1 Знакомство с API

В этой главе

- ✓ Что такое интерфейсы.
- ✓ Что такое API.
- ✓ Что такое ориентация на ресурсы.
- ✓ Что делает API «хорошим».

Раз вы выбрали эту книгу, то, скорее всего, уже в целом знакомы с API. Кроме того, вы наверняка уже знаете, что API означает «программный интерфейс приложения», так что в первой главе мы сосредоточимся на тщательном разъяснении значений этих базовых понятий и их значимости. Начнем с более подробного разбора идеи API.

1.1. ЧТО ТАКОЕ ВЕБ-API

API определяет способ взаимодействия компьютерных систем. А поскольку в изолированном режиме функционирует очень небольшое количество систем, то можно не удивляться тому, что API буквально повсюду. Их можно найти в библиотеках, которые мы используем из языковых диспетчеров пакетов (например, в библиотеке шифрования, предоставляющей метод наподобие `function encrypt(input: string): string`), а также в коде, который пишем сами, даже если

он и не предназначается для использования другими. Но существует особый тип API, который создается для раскрытия по сети и используется удаленно многими людьми. Именно такие программные интерфейсы, зачастую называемые веб-API, и рассматриваются в этой книге.

Веб-API привлекают внимание благодаря нескольким аспектам, но самым интересным из них, вероятно, является то, что создатели таких API имеют практически полный контроль над ними, при том, что конечные пользователи, наоборот, в контроле ограничены. Используя библиотеку, мы работаем с ее локальными копиями, в связи с чем создатели API могут в любое время делать все, что захотят, без риска навредить пользователям. Веб-API отличаются тем, что никаких копий не используется. В результате при внесении в веб-API изменений последние сказываются на пользователях вне зависимости от того, просили они о них или нет.

Представьте, к примеру, вызов веб-API, который позволяет вам зашифровать данные. Если работающая над ним команда решит начать использовать для шифрования другой алгоритм, то в этом случае у вас не будет выбора. При вызове соответствующего метода шифрования ваши данные будут зашифрованы последним заданным алгоритмом. В более экстремальном примере авторы API могут решить полностью закрыть его и игнорировать ваши запросы. В этот момент ваше приложение внезапно перестанет работать, и вы ничего не сможете с этим поделать. Оба описанных сценария показаны на рис. 1.1.

