The Design of Everyday APIs

Arnaud Lauret

MEAP

MANNING

MEAP Edition
Manning Early Access Program

The Design of Everyday APIs

Version 1

welcome

Thank you for purchasing *The Design of Everyday APIs*. APIs are everywhere. If there are thousands of public APIs, then there are millions of private ones, and he numbers keep growing every day. And whether public or private, the design of an API is a wonderful thing to achieve, but also a tricky thing if you're not aware of all aspects of API design.

Designing an API is far more than just designing a programming interface giving access to capabilities and data of an underlying system. Forget the SOAP, REST, gRPC, of GraphQL discussions… an API designer's job is far more than just designing a programming interface using a given type.

An API is obviously an interface for software, but not only that. This interface is first and foremost an interface for people: developers. These people want to do things that matter to them. They will choose to use an API not based on its design. These people may at best lose time using it or at worst misuse it depending on its design. Such software interface may be used in a hostile environment, which may impact its design. This interface will be implemented, which also may impact design. And this interface will evolve, which is also a concern for the API designer.

This book is designed to uncover all of these aspects (and more) of API design to help you build an API designer mindset, allowing you to design any API. This book is intended to teach fundamental principles of API design that you can apply to any type of API. But learning theory without practice can be quite boring and counterproductive. So all principles will be applied to the design of REST APIs.

I want to make sure that this book really helps you with API design challenges and helps you build the API designer mindset. So, please share your experiences in the author online forums. Let me know if there is anything I can do to help you achieve this and thank you for reading the book.

—Arnaud Lauret

brief contents

1 What really is API Design

In this chapter:

- What is an API
- What is API design

APIs are an essential pillar of our connected world. They are used by software to communicate with each other. From applications on smartphones to deeply hidden backend servers or internet of thing devices, they are absolutely everywhere. Whole systems whatever their size and purpose rely heavily on them. From tech startups and giants to non tech SMEs, big corporations and government organizations, whole companies, whole organizations rely heavily on APIs.

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

The design of anything is a very delicate task that must consider and reconcile many different aspects. APIs are definitely no exception. API design may be learned through experience but at the cost of a some failures that could lead to a few scars. Without guidance, it could even twist your API designer mind in such a way that you may focus on some aspects and totally miss others. Leaving you with an incomplete view of what really is API design. Leaving you with a distorted vision of what really is good API design. This book aims to give you the eye of the API designer, to help you build the API designer mindset that will allow you to survive the design of any type of API.

To design a *thing*, you need to deeply understand what this *thing* is. To design a *thing* , you need to deeply understand what *design* represents for this *thing*. So to learn API design we need to deeply understand what an API is and what designing an API means. This is what we start to uncover in this chapter and what will study throughout this book.

1.1 What is an API

API is an acronym standing for *Application Programming Interface*. This is just one description of what an API is and it is not really a self explanatory one. Nevertheless, whatever an accurate API's description could be, you probably use some everyday. APIs are everywhere. In our connected world almost everyone may be a modern Molière's Mr Jourdain.

For more than forty years I have been speaking prose without knowing anything about it.
-- Mr Jourdain The Middleclass Gentleman by Molière

Like Molière's Mr Jourdain who discovered while learning poetry that he has been speaking prose (normal every day speech) all his life without knowing it, almost everyone may use APIs everyday without really realizing it and even without knowing anything about it. You do not believe me?

1.1.1 Everyday APIs

May I ask you a few questions?

- Do you try to get or stay in shape using a fitness tracker?
- Do you use social networks?
- Do you own a smart bulb?
- Do you spend hours seeking the best hotel at the cheapest price when you prepare your next vacations?
- Do you frenetically track your online shopping orders' shipment?
- Do you read posts on blogs and news websites?
- Do you look for anything you need with an internet search engine?
- Do you own a computer or a smartphone?

I answer yes to all questions, what about you? What about your friends and familly? I'm pretty sure that anyone can say yes to at least one of them. This turns almost anybody into a modern Molière's Mr Jourdain. Anyone may use API without realizing it. How could this be possible? Let's seek common features into all these use cases to answer to this question.

