API может определить, что пространство значений для языковых тегов включает в себя то, что ISO определяет как возможные языковые теги в конкретный момент времени. В этом случае ISO гарантирует, что теги будут развиваться, не вредя вашей системе, так как вам не понадобится удалять или менять определения существующих тегов.
- *Поддержка системы API*. Не у всех концепций есть внешние сущности и менеджеры, такие как ISO для списка языковых тегов, но вы можете использовать ту же схему и для других словарей. Системы API могут поддерживать реестры, позволяя API разъединить определение API и развивающееся пространство значений терминов. Работа с подобным реестром не является сверхсложной

задачей, тем не менее она не должна быть зоной ответственности отдельной команды по API[1]. Вместо этого на системном уровне должна существовать поддержка реестров. Таким образом, например, Инженерный совет Интернета (Internet Engineering Task Force, IETF) управляет своими спецификациями с помощью более чем 2000 реестров, находящихся в ведении Агентства по выделению имен и уникальных параметров протоколов Интернета (Internet Assigned Numbers Authority, IANA)[2].

Роль архитектуры в управлении словарем такая же, как и в других аспектах создания продуктивной и поддерживающей среды для API: следить за требованиями и методиками существующих API и предлагать удачные методы и поддержку, когда эволюция словаря (или отсутствие ответственной модели управления ею) начинает повторяться во многих API.

Изначальный качественный метод должен заключаться в том, чтобы как минимум определить потенциально способные к развитию словари в API и документировать их, что также является частью качественного метода общего расширения. Если в эволюции API возникают повторяющиеся случаи в качестве неожиданных последствий эволюции словаря, это может означать, что системная поддержка эволюции словаря может помочь уменьшить необходимость в обновлениях API.

Может оказаться труднее достичь развития в аспекте терминологии, потому что непросто придумать, как наблюдать за использованием словаря в API. Это может быть одним из тех случаев, когда инвестиции на раннем этапе помогают улучшить наблюдаемость. Например, с помощью создания инструментов для документации словаря реально облегчить наблюдение за его применением и эволюцией в нескольких API. Но это предполагает, что команды по API сочтут такую поддержку с помощью документации достаточно полезной для того, чтобы ее использовать, что, в свою очередь, может потребовать наблюдений за тем, как команды обычно документируют свои API.

Как видите, развитие часто не сводится к поддержке и инструментам на системном уровне. Оно может начинаться с понимания того, за чем следует наблюдать, и последующей разработки способов наблюдения.

---

[1] В конце концов, один из главных мотивов создания реестра — разъединение управления списком известных значений и мест их использования.

[2] Модель реестров IANA очень хорошо иллюстрирует простоту и эффективность подобной инфраструктуры. В то же время это прекрасная демонстрация того, как, применяя такую схему дизайна систематически для многих спецификаций и определений API, можно стабилизировать эти спецификации, поскольку при большом количестве изменений необходимо обновить только реестры.

СТРАТЕГИЯ РАЗВИТИЯ СЛОВАРЯ

Продвигайте полезные методы, разъединяющие *дизайн API и эволюцию словаря*, если это возможно. Начните с продвижения *повторного использования словарей, определенных и управляемых извне*, например, организациями, определяющими стандарты. Отслеживайте, сколько изменений API могут быть вдохновлены (в основном) необходимостью обновлять терминологию, и обдумайте обеспечение поддержки для *управления словарем в системе API независимо от конкретных API*, предоставив для этого инфраструктуру.

Объем

В подразделе «Объем» раздела «Восемь аспектов систем API» главы 8 предполагалось, что больше API — лучше, чем меньше. Конечно, это не всегда верно, но намекает на то, что решения по количеству API в системе не должны зависеть от предположений о том, что система просто не справится с объемом. Большой объем не обязательно лучше, но точно не нужно автоматически считать, что он хуже.

Вместо этого следует стремиться к тому, чтобы всегда позволять создавать, изменять и удалять API и принимать решения о том, как это влияет на общую ценность системы API, основываясь на уровне сложности и динамике, которые, по вашему мнению, принесут больший доход. Система API должна иметь возможность расширяться до любого уровня, и ее способность справиться с объемом в идеале не должна влиять на стратегические решения по поводу размера и уровня изменений.

Управление объемом в системе API в основном вращается вокруг экономических мотивов расширения. То, что логично будет не поддерживать или не автоматизировать при небольших масштабах, может стать мишенью вложений, когда система начнет расти. Это обычная схема получения дохода от инвестиций: вкладывать в поддержку или автоматизацию имеет смысл после преодоления определенного порога, когда расходы на решение проблем по отдельности (снова и снова) становятся выше, чем расходы на поддержку и автоматизацию.

Ненадолго вернемся к аналогии с Интернетом — именно поэтому создаются и меняются со временем фреймворки создания скриптов. Они разрабатываются для конкретных распространенных шаблонов и приобретают популярность благодаря популярности этих шаблонов. Например, такие фреймворки, как Ruby on Rails (с серверной стороны) и jQuery (с клиентской стороны) были в зените славы, когда использовались более традиционные веб-приложения. С появлением одностраничного шаблона приложений распространились такие фреймворки, как React и Angular, и связанные с ними принципы протокола, такие как GraphQL. Популярность и доступность этих фреймворков зависит от популярности их основных шаблонов, а итоговые усилия направляются на создание и поддержание самих фреймворков.

Эта аналогия подчеркивает, что как только объем начинает влиять на какие-то виды поддержки или автоматизации, в системе происходит некоторая стандартизация, поскольку все больше API начинают применять эти поддерживаемые механизмы. От этого они становятся более похожими, тем самым помогая пользователям системы понимать API и работать с ними, потому что они решают определенные проблемы определенным способом.

Однако, как упоминалось в разделе «Центр подготовки» ранее, важно помнить, что поддержка или автоматизация (ответ на вопрос «Как?») никогда не должны быть единственным разрешенным способом выполнения задачи. C4E должен определить и предоставить это как часть общей поддержки платформы API, но всегда должно быть что-то, что можно заменить более подходящим способом решения той же проблемы, если было найдено лучшее решение.

И вновь проиллюстрировать сказанное можно примером с Интернетом. Может произойти так, что в течение некоторого времени создание веб-приложений будет сосредоточено на использовании общего одностраничного шаблона, упомянутого ранее. Этот ответ на вопрос «Что?», однако, будет отдельным руководством по отношению к возможному руководству по реализации данного шаблона, включающему фреймворки React или Angular или другие, которые их заменят, потому что будут предоставлять более качественную поддержку при создании одностраничных приложений. Чем лучше руководство разделяет «Что?» и «Как?», тем сильнее оно поддерживает эволюцию как на уровне шаблонов, так и на уровне реализации последних.

Это снова отсылает нас к концепции непрерывной разработки архитектуры: работа с объемом путем стратегических вложений в поддержку или автоматизацию всегда должна считаться эволюционным изменением системы API. Так происходит потому, что она обеспечивает высокий доход от инвестиций в эту конкретную проблему в данный конкретный момент, но все может измениться при дальнейшем увеличении объема, когда появятся более удачные решения.

РАЗВИТИЕ ОБЪЕМА

- Отслеживайте, как команды по API решают проблемы, связанные с разработкой и созданием своих продуктов, а также управлением ими, и обдумайте *инвестиции в поддержку или автоматизацию* при необходимости, то есть когда это становится полезно с точки зрения дохода от инвестиций.
- Для потенциальной поддержки или автоматизации рассмотрите потенциальный доход и для *команд, создающих API*, и для *его пользователей*. Общая прибыль от ввода поддержки или автоматизации будет складываться из суммы этих двух доходов.
- Самое важное с точки зрения системы API — определить повторяющиеся действия команд в процессе дизайна или реализации и исследовать возможность повысить продуктивность с помощью вложений в поддержку или автоматизацию.

Как мы говорили ранее, важнейший аспект развития объема заключается в следующем: не допускать, чтобы он мешал принятию решения по поводу того, позволять ли расти системе API. Лучше всего сделать это с помощью контроля текущей эволюции системы API, отслеживания того, что реализуют команды и как они это делают, и инвестиций, когда вам кажется, что в данный момент поддержка или автоматизация улучшат продуктивность команд.

Этот подход подразумевает, что система API активно отслеживается и, таким образом, эти решения принимают, основываясь на полученных данных. Хорошая схема, позволяющая масштабировать этот подход, состоит в том, чтобы придерживаться принципа «выражение API через API», описанного в предыдущей главе, и убедиться, что сами API предоставляют информацию о себе. Таким образом, станет возможно встраивать поддержку и автоматизацию в мониторинг API, и это поможет решить, когда нужно инвестировать в поддержку и автоматизацию дизайна и разработки API.

Хороший способ оценивать развитие аспекта объема — следить за тем, какая информация о системе всегда доступна тем, кто к ней обращается. Помните о том, что эту информацию можно собирать любым способом, лишь бы она была доступной. Ее можно предоставить через сами API (выражать API через API), через инструментарий управления работающей инфраструктурой (например, собирать данные из шлюзов API) или инструментарий управления инфраструктурой времен разработки этого API (например, собирать данные с общих платформ для разработки/развертывания продуктов с API). Если информация доступна, становится проще понимать динамику развития системы и управлять ею.

СТРАТЕГИЯ РАЗВИТИЯ ОБЪЕМА

Работа с объемом требует основы, которую можно применять для масштабируемого *наблюдения* за API и таким образом понимать динамику развития системы API. Наблюдаемость должна включать в себя информацию об API, которую можно использовать для принятия *решений об инвестициях*, основываясь на тенденциях в системе. Само управление объемом может быть масштабировано для работы с большими объемами, если необходимая для понимания системы информация является частью самих API. Подход «выражать API через API» со временем будет развиваться, меняя содержание наблюдаемой информации, которую применяют, чтобы понять текущее развитие системы API.

Скорость

Как говорилось в подразделе «Скорость» раздела «Восемь аспектов систем API» главы 8, *скорость* относится к тому факту, что системы API меняются постоянно и довольно высокими темпами. С одной стороны, это результат ориентации на API, из-за чего создается и используется все больше и больше API, но с другой — это

происходит также из-за отношения к API как к продуктам, когда за ними наблюдают и меняют их в соответствии с отзывами и запросами пользователей. Эти изменения в большинстве случаев происходят нескоординированно, поскольку одна из целей систем API — позволять отдельным сервисам развиваться независимо друг от друга, а не создавать сложные скоординированные процессы релизов, которые позволяют им развиваться только в зависимости друг от друга.

Высокоразвитая работа со скоростью означает, что релизы и обновления API происходят, когда в этом возникает необходимость, и система API способна поддерживать высокий уровень изменений. Развитие по этой оси должно происходить само собой, так же как и по другим осям. Изначально система API может быть довольно маленькой, чтобы даже при относительно большом числе изменений их количество за день оставалось разумным, однако со временем это изменится. В частности, сочетание увеличения объема (которое мы обсуждали в подразделе «Объем» на с. 192) и скорости изменений API означает, что работа со скоростью становится все важнее, поскольку система API растет и развивается.

РАЗВИТИЕ СКОРОСТИ

- API всегда должны быть *разработаны так, чтобы можно было вносить в них изменения*. В зависимости от стиля API это может иметь разные значения, но узнать у команд об их планах по расширению API — прекрасный первый шаг к тому, чтобы взгляд на *эволюцию API как на естественную часть жизненного цикла API* стал частью культуры дизайна API.
- Развитие API меняет и подход пользователей: применение API должно стать достаточно устойчивым для поддержки эволюции API, чтобы эволюции API и его пользователей были разъединены.
- *Увеличить скорость изменений API* может помочь применение *микросервисов или хотя бы DevOps* в качестве способа реализации сервисов. Используя подобные подходы, можно вносить изменения, *основываясь на отзывах и требованиях*, и не замедляться из-за взаимосвязей и координации между процессами реализации и развертывания.

Эти идеи показывают, что скорость влияет на создателей так же, как и на пользователей. Поэтому с ростом размеров и популярности системы API (и количества пользователей) высокоразвитая работа со скоростью становится все важнее. Координация обновлений сервиса между самим сервисом и всеми пользователями становится все дороже и быстро достигает той точки, где затраты на координацию становятся запредельными.

Как говорилось в подразделе «Контроль версий» на с. 197, существуют различные стратегии работы с меняющимися API. Они могут меняться прозрачно, и пользователи просто увидят изменения, поскольку это будет понятно благодаря семантическому номеру версии API (см. пункт «Семантический контроль версий» на с. 197). Другой вариант — API обещает стабильные версии, и тогда скорость

относится к текущему потоку новых версий, выпускаемых и доступных параллельно. Последняя схема предполагает, что пользователям будет легко узнать о новых версиях и найти о них информацию, что более подробно обсуждалось в подразделе «Видимость» на с. 195.

Поддерживать скорость довольно важно, не менее важно управлять ею. Если она увеличивается, пользователи должны иметь возможность успевать за ней. Это можно сделать по-разному, например пообещать, что API стабильны и будут работать в течение определенного времени[1] или что они станут постоянно эволюционировать и поэтому старые версии не нужно поддерживать в рабочем состоянии[2]. По какой бы схеме ни развивалась ваша система, в этой области отдельные API могут много выиграть от поддержки на системном уровне, поэтому выбор этих методик и их поддержка довольно быстро становятся ценной инвестицией.

СТРАТЕГИЯ РАЗВИТИЯ СКОРОСТИ

Улучшение гибкости, то есть способности к быстрому изменению, основанному на отзывах и требованиях, — один из главных движущих факторов систем API. С одной стороны, *проектирование с учетом будущих изменений* означает разработку API и сервисов, которые *создатели* смогут быстро и просто менять. С другой стороны, *использование меняющихся сервисов* означает, что система должна предоставить модель применения, *позволяющую пользователям* работать с меняющимися сервисами. *Все, что усложняет изменения*, необходимо внимательно определять и изучать. Этот процесс может проходить *постепенно* — вначале найти и улучшить один снижающий скорость фактор, затем при необходимости повторить процесс.

Уязвимость

Как говорилось в подразделе «Уязвимость» на с. 194, растущая *уязвимость* является логическим следствием увеличения системы API. Если у вас нет ни одного API, значит, нет и потенциальных уязвимых мест, но любой API, который вы добавляете, становится таким местом. Если вы знаете об этом простом и неизбежном факте, значит, уже сделали первый шаг к развитию аспекта уязвимости.

В зависимости от аудитории API могут быть открыты для внутренних (внутренние API) или для внешних (партнерские и внешние API) пользователей. Во многих случаях эти два или даже три сценария защищают по-разному, часто с использованием разных компонентов (рис. 9.1).

---

[1] В этом случае удобнее пользователям, а создатели должны инвестировать в запуск нескольких версий API параллельно.

[2] В этом случае удобнее создателям, а пользователи должны инвестировать в то, чтобы убедиться, что их приложения правильно поддерживают непрерывную эволюцию API.

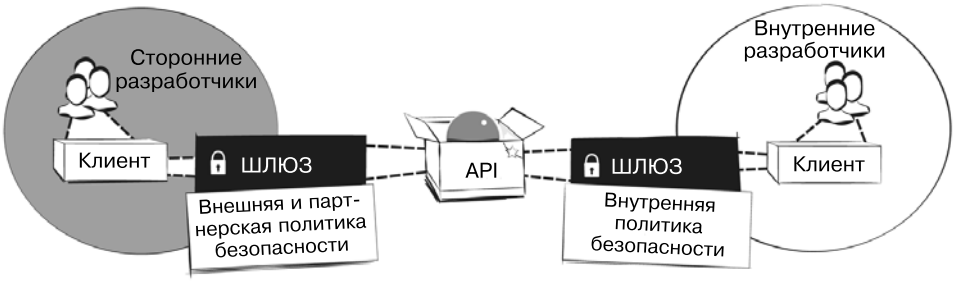


Рис. 9.1. Защита API с помощью шлюзов API

Понятно, что для обеспечения безопасности этот процесс организован относительно централизованно, чтобы у вас была возможность наблюдать за трафиком, управлять им (и, если понадобится, прерывать его) и таким образом лучше понимать, как его применять и каковы его потенциальные проблемы. Такая централизация в интересах безопасности вступает в конфликт с общими усилиями по децентрализации, и возникает вопрос о том, насколько от отдельных продуктов с API могут (и должны) зависеть контроль и конфигурация централизованного пункта усиления безопасности. Здесь сложно сбалансировать скорость и безопасность, но опять же в основном нужно ориентироваться на потребности организации и требования безопасности, а не на технические ограничения архитектуры.

Придерживаясь общей схемы, по которой мы смотрим на путь развития, мы можем применить ее и к уязвимости: важно наблюдать разработку API в системе и отслеживать общие темы и области, где могут помочь системная поддержка и инструменты. Единственное исключение из этого общего правила в том, что уязвимость — это аспект с высоким уровнем риска, поэтому необходимо жестче регламентировать наблюдение за системой и действия по этому поводу.

