Part 1

Fundamentals of API design

Every journey starts with a first step, and the API design journey is no exception. API designers need many skills and need to take many topics into account when creating APIs, but without a solid foundation, all advanced skills and topics are worth nothing. That is what you'll learn in this first part.

We will first set the scene by explaining what an API is, why it actually must be designed, and what *learning to design an API* actually means. You will discover that although these actually are programming interfaces, APIs are more than "technical plumbing" and that you must learn fundamental principles to design any type of API.

Even before thinking about the programming side, you will see that an API has to be thought of from its users' perspectives. An API is supposed to let your users easily achieve their goals, not the ones of the system exposing the API. Only once these goals are known and accurately described can the actual programming interface, such as a REST API, be designed. And like any programming, describing a programming interface should be done with an adapted tool like the OpenAPI Specification for REST APIs.

1 What is API design?

This chapter covers

- What an API is
- Why API design matters
- What designing an API means

Web application programming interfaces (APIs) are an essential pillar of our connected world. Software uses these interfaces to communicate—from applications on smartphones to deeply hidden backend servers, APIs are absolutely everywhere. Whether these are seen as simple technical interfaces or products in their own right, whole systems, whatever their size and purpose, rely on them. So do entire companies and organizations from tech startups and internet giants to non-tech small and medium-sized enterprises, big corporations, and government entities.

If APIs are an essential pillar of our connected world, API design is its foundation. When building and evolving an API-based system, whether it is visible to anyone or deeply hidden, whether it creates a single or many APIs, design must always be a major concern. The success or failure of such a system depends directly on the quality of the design of all its APIs.

But what does designing APIs really mean? And what do you have to learn to design APIs? To answer these questions, we need to consider what an API is and for whom it's designed, and we also need to realize that designing an API is more than just designing a programming interface for applications.

1.1 What is an API?

Billions of people own smartphones and use them to share photos on social networks. This wouldn't be possible without APIs. Sharing photos using a mobile social networking application involves the use of different types of APIs, as shown in figure 1.1.

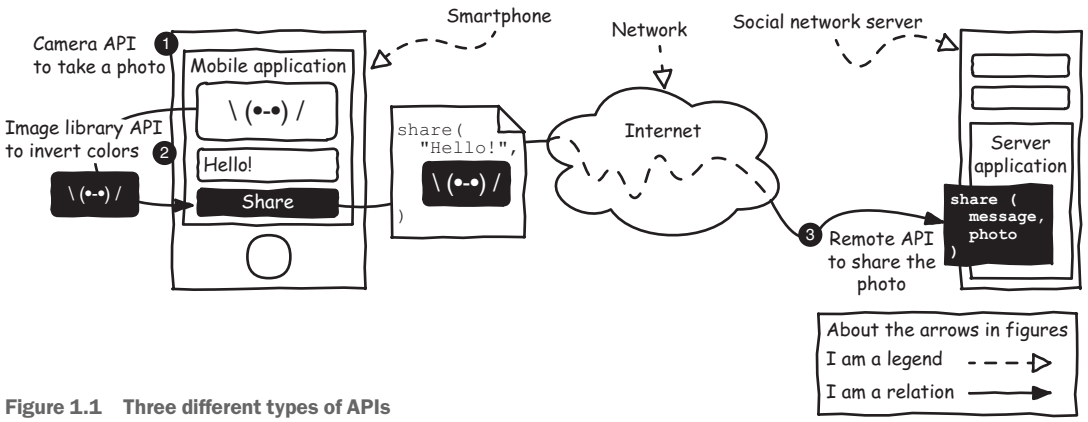

Figure 1.1 Three different types of APIs

First, to take a photo, the social networking mobile application uses the smartphone's camera via its API. Then, through its API, it can use some image library embedded in the application to invert the photo's colors. And, eventually, it shares the modified photo by sending it to a server application hosted on the social network server using a remote API accessible via a network, usually the internet. So, in this scenario, three different types of API are involved: respectively, a hardware API, a library, and a remote API. This book is about the latter.

APIs, whatever their types, simplify the creation of software, but remote APIs, and especially web ones, have revolutionized the way we create software. Nowadays, anyone can easily create anything by assembling remote pieces of software. But before we talk about the infinite possibilities offered by such APIs, let's clarify what the term *API* actually means in this book.

1.1.1 An API is a web interface for software

In this book, an API is a remote API and, more precisely, a *web API*—a web interface for software. An API, whatever its type, is first and foremost an interface: *a point where two systems, subjects, organizations, and so forth meet and interact*. The concept of an API might not be easy to grasp at first, but figure 1.2 makes it more tangible by comparing it to an application's user interface (UI).

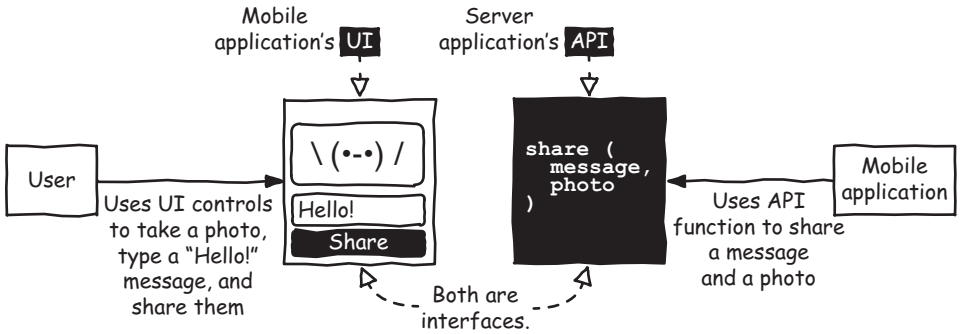

Figure 1.2 Comparing an application's user interface (UI) to an application programming interface (API)

As a user of a mobile application, you interact with it by touching your smartphone's screen, which displays the application's UI. A mobile application's UI can provide elements like buttons, text fields, or labels on the screen. These elements let users interact with the application to see or provide information, or to trigger actions such as sharing a message and a photo.

Just as we (human beings) use an application's UI to interact with it, that application can use another application through its programming interface. Whereas a UI provides input fields, labels, and buttons to provide some feedback that can evolve as you use them, an API provides functions that may need input data or that may return output data as feedback. These functions allow other applications to interact with the application providing the API to retrieve or send information or to trigger actions.

Strictly speaking, an API is *only* an interface exposed by some software. It's an abstraction of the underlying *implementation* (the underlying code—what actually happens inside the software product when the API is used). But note that the term *API* is often used to name the whole software product, including the API and its implementation. So APIs are interfaces for software, but the APIs we talk about in this book are more than just APIs: they are web APIs, as shown in figure 1.3.

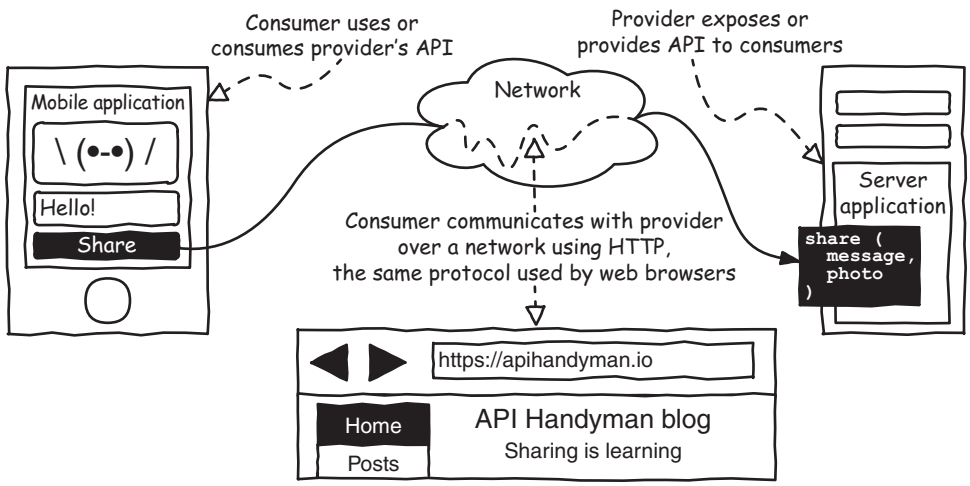

Figure 1.3 Web APIs are remote APIs that can be used with the HTTP protocol.

The mobile application running on a smartphone uses or consumes the API exposed or provided by the server application (often called a *backend application* or simply *backend*) that's hosted on a remote server. The mobile application is therefore called a *consumer*, and the backend is called a *provider*. These terms also apply respectively to the companies and the people creating the applications, or consuming or providing APIs. Here that means the developers of the mobile application are consumers and the ones developing the backend are providers.

To communicate with its backend, the mobile application usually uses a famous network: the internet. The interesting thing here is not the internet itself—such communication can also be done over a local network—but *how* these two applications communicate over the network. When a mobile application sends a photo and message to the backend application, it does so using the Hypertext Transfer Protocol (HTTP). If you've ever opened a web browser on a computer or a smartphone, you have used HTTP (indirectly). This is the protocol that is used by any website. When you type a website's address, like http://apihandyman.io or its secured version, https://apihandyman.io, into the address bar and press Enter or click a link in a browser, the browser uses HTTP to communicate with the remote server hosting the website in order to show you the site's content. Remote APIs, or at least the ones we're talking about in this book, rely on this protocol just like websites; that's why these are called *web APIs*.

So, in this book, APIs are web APIs. They are web interfaces for software. But why are such APIs so interesting?

1.1.2 APIs turn software into LEGO® bricks

Thousands, even millions, of mobile applications and their backends have been created thanks to web APIs, but there's more to the story. Indeed, web APIs unleash creativity and innovation by turning software into reusable bricks that can be easily assembled. Let's go back to our example and see what might happen when sharing a photo on a social network.