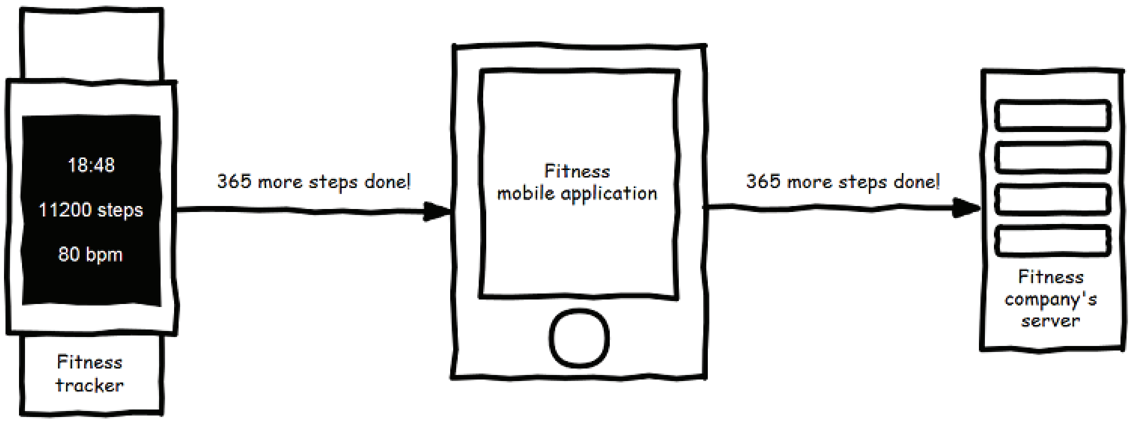


Figure 1.1 Counting steps with a fitness tracker

Fitness trackers collect data such as steps count or heart rate and communicate them to an application which is installed on your smartphone. This application can also send all these data to the servers of the company which created this fitness tracker.

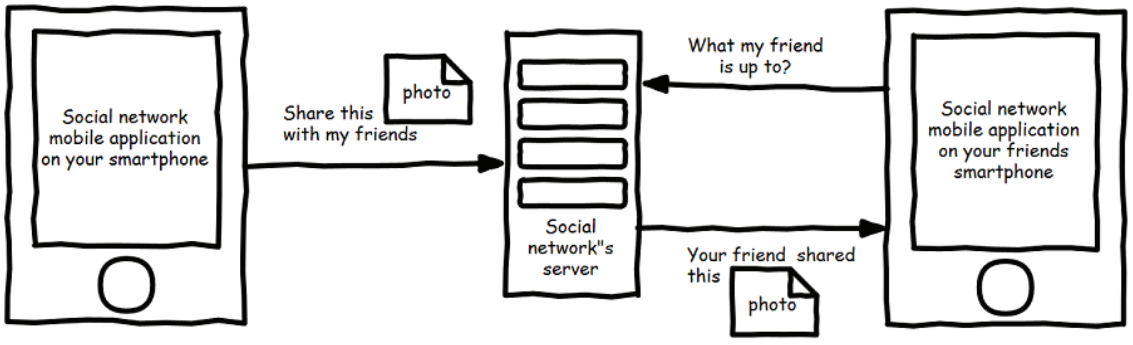

Social network mobile application on your smartphone
Share this with my friends
photo
Social network"s server
What my friend is up to?
Your friend shared this
photo
Social network mobile application on your friends smartphone

Figure 1.2 Sharing a photo on social network

When you share a photo using a social network's mobile application, it is sent to the social network's server so all your friends or followers can see it when they use the same application on their smartphone.

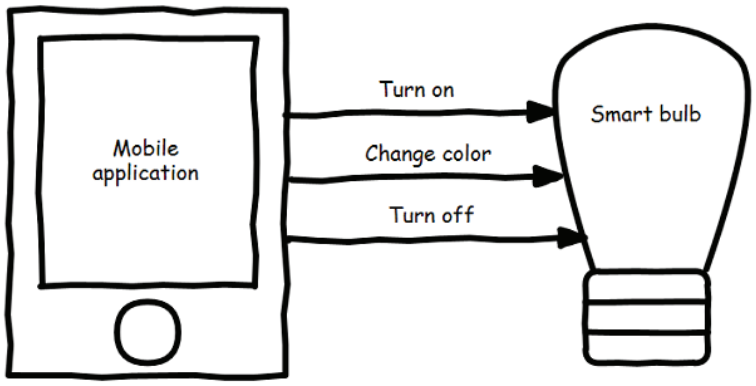


Figure 1.3 Turning on a smart bulb

A smart bulb can be controlled by a mobile application to be turned on and off or to change its color as you wish.

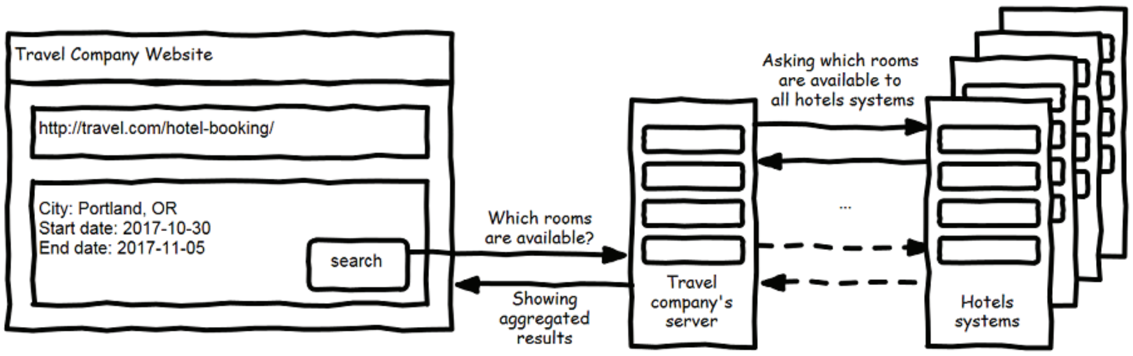

Travel Company Website
http://travel.com/hotel-booking/
City: Portland, OR
Start date: 2017-10-30
End date: 2017-11-05
search
Which rooms are available?
Showing aggregated results
Travel company's server
Asking which rooms are available to all hotels systems
...
Hotels systems

Figure 1.4 Booking an hotel

Travel websites are connected to thousands hotels companies or individual hotels servers in order to show you available rooms and their prices in your next vacations destination.