Примером такого подхода служат недавние разработки по поводу раскрытия в API персональных данных. Растущая популярность API приводит к повышению риска раскрытия через них персональных данных. Это опасно для организаций из-за возможных проблем с законодательством и органами надзора. Они не всегда могут быть видны командам, создающим продукты с API. К тому же хотя информация, раскрытая с помощью одного API, может казаться анонимной, чтобы не считаться персональными данными, растущая доступность дополнительной информации через другие API означает появление риска деанонимизации. И это обычно проще увидеть на уровне системы, а не на уровне отдельных API.

Существует также проблема непредсказуемых последствий раскрытия определенных данных с помощью API. Недавно Европейский союз выпустил Общий регламент по защите данных. Документ относится к обработке личных данных и требует, чтобы все организации в ЕС предоставили информацию об имеющихся в их распоряжении персональных данных и открывали к ним доступ по запросу.

Это означает, что создание продуктов с API, принимающих персональные данные, имеет далеко идущие последствия для организации, а в зависимости от размера и степени ее развития добиться соответствия данному регламенту может оказаться непросто.

Эти примеры показывают: несмотря на то что скорость полезна для системы API и является одной из причин, по которой организации и переходят на стратегии API, все равно управлять рисками необходимо. И в зависимости от сферы деятельности организации, разработанных API и их целевой аудитории во многих случаях для ответственного управления рисками необходимо принимать во внимание уязвимость и управлять ею.

РАЗВИТИЕ УЯЗВИМОСТИ

- API по умолчанию *раскрывают бизнес-функции*, которые до этого были недоступны или труднодоступны для пользователей. *Оценивать риск каждого нового API* необходимо, чтобы избежать *утечек информации* или других проблем, которые создадут основные риски для организации.
- Продукты с API должны документировать *всю хранящуюся в них информацию* и причины ее хранения. Информация *потенциально ценна*, однако может и увеличивать риски. К управлению информацией всегда нужно относиться как к тому, что может *потенциально создать риск в области законодательства, норм или репутации*.
- *Безопасность API* необходима в ответственной стратегии API, и к ней нужно относиться как к *важнейшему компоненту* общей *стратегии информационной безопасности* вашей организации.

В сравнении с другими аспектами системы уязвимость выделяется, так как увеличивает риск, поскольку API неизбежно создают проблемы, предоставляя доступ к бизнес-функциям.

Помимо вопросов защиты от злоумышленников или потенциального риска в отношении законодательства, норм и репутации, существует также вопрос стабильности сервисов и тестирования. Как говорится, с большой силой приходит большая ответственность, поэтому, если продукты с API становятся автономными с точки зрения разработки и развертывания, появляются и вероятные новые сценарии провалов. В своей знаменитой «Критике платформ Google» Стив Йегги писал: «Каждая из равных вам команд внезапно становится потенциальным исполнителем DOS-атаки. Никто не сможет нормально двигаться вперед, пока в каждом сервисе не будут введены серьезные квоты и регулирование». Эти слова подчеркивают, что надежность и устойчивость играют важную роль в стабильности системы API и она становится уязвимой для моделей неисправностей, которые вводятся с децентрализацией. С ней будет проще работать при интеграции.

Одна из самых больших сложностей в работе с уязвимостью системы — приспособиться к изменившейся реальности с более доступными бизнес-функциями. Поскольку у системы API есть явная цель открыть к ним доступ, с этой реальностью придется работать. С помощью оценки каждого API и управления его уязвимостью и упрощения встраивания продуктов API в эту архитектуру управление уязвимостью может встроиться в новое децентрализованное видение IT-системы, ориентированной на API.

СТРАТЕГИЯ РАЗВИТИЯ УЯЗВИМОСТИ

Переход к системам API требует другого подхода к управлению уязвимостью. Традиционную модель «внутренние API против внешних» нужно заменить моделью *индивидуального подхода ко всем сервисам как к компонентам, которые могут стать внешними*. Эта модель позволяет применять ко всем сервисам одни и те же модели безопасности. Система должна облегчить процесс защиты сервисов от злоумышленников или проблематичного поведения. Они могут быть внутренними, партнерскими или внешними, но изменение их категории в идеале не должно означать ничего, кроме *задействования другой политики безопасности*.

Видимость

Один из важнейших аспектов *видимости*, как говорилось в подразделе «Видимость» на с. 195, — это *наблюдаемость* API. Она соответствует принципу, описанному в подразделе «Выражать API через API» на с. 183, — он утверждает, что «все, что должно быть сказано об API, нужно выражать через этот API». Поэтому все, что нужно, становится видимым с помощью самого API, открывая доступ к информации другим сервисам системы, чтобы со временем при необходимости на ее основе можно было создать дополнительные сервисы.

Это эволюционный постепенный процесс, как и все в системе API. Изначально информации *об* API, которую нужно выразить *через* него, может быть не так уж много. Однако со временем все может измениться, поскольку увеличение объема и скорости диктует необходимость в лучшей поддержке и автоматизации определенных аспектов системы. Если эту поддержку и автоматизацию можно создать на основе API, они становятся частью самой системы API, то есть им не нужны особенные способы взаимодействия с сервисами.

В конце концов, одна из основных характеристик API — предоставление инкапсуляции, то есть API должен инкапсулировать *все* относящееся к его реализации, становясь единственным ресурсом для взаимодействия. Если придираться, то любой обходной путь, даже для внутренних целей, может считаться нарушением системного подхода.

Если API — это подход, который организация выбрала, чтобы *все* взаимозависимости стали открытыми, четко определенными и поэтому видимыми и управляемыми, тогда *любой* метод, создающий невидимые взаимозависимости, подрывает

его. Именно поэтому заключительная мысль знаменитого «Устава API» Джеффа Безоса состоит в том, что нельзя терпеть никакие методы, обходящие переход API: «Любой, кто не станет этого делать, будет уволен».

Распространенный способ создания взаимозависимостей, невидимых в системе API, — использование библиотек[1] (потенциально общих). Хотя многие разработчики считают, что применение библиотек отличается от создания взаимозависимостей на уровне API, на самом деле это не так, особенно при задействовании библиотек, которые или являются общими для нескольких продуктов с API, или предполагают наличие работающих взаимозависимостей с другими компонентами. Таким образом, если относиться к библиотекам так же, как к другим API, это может помочь избежать сценариев, где в систему возвращаются проблемы с управлением взаимозависимостями, находящиеся даже не на видимом и управляемом уровне API.

РАЗВИТИЕ ВИДИМОСТИ

- Важный аспект видимости — открывать *всю информацию об API*, которую необходимо *использовать или которой управлять в системе API*. Эта информация, скорее всего, со временем будет развиваться, поэтому важна возможность *легко менять API* в ответ на изменение требований пользователей и менеджеров к информации.
- Еще один важный аспект — открывать *все взаимозависимости с помощью API*, чтобы он стал *их точным отражением* и не существовало бы *скрытых взаимозависимостей*, использующих сторонние каналы.
- С ростом системы API *видимость на уровне API* необходимо дополнять *видимостью на системном уровне*, то есть способностью находить API, основываясь на видимой информации о них.
- И наконец, видимость также относится к легкости применения информации об API. Когда *API стандартизируют определенные функции*, например демонстрацию сообщения об ошибке или информацию о состоянии, такую информацию *проще использовать и собирать на системном уровне*, и это увеличивает видимость данных аспектов API.

Видимость на уровне API помогает развитию видимости на системном уровне: видимое на уровне API может задействоваться на системном уровне для облегчения обнаружения API, что делает их лучше видимыми.

[1] Смешно, но в этом и заключается изначальное значение термина «интерфейс для программирования приложений». API был создан в качестве интерфейса между двумя запущенными параллельно компонентами ПО — обычно пользовательским кодом и библиотекой. В последнее время (и в этой книге) API называют сетевые интерфейсы, изменив значение термина. Теперь он больше не охватывает традиционный сценарий локального API.

К примеру, если API четко и явно показывают все свои взаимозависимости на уровне API (принимая во внимание принцип, согласно которому все взаимозависимости должны считаться взаимозависимостями с API)[1], то эту информацию можно использовать для создания графика зависимости и даже создать информацию более высокого уровня, например высчитать популярность API[2].

В петле обратной связи потребность в видимости на системном уровне могут подпитывать требования к видимости на уровне API, а наблюдения за потребностями пользователей и методиками создателей API позволят системе адаптироваться к новым потребностям.

С развитием видимости до той стадии, когда API резко улучшаются как компоненты системы (адаптируясь к требованиям к видимости, появившимся из-за проблем с ней на системном уровне), отделение помогающих системе частей API от его функциональных аспектов способно стать повторяющейся схемой. Этот метод может сопровождаться методами, борющимися с уязвимостью, если помогающая системе часть должна быть доступна только инструментам системы.

СТРАТЕГИЯ РАЗВИТИЯ ВИДИМОСТИ

Чтобы API могли приносить доход, они должны быть *полезными* и их должно быть *легко найти*. Чтобы приносить пользу *системе*, им может понадобиться улучшить видимость части информации. Любая проблема в системе API должна вызывать вопрос: «Какая информация поможет решить эту проблему?» А если с видимостью на уровне API или системы возникают какие-то сложности, это должно вызывать обновление *руководства* (по улучшению видимости API) или *инструментов системы* (по улучшению видимости системы).

Контроль версий

Скорость — способность продуктов с API быстро меняться в ответ на отзывы или новые требования — это важная мотивация перехода к системе API. *Версии* являются ее неотъемлемой частью, поскольку каждый раз, когда продукт с API меняется, он становится новой версией. Как сказано в пункте «Семантический контроль версий» на с. 197, это не обязательно означает, что пользователь должен что-либо предпринимать по этому поводу (номер версии младшего уровня означает, что изменения совместимы с предыдущими версиями) или даже знать об этом (номер версии уровня «патч» означает, что интерфейс не изменился). Но управлять этими

[1] Показать взаимозависимости можно по-разному: разработчики могут открыто указать информацию о них, также можно использовать инструменты, проверяющие код реализации, или наблюдать за работающим API.

[2] График зависимости можно применять так же, как график ссылки в Интернете, в качестве вклада в методы вычисления, такие как алгоритм PageRank, упомянутый в главе 8.

версиями для уменьшения отрицательного влияния на систему обязательно, чтобы скорость не снижалась сильнее необходимого.

Контроль версий применим к API всех стилей. В дизайне на RPC и URI/гипермедиа он связан со сменой интерфейса ресурса, используемого для взаимодействий (процедур в случае API на RPC или взаимодействий с ресурсами для API на URI/гипермедиа). В API, построенных на запросах, контроль версий — это не часть самого интерфейса, представляющего собой общий язык запросов, а наука управления схемой данных, которые нужно запросить таким образом, чтобы существующие запросы продолжили работать. В событийно-ориентированных API контроль версий относится к дизайну сообщений, чтобы получатели новых сообщений могли обращаться с ними как со старыми, а не отклонять их из-за изменений в схеме.

Управлять версиями API можно различными способами, и отчасти это продиктовано разными целями API. Обещание стабильных API, которые никогда не изменятся (как это делает Salesforce), привлекает пользователей и может стать хорошей инвестицией, однако при реализации этой стратегии возникают затраты на запуск нескольких разных версий параллельно. Другая стратегия — следовать по пути Google и не обещать полной стабильности API, но реализовывать методику контролируемых изменений API. Существует большой риск, что пользователи не поймут эту стратегию, однако она снижает сложность операций.

РАЗВИТИЕ КОНТРОЛЯ ВЕРСИЙ

- Убедитесь, что у каждого API *есть стратегия создания версий*, — это первый шаг к развитию контроля версий. Она может включать в себя принятие того, что *API сознательно решают не поддерживать создание новых версий*, и любые обновления навредят изменениям, в особенности новым продуктам.
- Модели контроля версий во многом зависят от *стиля* и *модели использования API*, а также от *баланса между расходами и доходами* по инвестициям, которые создатели и пользователи должны вложить в эти модели.
- Контролю версий идет на пользу *согласованное отношение* к нему если не во всей системе API, то хотя бы в определенных классах API и пользователей.
- В зависимости от модели контроля версий *поддержка системы* API может помочь создателям и/или пользователям API убедиться, что *модель контроля версий системы поддерживается и правильно используется*.

Общие модели контроля версий делают лишь первые шаги и в плане стандартов, и в плане инструментов. Например, у популярного стандарта описаний OpenAPI нет модели версий или различий, из-за чего его трудно использовать в качестве прочной основы для контроля версий. Вместо этого стандарт поощряет генерирование кода по описанию, не затрагивая вопрос о том, что делать, если API развивается и у него появляется новое описание. Это, в свою очередь, поднимает вопрос о том, как менять код пользователя, чтобы адаптировать его к изменившемуся API.

Возможно, с ростом сложности и динамики систем API важность контроля версий также вырастет и стандартам и инструментам придется адаптироваться к этому. А пока развитие контроля версий все еще требует внимания и качественного управления на системном уровне. Отдельные API также значительно выиграют от руководства и/или инструментов, предоставленных им на данном уровне.

Как и ко всему на системном уровне, к контролю версий не стоит относиться так, словно существует только один верный способ его выполнения. Нормально иметь стратегию и поддерживать ее, но еще важнее быть открытым для новых моделей и иметь возможность перейти к их использованию. Контроль версий может значительно различаться в зависимости от стиля, конкретного класса или определенной группы пользователей API, поэтому важно уметь развивать эти стратегии и их применение с течением времени.

СТРАТЕГИЯ РАЗВИТИЯ КОНТРОЛЯ ВЕРСИЙ

Чтобы создание версий было *наименее вредным* для системы API, следует как можно реже задействовать *жесткий контроль версий* и как можно сильнее поддерживать *мягкий контроль*. Существует *огромное разнообразие* подходов к мягкому контролю версий, но основной принцип заключается в том, что само по себе создание версий довольно вредно, поэтому *согласованная стратегия контроля версий* (и возможные инструменты и поддержка для нее на системном уровне) должна быть найдена *как можно раньше*.

Изменяемость

Модели программирования сложно изменить, особенно в мыслях тех, кто занимается программированием. Как мы видели в подразделе «Изменяемость» на с. 198, программирование в децентрализованных системах — не самая простая задача, потому что в них существует много моделей неполадок, которых не было бы в более интегрированных системах, где модели неполадок значительно менее сложные. Работа с неизбежной *изменяемостью* сервисов в системе API требует изменений в образе мыслей разработчиков.

Когда вы изначально переходите к системе API, возможно, разработчики будут попрежнему использовать свои модели программирования и писать приложения, чересчур оптимистично представляя доступность компонентов. В системах с большим количеством взаимозависимостей может быть особенно сложно даже обнаружить источник проблем: при их появлении в одном из приложений для поиска причин может потребоваться отслеживать несколько сервисов — и это задание отличается от более традиционного подхода, заключающегося в последовательной отладке монолитного кода.

В идеале приложения будут ответственно справляться со всеми взаимозависимостями с API, и в них будет встроена устойчивость, включающая такие подходы, как *постепенное сокращение возможностей*. Чтобы лучше справляться с изменяемостью, надо также разрабатывать API в более устойчивом стиле и пытаться ма-

ксимально использовать существующие в системе операции. Не все взаимозависимости приложения крайне важны, таким образом, можно разработать приложение, поддерживающее откат к предыдущей версии и в то же время приносящее доход, даже если некоторые сервисы недоступны.

РАЗВИТИЕ ИЗМЕНЯЕМОСТИ

- *Локализация условий возникновения ошибок* — основное требование, которое необходимо выполнять в системе API. Возможность *отслеживать трафик и таким образом получать информацию* крайне важна для локализации проблем.
- Изменение методов разработки, позволяющее *лучше справляться с децентрализованными режимами неполадок*, помогает убедиться в том, что *отдельные неполадки* не обязательно превращаются в *каскад по всей цепи взаимозависимостей*.
- Чем больше количество разработчиков, которым вы сможете помочь или которых сориентировать на *более устойчивые методы разработки*, тем *более прочной будет система API*.

Поскольку изменяемость при децентрализации неизбежна, необходимо помнить, что переход с интегрированной на децентрализованную модель меняет модель неполадок для всей системы. Учитывая, что отдельные сервисы теперь могут портиться, их надежность более сложным образом влияет на надежность всей системы. И общий эффект от отдельных неполадок увеличивается с ростом системы API и изменением графика зависимости между API.

Чтобы разобраться с возможным ростом общего уровня неполадок, необходимо уменьшить распространение ошибок. Частично это можно сделать, если убедиться, что компоненты устойчивы к неполадкам и изолируют их, а не распространяют.