When a social network backend receives a photo and message, it might store the photo on the server's filesystem and store the message and the photo identifier (to retrieve the actual file later) in a database. It could also process the photo using some homemade facial recognition algorithm to detect if it contains any friends of yours before storing the photo. That's one possibility—a single application handling everything for another solitary application. Let's consider something different, as shown in figure 1.4.

The backend API could be used by both a social network mobile application and a website, and its implementation could be totally different. When the backend receives a photo and message to share (whichever application sent it), it could delegate the photo's storage as a service company through its API. It could also delegate the storage of the message and photo identifier to an in-house timeline software module through its API. How could the facial recognition be handled? Well, that could be delegated to some expert in facial recognition offering their services via … you guessed it … an API.

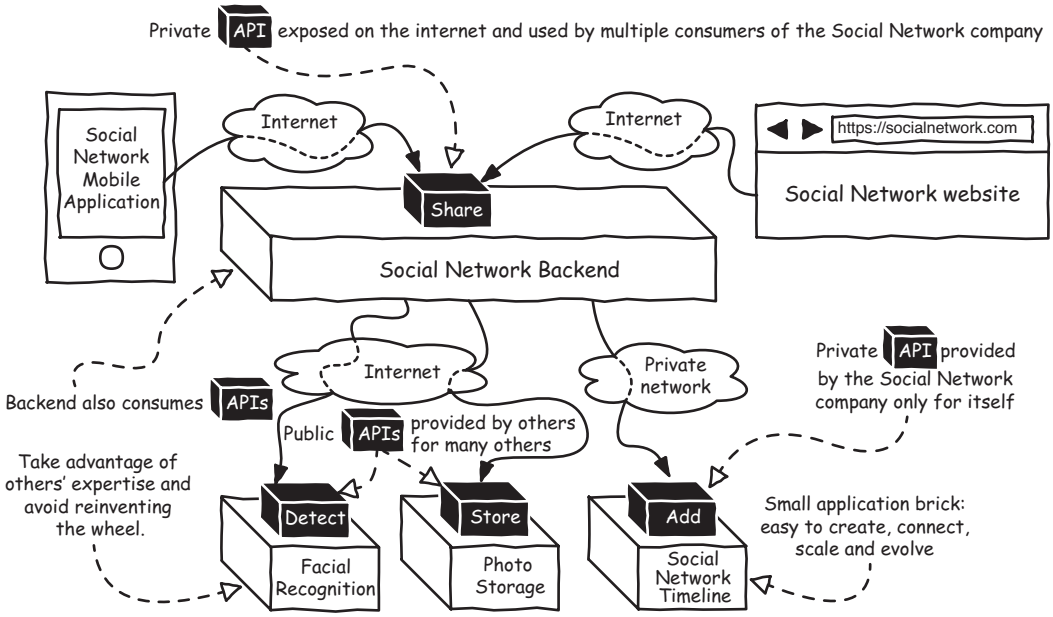

Figure 1.4 A system composed of public and private software LEGO® bricks connected via APIs.

Note that in figure 1.4, each API only exposes one function. This keeps the figure as simple as possible: a single API can expose many functions. The backend can, for example, expose functions such as Add Friend, List Friends, or Get Timeline.

This looks like a software version of the LEGO bricks system; you know, those plastic blocks that you can put together to create new things. (Aristotle was probably playing with some of these when he had the realization that "the whole is greater than the sum of its parts.") The possibilities are endless; the only limit is your imagination.

When I was a child, I used to play with LEGO bricks for endless hours—creating buildings, cars, planes, spaceships, or whatever I wanted. When I was bored with of one of my creations, I could destroy it completely and start something new from scratch, or I could transform it by replacing some parts. I could even put existing structures together to create a massive spaceship, for example. It's the same in the API world: you can decompose huge systems of software bricks that can be easily assembled and even replaced thanks to APIs, but there are some minor differences.

Each software brick can be used at the same time by many others. In our example, the backend API can be used by a mobile application and a web site. An API is usually not made to be consumed by a single consumer but by many. That way, you don't have to redevelop everything all the time.

Each software brick can run anywhere on its own as long as it's connected to a network in order to be accessible via its API. That offers a good way to manage performance and scalability; indeed, a software brick like the Facial Recognition one in figure 1.4 will probably need far more processing resources than the Social Network Timeline

one. If the former is run by the Social Network company, it could be installed on a different, dedicated, and more powerful server, while the Social Network Timeline runs on a smaller one. And being accessible via a simple network connection allows any API provided by anyone to be used by anyone.

In the API world, there are two types of bricks exposing two types of APIs: *public APIs* and *private APIs*. The Facial Recognition and Photo Storage software bricks are not built and not even run by the Social Network company but by third parties. The APIs they provide are public ones.

Public APIs are proposed *as a service* or *as a product* by others; you don't build them, you don't install them, you don't run them—you only use them. Public APIs are provided to anyone who needs them and is willing to accept the terms and conditions of the third-party supplier. Depending on the business model of the API providers, such APIs can be free or paid for, just like any other software product. Such public APIs unleash creativity and innovation and can also greatly accelerate the creation of anything you can dream of. To paraphrase Steve Jobs, there's an API for that. Why lose time trying to reinvent a wheel that will, irremediably, not be round enough? In our example, the Social Network company chose to focus on its core expertise, connecting people, and delegated facial recognition to a third party.

But public APIs are only the tip of the API iceberg. The Social Network Backend and Social Network Timeline bricks were created by the Social Network company for its own use. The timeline API is only consumed by applications created by the Social Network company: the mobile Social Network Backend in figure 1.4. The same goes for the mobile backend, which is consumed by the Social Network Mobile Application. These APIs are *private APIs*, and there are billions of them out there. A private API is one you build for yourself: only applications created by you or people inside your team, department, or company use it. In this case, you are your own API provider and API consumer.

> **NOTE** The public/private question is not a matter of *how* an API is exposed, but *to whom*. Even if it's exposed on the internet, the mobile backend API is still a private one.

There can be various kinds of interactions among *true* private APIs and public ones. For example, you can install commercial off-the-shelf software like a content management system (CMS) or a customer relationship management system (CRM) on your own servers (such an installation is often call *on premise*), and these applications can (even must!) provide APIs. These APIs are private ones, but you do not build those yourself. Still, you can use them as you wish and, especially, to connect more bricks to your bricks. As another example, you can expose some of your APIs to customers or selected partners. Such *almost public* APIs are often called *partner APIs*.

But whatever the situation, APIs basically turn software into reusable software bricks that can be easily assembled by you or by others to create modular systems that can do absolutely anything. That's why APIs are so interesting. But why should their design matter?

1.2 Why API design matters

Even if it is useful, an API seems to be only a technical interface for software. So why should the design of such an interface be important?

APIs are used by software, it's true. But who builds the software that uses them? Developers. People. These people expect these programming interfaces to be helpful and simple, just like any other (well-designed) interface. Think about how you react when faced with a poorly designed website or mobile application UI. How do you feel when faced with a poorly designed everyday thing, such as a remote control or even a door? You may be annoyed, even become angry or rant, and probably want to never use it again. And, in some cases, a poorly designed interface can even be dangerous. That's why the design of any interface matters, and APIs are no exception.

1.2.1 A public or private API is an interface for other developers

You learned in section 1.1.1 that the API consumer can be either the software using the API or the company or individuals developing that software. All these consumers are important, but the first consumer to take into consideration is the developer.

As you've seen, different APIs are involved in the example social networking use case. There is the timeline module that handles data storage and exposes a private API. And there are the public (provided by other companies) facial recognition and photo storage APIs. The backend that calls these three APIs does not pop into the air by itself; it's developed by the Social Network company.

To use the facial recognition API, for example, developers write code in the Social Network software in order to send the photos to have faces detected and to handle the result of the photo processing, just like when using a software library. These developers are not the ones who created the facial recognition API, and they probably don't know each other because they are from different companies. We can also imagine that the mobile application, website, backend, and the data storage module are developed by different teams within the company. Depending on the company's organization, those teams might know each other well or not at all. And even if every developer in the company knows every little secret of every API it has developed, new developers will inevitably arrive.

So, whether public or private, whatever the reason an API is created and whatever the company's organization, it will sooner or later be used by other developers—people who have *not* participated in the creation of the software exposing the API. That is why everything must be done in order to facilitate these newcomers when writing code to use the API. Developers expect APIs to be helpful and simple, just like any interface they have to interact with. That is why API design matters.

Developer experience

An API's *developer experience* (DX) is the experience developers have when they use an API. It encompasses many different topics such as registration (to use the API), documentation (to know what the API does and how to use it), or support (to get help when having trouble). But all effort put into any DX topic is worth nothing if the most important one is not properly handled—API design.

1.2.2 An API is made to hide the implementation

API design matters because when people use an API, they want to use it without being bothered by petty details that have absolutely nothing to do with them. And, to do so, the design of an API must conceal implementation details (what actually happens). Let me use a real-life analogy to explain this.

Say you decide to go to a restaurant. What about a French one? When you go to a restaurant, you become a *customer*. As a restaurant's customer, you read its menu to find out what kinds of food you can order. You decide to try the Bordeaux-style lamprey (it's a famous French fish dish from the Gascony region). To order the meal you have chosen, you talk to a (usually very kind and friendly) person called a *waiter* or *waitress*. A while later, the waiter comes back and gives you the meal you have ordered—the Bordeaux-style lamprey—that has been prepared in the *kitchen*. While you eat this delicious meal, may I ask you two questions?

First, do you know how to cook Bordeaux-style lamprey? Probably not, and that may be the reason why you go to a restaurant. And even if you do know how to cook this recipe, you may not want to cook it because it's complex and requires hard-to-find ingredients. You go to a restaurant to eat food you don't know how to cook or don't want to cook.