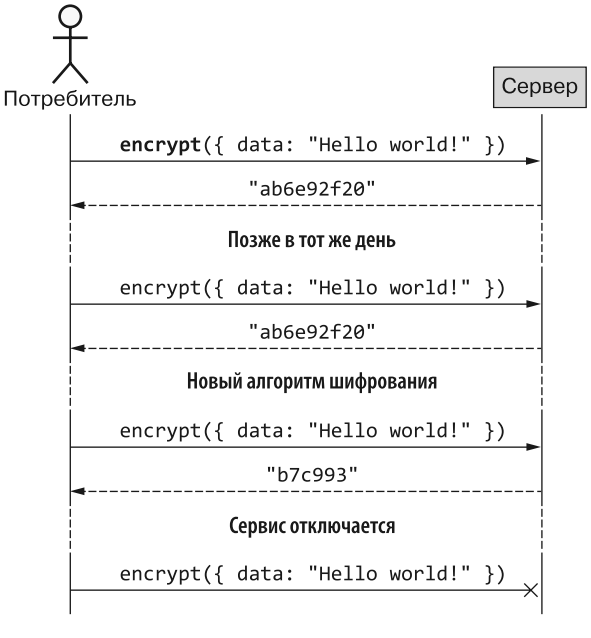

Рис. 1.1. Возможные сценарии при работе с веб-API со стороны потребителя

Тем не менее эти особенности, оказывающиеся для потребителей недостатками, для разработчиков API обычно выступают главными преимуществами, так как позволяют сохранять полный контроль над своим детищем. К примеру, если в API шифрования используется новый суперсекретный алгоритм, то создатели этого интерфейса вряд ли захотят просто так раскрыть его код миру в форме библиотеки. Вместо этого они скорее предпочтут использовать веб-API, который позволит им предоставить пользователям *функциональность* этого суперсекретного алгоритма, не выдавая свою ценную интеллектуальную собственность. В иных случаях системе может требоваться невероятная вычислительная мощность, что приведет к очень длительному выполнению, если развернуть ее в качестве библиотеки и запускать на домашнем ПК или ноутбуке. В похожих случаях, например при работе со многими API машинного обучения, создание веб-API позволяет предоставить потребителям мощную функциональность, скрыв вычислительные требования. Схематично это представлено на рис. 1.2.

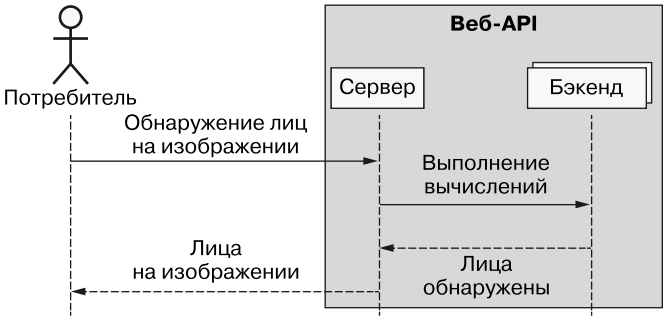

Рис. 1.2. Пример веб-API, скрывающего необходимую вычислительную мощность

Теперь, когда мы понимаем, что такое API (и, в частности, веб-API), возникает вопрос: в чем их значимость?

1.2. ПОЧЕМУ API ВАЖНЫ

Программное обеспечение нередко создается исключительно для использования человеком, и в этом нет ничего плохого. Однако за последние несколько лет мы видим все больший уклон в сторону автоматизации, когда компьютерные программы создаются для того, чтобы делать то же, что и человек, но быстрее и эффективнее. К сожалению, именно на этом рубеже ПО «только для людей» становится проблемой.

Разрабатывая нечто, предназначенное исключительно для использования человеком, и подразумевая взаимодействие с системой с помощью мыши и клавиатуры, мы склонны смешивать строение системы и визуальные аспекты

с сырыми данными и функциональными аспектами. Проблема в том, что может быть сложно объяснить компьютеру, как взаимодействовать с графическим интерфейсом. Причем она усугубляется тем, что изменение визуальных аспектов программы также может потребовать переобучения компьютера взаимодействию с этим *новым* графическим интерфейсом. Те изменения, которые кажутся чисто косметическими для нас, оказываются совершенно нераспознаваемыми для компьютера. Иными словами, для компьютера нет такого понятия, как «чисто косметически».

API — это интерфейсы, которые предназначены специально для компьютеров и имеют важные свойства, позволяющие компьютерам легко их использовать. К примеру, у этих интерфейсов отсутствуют визуальные составляющие, значит, не нужно беспокоиться о внешних изменениях. К тому же эти интерфейсы обычно развиваются только «совместимым» образом (подробнее см. в главе 24), значит, нет необходимости переучивать компьютер чему-либо в связи с изменениями. Если кратко, то API предоставляют способ говорить на языке, который нужен компьютерам, чтобы осуществлять безопасное и стабильное взаимодействие.

Но все это не ограничивается простой автоматизацией. API также открывают возможности для композиции, которая позволяет нам рассматривать функциональность как кубики лего, соединяя детали все новыми и новыми способами, чтобы получить нечто, в разы превосходящее сумму своих частей. Более того, эти получаемые композиции API также можно рассматривать как строительные блоки, формируя из них еще более сложные и невероятные проекты.

Но здесь возникает важный вопрос: как сделать так, чтобы создаваемые нами API сочетались подобно кубикам лего? Начнем с рассмотрения используемой для этого *ресурсно-ориентированной* стратегии.