Изменяемость — это один из тех аспектов, которые вы можете отложить на потом, но проблемы быстро выходят из-под контроля, особенно когда динамика системы API ускоряется. И разумеется, худший из вариантов, когда могут проявиться системные проблемы с надежностью, — это момент, когда система API и уровень ее изменений резко растут. Поэтому хорошо бы начинать инвестировать в работу с изменяемостью на ранних этапах и считать это необходимым первым шагом на пути развития системы API.

СТРАТЕГИЯ РАЗВИТИЯ ИЗМЕНЯЕМОСТИ

Для управления изменяемостью в системе API требуется *изменить методы разработки* на написание приложений, которые *ответственно работают в децентрализованной системе*. Требуется также, чтобы *система предоставила инструменты*, необходимые для того, чтобы *справляться с изменяемостью в подобной системе*, например отслеживать ошибки. Если вы *не справитесь с изменяемостью*, эффект почувствуете очень быстро, особенно когда *система быстро растет*, и/или когда *сервисы в этой системе начинают быстро меняться*, или когда у них *разная операционная стабильность*.

Выводы

В этой главе мы исследовали путь развития системы API, используя аспекты, представленные в главе 8, чтобы очертить путь к более высокоразвитой системе API. Мы ознакомились с отдельными факторами, влияющими на развитие каждого аспекта и на то, как инвестиции в улучшение развития проявятся на системном уровне. Мы решили не указывать один линейный путь к развитию системы API. Вместо этого использовали аспекты системы, чтобы показать многогранность этого пути. Главное, о чем нужно помнить: все аспекты влияют на общее развитие системы API, поэтому важно учитывать их все при оценке текущего уровня развития системы и возможных путей его улучшения.

В следующей главе мы соединим точки — рассмотрим, как развитие системы API взаимодействует с жизненным циклом продукта с API, представленным на рис. 6.1. В конце концов, цель систем API — обеспечивать удобные условия для разработки, использования и улучшения API, поэтому нужно понимать, как на методы создания отдельных продуктов влияет система, в которой они разрабатываются и развертываются.

10 Управление жизненным циклом API в развивающейся системе

> Из многих капель собирается ведро, из многих ведер собирается пруд, из многих прудов — озеро, а из многих озер — океан.
>
> *Перси Росс*

Мы надеемся, что если вы внимательно читали книгу до этого момента, то уже заметили, что между управлением жизненным циклом одного API и управлением жизненным циклом целой системы API существует несколько важных различий. Наш опыт свидетельствует: у компаний, которые замечают различия между одним и множеством и действуют, учитывая их, лучшие шансы на долгосрочный успех в процессе цифровой трансформации.

В этой главе мы кратко затронем эти различия, а затем углубимся в столпы жизненного цикла, которые описывали в главе 4. Сосредоточимся на системе — множестве API, которое описывали в главе 8, а не на одном API. Это должно нарисовать картину того, как проблемы масштаба, объема и стандартов, которые обсуждались в главе 1, вступают в игру, когда система API начинает расти.

На протяжении всего пути мы будем возвращаться к восьми аспектам системы API, описанным в главе 8 и уточненным в главе 9, и затронем некоторые элементы принятия решения (рассмотрены в главе 2). Собранные вместе, эти части помогут вам понять, как лучше всего применять эти схемы и методы в своих командах, для своих продуктов так, как будет оптимально для культуры и ценностей вашей компании.

Продукты с API и столпы жизненного цикла

Как отмечалось в главе 3, применение подходящего уровня мышления в интересах продукта к работе над API помогает вашим командам сосредоточиться на клиентоцентричном использовании интерфейса. Для вас это первая возможность разработать и реализовать API с помощью подхода Клейтона Кристенсена «Работа, которую нужно выполнить» (Jobs-to-be-Done, JTBD) (https://www.christenseninstitute.org/jobs-to-be-done/). На уровне бизнеса то, что вы научите команды по дизайну и архитектуре задавать вопросы: «Как это поможет нам достичь целей нашего бизнеса?» и «Каких OKR мы хотим достичь с помощью этого дизайна?» — значительно снизит вероятность того, что команды, занятые серверной частью приложения, просто выпустят интерфейсы, основанные на данных, без четкого пользовательского потока. Фокусируясь на JTBD и пользователях, руководителям IT проще опрашивать команды об их вкладе в общую работу и привязывать инициативы в области API не только к KPI, сконцентрированным на технологиях, но и к общим OKR на уровне бизнеса.

Помимо стратегии «API как продукт», мы также рассматривали так называемые столпы жизненного цикла API (см. главу 4). Хотя на самом деле они не составляют цикла, то есть четкого порядка, повторяющегося раз за разом, столпы жизненного цикла определяют важнейшие элементы, поддерживающие жизнеспособную программу с API. Их можно использовать в качестве руководства по созданию API. Таким образом, модель AaaP и столпы жизненного цикла поддерживают друг друга на уровне одного API.

Однако, как вы видели в последних двух главах, знакомящих с понятием системы API, компании редко работают с несколькими отдельными API. Напротив, большинство организаций создают системы взаимозависимых и функционально совместимых API, которые помогут в получении дохода и снижении затрат и рисков при использовании API и сервисов для достижения целей бизнеса (OKR).

Системы API

Пока в этой книге рассказывалось в основном о том, как непрерывно управлять отдельными API и как они встраиваются в большую картину системы API. Восемь аспектов, описанных в главах 8 и 9, использовались для того, чтобы позволить вам внимательнее рассмотреть отдельные элементы этой картины и обдумать все важные задачи управления растущей и непрерывно развивающейся системой API.

В этой главе мы хотим соединить перспективу управления отдельными API и рабочее окружение системы API, в которой они находятся. Мы сфокусируемся

на основной теме — на том, что связь между отдельными API и системами всегда должна быть обоюдной. *API вносят свой вклад в систему*, и за ними должно быть максимально просто наблюдать, чтобы получать информацию о том, как они разрабатываются и развиваются. *Система направляет и поддерживает команды продуктов с API*, предоставляя им информацию, формирующую общую картину принципов, протоколов и методов системы, а также руководство и поддержку.

Восемь аспектов системы можно использовать, чтобы сосредоточиться на отдельных столпах жизненного цикла API. Мы соединим перспективы отдельного API и системы, обсуждая, как нахождение API в системе влияет на эти столпы и какие аспекты системы важнее, когда вы снова обдумываете этот конкретный столп (и как поддержать команды продукта с API, опираясь на него) в контексте всей системы.

Момент принятия решения и развитие

В подразделе «Решения» на с. 34 мы заявляли, что код и интерфейсы, создаваемые сегодня, — тупые: релизы выполняют в точности то, что им велят, и ничего больше. Код не способен *решать*, *исследовать* или *экспериментировать*, он может только *выполнять*. Решения и творческий подход исходят от человека, и их в какой-то момент необходимо перевести в код и API, чтобы реализовать.

Чтобы принять решение, очень важно знать, *когда* нужно это сделать. Оказывается, откладывать решение зачастую разумно, если речь идет о реализации системы API. В растущей системе так много взаимозависимых и совместно функционирующих элементов, что поспешное решение может уменьшить количество будущих возможностей и даже сделать невозможными некоторые из лучших вариантов решения проблем в вашей архитектуре. Том и Мэри Поппендик, авторы нескольких книг, в том числе «Бережливая разработка программного обеспечения: гибкий набор инструментов» (*Lean Software Development: An Agile Toolkit*, издательство Addison-Wesley), рекомендуют «откладывать подтверждение на позднейший ответственный момент, то есть момент, когда непринятие решения исключает важную альтернативу».

Однако применение метода откладывания подтверждения может оказаться трудным для IT-менеджеров и дизайнеров системного уровня. Часто спрашивают: *когда* именно наступает этот позднейший ответственный момент? Цель данной главы — помочь вам распознать распространенные изменения формы вашей системы API, пройдясь по столпам и продемонстрировав часто встречающиеся сложности, связанные с разными аспектами системы. Это должно помочь вам сравнить аспекты, наблюдаемые в своей компании (разнообразие, объем и т. д.), с приведенными здесь примерами. Мы надеемся, что эти примеры помогут вам понять, какие столпы требуют особого внимания из-за роста вашей системы.

Аспекты системы и столпы жизненного цикла API

Столпы жизненного цикла и подход «API как продукт» совместно формируют методы, применимые в дизайне, разработке и выпуске API в вашей организации, и те же самые столпы могут помочь в управлении растущей системой API. Можно создать простую матрицу (табл. 10.1), в которой вышеупомянутые аспекты системы и столпы жизненного цикла API сочетаются, чтобы показать предметные области, которыми нужно управлять в системе API.

Таблица 10.1. Столпы жизненного цикла и аспекты системы API

Столп	**Аспект**							
	Разнообразие	**Объем**	**Словарь**	**Скорость**	**Уязвимость**	**Видимость**	**Контроль версий**	**Изменяемость**
Стратегия	✓	✓		✓				
Дизайн	✓		✓					
Документация	✓		✓			✓		
Разработка	✓			✓				✓
Тестирование		✓		✓	✓			✓
Развертывание	✓			✓				✓
Безопасность				✓	✓	✓		
Мониторинг		✓				✓		✓
Обнаружение	✓	✓	✓			✓		
Управление изменениями			✓	✓		✓		

Каждый столп, упомянутый в главе 4, и каждый аспект, описанный в главе 8, заслуживают внимания. Когда система растет, она не просто увеличивается. Она *меняет форму*. В небольшой системе API (например, если одна команда работает над одним взаимосвязанным набором API) не придется тратить много времени на создание руководств и стандартов по стилям API или форматам сообщений, потому что все работают в одной команде, используют одни и те же инструменты и стремятся к одним и тем же результатам.

Однако во время роста системы API появятся новые команды с другими целями. Некоторые из них могут находиться далеко от вас, использовать другие наборы инструментов, стили API и руководства. В растущей системе могут появиться большее разнообразие, более полный словарь и различные уровни видимости, объема, скорости и уязвимости. Со временем растущие системы меняют свою форму.

Рассмотреть каждый из аспектов системы в применении к столпам жизненного цикла, несомненно, важно, но для этого потребовалась бы куда более толстая книга, чем та, которую мы вам предлагаем. Вместо этого мы решили сосредоточиться на избранных аспектах каждого из столпов, привести полезные примеры и в некоторых случаях посоветовать, как справиться со сложностями.

Надеемся, что вы сможете использовать следующие идеи в качестве руководства, когда ваша система начнет увеличиваться в масштабах и объеме и станет проходить через стадии развития, описанные в предыдущей главе. Поскольку в каждой компании своя культура, может оказаться полезно создать собственные чистые шаблоны матриц, чтобы активизировать обсуждение внутри команд и начать дискуссию, которая позволит всем сотрудникам организации привести свои примеры и советы, помимо предложенных здесь.

Запомнив это, давайте пройдемся по столпам жизненного цикла API и выделим несколько самых распространенных аспектов системы, с которыми вы столкнетесь в ходе ее роста и изменения.

Стратегия

Во время роста системы API компании тактика и даже краткосрочные цели ее стратегии API могут изменяться. В подразделе «OKR и KPI» на с. 123 мы показали, как OKR и KPI могут повлиять на общую цифровую стратегию и на использование API и их целевую аудиторию — внутреннюю, внешнюю или партнерскую. Хотя это все еще важно при расширении системы, изменятся детали.

К примеру, KPI для ваших первых API, скорее всего, фокусировались на надежности, увеличении использования внутри компании и вкладе в доход или снижение затрат. Когда ваша программа расширяется до десятков, сотен или даже тысяч API, вам может понадобиться перестроить KPI, чтобы они сфокусировались на вопросах, уникальных для больших систем API. Здесь вы и можете применить аспекты системы, описанные в главе 8, чтобы сменить свою тактику и требования к реализации на соответствующие текущим задачам системы.

Среди аспектов системы, которые нужно принимать во внимание при перестройке стратегии, особого внимания требуют три: разнообразие, объем и скорость. Кратко рассмотрим их.

Разнообразие

При добавлении в систему новых групп продуктов, команд и пользователей API ваша способность контролировать и ограничивать каждый элемент дизайна и реализации API станет менее эффективной из-за естественной тенденции к росту разнообразия. Заставить одну команду или небольшую группу параллельных команд выполнять одни и те же указания по дизайну и использовать одинаковые

форматы, инструменты и методы тестирования и релиза, довольно легко. Однако, если вы добавите новые локации (например, офис, находящийся через полмира от вас), начнете поддерживать API групп других продуктов (например, приобретенных компаний) и работать с продуктами, использующими широкое разнообразие технологий (STFP, системы мейнфрейма и т. д.), то поймете, что больше не можете просто диктовать, как выполнять задания.

С ростом системы разнообразие возрастает естественным образом. Вместо того чтобы пытаться избежать этого здорового разнообразия, важно изменить стратегию API так, чтобы приветствовать различия и сосредоточиться на отрицании общих *принципов* для всех команд, а не заставлять всех использовать одни и те же методики во всех спектрах технологий и группах продуктов (см. подраздел «Принципы, протоколы и шаблоны» на с. 179).

Увеличение разнообразия — это нормально.

Объем

При увеличении системы API растет и объем многих элементов — становится больше API, больше трафика, больше команд и т. д. Однако ваши ресурсы для управления этим ростом могут все еще быть ограничены. Поэтому обычно приходится выбирать, какие из новых инициатив поддержать, какие из старых деактивировать, а какие API пока оставить без изменений.

Когда дело доходит до выбора, фокусируйтесь на API, которые точно приносят положительные результаты и которые проще обновлять и поддерживать, — это поможет вам разобраться с растущим объемом. Возможно, придется начать инвестировать в платформы, лучше масштабируемые при высоком объеме трафика. Вам также может понадобиться перейти с локальных релизов на виртуальные облачные устройства, которые проще масштабировать. Возможно, некоторые из API будут работать лучше в среде «функции как услуга» и др. А в некоторых случаях разумнее вернуть трафик обратно в локальную инфраструктуру, чтобы снизить затраты.

Объем — один из самых распространенных аспектов, с которыми вы столкнетесь при росте системы, существует множество способов справиться с ним.

Скорость

Последний аспект системы, который мы здесь рассмотрим, — это скорость. Некоторые пользователи сообщают нам, что все становится быстрее и быстрее и выдерживать темп непросто. Иногда это действительно так, но иногда проблема не в скорости изменений, а в их *количестве*. Скорость может проявляться в нескольких формах.

При росте системы понадобится вносить изменения во многие ее части, поэтому вы будете чаще их замечать. Нужно будет перестроить стратегию API так, чтобы изменения не были разрушительными и при этом становились более дешевыми и менее рискованными. Обычно это означает установку ограничений для масштабных изменений (например, формальных предложений, внимательного изучения и утверждения) и их ликвидацию для небольших корректив (например, незначительных изменений макета UI, исправлений ошибок, изменений, не вредящих API, и т. д.).

Скорость может проявляться также в виде роста количества пользователей. Если вы реализовали успешные API, которые приносят новые заказы, но команды, занятые серверной частью приложения, все еще выполняют такие задачи, как изучение и утверждение клиентов, вручную, то вскоре обнаружите огромное количество невыполненных заказов и можете потерять значительный доход. Стратегия вашего API должна простираться за пределы технических подробностей разработки и релиза кода — она должна включать в себя всех сотрудников организации.

Скорость выражается не только в простом увеличении темпа, также она может создать иллюзию замедления в некоторых частях компании.

Дизайн

Как говорилось в подразделе «Дизайн» на с. 81, дизайн интерфейса является важным столпом работы над продуктом с API. А когда API разрабатываются в контексте системы, появляются новые сложности. Часть системы API — это уже не отдельный продукт, а часть семьи продуктов. Насколько такое видение должно влиять на дизайн отдельных API, во многом зависит от предполагаемого применения API.

- API, которые ожидает многократное использование, проектируются с высоким уровнем видимости и будут задействоваться в качестве контактной точки отдельного пользователя. Их можно по-прежнему разрабатывать во многом как отдельные продукты, оптимизируя дизайн в первую очередь для пользователей API. Однако с точки зрения реализации, невидимой пользователям, все же имеет смысл синхронизировать его с системой, позволяя разработчикам применять все преимущества поддержки и инструментов.
- API, который, скорее всего, будет использоваться как часть системы, например, в соединении с другими API той же семьи продуктов, нужно разрабатывать, помня об этой схеме использования. Знакомый дизайн будет играть важную роль: для разработчиков, использующих и сочетающих различные API из одной системы, полезнее синхронизация на уровне дизайна API.

Если посмотреть на два этих сценария, становится понятно, что дизайн API в обоих случаях играет важную роль, но совершенно по-разному. В первом случае синхронизация важна в основном при реализации, хотя и при создании дизайна можно

пользоваться советами из руководства, определяя и решая проблемы. Во втором случае синхронизация важна и на уровне дизайна, и на уровне реализации, поэтому руководство в этом сценарии важнее.