Second, do you know what happened between the point in time when the waiter took your order and when they brought it back to you? You might guess that the waiter has been to the kitchen to give your order to a cook, who works alone. This cook prepares your meal and notifies the waiter when it is done by ringing a small bell and yelling, "Order for table number 2 is ready." But this scenario could be slightly different.

The waiter might use a smartphone to take your order, which is instantly shown on a touchscreen in the kitchen. In the kitchen, there's not a lonely cook, but a whole kitchen brigade. Once the kitchen brigade prepares your meal, one of them marks your order as Done on the kitchen's touchscreen, and the waiter is notified on the smartphone. Whatever the number of cooks, you are unaware of the recipe and ingredients used to cook your meal. Regardless of the scenario, the meal you've ordered by talking to the waiter has been cooked in the kitchen, and the waiter has delivered it to you. At a restaurant, you only speak to the waiter (or waitress), and you don't need to know what happens in the kitchen. What does all this have to do with APIs? Everything, as shown in figure 1.5.

From the social networking mobile application developer team's perspective, *sharing a photo* using the backend API is exactly the same, however the backend is implemented. The developers don't need to know the recipe and its ingredients; they only provide the photo and a message. They don't care if the photo is passed through image recognition before or after being added to the timeline. They also don't care if the backend is a single application written in Go or Java that handles everything, or relies on other APIs written in NodeJS, Python, or whatever language. When a developer creates a *consumer application* (the customer) that uses a *provider application* (the restaurant) through its API (the waiter or waitress) to do something (like ordering a meal), the developer and the application are only aware of the API; they don't need to know how to *do something* themselves or how the provider software (the kitchen) will actually do it. But hiding the implementation isn't enough. It's important, but it's not what people using APIs really seek.

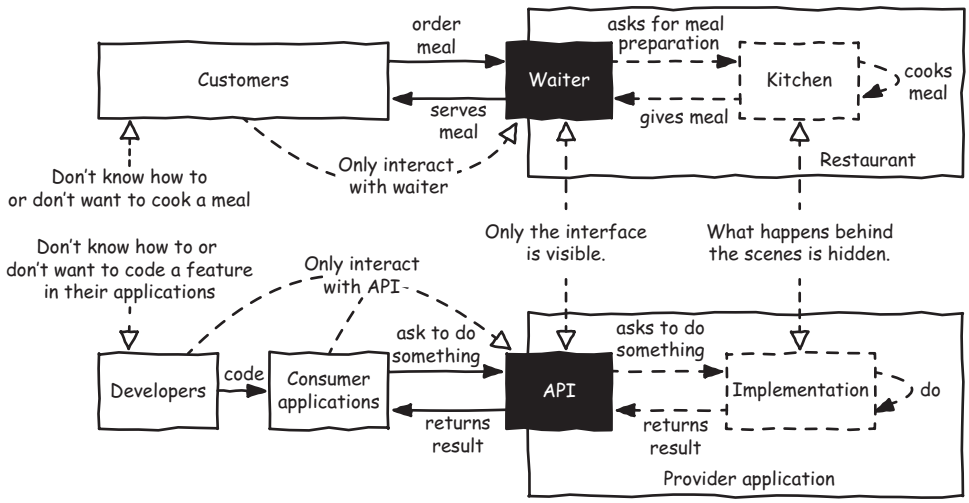

Figure 1.5 The parallels between dining at a restaurant and using an API

1.2.3 The terrible consequences of poorly designed APIs

What do you do when you use an everyday thing you've never used before? You take a close look at its interface to determine its purpose and how to use it based on what you can see and your past experience. And this is where design matters.

Let's consider a hypothetical example: a device called the UDRC 1138. What could this device be? What might be its purpose? Well, its name does not really help us to guess it. Maybe its interface can give us more clues. Take a look at figure 1.6.

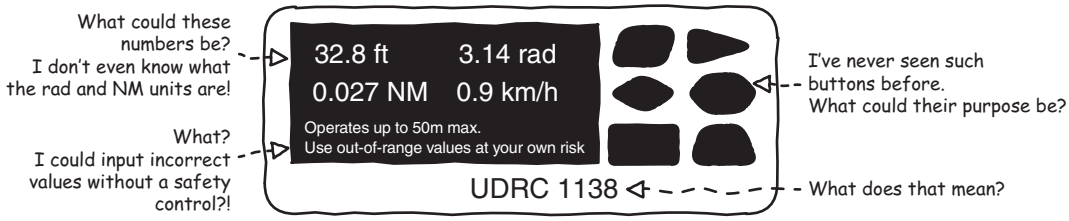

Figure 1.6 The cryptic interface of the UDRC 1138 device

On the right, there are six unlabeled, triangular and rectangular buttons. Could they mean something like start and stop? You know, like media player buttons. The four other buttons have unfamiliar shapes that do not give any hint about their purpose. The LCD display screen shows unlabeled numbers with units such as ft, NM, rad, and km/h. We might guess that *ft* is a distance in feet, and *km/h* is a speed in kilometers per hour, but what might *rad* and *NM* mean? And, at the bottom of the LCD screen, there's also a troublesome warning message telling us that we can input out-of-range values without a safety control.

This interface is definitely hard to decipher and not very intuitive. Let's take a look at the documentation, shown in figure 1.7, to see what it tells us about the device.

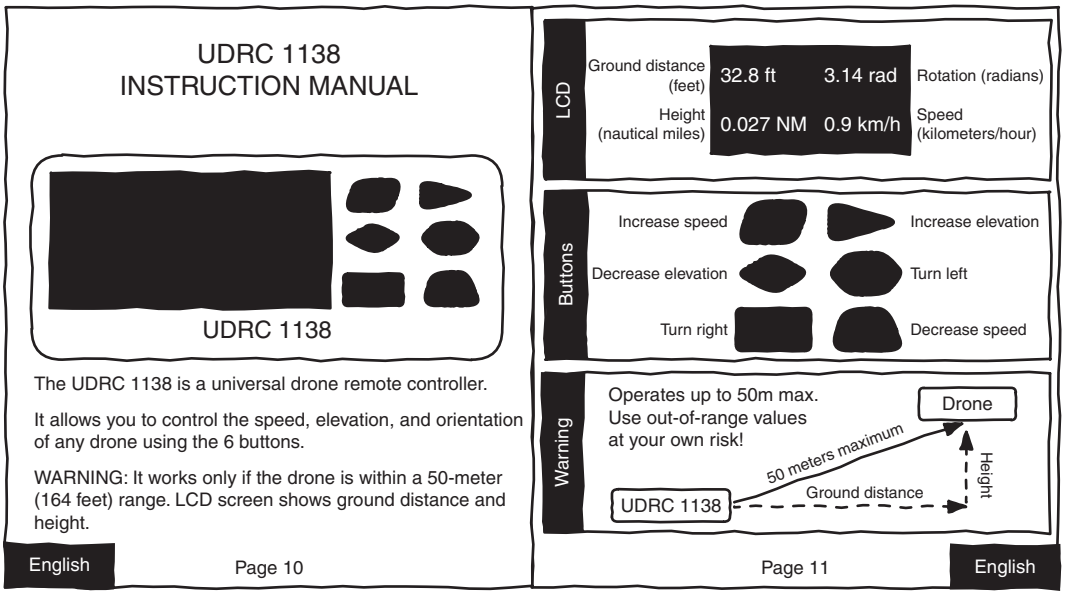

Figure 1.7 The UDRC 1138's documentation

According to the description, this device is a universal drone remote controller—hence the UDRC 1138 name, I suppose—that can be used to control any drone. Well, it sounds interesting. The page on the right gives a few explanations of the LCD screen, the buttons, and the warning message.

The description of the LCD screen is quite puzzling. Ground distance is in feet, height in nautical miles, drone orientation is provided with radians, and speed is in kilometers per hour. I'm not familiar with aeronautical units of measurement, but I sense there's something wrong with the chosen ones. They seem inconsistent. Mixing feet and meters? And in all the movies I've seen about airplanes, feet measure height and nautical miles measure distance, not the inverse. This interface is definitely not intuitive.

Looking at the button's descriptions, we can see that to increase elevation we use the triangular button at the top right, and to decrease it we use the diamond-shaped button in the second row. The connection isn't obvious—why aren't these related controls next to each other? The other controls also use shapes that mean absolutely nothing and seem to be placed randomly. This is insane! How is a user supposed to use the device easily? Why not use good old joysticks or directional pads instead of buttons?

And last, but not least, the explanation of the warning message. It seems that the drone can only be operated withing a 50 meter range—the range being calculated based on the ground distance and height provided by the LCD screen. Wait, what? We have to calculate the distance between the remote controller and the drone using the Pythagorean theorem?! This is total nonsense; the device should do that for us.

Would you buy or even try to use such a device? Probably not. But what if you have no other option than to use this terrible device? Well, I wish you the best of luck; you'll need it to achieve anything with such a poorly designed interface!

You're probably thinking that such a disastrous interface couldn't actually exist in real life. Surely designers couldn't create such a terrible device. And even if they did, surely the quality assurance department would never let it go into production! But let's face the truth. Poorly designed devices—everyday things with poorly designed interfaces—*do* go into production all the time.

Think about how many times you have been puzzled or have grumbled when using a device, a website, or an application because its design was flawed. How many times have you chosen not to buy or not to use something because of its design? How many times were you unable to use something, or to use it correctly or how you wanted to, because its interface was incomprehensible?

A poorly designed product can be misused, underused, or not used at all. It can even be dangerous for its users and for the organization that created it, whose reputation is on the line. And if it's a physical device, once it has gone into production, it's too late to fix it.