1.3. ЧТО ЗНАЧИТ РЕСУРСНО-ОРИЕНТИРОВАННЫЕ API

Многие из существующих сегодня веб-API действуют подобно слугам: вы просите их выполнить что-либо, и они это делают. К примеру, если мы хотим узнать прогноз погоды в своем городе, то можем приказать веб-API `predictWeather(postalCode=10011)`. Такой вид передачи приказа другому удаленному компьютеру с помощью вызова предварительно настроенной подпрограммы или метода обычно называется выполнением «удаленного вызова процедуры» (remote procedure call, RPC), так как, по существу, мы *вызываем* библиотечную функцию (или *процедуру*) для выполнения на другом компьютере, находящемся в другом (*удаленном*) месте. Важнейший аспект подобных API — акцент на выполняемых действиях. То есть мы думаем о вычислении погоды (`predictWeather(postalCode=...)`), либо

шифровании данных (`encrypt(data=...)`), либо отправке электронной почты (`sendEmail(to=...)`), во всех случаях делая акцент на «выполнении» чего-либо.

Тогда почему не все API ориентированы на RPC? Одна из основных причин связана с идеей «сохранения состояния» (statefulness), когда вызовы API могут быть либо «с сохранением состояния» (stateful), либо «без сохранения состояния» (stateless). Вызов API рассматривается как stateless, когда его можно совершить независимо от всех других запросов API без дополнительного контекста или данных. Например, вызов веб-API для прогнозирования погоды задействует только один независимый ввод данных (почтовый код), в связи с чем считается stateless. С другой стороны, веб-API, который сохраняет выбранные пользователем города и предоставляет для них прогнозы погоды, не имеет входных данных в среде выполнения, но требует, чтобы у пользователя уже были сохранены интересующие его города. В результате такой вид запроса API, подразумевающего предварительные запросы или использование ранее сохраненных данных, будет считаться stateful. На деле оказывается, что RPC-ориентированные API отлично подходят для stateless-функциональности, но при этом гораздо менее эффективны, когда мы вводим stateful-методы API.

ПРИМЕЧАНИЕ

Если вы знакомы с концепцией REST, то сейчас будет кстати сказать, что этот раздел не посвящен конкретно REST и RESTful API, а является более общим для всех API, которые выделяют «ресурсы» (как большинство RESTful API). Другими словами, хоть здесь и будет много пересечений с темой REST, данный раздел все же охватывает более общий материал.

Чтобы понять, почему это так, мы рассмотрим пример stateful-API для бронирования билетов на самолет. В табл. 1.1 отражен список RPC для взаимодействия с планированием авиаперелетов, включая такие действия, как планирование новых броней, просмотр существующих и отмена полета.

Таблица 1.1. Перечень методов для примера API бронирования авиаперелетов

Метод	Описание
`ScheduleFlight()`	Планирует новый перелет
`GetFlightDetails()`	Показывает информацию о конкретном перелете
`ShowAllBookings()`	Показывает все забронированные на данный момент перелеты
`CancelReservation()`	Отменяет резервирование перелета
`RescheduleFlight()`	Переназначает существующий перелет на другую дату или время
`UpgradeTrip()`	Переводит из эконом-класса в первый класс

Каждый из этих RPC отлично говорит сам за себя, но здесь никак не обойтись без запоминания этих методов API, многие из которых очень похожи между собой. Так, иногда метод говорит о «перелете» (тот же `RescheduleFlight()`), а в других случаях оперирует словом «резервирование» (например, `CancelReservation()`). Кроме того, нужно помнить, какие из множества синонимичных форм этих действий использовались. К примеру, следует фиксировать, с помощью какого из методов мы просматриваем все свои брони: `ShowFlights()`, `ShowAllFlights()`, `ListFlights()` или `ListAllFlights()` (в данном случае это `ShowAllFlights()`). Но как можно решить такую проблему? С помощью стандартизации.