Когда речь идет о подходе к дизайну в системе API, упомянутые здесь размышления можно дополнить с помощью аспектов системы, описанных в главе 9. Из этих аспектов на дизайн сильнее всего влияют разнообразие, словарь и контроль версий.

Разнообразие

Разнообразие — естественный результат развития системы API. Причиной является изменение схем дизайна со временем, отчего одна и та же проблема решается по-разному, в зависимости от методик, принятых на момент создания дизайна. Вторая причина состоит в том, что разные продукты соответствуют различным потребностям и нацелены на разных пользователей, поэтому одной методики, идеальной для всех групп пользователей, не существует.

По этим причинам разнообразие следует разрешить, а не пытаться ограничить пространство решений одним возможным дизайном. Однако управление сферой дизайна с помощью руководства, управляемого C4E (что обсуждалось в разделе «Центр подготовки» главы 9), помогает ограничивать его с пользой для дизайнеров, перед которыми находится набор вариантов дизайна, из которых они могут выбрать, основываясь на том, как те применяются и какие инструменты поддерживают.

Одна из антисхем разнообразия появляется, когда одну и ту же проблему в разных API решают по-разному без веской причины. Это не только пустая трата ресурсов команды на поиск новых решений вместо использования существующих — это отрицательно влияет на продуктивность пользователей, потому что им приходится заново учиться работать с каждым API. Такое отношение к отдельному API часто называется «прекрасные снежинки» — каждый из них уникален и отличается самыми тонкими деталями. Данный подход совершенно не продвигает и не поддерживает повторное использование там, где это необходимо.

В общем, разнообразие дизайна имеет смысл и должно приниматься там, где на это есть *причина, основанная на самом продукте*. В другом случае экономичнее и полезнее для системы использовать устоявшиеся схемы дизайна.

Словарь

Объединение терминологии дизайна по всей организации может помочь создать более согласованную методику и не допустить повторных, а в худшем случае конфликтующих между собой попыток создания моделей для одной и той же сферы.

В то же время определение и синхронизация словарей требуют затрат, поэтому попытка синхронизировать все внутри организации может быть не самым экономичным выбором[1].

Обычно концепции предметных сфер, которые можно безопасно инкапсулировать внутри сервиса (то есть те, которые не демонстрируются с помощью API этого сервиса), вообще не нужно синхронизировать. Это детали реализации сервиса, и если сделать их видимыми за пределами сервиса, это будет противоречить принципу инкапсуляции.

Концепции, важные для API сервиса, можно разделить на два типа.

- *Словари, не зависящие от предметной сферы*, должно быть легко найти внутри организации. Простой пример — список стран или языков. В любом API, использующем словари, нужно пытаться разграничить API и словарь, а затем составить список словарей, чтобы можно было наблюдать за их применением.
- *Словари, зависящие от предметной сферы*, должны быть установлены (а не просто оформлены в виде ссылки) как часть дизайна API. В зависимости от развития этого аспекта (как сказано в разделе «Словарь» главы 9) система может обеспечивать поддержку, чтобы облегчить командам определение и распространение нового словаря.

В целом управление словарями для дизайна API следует общему принципу, согласно которому синхронизация словарей полезна и в этом могут помочь наблюдения и поддержка. API, фиксирующий использование словарей, помогает руководителям системы наблюдать за их применением и развитием, что может привести к упрощению дизайна API посредством использования устоявшихся словарей вместо создания новых.

Контроль версий

Одна из главных целей высокоразвитой стратегии API — разъединить эволюцию сервисов, чтобы они могли развиваться индивидуально на оптимальной для них скорости. Такая эволюция означает, что реализации сервисов могут меняться, как и сами API. С точки зрения дизайна первое все же важнее, поскольку может означать, что команда продукта решила справиться с проблемой по-новому.

---

[1] Примером могут служить многочисленные попытки создать модель информации предприятия в больших организациях (см. врезку «EIM и API: идеал против прагматизма» в главе 8). В некоторых случаях предприятие меняется довольно медленно для того, чтобы эта задача была выполнимой, но даже тогда инициативы создания EIM редко считаются удачными.

Чтобы понять, каков уровень изменений в системе API, важно отслеживать версии, как мы уже обсудили в подразделе «Контроль версий» на с. 224. По принципу выражения API через API (см. подраздел «Выражать API через API» на с. 183) это означает, что API должны показывать номера своих версий, чтобы изменения в их реализации и дизайне стали видимыми.

Управление версиями тоже важно с точки зрения клиента, об этом подробнее — в подразделе «Управление изменениями» далее. Таким образом, важной частью дизайна становится следование принципам дизайна, которые облегчают работу с новыми версиями для пользователей, в идеале так, чтобы они всегда могли узнать об открытии доступа к новой версии, но чтобы при этом им приходилось что-то делать только в том случае, если они решают воспользоваться новыми функциями.

В общем, при разработке дизайна в системах API всегда нужно помнить о том, что сервисы будут непрерывно развиваться. В таком случае *проектирование с учетом будущих изменений* означает упрощение получения информации о новых версиях и для руководства системы, и для пользователей сервиса. В то же время это значит, что дизайн должен требовать от пользователей как можно меньше усилий при адаптации к новым версиям.

Документация

Как говорилось в подразделе «Документация» на с. 85, объем документации прямо пропорционален степени проработки API, хотя наличие базовой документации всегда неплохо, а в случае подхода «API как продукт» (см. главу 3) с нее и нужно начинать.

Документация может существовать в самых разнообразных формах: минимальную можно сгенерировать с помощью технических приспособлений, например описаний OpenAPI, ее можно дополнить комментариями и примерами и даже интегрировать в сам API с целью оптимизировать опыт разработчика до такой степени, чтобы убрать почти все препятствия для использования и сделать API крайне оптимизированным самодостаточным продуктом.

Однако последний шаг по вложению в документацию может быть довольно дорогим, а инвестиция считается полезной, только когда усилия по созданию идеальной документации перевешиваются количеством разработчиков, которым она принесла пользу.

Уровень инвестиций в документирование отдельных API зависит в основном от развития API и намеченной целевой аудитории, как говорилось в главе 6. Однако система должна предоставить руководство и поддержку, как только решение об инвестиции принято. Из всех аспектов системы самыми важными при поддержке документации отдельных API являются *разнообразие*, *словарь*, *контроль версий* и *видимость*.

Разнообразие

Если вы будете поддерживать разные стили документации, это поможет каждой команде выбрать оптимальный стиль для API, который она разрабатывает и развивает. Выбор будет зависеть как от *стиля* API, так и от развития и предполагаемой ЦА отдельных API.

Позволяя командам выбирать инструменты для документации, подходящие для дизайна их API, его стадии развития и целевой аудитории, вы поможете им опубликовать документацию, отвечающую текущим потребностям.

Если у вас есть комплекты требований к документации, полезно иметь конкретные руководства и инструменты, а также инструменты для утверждения, позволяющие командам интегрировать проверку документации в процесс создания API. В большинстве случаев нельзя автоматически проверить все аспекты документации, но часто можно включить хотя бы несколько проверок на адекватность (есть ли в ней разделы о расширении и создании версий, и точно ли они явно подчеркнуты и выделены). В таком случае команды будут лучше понимать, чего от них ожидают.

Возможно, разнообразие документации будет развиваться со временем. Некоторые стили документации могут устареть, а другие — появиться. В идеале разнообразие в стилях документации должно быть отделено от ее рабочего окружения, чтобы, к примеру, какие-то важные принципы, например указания по разделам, посвященным расширению и контролю версий, можно было распространять на разные стили документации.

Словарь

Как мы обсудили в подразделе «Словарь» на с. 212, одним из признаков развития системы API является управление словарями нескольких API, чтобы создателям API было проще найти и повторно задействовать существующие словари, а пользователям — применить свое знание терминологии в нескольких API.

Управление словарями для улучшения документации означает *управление документацией этих словарей*, чтобы любой API мог повторно использовать ее, применяя предоставленный ему словарь. В зависимости от способа управления документацией, повторное использование может быть «паутинным» и представлять собой просто ссылки на документацию, доступную в другом месте. А если стиль документации чуть менее «паутинный» и относится скорее к созданию автономной документации для API, тогда повторное использование может означать включение ее в документацию API[1].

[1] «Паутинный» способ, который мы упомянули первым, называется виртуальным включением. При его использовании доступ к документации открыт через документацию API, но она продолжает оставаться документацией словаря.

Важно помнить, что управление документацией словарей очень полезно с точки зрения наблюдения: если API документируют используемые термины так, что это *поддерживается* или как минимум *наблюдается* в системе, становится гораздо проще разбираться в использовании терминов в разных API, предлагать новые словари и понимать, какие из них уже не используются широко. Еще раз: *наблюдение за API, чтобы лучше поддерживать систему*, и *поддержка API, чтобы облегчить наблюдение*, — это две стороны одной монеты, которые обеспечивают лучший способ сочетания преимуществ для индивидуальных API и для системы.

Контроль версий

Одна из главных целей API и систем API — разделить реализации, чтобы отдельные продукты могли развиваться индивидуально. Такой рост скорости продукта ведет к росту количества его *версий*, что мы уже обсудили в подразделе «Контроль версий» на с. 224. Хотя в идеале бóльшая часть версий не должна вредить API, каждую версию, которая меняет младшую версию при использовании семантического контроля версий, все же нужно документировать. Это помогает пользователям понимать, какие изменения могли произойти, а создателям — делать документацию полезной для нескольких версий.

Как сказано в подразделе «Выражать API через API» на с. 183, документация, как и любая информация *об* API, должна быть частью API. Согласно этому принципу *история документации* также должна быть частью API, позволяя пользователям понимать эволюцию API, отслеживая ее[1].

Руководства по управлению версиями могут упростить использование API и помочь пользователям получать сведения о новых версиях и принимать решения о том, когда и как они хотят подстроиться под обновление. Если вы даете командам продуктов с API руководство, рассказывающее о том, как создавать и публиковать документацию для нескольких версий, им будет проще следовать этим методам. Система может предоставлять поддержку и инструменты для тестирования, чтобы разработчики API могли сразу же получить обратную связь по своей документации для нескольких версий.

Если ваша система использует семантический контроль версий и придерживается «паутинных» принципов, можно рекомендовать создать навигацию по всем версиям документации API с помощью ссылок RFC 5829 (https://tools.ietf.org/html/rfc5829). Эта схема включает в себя навигацию по истории документации, а также документацию отдельных версий со ссылками друг на друга. Тестирование на на-

[1] Это относится как минимум к изменениям, не вмешивающимся в работу API. Для тех, которые вмешиваются (старшая версия в семантическом контроле версий), может сработать другая схема — архивирование документации старых версий, что само по себе может быть сервисом, предоставляемым системой API.

личие этих ссылок можно провести после развертывания API и их документации, чтобы можно было утвердить хотя бы схематическую часть этой методики.

В общем, документация в системах API, естественно, будет включать в себя аспект контроля версий, и руководители системы могут получить информацию о версиях и их документации, предоставляя руководство и поддержку для наблюдаемого документирования версий.

Видимость

Документация сама по себе может быть довольно сложным ресурсом, содержащим большое количество информации и структур. Как мы уже обсудили, полезно иметь существующие инструменты и поддержку документации, чтобы команды по API могли положиться на определенные процессы при создании документации, а не выбирать и не создавать собственные.

Важно помнить (см. подраздел «Разнообразие» на с. 210), что главной целью документации по руководству и поддержке должно быть объяснение командам не как что-то создавать, а что нужно создавать. У них должны быть поддерживаемые наборы инструментальных средств, позволяющие легко ответить на вопрос «Что?», следуя инструкции, но важно отделить набор инструментальных средств от того, что он должен создавать.

Документация по API всегда должна быть видимой, а вместе с ней — все аспекты, значимые с точки зрения системы. Это важно и для пользователей документации, и для самой системы, которая может задействовать эту видимость, чтобы получить подробную информацию об API и при необходимости создать глубинные ссылки на документацию. Это означает, что важно обозначить в руководстве, какие требования должны быть наблюдаемыми с точки зрения системы, и добавить эти аспекты к руководству и инструментам.

Разработка

Со стороны кажется, что *разработка* не играет такой уж важной роли в системах API. В конце концов, задача API — *инкапсулировать* реализации, поэтому способ разработки не рассматривается с точки зрения чистого API. Однако без разработки реализации API точно не существовало бы, поэтому столп «Разработка», который мы обсуждали в подразделе «Разработка» на с. 88, — важнейшая часть систем API. Повышение общей эффективности системы API — одна из главных целей управления системами, поэтому важно изучать, как разработка встроена в перспективу системы.

И снова посмотрим на Интернет как на самую крупную из известных нам систем API. Если бы кто-то ввел правило, что все веб-приложения должны программироваться на одном и том же языке с помощью одних и тех же инструментов

разработки, это быстро погубило бы его. В конце концов, один из главных рецептов успеха Сети, особенно в сравнении с конкурентами на раннем этапе, — то, что здесь командам разработчиков обеспечена полная свобода разработки выбранных решений, если их продукт в итоге работает как приложение, используемое через браузер.

И хотя неотъемлемой частью успеха Интернета стало то, что веб-архитектура не диктовала выбор языков и инструментов для разработки, но, когда веб-приложения стали основным его направлением, поддержка разработки оказалась главным фактором эффективности их создания. Веб-ориентированные языки и фреймворки: PHP, ASP, JSP, JSF, Django, Flask, Ruby on Rails, Node.js и многие другие — сформировали способ разработки веб-приложений в прошлом и настоящем. Но они и сами проходят свой жизненный цикл, поэтому с полной уверенностью можно заявить, что Интернет не только пережил множество языков и инструментов, которые использовались для веб-приложений, но переживет и нынешние, и те, что появятся в будущем.

Этот урок, полученный благодаря обзору методов разработки и поддержки Интернета, напрямую применим к системам API: существование языков и инструментов, поддерживающих разработку отдельных продуктов, увеличивает продуктивность и во многом помогает с определенными протоколами и шаблонами. Однако важно помнить о том, что раз протоколы и шаблоны меняются, то меняются и языки, и инструменты. И даже без изменений в протоколах и шаблонах все равно появится постоянный поток языков и инструментов, претендующих на роль лучшего решения существующих задач.

Поэтому, если вы не будете смотреть на разработку с точки зрения системы, то можете упустить значительные возможности получить большую экономию, установить методы разработки и поделиться ими со всеми командами, оценить и принять новые языки и инструменты, когда они становятся доступными. Чтобы система поддерживала и реализовывала эти возможности, нужно помнить о самых важных аспектах — разнообразии, скорости, контроле версий и изменяемости.

Разнообразие

Если смотреть с точки зрения подхода «API как продукт», то на начальных стадиях создания API никто не думает о деталях реализации или методах разработки продукта. Всех волнуют только дизайн, обсуждение протоколов и наблюдение за тем, как первая обратная связь может повлиять на ранние стадии дизайна.

Когда становится ясно, как должен выглядеть продукт с API, следующий шаг в этой идеализированной картинке — продумать оптимальный способ самого создания продукта. Это решение должно быть основано на дизайне продукта (убедитесь в выборе правильных инструментов для работы) и команде (убедитесь в том, что команду устраивает выбор инструментов).

Поскольку разные API служат различным целям, имеют разную целевую аудиторию и разрабатываются разными командами продуктов API, вероятно, единственного оптимального языка и набора инструментов для разработки каждого отдельного API не существует. Поэтому очень важно поддерживать разнообразие в системе и позволять командам экспериментировать с новыми подходами, когда они чувствуют в этом необходимость. Поиск баланса между разнообразием и необходимостью не создавать систему, состоящую из реализаций, — это непросто. Прежде всего нужно принимать во внимание необходимость *согласованности* в отношении аспекта разнообразия. Использовать различные языки и инструменты — это вполне нормально, но мы советуем ограничить количество вариантов, чтобы обеспечить согласованность между ними. Так инвестиции и образование станут более результативными благодаря ограничению масштаба. И поэтому вокруг вариантов разработки всегда накапливается некая критическая масса. Если определенными языками или инструментами разработки пользуется больше определенного числа продуктов с API, они могут сменить статус экспериментальных на статус вариантов для реализации.

Скорость

В случае отдельных API скорость описывает, как быстро продукт с API может быть реализован и как быстро его можно изменять и адаптировать к меняющимся требованиям. На скорость влияет весь процесс разработки и развертывания (что будет обсуждаться также в подразделе «Развертывание» далее). Поскольку очень важно использовать языки и инструменты, которые являются хорошими решениями проблемы реализации, система может помочь вам, предоставляя поддержку и инструменты для облегчения и ускорения процессов разработки и развертывания.

На скорость также влияет размер сообщества разработчиков: чем больше команд используют языки и инструменты, тем быстрее они развиваются и тем быстрее их проблемы могут быть решены. Таким образом, скорость зависит не только от выбора языков и инструментов, подходящих для решения проблемы. На нее также влияет управление разнообразием, которое обсуждалось в предыдущем пункте. Если разнообразие допускает наличие критической массы для распознавания и решения вопросов, тогда управление скоростью можно описать как выбор лучшего решения данной проблемы, *когда на уровне организации появляется критическая масса*.