Terrible design flaws are not reserved to the interfaces of everyday things. Unfortunately, APIs can also suffer from the disease of poor design. Recall that an API is an interface that developers are supposed to use within their software. Design matters, whatever the type of interface, and APIs are no exception. Poorly designed APIs can be just as frustrating as a device like the UDRC 1138. A poorly designed API can be a real pain to understand and use, and this can have terrible consequences.

How do people choose a public API, an API as a product? Like with an everyday thing, they look at its interface and documentation. They go to the API's developer portal, read the documentation, and analyze the API to understand what it allows them to do and how to use it. They evaluate whether it will let them achieve their purpose effectively and simply. Even the best documentation will not be able to hide design flaws that make an API hard or even dangerous to use. And if they spot such flaws, these potential users, these potential customers, will not choose the API. No customers, no revenue. This could lead to a company going bankrupt.

What if some people decide to use the flawed API anyway? Sometimes, users may simply not detect flaws at first sight. Sometimes, users may have no other choice than using such terrible APIs. This can happen, for example, inside an organization. Most of the time people have no other choice but to use terrible private APIs or terrible APIs provided by commercial off-the-shelf applications.

Whatever the context, design flaws increase the time, effort, and money needed to build software using the API. The API can be misused or underused. The users may need extensive support from the API provider, raising costs on the provider side. These are potentially grave consequences for both private and public APIs. What's more, with public APIs, users can complain publicly or simply stop using them, resulting in fewer customers and less revenue for the API providers.

Flawed API design can also lead to security vulnerabilities in the API, such as exposing sensitive data inadvertently, neglecting access rights and group privileges, or placing too much trust in consumers. What if some people decided to exploit such vulnerabilities? Again, the consequences could be disastrous for the API consumers and providers.

In the API world, it's often possible to fix a poor design after the API goes into production. But there is a cost: it will take time and money for the provider to fix the mess, and it might also seriously bother the API's consumers.

These are only a few examples of harmful impacts. Poorly designed APIs are doomed to fail. What can be done to avoid such a fate? It's simple: *learn how to design APIs properly*.

1.3 The elements of API design

Learning to design APIs is about far more than just learning to design programming interfaces. Learning to design APIs requires learning principles, and not only technologies, but also requires knowing all facets of API design. Designing APIs requires one to not only focus on the interfaces themselves, but to also know the whole context surrounding those and to show empathy for all users and the software involved. Designing APIs without principles, totally out of context, and without taking into consideration both sides of the interface—the consumer's side and also the provider's—is the best way to ensure a total failure.

1.3.1 Learning the principles beyond programming interface design

When (good) designers put a button in a specific spot, choose a specific form, or decide to add a red LED on an object, there is a reason. Knowing these reasons helps designers to create good interfaces for everyday items that will let people achieve their goals as simply as possible—whether these items are doors, washing machines, mobile applications, or anything else. It's the same for APIs.

The purpose of an API is to let people achieve their goals as simply as possible, whatever the *programming* part. Fashions come and go in software. There have been and there will be many different ways of exposing data and capabilities through software. There have been and there will be many different ways of enabling software communication over a network. You may have heard about RPC, SOAP, REST, gRPC, or GraphQL. You can create APIs with all of these technologies: some are architectural styles, others are protocols or query languages. To make it simpler, let's call them *API styles*.

Each API style can come with some (more or less) common practices that you can follow, but they will not stop you from making mistakes. Without knowing fundamental principles, you can be a bit lost when choosing a so-called common practice; you can struggle to find solutions when facing unusual use cases or contexts not covered by common practices. And if you switch to a new API style, you will have to relearn everything.

Knowing the fundamental principles of API design gives you a solid foundation with which to design APIs of any style and to face any design challenge. But knowing such design principles is only one facet of API design.

1.3.2 Exploring all facets of API design

Designing an interface is about far more than placing buttons on the surface of an object. Designing an object such as a drone remote controller requires designers to know what its purpose is and what users want to achieve using it. Such a device is supposed to control the speed, elevation, and direction of a flying object. This is what users want to do, and they don't care if it is done using radio waves or any other technology.

All these actions must be represented by a user interface with buttons, joysticks, sliders, or some other type of control. The purpose of these controls and their representations must make sense, they must be easily usable by users, and most importantly, they must be totally secure. The UDRC 1138's interface is a perfect example of a totally unusable and unsecure interface with its perplexing LCD or buttons and its absence of a safety control.

The design of such a remote control must also take into consideration the whole context. How will it be used? For example, if it is to be used in extreme cold, it would be wise to use controls that can be operated with bulky gloves. Also, the underlying technology may add some constraints on the interface. For example, transmitting an order to the drone cannot be done more than *X* times per second.

Finally, how could such a terrible design be put into production? Maybe the designers involved were not trained sufficiently and did not get enough guidance. Maybe the design never was validated. If only some people, maybe potential users, had reviewed this design, its blatant flaws probably could have been fixed. Maybe the designers created a good model but, in the end, their plan was not respected at all.

Whatever its quality, once people get used to an object and its interface, changes must be made with extreme caution. If a new version of this remote controller were to come out with a totally different button organization, users might not be willing to buy the new version because they would have to relearn how to operate the controller.

As you can see, designing an object's interface requires focusing on more than just buttons. It's the same with API design.

Designing an API is about far more than just designing an easy-to-understand and easy-to-use interface on your own. We must design a totally secure interface—not unduly exposing sensitive data or actions to consumers. We must take the whole context into consideration—what the constraints are, how the API will be used and by whom, how the API is built, and how it could evolve. We have to participate in the whole API lifecycle, from early discussions to development, documentation, evolution, or retirement, and everything in between. And as organizations usually build many APIs, we should work together with all other API designers to ensure that all of the organization's APIs have a similar look and feel in order to build individual APIs that are as consistent as possible, thus ensuring that the sum of all the APIs is as easy to understand and easy to use as each individual one.

Summary

- Web APIs turn software into reusable bricks that can be used over a network with the HTTP protocol.
- APIs are interfaces for developers who build the applications consuming them.
- API design matters for all APIs—public or private.
- Poorly designed APIs can be underused, misused, or not used at all, and even unsecure.
- Designing a good API requires that you take the whole context of the application into consideration, not only the interface itself.

Часть I

Основы проектирования API

Каждое путешествие начинается с первого шага, и проектирование API не является исключением. Проектировщикам API необходимо обладать множеством навыков и нужно учитывать множество тем при создании API, но без прочной основы все передовые навыки и темы ничего не стоят. Это то, о чем вы узнаете в первой части.

Сначала мы рассмотрим ситуацию, объяснив, что такое API, почему его нужно проектировать и что на самом деле означает *обучение проектированию API*. Вы узнаете, что, будучи программными интерфейсами, API – это больше, чем просто «связующее звено», и что для проектирования любого типа API необходимо изучить фундаментальные принципы.

Еще до того, как задуматься о программировании, вы увидите, что API нужно рассматривать с точки зрения его пользователей. Предполагается, что API позволяет вашим пользователям легко достигать своих целей, а не системам, представляющим API. Только после того, как эти цели станут известны и точно описаны, можно спроектировать фактический интерфейс программирования, такой как REST API. И, как и любое программирование, описание интерфейса программирования должно выполняться с помощью адаптированного инструмента, такого как спецификация OpenAPI для REST API.

1 Что такое проектирование API

В этой главе вы узнаете:

- что такое API;
- почему важно проектирование API;
- что означает проектирование API.

Веб-API (программные интерфейсы приложения) являются неотъемлемой частью нашего взаимосвязанного мира. Программное обеспечение использует эти интерфейсы для обмена данными – от приложений на смартфонах до глубоко скрытых серверов баз данных, API-интерфейсы присутствуют абсолютно везде. Считаются ли они простыми техническими интерфейсами или продуктами сами по себе, целые системы, независимо от размера и назначения, зависят от них. То же самое делают целые компании и организации, начиная с технологических стартапов и интернет-гигантов, заканчивая нетехническими малыми и средними предприятиями, крупными корпорациями и государственными структурами.

Если API-интерфейсы являются неотъемлемой частью нашего взаимосвязанного мира, их проектирование – это его основа. При создании и развитии системы на базе API, независимо от того, является ли она видимой для кого-либо или глубоко скрытой, создает ли она один или несколько API-интерфейсов, проектирование всегда должно быть основной задачей. Успех или неудача такой системы напрямую зависит от качества проектирования всех ее API. Но что на самом деле означает проектирование API? И что нужно изучать, чтобы проектировать

их? Чтобы ответить на эти вопросы, нужно рассмотреть, что такое API и для кого они предназначены, а также понять, что проектирование API – это больше, чем просто проектирование программного интерфейса для приложений.

1.1. Что такое API?

Миллиарды людей владеют смартфонами и используют их для обмена фотографиями в социальных сетях. Без API это было бы невозможно. Обмен фотографиями с помощью мобильного приложения для социальных сетей предполагает использование различных типов API, как показано на рис. 1.1.