Ориентированность на ресурсы призвана помочь решить эту проблему в двух направлениях, предоставляя стандартный набор строительных блоков для использования при проектировании API. Во-первых, ресурсно-ориентированные API опираются на идею «ресурсов», являющихся ключевыми единицами, которые мы сохраняем и с которыми взаимодействуем, стандартизируя все, что реализует API. Во-вторых, вместо того чтобы использовать для любого нужного действия произвольных имен RPC, ресурсно-ориентированные API ограничивают действия до небольшого стандартного набора (описанного в табл. 1.2), который применяется к каждому ресурсу, формируя полезные действия в API. Если посмотреть немного под другим углом, то ресурсно-ориентированные API — это просто особый тип API, основанных на RPC, в которых каждый RPC следует понятному и стандартизированному паттерну: `<StandardMethod><Resource>()`.

Таблица 1.2. Перечень стандартных методов и их значения

RPC	Описание
`Create<Resource>()`	Создает новый `<Resource>`
`Get<Resource>()`	Показывает информацию о конкретном `<Resource>`
`List<Resources>()`	Показывает список всех существующих `<Resources>`
`Delete<Resource>()`	Удаляет существующий `<Resource>`
`Update<Resource>()`	Обновляет существующий `<Resource>` на месте

Если пойти дальше по этому пути особых, ограниченных RPC, то получится, что вместо разнообразия всевозможных методов RPC, показанных в табл. 1.1, мы можем придумать один ресурс (например, `FlightReservation`) и получить равнозначную функциональность с набором стандартных методов, показанных в табл. 1.3.

Таблица 1.3. Перечень стандартных методов, применяемых к ресурсу перелетов

Метод		Ресурс		Методы
`Create`				`CreateFlightReservation()`
`Get`				`GetFlightReservation()`
`List`	×	`FlightReservation`	=	`ListFlightReservations()`
`Delete`				`DeleteFlightReservation()`
`Update`				`UpdateFlightReservation()`

Очевидно, что стандартизация оказывается более организованной, но значит ли это, что все ресурсно-ориентированные API лучше, чем RPC-ориентированные? Вообще-то нет. В некоторых сценариях RPC-ориентированные API окажутся более эффективными (особенно в случае stateless-методов API). Однако во многих других случаях пользователям будет намного проще понять, изучить и запомнить именно ресурсно-ориентированные API. Причина в том, что предоставляемая ими стандартизация упрощает совмещение того, что вы уже знаете (например, набора стандартных методов), с тем, что вы можете легко усвоить (например, названием нового ресурса), позволяя сразу начать взаимодействовать с API. Говоря в цифрах, если вы знаете, скажем, пять стандартных методов, то благодаря надежному паттерну освоение одного нового ресурса в действительности будет приравнено к освоению пяти новых RPC.

Конечно же, не все API одинаковы, и было бы несколько опрометчиво определять их сложность в формате размера списка того, что «нужно изучить». С другой стороны, здесь работает один важный принцип: сила паттернов. На деле изучать совмещаемые части и объединять их в более сложные компоненты, которые следуют установленному образцу, оказывается легче, чем изучать предварительно собранные компоненты, которые каждый раз соответствуют особой структуре. Поскольку ресурсно-ориентированные API зиждутся на проверенных временем паттернах проектирования, зачастую их проще выучить, а значит, они оказываются «лучше», чем их RPC-ориентированные аналоги. Но вслед за этим возникает важный вопрос: что в данном случае значит «лучше»? Как нам понять, является ли API «хорошим»? Что здесь вообще значит «хороший»?

1.4. ЧТО ДЕЛАЕТ API «ХОРОШИМ»

Прежде чем рассматривать различные аспекты, делающие API «хорошим», нужно разобраться, зачем мы вообще используем программный интерфейс. Иначе говоря, какова цель его создания? Обычно она сводится к двум простым причинам:

1) у нас есть востребованная определенными пользователями функциональность;
2) эти пользователи хотят использовать данную функциональность программно.