Контроль версий

Методы создания версий определенно воздействуют на методы разработки API. Как сказано в подразделе «Контроль версий» на с. 197, это важный аспект систем API, и ответственный подход к контролю версий необходим, когда система усложняется в смысле размера, взаимозависимостей между сервисами и количества изменений, которые вносятся с помощью системы.

С точки зрения пользователя создание версий может быть очень полезно (когда API улучшают, чтобы предоставить сервисы, в которых пользователи заинтересованы), а может и нарушать работу системы без необходимости (когда API меняются, но пользователи не заинтересованы в изменении способа использования данного сервиса). Как сказано в пункте «Скорость» ранее, если в процессе разработки скорость позволяет быстро вносить изменения и уменьшается отрицательный эффект от новых версий, общая ценность, производимая гибкостью сервиса, максимально увеличивается.

На уровне системы важно убедиться, что за скоростью разработки можно наблюдать, таким образом, вы можете следить и за уровнем изменений и открывать доступ к информации о различных версиях. Для начала можно использовать стандартные способы осмысленной идентификации, описанные в пункте «Семантический контроль версий» на с. 197, и продемонстрировать номера версий. Это уже даст информацию о динамике всей системы API.

Изменяемость

Изменяемость сервисов в системе API, которую мы обсудили в подразделе «Изменяемость» на с. 198, создает ряд сложностей для существующих методов разработки. Главная проблема состоит в том, что по определению системы API децентрализованы и сталкиваются со всеми основными сложностями децентрализованной модели. Ответственное отношение к изменяемости систем API требует другого подхода к программированию в условиях с более прочной связанностью, а если не обратить на это внимания, пострадает общая стабильность системы, например, произойдет каскад ошибок.

В правильной работе с изменяемостью большую роль играют языки и инструменты, а также методы разработки. Таким образом, система должна распознавать и поощрять разработку, подходящую для присущей системе изменяемости. Возможно, это необходимо не во всех сценариях, но таким образом вы можете убедиться, что правильно работаете с изменяемостью.

GRAPHQL И ДОСТУПНОСТЬ API

Изменяемость можно каким-то образом изолировать, что означает, что некоторым командам придется подходить к ней более ответственно, а некоторым — менее. Так, если вы придерживаетесь схемы «серверная часть для клиентской части» (backend for frondends, BFF) (https://thght.works/2OHkpas), одно приложение для серверной части будет служить агрегатором для нескольких API, а затем передавать *один* API в приложения клиентской части, возможно, с помощью гибкой модели API, основанной на запросах, например GraphQL. В этом сценарии у клиентской части очень простая модель доступа к API: GraphQL или доступен, или нет. Однако серверной части придется столкнуться с гораздо более сложной моделью. При переводе запроса GraphQL в различные запросы к API все эти API должны считаться изменчивыми. Хорошо написанный распознаватель GraphQL сможет справиться с частичным отключением базовых API, реагируя с помощью частичных ответов GraphQL.

В этом сценарии управление изменяемости на уровне API было передано серверной части BFF, а его клиентской части нужно работать только с изменяемостью на уровне данных (если брать сценарий частичных ответов GraphQL), что значительно облегчает эту работу для команды по разработке.

Управление изменяемостью систем API как часть процесса разработки может сильно влиять на качество продуктов с API и общую стабильность системы. Необходимо убедиться, что языки, инструменты и методы разработки учитывают этот аспект, чтобы продукты начали лучше работать в постоянно изменчивой среде системы API.

Тестирование

В подразделе «Тестирование» на с. 91 мы объяснили, почему оно так важно. А также установили довольно высокую планку, заявив: «Ни один API не должен отправляться в производство без тестирования». При росте системы API это может стать сложной задачей. Поскольку время и сроки выполнения работы всегда играли значимую роль в разработке ПО, от вас могут потребовать не только ускорить процесс тестирования, но и *сократить* его, пропустив какие-то шаги или уменьшив глубину и тщательность тестирования. Но это крайне неудачная мысль. Однако сейчас вы увидите, как нужно развивать методы тестирования при росте системы.

Еще одна сложность, с которой вы столкнетесь при расширении системы API, — повышение затрат на тестирование — не только траты времени и сил на выполнение тестов, но и издержек при их *невыполнении* (например, если вы не обнаружите существующих проблем при тестировании). Другими словами, рост цены неудачи. Это может особенно беспокоить разработчиков архитектуры на системном уровне, так как неполадки в системе обычно четко видны и, если с ними не справиться, могут очень дорого обойтись вашей компании.

К счастью, опытные компании уже многократно сталкивались с этой проблемой, поэтому решения задачи о том, как проводить тестирование в системе API, уже найдены и доступны. Здесь мы выделим самые распространенные из них. Надеемся, что подскажем вам, как начать смотреть на задачи тестирования и находить решения, подходящие вашей компании. Для тестирования самыми важными аспектами системы являются объем, скорость, уязвимость и изменяемость.

Объем

Часто встречающаяся при росте системы API сложность заключается в том, что общее количество тестов, необходимых для ее охвата, начинает быстро увеличиваться. А поскольку большинство систем используют запросы к другим API, количество тестов растет *нелинейно*. При добавлении одной конечной точки API,

используемого 12 другими API, не просто добавляется набор тестов — требуется модифицировать еще 12 наборов тестов! Если ваш метод тестирования опирается в основном на тесты, управляемые человеком (то есть когда сотрудники что-то печатают на экране и записывают результаты), спрос на тестирование в растущей системе API быстро превысит возможности вашего отдела ОК.

Нелинейный рост количества тестов — один из основных поводов для того, чтобы компания начала чаще применять автоматическое тестирование. Гораздо проще масштабировать автоматические тесты, например запуская параллельно больше тестов, чем принимать новых сотрудников в команду по ОК, обучать их и наблюдать, как они проводят тесты вручную. Конечно, автоматическое тестирование требует затрат на начальном уровне, но они быстро окупятся при росте системы.

Другая сложность, связанная с объемом, — это количество трафика, необходимое для того, чтобы ваши тесты выдавали надежные результаты. На ранних этапах авантюры с API симуляция 100 запросов в секунду (requests per second, RPS) может отражать трафик производства. Однако при появлении новых команд, создающих свои API, которые начинают чаще отправлять друг другу запросы, общий объем трафика быстро раздуется. Когда трафик производства достигает 1000 RPS, по тестированию на 100 RPS уже нельзя предсказать, что произойдет после публикации этого компонента. Важно четко отслеживать уровень запросов во время производства и убеждаться, что тестовая среда все еще соответствует этим требованиям после расширения системы.

Скорость

Как мы уже упоминали, скорость тестирования — способность выполнять тесты за разумный период времени — может стать проблемой при росте системы. Проверенное правило таково: тест одной единицы или одного блока должен выполняться за несколько секунд, тест поведения или соответствия целям бизнеса — менее чем за 30 секунд, тест на интеграцию — менее чем за 5 минут, а тест на масштаб/производительность — менее чем за 30 минут. Если платформа для тестирования не может поддерживать такой темп, а команды по разработке обновляют их в стиле непрерывного изменения, вы столкнетесь с тем, что большое количество запросов в области тестирования/ОК осталось без ответа.

Есть несколько способов решения проблемы с объемом. Выделим три:

- параллельное тестирование;
- виртуализацию;
- канареечный тест.

Первый способ улучшить скорость тестирования — создать *параллельные* тесты. Самый прямой метод — распределить автоматические тесты по нескольким устройствам и запустить их одновременно. Например, если после создания каждого ком-

понента нужно запустить 35 тестов, их можно запустить подряд на одном устройстве или по одному параллельно на 35 устройствах. Если каждый тест выполняется за 10 секунд или быстрее, то при втором подходе вы переходите от тестирования, занимающего больше 5 минут, к занимающему менее 10 секунд. Это и есть скорость. Конечно, если предполагать, что все тесты можно запускать параллельно, то есть не придерживаться четкого порядка и тест 13 обязательно выполнять перед тестом 14 и т. д., что, кстати, тоже хороший метод. Параллельное тестирование помогает увеличить скорость тестирования перед запуском в производство.

Параллельное тестирование может помочь со скоростью на уровне тестов отдельных единиц и поведения, а с тестированием на взаимодействие и производительность можно справиться с помощью элементов виртуализации. Проводить тестирование нового компонента на совместимость в сравнении с другими сервисами может быть и дорого, и рискованно. А если новый компонент не справится с данными о производстве? А если при взаимодействии с другими компонентами он неожиданно навредит им? В небольших системах использование имитации сервисов в качестве замены компонентов производства хорошо масштабируется. Однако при росте системы имитациям может быть сложно успевать за скоростью изменений.

Удачное решение для улучшения скорости тестирования при сохранении безопасности — использовать *виртуальные* сервисы, обычно через общую платформу виртуализации, которая может задействовать трафик производства, а затем имитировать его по запросу в безопасной тестовой среде. Это позволяет разработчикам тратить меньше сил на синхронизацию имитаций сервисов с характеристиками и функциями производства при увеличении объемов настоящих компонентов. В качестве бонуса хорошие платформы виртуализации позволяют разработчикам создавать *синтезированный трафик*, который имитирует не только правильное поведение сервисов производства, но и искаженные или даже мошеннические взаимодействия в сети, чтобы можно было протестировать API и сервисы в неидеальных условиях. Это помогает справляться с аспектами уязвимости и изменчивости, о которых мы поговорим далее.

Еще один способ улучшить скорость тестирования — выполнить некоторые тесты уже после запуска сервиса в производство. Иногда это называют *канареечным тестом*, или *канареечным развертыванием* (https://martinfowler.com/bliki/CanaryRelease.html). В этом случае после основных тестов на поведение вы запускаете новый сервис для ограниченного набора аккаунтов (которые, возможно, захотели стать бета-тестерами). После такого частичного запуска вы отслеживаете результаты (см. подраздел «Мониторинг» далее) и, если все идет хорошо, выкатываете обновление для более широкой аудитории.

При проведении канареечного теста нужно учесть несколько важных моментов.

- Вы все равно выполняете основные тесты, чтобы одобрить этот компонент.
- Вы можете реализовать частичный запуск подраздела своей системы.

- У вас налажен механизм мониторинга для оценки влияния частичного запуска.
- Вы можете быстро — в течение нескольких секунд — откатить изменения и вернуться к предыдущей версии, не вредя функциональности и хранению данных.

Уязвимость

При росте объема (то есть увеличении количества конечных точек) и масштаба (то есть увеличении количества команд, использующих эти конечные точки) ваша система становится более уязвимой. Мы обсудим это подробнее в разделе «Безопасность» далее, а сейчас важно сфокусироваться на том, как рост системы ведет к увеличению уязвимости и что с этим делать.

Мы уже упоминали, что при масштабировании тестов нужно убедиться, что объем трафика в тестах соответствует объему производства. Это относится и к увеличению количества команд или других сервисов, которые будут использовать этот конкретный API. Ваш режим тестирования должен учитывать, что множество *разных* пользователей API будут делать запросы одновременно. Например, в некоторых случаях какое-то состояние, на которое влияют действия пользователей, передается от одного компонента к другому. В случае запуска API для более широкой аудитории стоимость поддержания этого состояния резко вырастет. Вам также может понадобиться больше тестировать, чтобы убедиться, что это состояние остается изолированным между пользователями API и что большие объемы не переполнят оперативную память при росте количества пользователей. Уязвимость может обнаружиться при увеличении объемов использования.

Она возрастает также при изменении *типа* пользователей API. В какой-то момент вашими пользователями становятся не только внутренние команды, но и ключевые внешние партнеры и даже сторонние разработчики, которых вы практически не можете контролировать. Тесты должны отражать эти изменения в системе и разрабатываться так, чтобы защитить ее и ваши данные от любых ошибочных или мошеннических действий сторонних пользователей.

В последние годы многие опираются на знаменитый «устав», в котором Джефф Безос учил свои команды в том числе тому, что «все интерфейсы сервисов без исключения должны изначально разрабатываться для внешнего использования». Хотя наш опыт работы с моделями развития свидетельствует, что вам может не понадобиться делать это в первый же день планирования API, но вы все же *должны* быть готовы разбираться с этим при росте системы.

Наконец, стоит упомянуть о том, что один из самых эффективных способов улучшить результаты тестирования — прежде всего писать код и псевдокод API так, чтобы уменьшить вероятность невыполнения теста. Поэтому, как мы узнали, многие компании, с которыми мы сотрудничаем, способные правильно подстроиться под рост объемов и масштаба, набирают в команды по разработке экспертов в об-

ласти тестирования. Другими словами, они «сдвигаются влево» (проводят тесты на ранних стадиях работы). Если добавить человека с навыками тестирования в команды, пишущие код и разрабатывающие дизайн, возможно избежать ошибок, из-за которых тесты долго запускаются и неточно отражают условия производства. Подумайте над тем, чтобы снизить уязвимость своей системы, улучшая навыки тестирования команды по разработке.

Изменяемость

В последнее время, как уже говорилось, рост системы API означает ее усложнение, а не просто увеличение в размерах. То, что было мелкой ошибкой, когда в мире API вашей компании существовала лишь горсточка конечных точек, управляемая одной командой, может вывести из строя бóльшую часть системы, если окажется, что все ваши сервисы зависят от одного из них, запущенного на одном устройстве в какой-нибудь отдаленной местности.

Большая система API рискует стать более изменчивой. Это часто происходит из-за того, что в систему незаметно проникают такие вещи, как фатальные взаимозависимости. Простой пример такой ситуации мы можем найти в кризисе из-за left-pad (http://bit.ly/2DDJTEt), произошедшем в 2016 году в сообществе Node.js. Не забираясь в дебри (если вам интересно, прочитайте статью по ссылке), можем сказать, что небольшая библиотека за короткий период времени была включена в тысячи проектов на Node.js. Из-за спора автор этой библиотеки вынудил их удалить ее из оборота, и почти сразу вышли из строя тысячи сборок, включая сборку самого Node.js! Хотя проблему нашли и исправили меньше чем за час, это событие стало суровым напоминанием о том, как большие системы становятся со временем более уязвимыми.

Подобные уязвимости возникают не только тогда, когда компоненты исчезают или почему-то оказываются недоступными. Возможно также, что API или компонент, от которого зависит ваша система, изменится, нарушив важные функции. Вы можете научить команды снижать вероятность этого, когда они обновляют код, но у вас не будет контроля над сторонними библиотеками или фреймворками, которые появятся в вашей системе. При росте системы, скорее всего, вы будете чаще пользоваться внешними API, и они станут увеличивать изменяемость системы.

Поэтому важно добавлять тесты, отыскивающие фатальные взаимозависимости и обращающие внимание на затраты из-за критических неполадок в основных компонентах. Лучше всего делать это на стадии тестирования на совместимость или производительность, когда проверяются ссылки на другие API и сервисы. Как мы уже упомянули в предыдущем разделе, лучший способ уменьшить уязвимость — добавить эксперта по тестированию в команды по дизайну и разработке, чтобы с этими проблемами можно было разобраться *до того*, как компонент или процесс будут запущены в производство.

В больших системах даже мелкие ошибки могут иметь далеко идущие последствия. Обязательно тестируйте на наличие моментов, когда важные компоненты пропадают или меняются и ими невозможно воспользоваться, и исправляйте эти проблемы.

Развертывание

Один из важнейших столпов любой программы с API — это развертывание. Неважно, какие процессы дизайна или сборки вы используете, программа не становится реальной, пока API или компонент не опубликован (см. подраздел «Стадия 2: публикация» на с. 130), а размещение — это и есть процесс публикации. На начальном этапе программы вы можете сосредоточиться на простом процессе запуска API в производство. Многие организации даже запускают их вручную (то есть щелкают на инструментах запуска, используют отбор и исполнение скриптов вручную и т. д.) на первом этапе. Но когда система начинает расти, запуск вручную становится сложно масштабировать и, таким образом, появляется новый уровень ненужной изменяемости в системе.

Самая распространенная тактика масштабирования развертывания — максимальная автоматизация процесса. Это одна из ключевых идей DevOps и движения за непрерывное развертывание. Джез Хамбл, соавтор книги «Непрерывное развертывание» (*Continuous Delivery*, издательство Addison-Wesley), говорил: «Наша цель — развертывать программы. В большой децентрализованной системе, в сложной среде производства, во встроенной системе или в приложении — неважно, это предсказуемое и рутинное дело, которое обязательно выполнять по запросу» (https://continuousdelivery.com/).

У автоматизации развертывания множество преимуществ, особенно при масштабировании развертывания в системе API. Мы сфокусируемся на четырех аспектах системы: разнообразии, скорости, контроле версий и изменяемости.