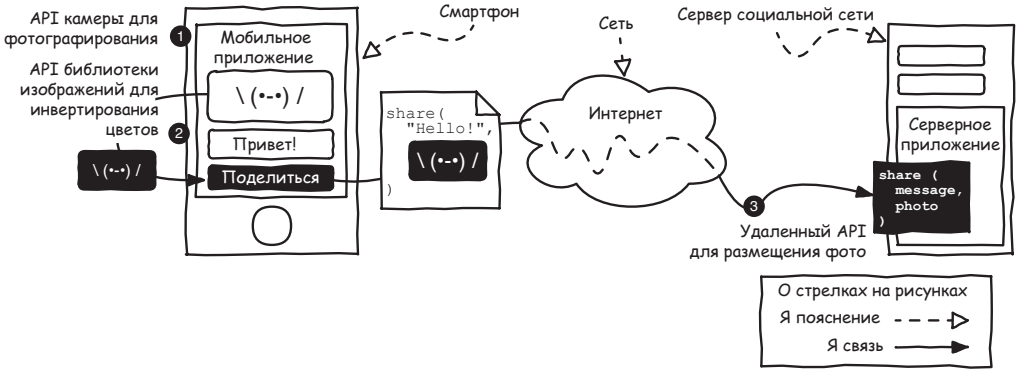

Рис. 1.1. Три разных типа API

Во-первых, чтобы сделать фото, мобильное приложение социальной сети использует камеру смартфона с помощью API. Затем через API оно может использовать некую библиотеку изображений, встроенную в приложение, для инвертирования цветов фотографии. И в итоге оно делится измененной фотографией, отправляя ее на серверное приложение, размещенное на сервере социальной сети, используя удаленный API, доступ к которому можно получить через сеть, обычно через интернет. Таким образом, в этом сценарии задействованы три различных типа API: аппаратный API, библиотека и удаленный API, соответственно. Эта книга посвящена последнему типу.

API-интерфейсы, независимо от своего типа, упрощают создание программного обеспечения, но удаленные API, особенно веб-API, изменили способ создания программного обеспечения. В настоящее время любой может легко создать что-либо, собрав удаленные части программного обеспечения. Но, прежде чем говорить о безграничных возможностях, предоставляемых такими API, давайте разберемся, что на самом деле подразумевается под термином *API* в этой книге.

1.1.1. API – это веб-интерфейс для программного обеспечения

В этой книге API – это удаленный API, а точнее, веб-API – веб-интерфейс для программного обеспечения. API, независимо от типа, прежде всего

является интерфейсом: точкой, где две системы, субъекта, организации и т. д. встречаются и взаимодействуют. Поначалу концепция API может быть непростой для понимания, но рис. 1.2 делает ее более осязаемой, сравнивая ее с пользовательским интерфейсом приложения.

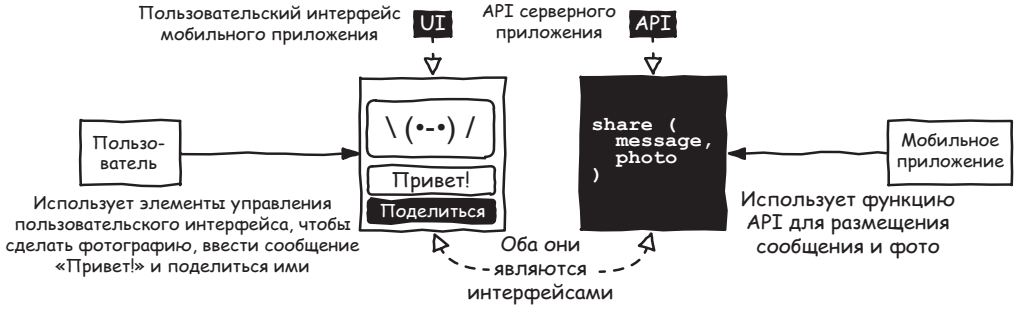

Рис. 1.2. Сравнение пользовательского интерфейса приложения с программным интерфейсом приложения (API)

Будучи пользователем мобильного приложения, вы взаимодействуете с ним, касаясь экрана вашего смартфона, на котором отображается пользовательский интерфейс приложения. Пользовательский интерфейс мобильного приложения может предоставлять такие элементы, как кнопки, текстовые поля или метки на экране. Эти элементы позволяют пользователям взаимодействовать с приложением для просмотра или предоставления информации, а также для запуска таких действий, как обмен сообщениями и фотографиями.

Так же как мы (люди) используем пользовательский интерфейс приложения для взаимодействия с ним, это приложение может использовать еще одно приложение с помощью своего программного интерфейса. Принимая во внимание, что пользовательский интерфейс предоставляет поля ввода данных, метки и кнопки, чтобы обеспечить обратную связь, которая может развиваться по мере их использования, API предоставляет функции, которые могут нуждаться во входных данных или которые могут возвращать выходные данные в качестве ответа. Эти функции позволяют другим приложениям взаимодействовать с приложением, предоставляющим API, для извлечения или отправки информации или для запуска действий.

Строго говоря, API – это *только* интерфейс, предоставляемый неким программным обеспечением. Это абстракция базовой *реализации* (базовый код – это то, что на самом деле происходит внутри программного продукта при использовании API). Но обратите внимание, что термин *API* часто используется для обозначения всего программного продукта, включая API и его реализацию.

Таким образом, API – это интерфейсы для программного обеспечения, но API, о которых мы говорим в этой книге, – это больше, чем просто API: это веб-API, как показано на рис. 1.3.

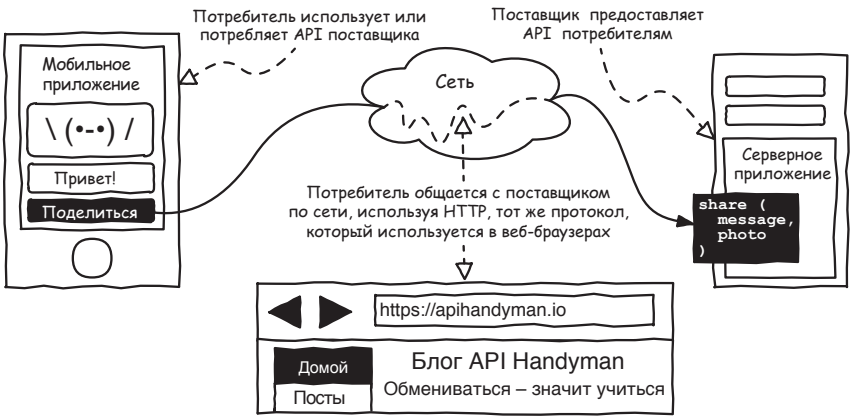

Рис. 1.3. Веб-API – это удаленные API, которые можно использовать с протоколом HTTP

Мобильное приложение, работающее в смартфоне, использует API, предоставляемый серверным приложением (часто именуемым бэкенд-приложением или просто *серверной частью*), которое размещено на удаленном сервере. Поэтому мобильное приложение называется *потребителем*, а серверная часть называется *поставщиком*. Данные термины также применимы к компаниям и людям, создающим приложения, или использующим или предоставляющим API. Здесь это означает, что разработчики мобильного приложения являются потребителями, а разработчики бэкенд-приложения – поставщиками.

Для обмена данными со своим бэкенд-приложением мобильное приложение обычно использует известную сеть: интернет. Здесь интересен не сам интернет – такой обмен данными также может осуществляться через локальную сеть, – а то, *как* эти два приложения обмениваются данными по сети. Когда мобильное приложение отправляет фотографию и сообщение бэкенд-приложению, оно использует Протокол передачи гипертекста (HTTP). Если вы открываете веб-браузер на компьютере или смартфоне, то используете HTTP (косвенно). Это протокол, который применяется любым веб-сайтом. Когда вы вводите адрес веб-сайта, например http://apihandyman.io или его защищенную версию, https://apihandyman.io, в адресной строке и нажимаете **Enter** или щелкаете по ссылке в браузере, браузер использует HTTP для связи с удаленным сервером, на котором размещен веб-сайт, чтобы показать вам содержимое сайта. Удаленные API или, по крайней мере, те, о которых мы говорим в этой книге, используют этот протокол так же, как веб-сайты; вот почему они называются *веб-API*.

Итак, в этой книге API – это веб-API. Это веб-интерфейсы для программного обеспечения. Но чем они так интересны?

1.1.2. API превращают программное обеспечение в детали конструктора LEGO®

Тысячи, даже миллионы мобильных приложений и их бэкендов были созданы благодаря веб-API, но это еще не все. Веб-API развивают твор-

ческий потенциал и инновации, превращая программное обеспечение в многократно используемые блоки, которые можно легко собрать. Давайте вернемся к нашему примеру и посмотрим, что может произойти, если разместить фото в социальной сети.

Когда бэкенд социальной сети получает фотографию и сообщение, он может сохранить фотографию в файловой системе сервера, а сообщение и идентификатор фотографии (для последующего извлечения фактического файла) – в базе данных. Он также может обработать фотографию, используя некий самодельный алгоритм распознавания лиц, чтобы определить, есть ли ней ваши друзья, перед тем как сохранить фотографию. Это одна из возможностей – одно приложение, обрабатывающее все для другого отдельного приложения. Давайте рассмотрим что-нибудь другое, как показано на рис. 1.4.

Внутренний API может использоваться как мобильным приложением социальной сети, так и веб-сайтом, и его реализация может быть совершенно иной. Когда серверная часть получает фотографию и сообщение для совместного использования (какое бы приложение его ни отправляло), она может делегировать хранение фотографии в качестве сервисной компании через свой API. Он также может делегировать хранение идентификатора сообщения и фотографии внутреннему программному модулю хроники через его API. Как обрабатывать распознавание лиц? Что же, это можно было бы делегировать какому-нибудь специалисту по распознаванию лиц, предлагающему свои услуги через... как вы уже догадались... API.