К примеру, у нас может быть система, которая прекрасно справляется с переводом текстов. В мире наверняка много людей, заинтересованных в такой возможности, но одного этого недостаточно. В конце концов, мы можем запустить вместо API мобильное приложение, предоставляющее эту прекрасную систему перевода. Чтобы получить API, люди, заинтересованные в этой функциональности, должны также захотеть написать программу, позволяющую использовать ее. Учитывая два этих критерия, какие качества будут желательными для нашего API?

1.4.1. Функциональность

Начнем с самого важного: вне зависимости от того, как будет выглядеть окончательный интерфейс, система в целом должна быть *функциональной*. Иначе говоря, она должна делать то, что пользователям действительно нужно. Если она предназначена для перевода текстов с одного языка на другой, то должна уметь это делать. Кроме того, к большинству систем может предъявляться множество *нефункциональных* требований. Например, если наша система переводит текст, то к ней могут предъявляться нефункциональные требования, связанные с такими аспектами, как задержка (например, процесс перевода должен занимать несколько миллисекунд, а не дней) или точность (например, переводы не должны оказываться ложными и вводить в заблуждение). Вот эти два аспекта могут составлять функциональную сторону системы.

1.4.2. Выразительность

Помимо функциональной эффективности, система также должна иметь интерфейс, который бы позволял пользователям выражать свои намерения ясно и просто. Другими словами, если система переводит текст, то ее API должен быть создан так, чтобы делать это можно было простым и понятным способом. В данном случае это может быть RPC под названием `TranslateText()`. Конечно, это может звучать вполне очевидно, но в действительности нередко оказывается сложнее, чем кажется.

К примерам подобной скрытой сложности относится случай, когда API уже поддерживает некую функциональность, но по недосмотру мы не замечаем, что она нужна пользователям, и не реализуем выразительного способа доступа к ней. В результате люди вынуждены использовать ее *обходными путями*, совершая

ряд непривычных и не всегда очевидных действий. К примеру, если API предоставляет возможность переводить текст, то пользователь может косвенно задействовать его исключительно для определения языка, даже если переводить ничего не хочет. Нетрудно представить, что прямой RPC `DetectLanguage()` оказался бы для пользователей куда удобнее, чем совершение множества вызовов API в попытке угадать язык.

Листинг 1.1. Функциональность, позволяющая определять язык с помощью только метода API TranslateText

```
function detectLanguage(inputText: string): string {
  const supportedLanguages: string[] = ['en', 'es', ... ];
  for (let language of supportedLanguages) {
    let translatedText = TranslateApi.TranslateText({
      text: inputText,
        targetLanguage: language
    });
    if (translatedText == inputText) {
      return language;
    }
  }
  return null;
}
```

Здесь подразумевается, что API определяет метод TranslateText, который получает вводный текст и целевой язык, в который нужно его перевести

Если переведенный текст совпадает с исходным, то мы понимаем, что перед нами один и тот же язык

Если мы не находим переведенный текст, совпадающий с исходным, то возвращаем null, указывая, что язык исходного текста определить невозможно

Как показывает этот пример, API, которые поддерживают определенную функциональность, но не упрощают пользователям доступ к ней, оказываются не самыми удачными. При этом выразительные API предоставляют пользователям возможность ясно выражать не только то, что они хотят сделать (например, перевести текст), но и каким именно образом (например, в течение 150 миллисекунд, с точностью 95 %).

1.4.3. Простота

Одним из важнейших факторов, связанных с удобством использования любой системы, является ее *простота*. И хотя логично предположить, что достичь простоты можно за счет сокращения *компонентов* (RPC, ресурсов и т. п.) в API, к сожалению, это не так. К примеру, API может опираться на один метод `ExecuteAcrion()`, обрабатывающий всю функциональность. Тем не менее это ничего не упрощает, а только перемещает сложность из одного места (множества разных RPC) в другое (много настроек в одном RPC). Как же тогда выглядит простой API?