Разнообразие

В значительной части этой книги мы подчеркивали необходимость поддержки разнообразия при росте системы. Однако во время сборки и разработки разнообразие может стать реальной угрозой ее жизнеспособности и стабильности. Запуск процесса развертывания должен быть устойчивым, определенным и повторяемым. Если сегодня ваша команда выполняет процесс `installOnboardingAPIs`, он должен выдать *точно такой же результат* и если его запустить через несколько дней. Развертывание не должно варьироваться.

Для этого нужно удалить разнообразие из системы. В процессе сборки и развертывания полезно применять такие подходы, как «Шесть сигм», «Кайдзен», «Бережливая разработка» и т. д., то есть сфокусироваться на удалении из релиза не очень важных вариаций и убедиться, что технология развертывания собирает

все артефакты релиза (код, конфигурацию и т. д.) в одном месте для облегчения публикации. Нужно также внимательно отслеживать операционную систему и другие взаимозависимости, чтобы они оставались неизменными при каждом повторе одного и того же развертывания. Хорошие платформы CI/CD позволяют разработать и реализовать надежный и повторяемый процесс развертывания.

«ШЕСТЬ СИГМ», «БЕРЕЖЛИВАЯ РАЗРАБОТКА», «НЕПРЕРЫВНОЕ УЛУЧШЕНИЕ»

Существует некоторое количество моделей, нацеленных на непрерывное улучшение и удаляющих разнообразие при улучшении качества. Самыми известными из них, вероятнее всего, являются «Шесть сигм», «Бережливая разработка» и «Непрерывное улучшение», а у каждой из них есть несколько вариаций (например, «Бережливые шесть сигм» и т. д.). Если у вашей компании еще нет программы, работающей по этим моделям, мы рекомендуем рассмотреть их. На Formaspace есть интересная статья (http://bit.ly/2QOUcbG), в которой сравниваются три основные модели, — можно начать их изучение с нее.

Хотя важно удалить разнообразие из процесса развертывания, вам все равно может понадобиться поддерживать различные наборы инструментальных средств CI/CD. Нет жестких требований к тому, чтобы во всех частях вашей организации от центрального сервера до КПК по всему миру использовалась *одна и та же платформа* для развертывания. Однако советуем максимально ограничить количество вариантов платформ, поскольку большинство из них требует больших вложений времени и денег.

Скорость

Ускорение процесса развертывания часто считается главной целью компаний, которые трансформируют свои IT-процессы. При расширении системы нужно принимать во внимание два аспекта скорости развертывания. Первый из них мы называем *типом 1*: сокращение промежутков между релизами одного API/компонента. Второй называем *типом 2*: наращивание общей скорости всех циклов релиза в вашей IT-группе.

Думая о скорости развертывания, большинство людей представляет себе первый случай (тип 1). Он очень важен. Мы часто обсуждаем с пользователями снижение количества случаев появления петли от обратной связи к характеристике или ускоренное продвижение от идеи к установке (для новых проектов). Более быстрое развертывание может снизить риски и затраты на эксперименты с новым продуктом или сервисом, что улучшит способность вашей компании обучаться и внедрять инновации. А автоматизированное и детерминированное развертывание поможет увеличить скорость релиза.

Второй случай (тип 2) значительно отличается от первого. Вам нужно будет запускать в производство *больше* программ за тот же период времени. Для этого нужно, чтобы больше команд выпускали релизы, больше релизов запускалось в производство и больше изменений вносилось в систему. Если вы опираетесь на одну центральную команду по релизам для развертывания всей продукции, вам будет сложно набрать скорость развертывания второго типа. Существуют пределы расширения одной команды по релизам.

Лучший подход — распределять ответственность за релизы среди более широкого сообщества разработчиков. Это еще одна причина, по которой автоматизация как можно большего объема процессов релиза крайне полезна. Чем значительнее автоматизация, тем больше людей смогут сосредоточиться на крайних случаях и ожиданиях, чтобы все работало как следует и единообразно. Так же как и в случае других столпов, описанных в этой главе, можно безопасно ускорить процесс, децентрализовав его или запустив параллельные процессы. Результатом будет большее число релизов в целом без ускорения каждого отдельного цикла релиза.

Децентрализованное управление релизами в компании Etsy

Майк Бриттейн, технический директор компании Etsy, создал интересный набор слайдов и видеопрезентацию об их версии децентрализованных релизов под названием «Децентрализованное управление релизами» (http://bit.ly/2qMSUmf). Если вы хотите реализовать эту идею, советуем начать с презентации Бриттейна.

Ускоряя развертывание, обязательно помните о ценности скорости первого и второго типов.

Контроль версий

Контроль версий в целом обсуждался в главе 8 (см. подраздел «Контроль версий» на с. 197). Хотя мы заявили, что при разработке дизайна и реализации следует избегать создания версий *внешнего интерфейса* API, с *внутренним интерфейсом* дело обстоит совсем иначе. Пользователей API не стоит беспокоить исправлением ошибок или мелкими, не нарушающими их работы, изменениями. Их следует тревожить, только если работа интерфейса нарушается и/или становятся доступными новые функции. Но внутренние пользователи (разработчики, дизайнеры, специалисты по архитектуре и пр.) должны иметь возможность видеть каждое мелкое отклонение и изменение в пакете релиза, даже такие небольшие, как изменения в ресурсах поддержки, например в логотипах, и т. п.

Убедиться, что вы открываете доступ к небольшим изменениям в пакетах развертывания, можно с помощью *создания версий релиза*, используя схему семантического контроля версий (см. пункт «Семантический контроль версий» на с. 197): СТАРШАЯ

(серьезные изменения), МЛАДШАЯ (новые функции, совместимые с предыдущими версиями) и ПАТЧ (без изменений в интерфейсе, исправление ошибок). Иногда пользователи добавляют уровень РЕЛИЗ (то есть СТАРШАЯ.МЛАДШАЯ.ПАТЧ.РЕЛИЗ). С таким дополнительным значением проще отследить каждую сборку и/или цикл релиза до малейшего изменения. А это может быть важно, когда релиз продукта ведет себя неожиданно. Возможность отследить пакет, используя номер релиза, может быть очень полезна при определении его отличительных черт.

Большинство инструментов для релиза позволяют присваивать дополнительный номер сборки каждому релизу. Таким образом, вам не нужно править схему семантического контроля версий, при этом вы все равно получаете подробное отслеживание каждого пакета сборки и производства. Что бы вы ни решили делать, помните, что *внутренние* релизы получают детальные идентификаторы, а вот идентификаторы *внешних* релизов следует менять, только если вносятся серьезные изменения.

Изменяемость

Как вы могли подумать, увеличение скорости развертывания и по типу 1, и по типу 2 (см. пункт «Скорость» ранее) вызывает риск роста общей изменяемости системы. Поэтому многие организации пытаются замедлить темп релиза. Однако обычно так поступать не стоит. Вместо этого, чтобы увидеть, что развертывание не вносит в систему неожиданную изменяемость, можно сделать три вещи:

- убедиться, что изменения в релизах не нарушают работу программы;
- поддержать определенные пакеты развертывания без вариаций;
- поддержать мгновенную обратимость установки.

Как говорилось в подразделе «Контроль версий» на с. 197, развертывание должно *избегать создания версий*, когда это возможно, в том смысле, в каком это видит большинство из нас. Наш опыт свидетельствует: можно вносить важные изменения в работающую систему, не нарушая ее работы. Джейсон Рэндольф из GitHub называет такой подход эволюционным дизайном и объясняет его ценность так: «Когда люди создают что-то на основе нашего API, мы просим их доверять нам, ведь они вкладывают время в создание своих приложений. И чтобы заслужить их доверие, мы не можем вносить изменения [в API], которые могут нарушить работу их кода» (https://events.yandex.com/lib/talks/42/).

Как говорит Рэндольф в своей лекции о GitHub, можно использовать элементы *дизайна*, чтобы облегчить внесение изменений, не нарушающих работу программы. Можно также создать тесты (см. подраздел «Тестирование» на с. 91), проверяющие наличие подобных изменений, и включить их в процесс сборки, чтобы уменьшить вероятность нарушения производства. Принимая обет не нарушать работы программы, вы ограничите возможность появления изменяемости при масштабировании развертывания.

Еще один ключ к снижению изменяемости при развертывании — убедиться, что каждый пакет релиза самодостаточен и, как мы уже говорили ранее в этой главе, определен (см. подраздел «Разнообразие» на с. 250). Когда вы способны предсказать результаты развертывания (например, когда уверены в том, на какие элементы производства повлияет этот релиз), можете снизить вероятность неожиданностей в обновлении. Кроме того, ответственный за развертывание пакета в производстве должен иметь возможность запустить релиз в производстве или другой среде (на устройстве разработчика, тестовом сервере и т. д.) и получить те же самые результаты. Это крайне важно для тестирования релиза и поиска ошибок совместимости и других внешних неполадок, которые могут появиться в производстве.

Наконец, важный аспект изменяемости развертывания связан с *обратимостью*. Как мы уже говорили, скорость второго типа, где *больше* общее количество релизов, может угрожать стабильности, увеличивая изменяемость. То, что вы убедитесь, что изменения не нарушают работу системы, а развертывание не имеет вариаций, поможет снизить количество неполадок, но не предотвратит их на 100 %. Если в системе появляется неожиданная ошибка, важно иметь возможность моментально отменить изменение, откатить его в течение нескольких секунд. Это решение на самый крайний случай. Откат изменения означает, что собранные или сохраненные данные не повреждены. Другими словами, все ваши развертывания должны рассчитывать на обратимость любых изменений в схеме или модели данных. Для этого нужно изменить способ обновлений разработки дизайна и реализации.

Безопасность

В подразделе «Безопасность» на с. 98 мы говорили о том, как важны основные элементы безопасности (идентификация, распознавание и авторизация). А также обсудили, как снизить количество уязвимых мест, которые могут стать мишенью атаки, и добавить в каждый компонент и/или API, запущенный в производство, устойчивость и изоляцию (см. подраздел «Уязвимость» на с. 194). Все эти элементы становятся все важнее с ростом системы. И разумеется, с каждым из них возникают проблемы. Безопасность — это всеобъемлющий и сложный предмет, здесь мы не сможем достаточно глубоко погрузиться в него. Однако выделим три самых важных аспекта системы и обсудим, как они влияют на общую безопасность системы.

Скорость

Скорость изменений затрудняет поддержание безопасности в расширяющейся системе API. Добавляются новые компоненты, обычно все более быстрыми темпами, а также новые взаимосвязи и новые пользователи. У многих наших клиентов инфраструктура безопасности требует определения явного контроля доступа, прежде чем компонент или интерфейс будет запущен в производство. Это работает, когда темп изменений и охват системы не слишком высоки. Но когда система растет в масштабе и объемах, определение контроля доступа на раннем этапе способно

стать узким местом. Оно может сдерживать релизы и замедлять выпуск новых функций и исправление ошибок.

Распространенный способ справиться с проблемой скорости в смысле безопасности — убедиться, что компоненты разработаны для безопасной работы, даже если профили контроля доступа еще не на месте.

Еще один способ — ввести автоматическое тестирование безопасности. Создание скриптов для тестов по безопасности не идеальное решение (с помощью скриптов трудно тестировать на поведение в случае попыток мошенничества), но может помочь. Тестирование безопасности во время цикла сборки поможет найти проблемы на ранней стадии, снизить затраты на их исправление и вероятность ущерба при запуске.

Уязвимость

При росте системы API увеличивается поверхность, а следовательно, и уязвимость. Если много команд быстро выпускают много компонентов, становится очень трудно следить за возможными уязвимыми местами, появляющимися в системе. Как мы только что сказали, может помочь добавление тестов на безопасность в фазе сборки, но это не единственное, что можно сделать, чтобы разобраться с растущей уязвимостью системы API.

Когда релиз каждого компонента рассматривается как единичное событие для безопасности, контролировать, одобрять и отслеживать уязвимые места может быть невероятно трудно. Важный способ справиться с увеличением масштаба и объема заключается в следующем:

- опираться на общую страховку в качестве старта для профилей безопасности на уровне системы;
- переложить ответственность за отслеживание и доклад о действиях в области безопасности на уровень, максимально близкий к командному.

Если вы опираетесь на реализации безопасности, зависящие от алгоритма, а не от кода, то получаете несколько преимуществ. Во-первых, описательные алгоритмы проще читать и отлаживать, чем предписывающий код. Во-вторых, большинство прокси-серверов позволяют относиться к любым алгоритмам как к единицам, которые можно использовать повторно, а затем сочетать в мощном пакете профилей, который гораздо проще контролировать и отслеживать. Наконец, многие платформы безопасности позволяют управлять алгоритмами с помощью скриптов и/или инструментов командной строки, совместимых с системами CI/CD, чтобы сделать свой алгоритм безопасности элементом пакета релиза, научиться работать с которым должны все команды.

Это приводит нас ко второй части подхода — к тому, чтобы перекладывать ответственность за отслеживание и доклады на команды разработчиков. Вряд ли одна команда по безопасности (особенно находясь на уровне корпорации на глобальном

предприятии) сможет достаточно успешно контролировать события на уровне компонентов и реагировать на них. Гораздо более разумно, если команда, разработавшая и выпустившая этот компонент или API, будет следить за ним. Однако делать это качественно они могут только при наличии соответствующих инструментов и поддержки. При росте системы важно сменить централизованную экспертизу на децентрализованные инструменты и методы. Для этого можно перевести некоторых экспертов по безопасности в команды по разработке, как мы рекомендовали в подразделе «Тестирование». Другой способ расширить навыки безопасности — создать такие инструменты, как методы на уровне дизайна, тестирование на уровне сборки и компактное отображение информации на уровне производства. Инвестируя в инструменты, чтобы масштабировать существующие знания в области безопасности, вы можете успешно расширить область, которую способны охватить, и улучшить общую безопасность системы API.

Видимость

Все это подводит нас к последнему аспекту, который мы хотим здесь подчеркнуть, — видимости. В мире безопасности вам может повредить то, о чем вы не знаете. Все предусмотреть нельзя. Но можно вместе с такими методами, как «нулевое доверие», правила безопасности, основанные на алгоритмах, и тесты на безопасность во время сборки, добавить и повышенную видимость с помощью ведения системных журналов и компактного отображения информации.

Информационная панель предлагает обзор активности в сети в реальном времени. Благодаря ей возможность увидеть свои интерфейсы и компоненты в действии появляется у команд из всех отделов компании, в том числе у команд по безопасности. Вначале информационные панели полезны, потому что делают видимым общий трафик вашей сети. Со временем команды могут создать свои фильтры, чтобы сфокусироваться на том трафике, который, как они узнали, является важным показателем правильной или неправильной работы системы.

Часто точки ввода данных, которые появляются на информационных панелях, — это своего рода KPI и OKR (см. подраздел «OKR и KPI» на с. 123), которые мы обсуждали ранее. Команды по безопасности могут сами определить ключевые значения, за которыми стоит следить, и упростить передачу этой информации в реальном времени для всех команд разработчиков. Чаще всего эту информацию получают от шлюзов и прокси-серверов, используемых по всей системе, но иногда необходимо разработать индивидуальные компоненты, чтобы собирать и показывать важные параметры запросов на распознавание, одобрение, разрешение доступа, отказ в доступе и т. д. Предоставляя командам разработчиков точки ввода данных и шаблоны, которые эксперты по безопасности ожидают увидеть, и обеспечивая согласованность на уровне компонентов по время тестирования перед запуском, вы добьетесь того, что ваши действия в области безопасности будут соответствовать новому масштабу и объему растущей системы API компании.

Обычно системные журналы можно использовать как часть анализа произошедшего, которая поможет вам и вашей команде по безопасности понять, что произошло, и, скорее всего, подскажет, как предотвратить появление таких же проблем в будущем. Если ввести практику устойчивого и надежного ведения системных журналов всеми командами по разработке, эта информация точно будет зафиксирована для возможного использования в обсуждении после произошедшего события. Но недостаточно просто собирать информацию в форме системных журналов. Нужно также убедиться, что она хранится в форме, облегчающей доступ к данным, фильтрацию и корреляцию, чтобы вы могли найти и изучить только нужные записи. Для этого можно применить метод *децентрализованного сбора* (в этом случае каждая команда отвечает за сбор информации по результатам отслеживания) и *централизованного хранения* (есть единая платформа, куда посылают все системные журналы, чтобы позже отфильтровать и изучить их). Центральное руководство по безопасности также может предоставить рекомендуемые требования точки ввода данных и действия по отслеживанию, а еще можно написать тесты на уровне сборки, чтобы убедиться, что работа всех команд согласована с этим руководством, прежде чем запустить ее в производство.