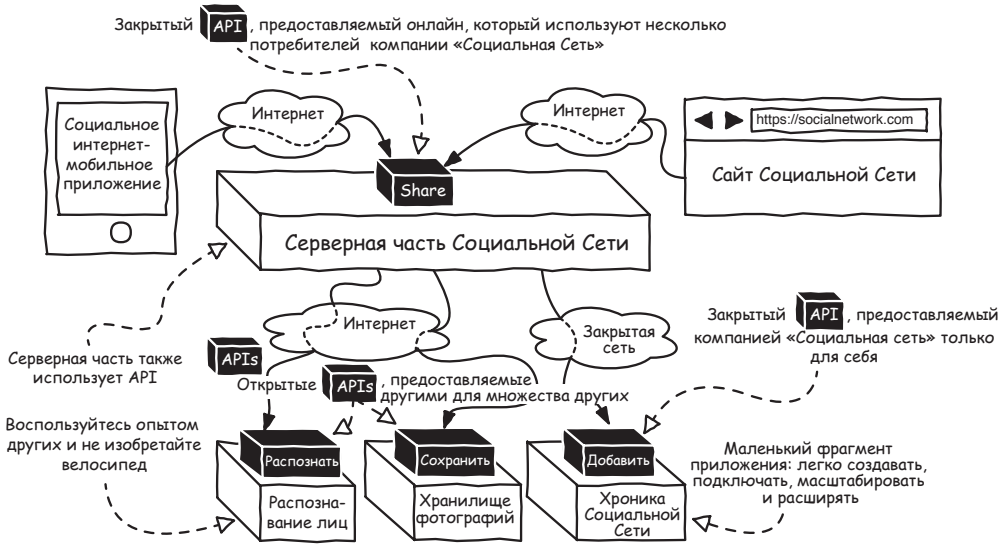

Рис. 1.4. Система, состоящая из открытых и закрытых программных деталей LEGO®, подключенных через API

Обратите внимание, что на рис. 1.4 каждый API предоставляет только одну функцию. Это делает рисунок максимально простым: один API мо-

жет предоставлять множество функций. Серверная часть может, например, предоставлять такие функции, как **Добавить друга**, **Перечислить друзей** или **Получить хронику**.

Это похоже на программную версию системы деталей конструктора LEGO; это те пластиковые блоки, которые можно собирать, чтобы создавать новые вещи. (Аристотель, вероятно, играл с некоторыми из них, когда понял, что «целое больше, чем сумма его частей».) Возможности тут бесконечны – единственным ограничением является ваше воображение.

Когда я был ребенком, я целыми часами играл в LEGO, собирая здания, машинки, самолеты, космические корабли или все что угодно. Когда мне надоедало одно из моих творений, я мог полностью уничтожить его и начать что-то новое с нуля или мог переделать его, заменив некоторые детали. Я мог бы даже собрать существующие структуры вместе, например чтобы создать массивный космический корабль. То же самое происходит и в мире API: вы можете разложить огромные системы программных блоков, которые можно легко собрать и даже заменить благодаря API, но есть небольшие отличия.

Каждый программный блок может одновременно использоваться и многими другими. В нашем примере внутренний API может использоваться мобильным приложением и веб-сайтом. Обычно API предназначен для использования не одним потребителем, а множеством. Таким образом, вам не нужно все время приступать к повторной разработке.

Каждый программный блок может работать где угодно самостоятельно, если он подключен к сети, чтобы быть доступным через API. Это хороший способ управления производительностью и масштабируемостью; программный блок, такой как блок распознавания лиц на рис. 1.4, вероятно, потребует гораздо больше ресурсов обработки по сравнению с хроникой социальной сети. Если первым управляет компания Social Network, его можно установить на другом, выделенном и более мощном сервере, тогда как хроника Social Network работает на сервере поменьше. А доступность через простое сетевое соединение позволяет любому API, предоставленному кем-либо, использоваться кем-либо.

В мире API существует два типа блоков, представляющих два типа API: *открытые API* и *закрытые API*. Программные блоки распознавания лиц и хранения фотографий создаются и даже управляются не компанией Social Network, а третьими лицами. Предоставляемые ими API являются открытыми.

Открытые API-интерфейсы предлагаются *в качестве сервиса* или *продукта* другими; вы не создаете их, не устанавливаете и не запускаете – вы только используете их. Открытые API предоставляются всем, кто в них нуждается и готов принять условия стороннего поставщика. В зависимости от бизнес-модели провайдеров API такие API могут быть бесплатными или платными, как и любой другой программный продукт. Такие открытые API-интерфейсы раскрывают творческий потенциал и инновации, а также могут значительно ускорить создание всего того, о чем вы можете мечтать. Перефразируя Стива Джобса, можно сказать,

что для этого есть API. Зачем терять время, пытаясь изобрести колесо, которое все равно будет недостаточно круглым? В нашем примере компания Social Network решила сосредоточиться на своем основном опыте, соединяя людей, и делегировала распознавание лиц третьей стороне.

Но открытые API – это только вершина айсберга. Серверная часть и временная шкала были созданы компанией Social Network для собственного использования. API хроники используются только приложениями, созданными этой компанией, как показано на рис. 1.4. То же самое касается мобильного бэкенда, который используется мобильным приложением Social Network. Эти API являются *закрытыми*, и их – миллиарды. Закрытый API – это API, который вы создаете для себя: его используют только приложения, созданные вами или сотрудниками вашей команды, отдела или компании. В данном случае вы являетесь поставщиком и потребителем собственного API.

ПРИМЕЧАНИЕ. Вопрос открытости/закрытости не в том, *как* API предоставляется, а *кому*. Даже будучи размещенным в интернете, API мобильного бэкенда по-прежнему является закрытым.

Между *настоящими* закрытыми и открытыми API могут быть различные виды взаимодействия. Например, вы можете установить коммерческое готовое программное обеспечение, такое как система управления контентом (CMS) или система управления взаимоотношениями с клиентами (CRM), на собственные серверы (такая установка часто называется *on-premise*), и эти приложения могут (и даже должны!) предоставлять API. Эти API являются закрытыми, но вы не создаете их сами. Тем не менее вы можете использовать их по своему усмотрению, в особенности для подключения большего количества блоков к своим блокам. Возьмем другой пример. Вы можете предоставить некоторые из ваших API-интерфейсов клиентам или выбранным партнерам. Такие *почти открытые* API часто называют *партнерскими*.

Но независимо от ситуации API в основном превращают программное обеспечение в программные блоки многократного использования, которые вы или другие пользователи могут собрать, чтобы создать модульные системы, способные выполнять абсолютно все. Вот почему API так интересны. Но почему их разработка так важно?

1.2. Чем важно проектирование API

Даже если это полезно, API кажется лишь техническим интерфейсом для программного обеспечения. Так в чем состоит важность проектирования такого интерфейса?

API-интерфейсы используются программным обеспечением, это правда. Но кто создает программное обеспечение, которое их использует? Разработчики. Люди. Эти люди ожидают, что эти программные интерфейсы будут полезными и простыми, как и любой другой (хорошо спроектированный) интерфейс. Подумайте о своей реакции, когда вы сталкиваетесь с плохо спроектированным веб-сайтом или пользователь-

ским интерфейсом мобильного приложения. Что вы чувствуете, когда сталкиваетесь с плохо продуманными повседневными вещами, такими как пульт дистанционного управления или даже дверь? Это может вызвать у вас раздражение, вероятно, вы даже рассердитесь или разразитесь тирадой и вряд ли захотите когда-либо пользоваться ими. А в некоторых случаях плохо спроектированный интерфейс может быть даже опасным. Вот почему проектирование любого интерфейса имеет значение, и API не являются исключением.

1.2.1. Открытый или закрытый API – это интерфейс для других разработчиков

В разделе 1.1.1 вы узнали, что потребителем API может быть либо программное обеспечение, использующее API, либо компания, либо отдельные лица, разрабатывающие это программное обеспечение. Все эти потребители важны, но первый, кого нужно принимать во внимание, – это разработчик.

Как вы уже видели, в этом примере с социальными сетями участвуют разные API. Существует модуль хроники, который управляет хранением данных и предоставляет закрытый API. И есть открытые (предоставляемые другими компаниями) API распознавания лиц и хранения фотографий. Бэкенд, который вызывает эти три API, не появляется сам по себе; он разработан компанией Social Network.

Например, чтобы использовать API распознавания лиц, разработчики пишут код в программном обеспечении Social Network, чтобы отправлять фотографии для распознавания лиц и обрабатывать результат обработки фотографий, как при использовании программной библиотеки. Эти разработчики не являются теми, кто создал API распознавания лиц, и они, вероятно, не знают друг друга, потому что они из разных компаний. Мы также можем представить, что мобильное приложение, веб-сайт, серверная часть и модуль хранения данных разрабатываются разными группами внутри компании. В зависимости от организации компании эти команды могут хорошо знать друг друга или не знать вообще. И даже если каждый разработчик в компании знает все маленькие секреты каждого API, который она разработала, неизбежно появятся новые разработчики.

Таким образом, будь то открытый или закрытый API независимо от того, по какой причине он создан и какой бы ни была организация компании, рано или поздно он будет использоваться другими разработчиками – людьми, которые не участвовали в создании программного обеспечения, предоставляющего API. Вот почему нужно сделать все, для того чтобы при написании кода этим новичкам было легко использовать API. Разработчики ожидают, что API будут полезны и просты, как и любой интерфейс, с которым им приходится взаимодействовать. Вот почему так важно проектирование API.

Опыт разработчика

API (DX) – это опыт, который получают разработчики при использовании API. Он охватывает множество различных тем, таких как регистрация (для использования API), документирование (чтобы узнать, что делает API и как его использовать) или поддержка (чтобы получить помощь при возникновении проблем). Но все усилия ничего не стоят, если не решен самый важный вопрос – проектирование API.

1.2.2 API создается, для того чтобы скрыть реализацию

Проектирование API имеет значение, потому что, когда люди используют API, они хотят использовать его, не беспокоясь о мелочах, которые не имеют к ним никакого отношения. А для этого разработка должна скрывать детали реализации (что на самом деле происходит). Позвольте мне использовать реальную аналогию в качестве объяснения.

Скажем, вы решили пойти в ресторан. Как насчет французского, например? Когда вы идете в ресторан, то становитесь клиентом. Как клиент ресторана, вы читаете его меню, чтобы узнать, какие блюда можно заказать. Вы решаете попробовать миногу по-бордосски (знаменитое французское рыбное блюдо из области Гасконь). Чтобы заказать выбранную еду, вы говорите с (обычно очень приятным и дружелюбным) человеком, которого называют официантом или официанткой. Спустя некоторое время официант возвращается и приносит заказанное вами блюдо – миногу по-бордосски, – которое приготовили на кухне. Пока вы едите свой вкусный обед, можно задать вам два вопроса?