Вместо того чтобы пытаться излишне сокращать количество RPC, API должен стремиться раскрывать нужную пользователям функциональность наиболее доступным способом, делая API максимально простым, но не более. Представьте, что в API перевода нужно добавить возможность определять язык входного

текста. Можно сделать это путем возвращения определенного исходного текста в ответ на перевод. Однако это все равно запутывает функциональность, скрывая ее внутри метода, предназначенного для другой цели. Вместо этого будет разумнее создать новый метод специально для поставленной задачи, например `DetectLanguage()`. (Обратите внимание, что мы также можем включать распознание языка при переводе содержимого, но это служит уже совсем для другой цели.)

Еще один универсальный подход к обеспечению простоты основан на старом правиле «стандартного случая» (делать обработку стандартных случаев быстрой), но вместо этого ориентируется на простоту использования, оставляя пространство для пограничных случаев. Такая измененная формулировка подразумевает, что нужно «реализовать типичный случай превосходно, а продвинутый сделать возможным». Это означает, что при каждом добавлении чего-либо, усложняющего API в угоду продвинутому пользователю, лучше все-таки скрывать это усложнение от типичного пользователя, который заинтересован лишь в стандартном случае. При таком подходе более частые сценарии получатся простыми и легкими в использовании, но при этом сохранятся и продвинутые возможности для тех, кому они нужны.

В качестве примера представим, что наш API перевода включает принцип использования модели машинного обучения, которая понадобится при переводе текста, когда вместо того, чтобы указать целевой язык, мы выбираем основанную на нем модель и используем ее как «механизм перевода». Несмотря на то что такая функциональность предоставляет гораздо больше гибкости и контроля для пользователей, она окажется и намного сложнее. Этот новый типичный случай показан на рис. 1.3.

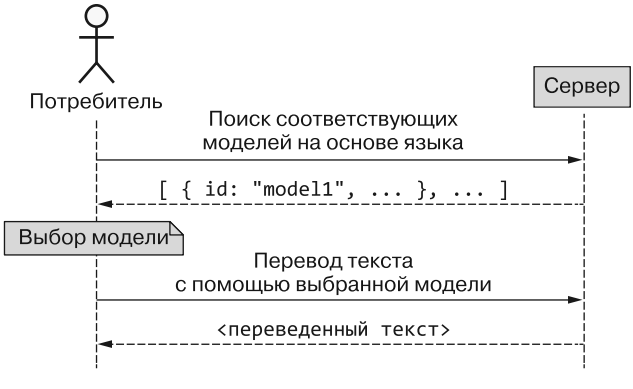

Рис. 1.3. Перевод текста после выбора модели

Как видите, в итоге в обмен на поддержку более продвинутой функциональности мы существенно усложнили процесс перевода. Чтобы увидеть это

более наглядно, сравните код из листинга 1.2 с простотой вызова `TranslateText("Hello world", "es")`.

Листинг 1.2. Перевод текста после выбора модели

```
function translateText(inputText: string,
                       targetLanguage: string): string {
  let sourceLanguage =
      TranslateAPI.DetectLanguage(inputText);
  let model = TranslateApi.ListModels({
    filter: `sourceLanguage:${sourceLanguage}
             targetLanguage:${targetLanguage}`,
  })[0];
  return TranslateApi.TranslateText({
    text: inputText,
    modelId: model.id
  });
}
```

Поскольку нам нужно выбирать модель, сначала необходимо узнать язык входного текста. Чтобы определить его, мы можем опереться на гипотетический метод DetectLanguage(), предоставленный API

Когда нам будут известны и исходный, и целевой языки, можно будет выбрать любую подходящую модель из тех, что предлагает API

Теперь, когда у нас наконец есть все необходимые входные данные, можно вернуться к переводу текста на целевой язык

Как сделать такой API максимально простым, одновременно сделав типичный случай превосходным, а продвинутый — возможным? Поскольку типичный случай подразумевает пользователей, которых не интересует конкретная модель, можно спроектировать API так, чтобы он принимал либо `targetLanguage`, либо `modelId`. Тогда продвинутый случай будет по-прежнему рабочим (код из листинга 1.2 продолжит функционировать), но типичный случай будет выглядеть намного проще, опираясь просто на параметр `targetLanguage` (и ожидая, что параметр `modelId` останется неопределенным) (листинг 1.3).