Мониторинг

У мониторинга несколько полезных целей, включая поиск узких мест, отслеживание внутренних KPI и внешних OKR и предупреждение об аномалиях в работе программы, например о необычных пиках трафика, неожиданных разрешениях на доступ и др. На раннем этапе существования вашей программы с API мониторингом можно заниматься с помощью немногих специальных инструментов и нескольких простых информационных панелей и методов изучения системных журналов. Однако, как и в случае многих других столпов API, с ростом системы проблемы объема, видимости и изменяемости могут превысить возможности ваших инструментов. Мы назовем несколько распространенных проблем и их возможных решений.

Объем

Распространенная сложность, возникающая в области мониторинга при расширении системы API, заключается в том, что общий объем информации перерастает вашу способность с ней справляться. Обычно это происходит, когда у организации есть централизованная модель управления мониторингом, где небольшая группа людей с навыками мониторинга выполняет задачи по сбору данных системных журналов — как поступающих в реальном времени, так и старых, управлению ими и их интерпретации. В какой-то момент одна команда уже не может справиться с объемом, и увеличение этой центральной команды не улучшает положения. Вместо этого, как мы предложили в пункте «Уязвимость» на с. 255, можно делегировать задачи по мониторингу командам, разрабатывающим API и компоненты. Первый шаг в этом деле — распределение работы по сбору данных отслеживания и управлению ими. Как только у команд появляются собственные данные отслеживания, они могут

начать разрабатывать фильтры и корреляции для получения полезной информации об этих данных.

Однако мониторинг одного компонента или одного API — это лишь часть задачи. При росте системы важно контролировать и взаимодействие между этими компонентами. Хорошая тактика при решении новой задачи — создание центрального хранилища данных отслеживания, где можно сфокусироваться на корреляциях между разными API в вашей организации. В этом случае центральное хранилище данных часто является отфильтрованным и коррелированным подразделом (обычно сжатым по времени) всех данных отслеживания команд различных компонентов или API. Централизованные данные — это избранное, выдержка из всего объема, которую можно использовать, чтобы увидеть повторяющиеся схемы и аномалии. После того как вы их определили, команды могут использовать детали отслеживания, чтобы искоренить проблемы и подтвердить информацию.

Отметьте, что для этого метода нужно:

- дать командам задание по сбору данных отслеживания и управлению ими;
- установить центральное хранилище, отбирающее отфильтрованные данные из хранилищ отдельных команд;
- при появлении тенденций или проблем на центральном уровне опираться на детали на уровне команд, чтобы подтвердить или решить любые проблемы.

Видимость

Метод создания центрального хранилища для отфильтрованных или коррелированных данных — основной элемент поддержания и даже улучшения видимости в области мониторинга. В подразделе «Безопасность» на с. 254 мы говорили о том, что невозможно предусмотреть все возможные проблемы в сфере безопасности. Отсюда можно сделать важный вывод: невозможно знать все точки ввода данных, которые надо отследить. Поэтому попытки ведения системных журналов и отслеживания на ранних этапах часто включают в себя множество значений, у которых нет известной вам связи с текущим состоянием вашей сети. Однако связь есть, просто такая, о которой человек в принципе не может узнать. Поэтому команды могут урезать сбор данных, выбрасывая якобы несвязанные значения из записей. Обычно это неудачное решение.

Вместо этого разумнее записывать все, а отслеживать только отдельный набор точек ввода данных. Это соответствует предыдущему совету — поощрять команды собирать и хранить свои данные, при этом позволяя центральным механизмам мониторинга отбирать отфильтрованные и коррелированные версии этих данных в общий набор информационных панелей, открытых для всех. Пока команды видят более детальный, но ограниченный образ системы, централизованный мониторинг

позволяет видеть ту же систему шире, но менее детально. Это простой пример принципа неопределенности Гейзенберга (http://bit.ly/2QHHnQF).

ПРИНЦИП НЕОПРЕДЕЛЕННОСТИ ГЕЙЗЕНБЕРГА

В 1927 году Вернер Гейзенберг опубликовал работу по физике, в которой содержалось следующее наблюдение: *чем более точно определено положение частицы, тем менее точно мы знаем ее импульс и наоборот*. В мире IT этот принцип обычно применяется в качестве компромисса между детальным пониманием небольшой части системы и знанием о том, как эта часть влияет на всю систему. Попытки получить одновременно и детальное понимание каждой части, и полное представление об их взаимодействии при росте системы становятся все менее и менее результативными. Поэтому мы советуем командам изучать подробный образ системы, а группам текущего мониторинга — широкий и менее подробный.

Еще один важный аспект видимости в системах мониторинга — это связанное с ней качество наблюдаемости. При росте системы становится все сложнее наблюдать за ее поведением. Можно использовать мониторинг, точнее, публикацию данных отслеживания, чтобы улучшить общую наблюдаемость. Неожиданные результаты, ошибки и другие странности часто появляются, потому что человек не видит, как работает система и/или как компоненты системы связаны друг с другом. Наш опыт свидетельствует: в случае обнаружения странной ошибки в сложной системе люди склонны говорить что-то наподобие: «Я и не знал, что такое возможно» или «Я не думал, что это так работает». Улучшение мониторинга и компактное отображение информации не предотвращают неполадки, но могут помочь не очень удивляться, когда что-то пойдет не по плану.

Изменяемость

Наконец, снова повторим то, что заявили в подразделе «Безопасность» на с. 254: по нашему опыту, в больших системах более вероятен высокий уровень изменяемости. Это наблюдение сделал специалист по теории вычислительных машин и систем Мэл Конвей в своей работе 1967 года «Как комитеты создают новое?» (http://bit.ly/2RSyMKS): «Структуры больших систем стремятся к распаду... значительно чаще, чем структуры небольших систем».

Хотя это наблюдение относилось прежде всего к фазе разработки больших проектов, мы видим то же поведение в работе больших систем. Одна небольшая ошибка способна нарушить работу всей системы, а с ростом последней такие нарушения обходятся все дороже и дороже. Из-за этого некоторые компании могут предположить, что качество их систем со временем падает, но на самом деле ошибки, скорее всего, всегда там были. Просто, когда вся система стала больше, увеличилось их влияние, а с ним и опасность. Вы можете снизить риск нарушения работы всей

системы из-за одной ошибки и лучше увидеть качество всей системы, если будете поддерживать надежную программу мониторинга.

Обнаружение

Как мы обсуждали в разделе «Обнаружение и продвижение» главы 4, этот столп жизненного цикла API состоит из всех действий, которые помогают пользователям найти и использовать API. Для отдельных API нужно понимать, в каком окружении их можно будет обнаружить, и упрощать этот процесс.

В сложных и постоянно растущих системах API обнаружение развивается с той же общей динамикой, что и сайты, и веб-страницы. Поначалу достаточно было иметь тщательно составленный список сайтов и страниц, распределенных по категориям в попытке обеспечить их нахождение. Этот подход использовал поисковик Yahoo! как изначальный механизм обнаружения в Сети, и он хорошо работал, когда количество сайтов и страниц было относительно небольшим, как и уровень изменений, и достаточно было одной схемы деления на категории для всей информации в Интернете.

Разумеется, этот подход плохо масштабировался и был быстро забыт из-за невероятного роста Интернета, частоты его изменений и разнообразия содержимого. Начавший действовать в 1996 году Google, который изначально назывался *BackRub* и был сервисом, предоставляемым только на территории Стэнфордского университета, радикально изменил обнаружение, введя два главных изменения.

- *Поиск по содержимому* заменил поиск по категории, то есть вместо того, чтобы опираться на схему деления на категории, созданную сторонними разработчиками, поисковик задействовал полнотекстовый поиск по самому содержимому.
- *Ранжирование по популярности* заменило ранжирование, выполняемое вручную, сменив решения сторонних разработчиков о том, как нужно сортировать содержимое, на вычисление его релевантности по популярности в Сети, полученной благодаря внутренним ссылкам.

Конечно, история о том, как развивалось обнаружение в Интернете, гораздо длиннее, но важно помнить об общей динамике. Хотя у этого подхода и есть издержки (популярные сайты становятся все более популярными, а менее популярные найти все труднее), он в целом работал достаточно хорошо для того, чтобы пользователи сочли его полезным, поэтому бóльшая часть задач по обнаружению в Интернете сегодня выполняется по данной общей модели.

В мире API у *содержимого* другое значение, поскольку API сами по себе не столько информация, сколько *описания сервисов*. Однако *популярность* — это концепция, которую довольно просто перенести в мир API, и график зависимости API вполне может служить эквивалентом структуры ссылок в Интернете.

Пока нет четких признаков того, кто может стать Google в мире API. И поскольку многие системы API в основном содержат внутренние или партнерские, а не внешние API, не совсем понятно, возникнет ли для API та же динамика обнаружения, какая появилась для Интернета. Но для того, чтобы обнаружение и продвижение могли масштабироваться, важно, чтобы релевантная информация была доступна в таком виде, который могут использовать автоматика и инструменты. Главные аспекты системы в этой области — разнообразие, объем, словарь, видимость и контроль версий.

Разнообразие

Документацию API можно создавать в разных формах, и форма продиктована решениями по инвестициям в этой области жизненного цикла. Инвестиции зависят от развития API и предполагаемой целевой аудитории. Кроме того, они зависят от поддержки и инструментов, которые могут стать доступными на уровне системы.

С ростом популярности и важности API появились высокотехнологичные решения для документации и обнаружения API. Обычно их наборы сочетают в себе аспекты документации, обнаружения, создания кода и другие элементы опыта разработчика. Хотя такие наборы очень полезны, необходимо помнить, что в них обязательно встроены какие-то предпочтения (например, по стилям API) и, как и любые инструменты, их следует использовать так, чтобы можно было при необходимости увеличить или удалить.

Идеальное решение, позволяющее убедиться, что инструменты помогут при обнаружении и при этом оно останется открытым и декларативным, — выразить всю нужную для обнаружения информацию *с помощью самого API* (см. подраздел «Выражать API через API» на с. 183), а затем позволить поддержке и инструментам собрать эту информацию и использовать ее для обнаружения с помощью системы. Это соответствует общему принципу раскрытия всей релевантной информации через сам API. Мы обсудим это подробнее в пункте «Словари» далее.

Объем

С ростом объема системы API важно обеспечить легкость обнаружения как аспект, который поможет не только *найти* API, но и *ранжировать* их. Находить их необходимо, но недостаточно, и это понимает каждый, кто когда-либо использовал поисковик в Интернете, который выдает миллионы ссылок на один поисковый запрос.

Поэтому для обнаружения нужны не только способы найти API, но и способы их лучше понять и ранжировать. Вероятно, понимание того, каким должно быть полезное ранжирование, будет со временем развиваться в любой системе API. Изначально оно может быть даже ненужным, поскольку API еще мало. Затем из-за растущего объема API такая необходимость возникает. Полезные способы

ранжирования тоже могут со временем изменяться, поскольку объем растет и определение того, что лучше всего соответствует поисковому запросу, развивается.

Чтобы иметь возможность расти по оси постоянно меняющегося и улучшающегося обнаружения, важно помнить, что API в любой момент должны выдавать максимум информации о себе и эта информация может постоянно эволюционировать, поскольку системе нужно, чтобы API со временем менялись.

С точки зрения системы важно предоставлять поддержку и инструменты, облегчающие обнаружение в системе API. В зависимости от модели обнаружения этот набор задач может удивительно походить на оптимизацию сайта в поисковых системах (Search Engine Optimization, SEO) в Интернете, где существует баланс между информацией, которую могут собрать и использовать сервисы по обнаружению, и уровнем сотрудничества, которого хотят отдельные партнеры.

Словари

При развитии обнаружения API становится проще найти, и, поскольку основной принцип, который мы здесь обсудили, соответствует схеме перехода обнаружения в больших структурах с модели категоризации Yahoo! на полнотекстовый поиск и модель ранжирования, основанную на популярности Google, полезно будет рассмотреть и некоторые другие веб-разработки.

В 2011 году в рамках сотрудничества основных поисковых систем — Bing, Google и Yahoo! — был основан сайт Schema.org (https://queue.acm.org/detail.cfm?id=2857276) в качестве попытки создать единую схему по ряду тематик, включавшему различных людей, места, события, продукты, предложения и т. п. Главная цель этого проекта — позволить людям, публикующим информацию в Интернете, пометить ее так, как они считают нужным, и позволить поисковикам использовать эти метки как дополнительный вклад в алгоритмы поиска и ранжирования.

Стоит отметить: все это применимо к информации в Сети, а не к API. Однако главное — *принцип*: Schema.org продолжает развиваться в качестве словаря терминов в Интернете, которые можно использовать, чтобы пометить свою информацию. Это стало возможным, поскольку сам словарь и его использование разъединены. С развитием словаря информацию можно помечать более сложными способами. Создание и использование терминов этого словаря связаны слабо.

Можно установить похожие принципы и для систем API. Система может поддерживать и продвигать использование каких-либо терминов, чтобы облегчить обнаружение, а еще обеспечить поддержку для создания маркированного содержимого (например, в документации и/или базовых документах API) и его одобрения. В ходе разработки можно даже автоматизировать тесты на наличие этих терминов, например, «стиль API», и если такой термин не найден, создать предупреждение и попросить команду по API добавить эту информацию, чтобы ее можно было найти.

Видимость

Как говорилось в пункте «Объем» на с. 261, один из главных аспектов, о которых надо помнить, думая об обнаружении в системах API, — это объем. И как мы там увидели, главный способ управлять объемом — сделать видимой через API столько информации о нем, сколько необходимо. Конечно, набор информации со временем будет меняться, поэтому главное, что надо принимать во внимание с точки зрения видимости, — следует начинать с малого и иметь план, как увеличить количество информации, раскрытой в отдельных API.

Помня о том, что набор видимой информации обычно эволюционирует со временем, можно непрерывно совершенствовать обнаружение в системе API так же, как оно постоянно улучшается в Интернете. Легкость обнаружения никогда не бывает окончательной, а достичь ее можно с помощью увеличения информации, доступной инструментам для обнаружения, наблюдения за развитием системы и размышлений о том, какие изменения лучше всего помогут улучшить обнаружение.

Контроль версий

API в системе обычно меняются, и одна из основных целей системы API — упрощение этих изменений. Возможность вносить изменения, не нарушая работы пользователей, с каждым новым релизом — тоже цель многих API и систем API. Если применять качественные методы управления изменениями, можно четче разъединить создателей и пользователей API и позволить им развиваться независимо друг от друга.

В идеале команды по API могут развивать свои продукты и выпускать новые версии в собственном темпе. Так проще для команд, однако пользователям сложнее уследить за выходом версий, найти старые версии и понять, в чем между ними разница.

Системы могут упростить понимание API и их версий, если будут требовать документировать все версии. Если доступ к их истории реально получить с помощью API, ее будет проще найти. Пользователи смогут обнаружить информацию о версии API и, возможно, понять, как он развивался, с помощью истории продукта.

При серьезных изменениях API, когда реализация, возможно, стала совершенно иной и труднее обеспечить обнаружение документации и версий, данный аспект становится сложнее. В этих случаях может быть полезно обеспечить поддержку, чтобы командам по API не приходилось тратить дополнительные усилия на сохранение устаревших версий. Вместо этого они смогут положиться на систему, оставив доступной лишь часть информации или сообщив пользователям, что старая версия все еще существует, но детальные сведения получить уже нельзя.

Управление изменениями

Одним из важных столпов жизненного цикла API является *управление изменениями*, как говорилось в подразделе «Управление изменениями» на с. 104. Часть общего пути API следует проделать, чтобы уменьшить нарушения работы системы API при развитии API. Часто для этого нужно придерживаться принципов управления изменениями, которые могут вращаться вокруг моделей расширения и внесения в соответствии с ними только безопасных изменений. Другой подход заключается в том, чтобы никогда не менять запущенные в производство API, а вместо этого создать модель операций, которая позволяет запускать много разных версий параллельно.

Планирование изменений — это один из центральных вопросов систем API и их непрерывной эволюции, поэтому управление изменениями сильно зависит от того, как система API поддерживает изменения с помощью руководства и инструментов. Чтобы помочь отдельным API с управлением изменениями, необходимо принимать во внимания аспекты словаря, скорости, видимости и контроля версий. Мы обсудим это в следующих пунктах.

Словари

При управлении изменениями в API важно помнить о влиянии *эволюции словаря* на *эволюцию API*. Словарь — это распространенный и известный аспект дизайна API, а разработка словаря в API всегда подразумевает, что его обновления приведут к обновлениям API. Хотя при правильной модели расширения эти обновления могут не нарушать работы системы, они все же запускают целую цепь обновлений API и связанных с ним ресурсов и, возможно, провоцируют такие же действия у пользователей API.

Управление словарем также можно отделить от API, превратив его в ресурс, который может развиваться, поэтому API может оставаться стабильным даже при изменениях в словаре. Как сказано в подразделе «Словарь» на с. 188, распространенный способ сделать это — дать ссылку на реестр, либо управляемый внешним органом власти, либо являющийся частью системы. В этой модели изменения в словаре не обязательно запускают изменения в API, кроме того, так проще *делиться словарем* с другими API.