Первое: вы знаете, как приготовить миногу по-бордосски? Наверное, нет, и, возможно, по этой причине вы и идете в ресторан. И даже зная рецепт приготовления, вы, вероятно, не захотите этого делать, потому что это сложно и для этого требуются труднодоступные ингредиенты. Вы отправляетесь в ресторан за блюдом, которое не умеете или не хотите готовить.

Второе: знаете ли вы, что произошло между моментом, когда официант принял ваш заказ и когда принес его вам? Вы можете догадаться, что официант был на кухне, чтобы отдать заказ повару, который работает один. Этот повар очень старается и уведомляет официанта, когда блюдо будет готово, звоня в маленький колокольчик и крича: «Заказ для столика № 2 готов!» Но сценарий может немного отличаться.

Официант может использовать смартфон, чтобы принять ваш заказ, который мгновенно отображается на сенсорном экране на кухне. А там не одинокий повар, а целая бригада. Как только блюдо готово, один из членов бригады помечает ваш заказ как готовый на сенсорном экране, а официант получает уведомление на свой смартфон. Независимо от количества поваров, вы не знаете рецепт и ингредиенты, используемые для приготовления еды. Независимо от сценария, еда, которую вы заказали, поговорив с официантом, была приготовлена на кухне, и официант принес ее вам. В ресторане вы говорите только с официантом (или офи-

цианткой), и вам не нужно знать, что происходит на кухне. Какое отношение все это имеет к API? Самое прямое, как показано на рис. 1.5.

С точки зрения группы разработчиков мобильных приложений для социальных сетей, *размещение фото* с использованием бэкенд-API абсолютно одинаково, только бэкенд уже реализован. Разработчикам не нужно знать рецепт и его ингредиенты; они только предоставляют фото и сообщение. Им все равно, будет ли фотография проходить через распознавание изображений до или после добавления в хронику. Они также не заботятся о том, является ли серверная часть единственным приложением, написанным на языке Go или Java, которое обрабатывает все или использует другие API, написанные на NodeJS, Python или любом другом языке. Когда разработчик создает *потребительское приложение* (клиент), которое использует *приложение провайдера* (ресторан) через свой API (официант или официантка), чтобы сделать что-то (например, заказать еду), разработчик и приложение знают только об API; им не нужно знать, как сделать что-то самостоятельно или как программное обеспечение провайдера (кухня) будет это делать. Но скрыть реализацию недостаточно. Это важно, но это не то, что на самом деле нужно тем, кто использует API.

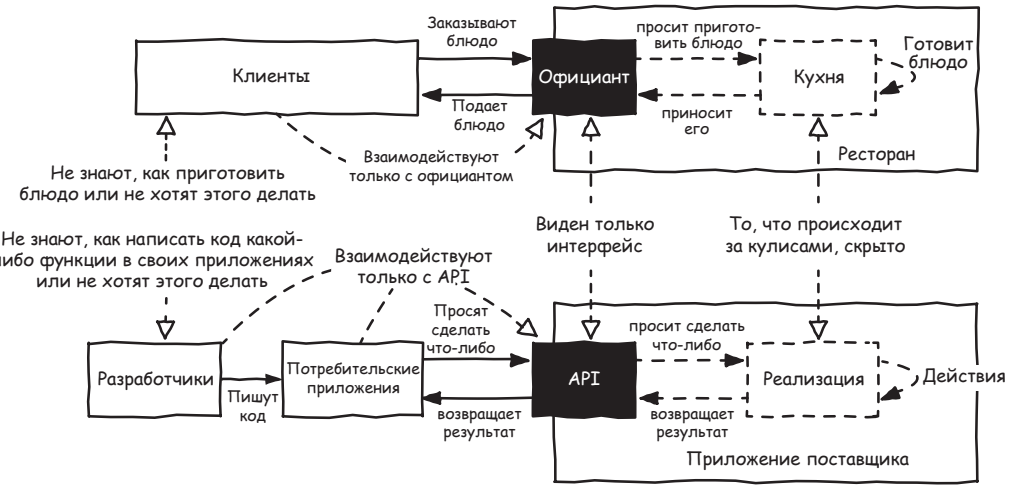

Рис. 1.5. Параллели между обедом в ресторане и использованием API

1.2.3. Страшные последствия плохо спроектированных API

Что вы делаете, когда впервые используете какую-либо повседневную вещь? Вы внимательно смотрите на ее интерфейс, чтобы определить ее назначение и то, как ее использовать, основываясь на том, что вы видите, и на своем прошлом опыте. И здесь важен дизайн.

Давайте рассмотрим гипотетический пример: устройство под названием UDRC 1138. Что это может быть за устройство? Какова его цель? Ну, его название не очень помогает. Возможно, его интерфейс может дать нам больше подсказок. Посмотрите на рис. 1.6.

Рис. 1.6. Загадочный интерфейс устройства UDRC 1138

Справа есть шесть немаркированных треугольных и прямоугольных кнопок. Может, это что-то вроде запуска и остановки? Ну, как кнопки медиаплеера. Четыре другие кнопки имеют незнакомую форму без малейшего намека на их назначение. На ЖК-дисплее отображаются номера без меток с такими единицами измерения, как ft, NM, rad и km/h. Можно предположить, что *ft* – это расстояние в футах, а *km/h* – скорость в километрах в час, но что могут означать *rad* и *NM*? Кроме того, в нижней части ЖК-экрана также появляется тревожное сообщение, предупреждающее о том, что мы можем вводить значения вне допустимого диапазона без контроля безопасности.

Этот интерфейс, безусловно, трудно расшифровать, и он не очень интуитивно понятен. Давайте посмотрим на документацию, изображенную на рис. 1.7, чтобы увидеть, что в ней говорится об этом устройстве.

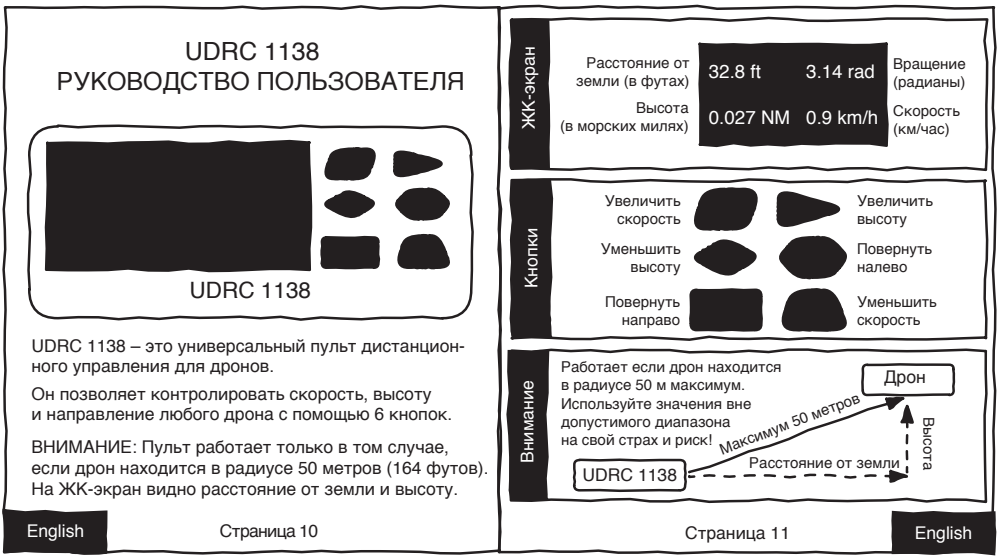

Рис. 1.7. Документация по UDRC 1138

Согласно описанию, это устройство представляет собой универсальный пульт дистанционного управления дроном – отсюда и название UDRC 1138, я полагаю, – который можно использовать для управления любым дроном. Ну что же, звучит интересно. На странице справа приве-

дено несколько пояснений по ЖК-экрану, кнопкам и сообщение с предупреждением.

Описание ЖК-экрана довольно загадочное. Расстояние от земли указано в футах, высота – в морских милях, ориентация дронов обеспечивается радианами, а скорость указана в километрах в час. Я не знаком с авиационными единицами измерения, но чувствую, что с выбранными единицами что-то не так. Они кажутся противоречивыми. Смесь футов и метров? И во всех фильмах о самолетах, которые я видел, футы используются для измерения высоты, а морские мили – для измерения расстояния, а не наоборот. Этот интерфейс определенно не интуитивно понятен.

Глядя на описания кнопки, видно, что для увеличения высоты используется треугольная кнопка в правом верхнем углу, а для ее уменьшения – ромбовидная кнопка во втором ряду. Связь неочевидна – почему эти элементы управления не идут друг за другом? Другие элементы управления также используют формы, которые абсолютно ничего не значат и кажутся расположенными случайным образом. Это безумие! Как использовать это устройство? Почему бы не взять старые добрые джойстики или геймпады вместо кнопок?

И последнее, но не менее важное объяснение предупреждающего сообщения. Похоже, что беспилотник может работать только в диапазоне 50 м – диапазон рассчитывается на основе расстояния до земли и высоты, обеспечиваемой ЖК-экраном. Подождите, что? Мы должны рассчитать расстояние между пультом дистанционного управления и дроном, используя теорему Пифагора?! Это полная чушь – устройство должно делать это за нас.

Вы бы купили или хотя бы попробовали в деле такой гаджет? Вероятно, нет. Но что, если у вас нет другого выбора, кроме как использовать это ужасное устройство? Что же, желаю удачи; вам понадобится что-то сделать с таким плохо спроектированным интерфейсом!