Листинг 1.3. Перевод текста на целевой язык (типичный случай)

```
function translateText(inputText: string,
                       targetLanguage: string,
                       modelId?: string): string {
  return TranslateApi.TranslateText({
    text: inputText,
    targetLanguage: targetLanguage,
    modelId: modelId,
  });
}
```

Теперь, когда у нас есть представление о том, насколько для «хорошего» API важна простота, рассмотрим заключительный критерий: предсказуемость.

1.4.4. Предсказуемость

Если в жизни сюрпризы иногда оказываются забавными, то в API им точно места нет, хоть в определении интерфейса, хоть во внутреннем поведении. Это утверждение можно сравнить с поговоркой об инвестировании: «Если вы очень им увлечены, значит, действуете неправильно». Так что же мы подразумеваем под API «без сюрпризов»?

API без сюрпризов опираются на повторяющиеся паттерны, применяемые и к внешнему определению API, и к его поведению. Например, если API перевода текста содержит метод `TranslateText()`, который в качестве параметра получает входное содержимое в поле `text`, то при добавлении метода `DetectLanguage()` входное содержимое должно также называться `text` (а не `inputText`, `content` или `textContent`). Хоть это и выглядит очевидным сейчас, помните, что многие API построены несколькими командами и выбор имен для полей при наличии множества параметров зачастую оказывается произвольным. В итоге, когда два отдельных человека отвечают за два таких отдельных поля, возникает большая вероятность, что они сделают разный выбор. И когда это происходит, мы получаем несогласованный API, то есть API с сюрпризом.

Несмотря на то что такая несогласованность может казаться незначительной, в действительности подобные проблемы оказываются намного более важными, чем выглядят на первый взгляд. Причина в том, что пользователи API довольно редко изучают каждую деталь, тщательно вчитываясь в документацию. Вместо этого они просматривают ровно столько, сколько нужно, чтобы добиться требуемого результата. То есть если человек узнает, что поле называется `text` в одном текстовом сообщении, он наверняка предположит, что оно так же называется и в другом. При этом человек будет строить аналогичные догадки, основанные на известных ему фактах, и в отношении остального. Если этот процесс потерпит неудачу (например, потому что в другом сообщении поле было названо `inputText`), то работа встанет и пользователю придется выяснять, почему его предположение не оправдалось.

Очевидный вывод здесь в том, что API, которые опираются на повторяющиеся, предсказуемые паттерны (например, согласованное именование полей), проще и быстрее освоить, а значит, они лучше. Аналогичные преимущества дают и более сложные паттерны, такие как стандартные действия, которые мы видели при изучении ресурсно-ориентированных API. Это дает нам представление об основной цели книги: научить использовать известные, четко определенные, понятные и (надеюсь) простые паттерны для создания предсказуемых, доступных, следовательно, и более качественных API. Теперь, когда мы разобрались с понятием API и выяснили, что делает их хорошими, пора начать рассматривать высокоуровневые паттерны, которым мы можем следовать при разработке наших API.

РЕЗЮМЕ

- Интерфейсы представляют собой контракты, определяющие, как две системы должны взаимодействовать друг с другом.
- API — это особые типы интерфейсов, которые определяют правила взаимодействия двух компьютерных систем. Они могут выражаться в разных формах, таких как скачиваемые библиотеки и веб-API.

- Веб-API отличаются тем, что предоставляют функциональность по сети, скрывая конкретную реализацию или вычислительные требования, необходимые для ее выполнения.
- Ресурсно-ориентированные API — это способ проектирования программных интерфейсов, позволяющий снизить сложность за счет использования стандартного набора действий, называемых *методами*, среди ограниченного набора компонентов, называемых *ресурсами*.
- Однозначно определить, что значит «хороший» API, сложно, но обычно они отличаются функциональностью, выразительностью, простотой и предсказуемостью.