Поэтому имеет смысл подумать о словарях на системном уровне. Это может поддержать и упростить работу команд по дизайну и развитию отдельных продуктов с API. В каком-то смысле идея поддержки словаря на системном уровне связана с концепцией *словаря базы данных*, но этот термин чаще всего применяют к конкретным аспектам схем баз данных, а словари, по идее, независимы от реализации и являются просто самоуправляемыми типами данных, которые могут повторно использоваться различными API.

Скорость

На внесение изменений должны влиять в основном планирование и циклы продукта, основанные на обратной связи и постепенном выпуске функций. Управление необходимо, чтобы упростить этот процесс, но прежде всего оно не должно препятствовать изменениям. Скорость, как сказано в подразделе «Скорость» на с. 217, — это одна из главных целей перехода на API и системы API, и важно понять, как система помогает или мешает увеличить скорость.

Команды продуктов с API нужно поощрять к тому, чтобы они сообщали, ощущают ли, что их ускоряли или замедляли. Эти отзывы необходимо учитывать при создании руководства по улучшению управления изменениями, а также поддержки и инструментов на уровне системы. Управление изменениями — это один из столпов, к которым надо всегда относиться серьезно, в том числе из-за его прямой связи со скоростью.

Опять же важно помнить о контексте управления изменениями и по возможности сдвигать применение более сложных способов на поздние стадии развития API. Ответ на вопрос: «Важно ли кому-то, что меняется API в системе?» — сильно зависит от того, кто зависит от самого API.

Если (пока) API никто не использует, можно поспорить о том, что любое управление изменениями в этот момент является пустой тратой времени и поэтому влияет на скорость. Однако после появления пользователей образуется парадокс: при большем объеме использования хорошее управление изменениями помогает удерживать скорость, но менять API для поддержки самого управления изменениями становится сложнее. Таким образом, чтобы поддерживать скорость, в особенности на поздних стадиях развития, полезно инвестировать в управление изменениями и поддерживать его на системном уровне с самого начала.

Контроль версий

В целом API должны соответствовать модели семантического контроля версий, описанного в пункте «Семантический контроль версий» на с. 197. Не обязательно применять эту схему в точности, но полезно разделять уровни изменений.

- В случае версий-*патчей* изменений на уровне API не наблюдается, поэтому пользователи ими интересуются, только если хотят проверить, изменили ли реализацию, чтобы исправить ошибку.
- В случае *младших* версий, которые по определению совместимы с предыдущими, пользователям может быть интересно узнать об изменениях, но также они могут решить игнорировать их, если состояние сервиса на данный момент их устраивает.
- В случае *старших* версий пользователи должны что-то предпринять, поскольку те вносят изменения, нарушающие работу программы. Необходимо уведомлять пользователей о выпуске старших версий, а также о том, когда текущая версия будет удалена.

Существует много разных способов работы с этими механизмами с помощью отдельных API. Поскольку управление изменениями и взаимозависимостями играет важную роль в устойчивости системы API, советуем в какой-то степени согласовать эти два вопроса, чтобы пользователям API было проще справляться со скоростью изменений в этой системе.

В конце концов, возможность менять и обновлять API на высокой скорости очень полезна, если контролировать негативные побочные эффекты.

Видимость

Управлять видимостью сложно, когда речь идет об управлении изменениями в системах API. В целом пользователи API хотели бы работать с API так, чтобы их как можно реже беспокоили, за исключением сообщений о новых функциях, которые им нужны, — в этом случае они готовы инвестировать в адаптацию своего API под использование изменений. Если сделать управление изменениями видимым и таким образом позволить пользователям писать код, который может реагировать на эту информацию, это поможет улучшить устойчивость системы API.

Как мы сказали в предыдущем разделе, важно встроить управление изменениями в систему API, чтобы скорость изменений не нарушала работу сервисов чаще, чем необходимо. Один из главных вопросов, которые мы обсудили, — *что именно* делать видимым. Рекомендуется использовать модель, отражающую схему семантического контроля версий: *патчи*, *младшие* и *старшие* версии.

С точки зрения безопасности можно посоветовать не демонстрировать версии-патчи внешним пользователям и даже партнерам. В конце концов, эти изменения должны не менять сам API или его поведение, а только решать проблемы в реализации. Младшие версии нужно делать видимыми, поскольку это помогает пользователям понять, что они доступны. У пользователей *всегда* должен быть способ исследовать историю версий, и в идеале различия между младшими версиями всегда требуется документировать. А старшие версии, разумеется, всегда должны быть видимыми, поскольку они вводят серьезные изменения.

То, что версии сделаны видимыми единообразно, поможет пользователям адаптироваться к этой модели управления изменениями в системе API и даже способно позволить им применять инструменты, чтобы отреагировать на нее. Поэтому одинаковый подход к версиям всех API значительно увеличивает ценность (в форме улучшения стабильности) системы API. То, что вы предоставляете руководство, говорящее, что надо делать, поддержку того, как это делать, и инструменты, чтобы убедиться, что API действительно следуют этому руководству, поможет упростить управление изменениями для создателей API. Пользователям же будет полезно наличие устойчивого управления изменениями во всей системе.

Выводы

В этой главе мы решили совместить столпы жизненного цикла, описанные в главе 4, с аспектами системы, представленными в главе 8. Самой важной целью этого упражнения было подчеркнуть переход от работы с одним API к работе со многими API, чтобы, с одной стороны, отдельные API могли процветать внутри системы, а с другой — система API, ограничивающая и поддерживающая API, могла постоянно развиваться. Основные факторы этой эволюции — *обратная связь с помощью наблюдения*, позволяющая системе понять, как развиваются методы API, и *поддержка с помощью руководств и инструментов*, позволяющая ей превратить наблюдения в способы упростить внесение изменений.

Матрица системы/жизненного цикла (см. раздел «Аспекты системы и столпы жизненного цикла API» данной главы) является способом продемонстрировать взаимосвязь между аспектами системы и столпами жизненного цикла. Разобравшись в получившейся сложной таблице, мы сосредоточились на тех комбинациях аспектов системы и столпов жизненного цикла, которые заслуживают особого внимания.

Основная модель соединения аспектов системы и столпов жизненного цикла заключается в рассмотрении *наблюдаемости* отдельных API, чтобы с точки зрения системы стало возможно наблюдать за поведением отдельных API. Это соответствует общему смыслу «выражение API через API». Второй шаг — использовать эти наблюдения, чтобы определить области, в которых разработку API можно направить и поддержать с лучшим результатом, если инвестировать в эти столпы. Мы всегда применяем модель «Зачем? Что? Как?» (см. раздел «Центр подготовки» главы 9), чтобы убедиться, что *руководство по API* и *руководство по реализации* четко разделены. Это позволяет методам создания API и реализации развиваться независимо друг от друга, что обеспечивает бóльшую согласованность в системе API, поскольку реализации могут меняться, не затрагивая при этом методы на уровне API.

Наконец, эта глава дала возможность совместить несколько элементов, которые обсуждались в других главах. Мы затронули понятие развития как индикатора изменения формы вашей системы, а также процесс распределения принятия решений как инструмент для разделения ответственности и направления действий в растущей организации. У каждой компании есть свои производственная культура, распространенные способы действий и уровни ожиданий. Мы надеемся, что данный материал даст вам несколько идей по развитию собственной уникальной матрицы системы или жизненного цикла и поможет научиться использовать внутренний процесс принятия решений в компании, чтобы непрерывно улучшать свою систему, в то время как она со временем развивается.

11 Продолжение путешествия

> Мы требуем строго определенных зон сомнения
> и неопределенности!
>
> *Дуглас Адамс*

Управление API — предмет сложный, и нам пришлось осветить в этой книге множество тем, чтобы исследовать его. Мы начали с изучения фундаментального понятия *руководства* API и работы, основанной на решениях. Сфокусировавшись на решениях, пришли к модели принятия решений с элементами, которые можно распределить или распланировать. Планирование решений обеспечило нам полезный и детальный способ управления работой по созданию API.

Создав основу, мы начали путешествие, представив вам первый важный фактор в управлении API — перспективу продукта. Подход к API как к продукту помогает понять, какие решения наиболее значимы. Мы начали с темы разработки одного продукта с API, потому что контекст *локальной оптимизации* важен и его проще понять, чем сложную систему, которая появится позже. Затем исследовали локальный контекст API, рассмотрев десять столпов работы (и решения), которые формируют продукт API так же, как и команды, которые выполняют эту работу, и их культура.

Рассмотрев локальный масштаб управления API, мы подняли планку, познакомив вас со вторым фактором управления API — временем. Принимая во внимание время, мы должны были внимательнее рассмотреть управление изменениями в продукте API, в особенности их влияние на пользователей, разработчиков и владельцев API. Это привело нас к модели жизненного цикла API, которая определяет, какой эффект от изменений в API наблюдается при его развитии на стадиях создания, публикации, окупаемости, поддержки и удаления.

Наконец, мы добавили третий фактор управления API — масштаб. Мы познакомили вас с системой API и обзором сложной системы с высоты 10 000 футов. Вторая

половина книги сосредоточена на объеме *системной оптимизации* и решениях, ей сопутствующих. Мы представили вам восемь аспектов системы: разнообразие, словарь, объем, скорость, уязвимость, видимость, контроль версий и изменяемость. Эта структура описывает самые важные переменные в сложной системе, которой вам нужно управлять, и непростые взаимодействия между ними. В конце книги мы сопоставили этот сложный набор факторов системы с десятью столпами, которые описывали в самом начале.

В этой книге много информации, структур и моделей, но в основе управления API лежат четыре главные составляющие: руководство, продукты, время и масштаб. Неважно, какие у вас API, в какой сфере вы работаете и какого размера ваша компания, все равно придется управлять API, учитывая их. Мы предоставили в книге инструменты, которые помогут вам это сделать.

Готовимся к будущему

Сейчас сложно сказать, что произойдет в будущем, но мы уверены: тенденция к связи между программами не исчезнет. Поскольку архитектура все больше опирается на взаимосвязанные или интегрированные компоненты, спрос на управление API будет расти, несмотря на изменения и развитие протоколов, форматов, стилей и языков, с помощью которых создаются API.

Мы попытались написать эту книгу таким образом, чтобы она была вам полезна независимо от конкретных технологий, которые вы используете. Ключевые концепции — руководство, перспектива продукта, время и масштаб — необходимы и вневременны в сфере управления API. Поэтому, даже если все вокруг изменится, у вас будут концепции и модели, которые помогут с этим разобраться.

Используйте подход «API как продукт», чтобы делать выбор в сфере дизайна и реализации. Он дает возможность создать API, подходящие к требованиям пользователей и вашему рабочему окружению, а не полагаться на милость тенденций и популярных веяний в своей индустрии. Распределяйте решения по API, основываясь на целях, квалификации сотрудников и обстановке в организации, а не пытайтесь скопировать корпоративную культуру последнего успешного стартапа.

А самое главное, примите сложность системы, которой управляете. Поймите, как время и масштаб меняют работу, которую необходимо выполнить. Используйте жизненный цикл продукта с API и восемь аспектов системы, чтобы очертить свое рабочее окружение. Поиграйте в «А что, если?..» с переменными системы, чтобы оценить ее. А что, если вырастет разнообразие? А что, если скорость больше не будет важна? Ответы на эти вопросы, может, и не будут провидческими, но определенно дадут новую информацию.

Начните управлять сегодня

Когда вы сталкиваетесь с большой проблемой в сложной комплексной области, неудивительно, что вы чувствуете себя обескураженными. В начале книги мы обсуждали качество решений. Один из самых важных элементов принятия качественного решения — это доступная информация. К ней относятся знание того, как другие люди решали похожие проблемы, понимание обстановки и эффекта, который произведет любое ваше решение.

Если вы потратите время на сбор подобной информации об управлении API, то сможете принимать лучшие решения в данной сфере. Но если потратите на это слишком много времени, то не сможете научиться на практике. Когда вы сталкиваетесь с неопределенностью в сложной адаптивной системе, единственный разумный путь к прогрессу — решать проблему поэтапно.

Лучший способ делать это — применение таких техник, как PDSA Деминга, описанная в подразделе «Постепенное улучшение» на с. 114. Используйте полученные данные и создайте теорию. В соответствии с этой теорией спланируйте эксперимент. Найдите в организации безопасное место и проведите его. Измерьте результаты и начните снова. Чтобы начать управление API, не нужны гибкие процессы, методы бережливой разработки, доски технологии Канбан, инструменты DevOps или архитектура, основанная на микросервисах. Все это полезно и уместно, но не требуется для начала работы. Все, что вам нужно, — это теория, точные измерения и желание работать и экспериментировать.

Столкнувшись с такой сложной сферой, как управление API, продвигаться таким образом — самое разумное. Кроме того, вы можете начать прямо сейчас. Найдите в своей системе API то, что, по вашему мнению, можно улучшить, и используйте информацию, полученную из этой книги, чтобы провести эксперимент. Учитесь на ходу и растите над собой по мере обучения. Вскоре вы получите систему управления API, которая работает на вас так же, как вы работаете над ней.

Это долгий путь. Но наш опыт и опыт большинства компаний, со специалистами которых мы общались, говорит, что он не обязательно должен быть трудным. Если вы вооружены знаниями об управлении API, собранными в книге, и стремитесь найти решение, подходящее для уникальных потребностей вашей компании, у вас будет большое преимущество. Ваш путь будет ясным, и вы получите значительные шансы на продвижение по нему.

Чем дальше мы продвигаемся, тем ближе к достижению целей. И это полезно для всех.

Об авторах

Мехди Меджуи — предприниматель в сфере API, один из основателей OAuth.io и создатель конференций APIDays — всемирной серии конференций по API, которые проводятся каждый год в семи странах. Как ведущий экономист API в Академии API, Мехди советует людям, принимающим решения в этой сфере, помнить о влиянии API на стратегию цифровой трансформации на микро- и макроуровнях. Он разработал дизайн системы индустрии API, является соавтором доклада «API для банков: состояние рынка», с 2015 года служит экспертом Европейской комиссии по API для проекта «Цифровое правительство». Кроме того, он читает лекции по предпринимательству в цифровую эпоху в парижской бизнес-школе HEC и является советником в нескольких стартапах по инструментам для API.

Эксперт в дизайне протоколов и структурированных данных **Эрик Уайлд** консультирует организации, помогая им получить максимальные результаты от работы с API и микросервисами. Эрик занимается разработкой инновационных технологий с момента появления Интернета и активно сотрудничает с IETF и W3C. Он получил докторскую степень в Высшей технической школе Цюриха, преподавал в ней и в университете Беркли, после чего работал в компаниях EMC, Siemens и в последнее время в CA Technologies.

Ронни Митра помогает крупным и мелким компаниям по всему миру улучшить дизайн и системную архитектуру в организации. В качестве ведущего дизайнера Академии API он сочетает работу с принципами UX и сложностью системы, чтобы решать задачи по созданию эффективных программ с API и установлению практических стратегий трансформации.

Майк Амундсен, известный писатель и лектор, путешествует по миру, дает консультации и читает лекции об архитектуре сетей, веб-разработке и пересечениях между технологиями и обществом. Как ведущий разработчик архитектуры API в Академии API он работает с компаниями, чтобы помочь им повысить доход, реализуя возможности, предоставляемые API пользователям и предприятию.

Об обложке

Иллюстрация на обложке взята из книги Д. Г. Вуда «Одушевленные создания» *(Animate Creation)*.

На обложке книги «Непрерывное развитие API» изображена валлийская овчарка (вал. *Ci Defaid Cymreig*) — представитель породы домашних пастушьих собак типа колли, ведущей свое происхождение из Уэльса. Они бывают черно-белого, рыже-белого и трехцветного окраса, часто с мраморными отметинами. Конечности у них длиннее, а грудная клетка и морда шире, чем у бордер-колли.

Валлийские овчарки — очень волевые и энергичные собаки. Обученные пастушьим обязанностям, могут исполнять их практически без указаний человека. Однако у них нет низкой стойки и сильного визуального контакта, как у бордер-колли (характерные для волков черты хищника, которые позволяют собаке легче управлять стадом), из-за чего их реже выбирают для присмотра за скотом.

Из-за того что их разводят ради поведенческих качеств, а не внешних черт, и разбавления крови в результате скрещивания с бордер-колли, валлийская овчарка не признается основными кинологическими организациями стандартизированной породой. В последние годы были предприняты попытки сохранить эту породу, в основном для домашних целей.

Многие из животных, изображаемых на обложках книг издательства O'Reilly, находятся под угрозой вымирания, и все они представляют ценность для нашего мира. Чтобы узнать о том, каким может быть ваш личный вклад в их спасение, зайдите на страницу animals.oreilly.com.