Вы, вероятно, думаете, что такой катастрофический интерфейс на самом деле не может существовать в реальной жизни. Конечно, проектировщики не могли создать такой ужасный прибор. И даже если бы они это сделали, конечно, отдел обеспечения качества никогда бы не пустил его в производство! Но давайте посмотрим правде в глаза. Плохо спроектированные устройства – вещи с плохо спроектированными интерфейсами для повседневного использования – все время *поступают* в производство.

Подумайте, сколько раз вы были озадачены или ворчали при работе с устройством, веб-сайтом или приложением, потому что в его дизайне были дефекты. Сколько раз вы решали не покупать или не использовать что-то из-за его дизайна? Сколько раз вы не могли применить что-то, или применить это правильно, или как вы хотели, потому что его интерфейс был непостижим?

Плохо спроектированный продукт может быть использован не по назначению, недоиспользован или не использован вообще. Это может даже быть опасно для его покупателей и для компании, которая созда-

ла его, и репутация такой компании может быть под угрозой. И если это физическое устройство, после того как оно будет запущено в производство, исправлять его будет слишком поздно.

Страшные недостатки дизайна не ограничиваются интерфейсами повседневных вещей. К сожалению, API также могут страдать от этого. Напомню, что API – это интерфейс, который разработчики должны использовать в своем программном обеспечении. Проектирование имеет значение независимо от типа интерфейса, и API-интерфейсы не являются исключением. Плохо спроектированные API-интерфейсы могут стать таким же разочарованием, что и устройства, подобные UDRC 1138; они могут быть очень трудным для понимания и использования, а это может иметь ужасные последствия.

Как люди выбирают открытый API, API в качестве продукта? Как и в повседневной жизни, они смотрят на его интерфейс и документацию. Они заходят на портал разработчиков API, читают документацию и анализируют API, чтобы понять, что он позволяет делать и как его использовать. Они оценивают, позволит ли он им эффективно и просто достичь своей цели. Даже самая совершенная документация не сможет скрыть недостатки проектирования, которые делают API трудным или даже опасным для использования. И если такие недостатки будут обнаружены, возможные пользователи (эти потенциальные клиенты) не станут выбирать этот API. Нет клиентов – нет дохода. Это может привести к банкротству компании.

Что, если кто-то решит воспользоваться некорректным API в любом случае? Иногда пользователи могут просто не обнаружить недостатки с первого взгляда. А иногда у них просто может не быть альтернативы. Такое может произойти, например, внутри какой-то компании. Чаще всего у людей нет другого выбора, кроме как использовать ужасные закрытые API или API, предоставляемые коммерческими готовыми приложениями.

Независимо от контекста недостатки проектирования увеличивают время, усилия и траты, необходимые для создания программного обеспечения с использованием API. API может быть использован неправильно или недоиспользован. Клиентам может потребоваться обширная поддержка со стороны поставщика API, что увеличивает затраты на стороне поставщика. Это потенциально серьезные последствия как для закрытых, так и для открытых API. Более того, благодаря открытым API-интерфейсам пользователи могут открыто пожаловаться или просто прекратить их применение, что приведет к уменьшению количества клиентов и снижению доходов для поставщиков API.

Ошибочное проектирование API также может привести к уязвимостям безопасности в API, таким как непреднамеренное раскрытие конфиденциальных данных, пренебрежение правами доступа и привилегиями группы или слишком большое доверие к потребителям. Что, если некоторые решат воспользоваться такими уязвимостями? Последствия для потребителей и поставщиков API могут быть катастрофическими.

В мире API плохое проектирование часто можно исправить, после того как API будет запущен в производство. Но издержки неизбежны:

поставщику понадобятся время и деньги, чтобы исправить ситуацию, и это может также серьезно беспокоить потребителей API.

Это лишь несколько примеров вредного воздействия. Плохо спроектированные API обречены на провал. Что можно сделать, чтобы избежать подобной участи? Все просто: *научиться правильно проектировать API*.

1.3. Элементы проектирования API

Научиться проектировать API – это гораздо больше, чем просто научиться проектировать программные интерфейсы. Обучение проектированию API требует изучения принципов, и не только технологий, но и знания всех аспектов проектирования. Для проектирования API важно не просто сосредоточиться на самих интерфейсах, но также знать весь контекст, окружающий их, и проявлять эмпатию ко всем пользователям и программному обеспечению. Проектирование API без принципов, полностью вне контекста и без учета обеих сторон интерфейса – как потребителя, так и поставщика, – лучший способ гарантировать полный провал.

1.3.1 Изучение принципов, выходящих за рамки проектирования программного интерфейса

Когда (хорошие) проектировщики помещают кнопку в определенное место, выбирают конкретную форму или решают добавить красный светодиод на объект, на то есть причина. Знание этих причин помогает проектировщикам создавать хорошие интерфейсы для повседневных вещей, которые позволят людям достичь своей цели как можно проще – будь то двери, стиральные машины, мобильные приложения или что-либо еще. То же самое происходит и в случае с API.

Цель API состоит в том, чтобы позволить людям достичь своих целей настолько просто, насколько это возможно, независимо от части, касающейся *программирования*. В программном обеспечении мода приходит и уходит. Было и будет много разных способов раскрытия данных и возможностей с помощью программного обеспечения. Было и будет много разных методов обеспечения обмена данными по сети. Возможно, вы слышали о RPC, SOAP, REST, gRPC или GraphQL. Вы можете создавать API-интерфейсы с помощью всех этих технологий: некоторые из них – это архитектурные стили, другие – протоколы или языки запросов. Чтобы было проще, будем называть их *стилями API*.

Каждый стиль API может сопровождаться некоторыми (более или менее) общепринятыми практиками, которым вы можете следовать, но они не защитят вас от ошибок. Не зная основополагающих принципов, вы можете растеряться при выборе так называемой общепринятой практики; вы можете изо всех сил пытаться найти решения, сталкиваясь с необычными случаями использования или контекстами, которые подобного рода практики не охватывают. И если вы переключитесь на новый стиль API, вам придется все изучать заново.

Знание основополагающих принципов проектирования API дает вам прочную основу для проектирования API любого стиля и решения лю-

бых задач проектирования. Но знание таких принципов – лишь один из аспектов проектирования.

1.3.2 Изучение всех аспектов проектирования API

Проектирование интерфейса – это гораздо больше, чем просто размещение кнопок на поверхности объекта. Создание такого объекта, как пульт дистанционного управления дроном, требует, чтобы проектировщики знали, какова его цель и чего хотят добиться люди, использующие его. Предполагается, что такое устройство должно контролировать скорость, высоту и направление летящего объекта. Это то, что нужно пользователям, и им все равно, будет ли это сделано с использованием радиоволн или любой другой технологии.

Все эти действия должны быть представлены в пользовательском интерфейсе с помощью кнопок, джойстиков, ползунков или других элементов управления. Назначение этих элементов управления и их представлений должно иметь смысл, чтобы у пользователей не возникало проблем при их применении, и, что самое важное, они должны быть полностью безопасными. Интерфейс UDRC 1138 является прекрасным примером абсолютно непригодного и небезопасного интерфейса с его ЖК-дисплеем, который сбивает с толку, или с кнопками и отсутствием управления безопасностью.

Конструкция такого пульта дистанционного управления также должна учитывать весь контекст. Как его станут использовать? Например, если он будет использоваться в условиях сильного холода, было бы разумно применять элементы управления, с которыми можно работать в громоздких перчатках. Кроме того, базовая технология может добавлять ограничения для интерфейса. Например, нельзя выполнять передачу приказа дрону более X раз в секунду.

Наконец, как такое можно запускать в производство? Возможно, принимавшие в этом участие проектировщики не были достаточно обучены и не получили должного руководства. Или этот дизайн так и не был утвержден. Если бы только кто-то – возможно, потенциальные пользователи – рассмотрел его, эти явные недостатки, вероятно, можно было бы исправить. Возможно, проектировщики создали хорошую модель, но в конце концов их план так и не был соблюден.

Независимо от качества, как только люди привыкнут к предмету и его интерфейсу, изменения нужно проводить с особой осторожностью. Если новая версия этого пульта дистанционного управления будет иметь совершенно иную организацию кнопок, пользователи, возможно, не захотят покупать новую версию, потому что им придется заново учиться работать с ним.

Как видите, проектирование интерфейса объекта требует сосредоточиться не только на кнопках. То же самое касается и проектирования API.

Проектирование API – это гораздо больше, чем просто проектирование простого и понятного интерфейса. Мы должны создать полностью безопасный интерфейс, ограничивая потребителям доступ к конфиденциальным данным или не давая им совершать лишние действия. Необ-

ходимо принимать во внимание весь контекст – каковы ограничения, как и кем будет использоваться API, как он создается и как может развиваться. Мы должны участвовать во всем жизненном цикле API – от первых обсуждений до проектирования, документирования, развития или вывода из эксплуатации. И поскольку компании обычно создают множество API, мы должны работать вместе с остальными проектировщиками, чтобы гарантировать, что все API организации имеют одинаковый внешний вид, чтобы создавать отдельные интерфейсы, которые были бы максимально согласованными, гарантируя таким образом, что все эти API так же просты для понимания и легки в использовании, как и каждый в отдельности.

Резюме

- Веб-API превращают программное обеспечение в блоки многократного использования, которые можно использовать по сети с помощью протокола HTTP.
- API – это интерфейсы для разработчиков, которые создают приложения, использующие их.
- Дизайн API важен для всех API – открытых или закрытых.
- Плохо спроектированные API-интерфейсы могут быть использованы недостаточно, неправильно или вообще не использоваться, и даже небезопасны.
- Проектирование хорошего API требует, чтобы вы учитывали весь контекст приложения, а не только сам интерфейс.