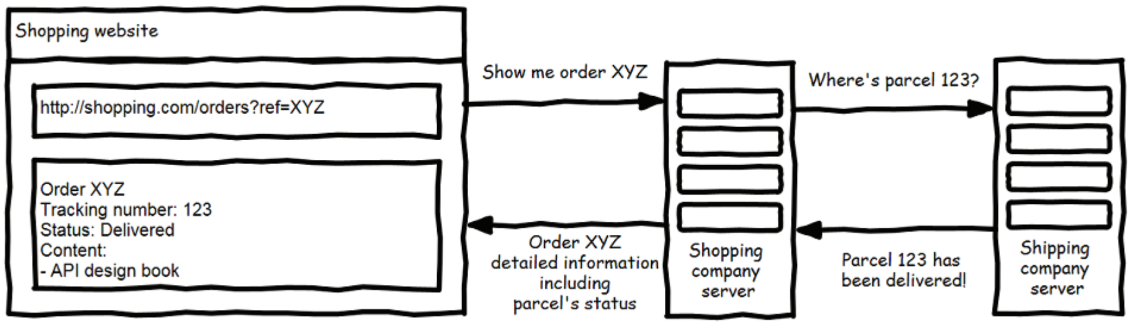


Figure 1.5 Tracking an order on a shopping website

Online shopping websites can provide information about your order delivery once it has been shipped because they are connected to the shipping company's servers.

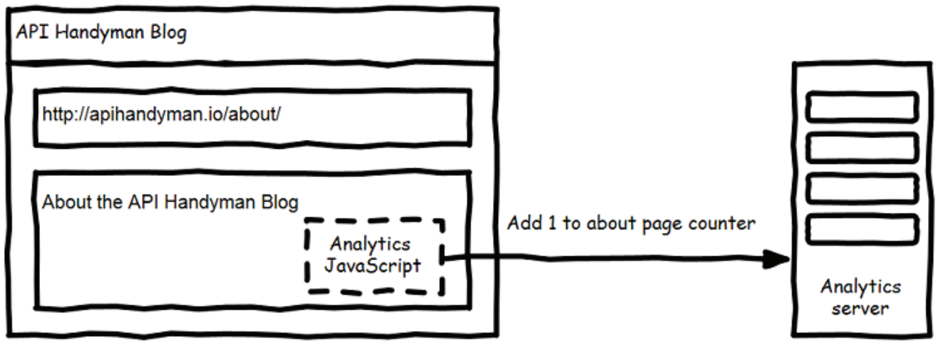

API Handyman Blog
http://apihandyman.io/about/
About the API Handyman Blog
Analytics JavaScript
Add 1 to about page counter
Analytics server

Figure 1.6 Counting how many people see a web page

Blogs and news website are aware of the most read articles because each web page contains a little piece of software that track your activity by sending browsed paged URL to an analytics server.

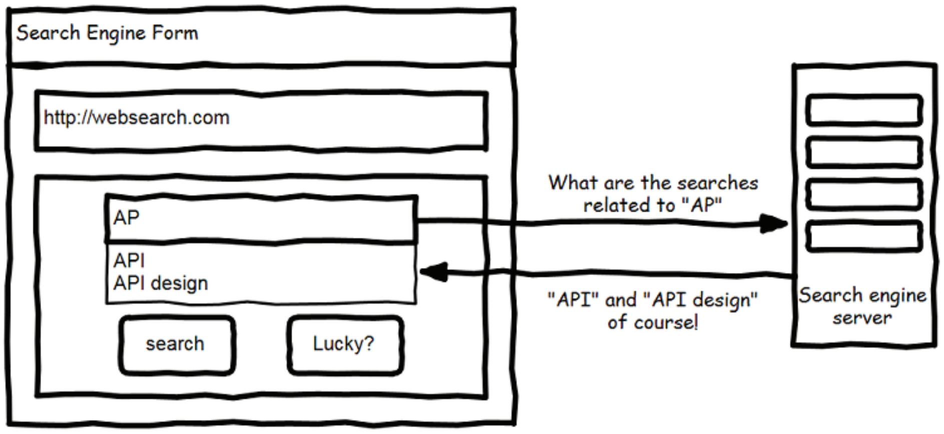


Figure 1.7 A search engine showing hints

When you use a search engine, the search engine web page calls its server to retrieve related searches and provide you some hints as you type your search query.

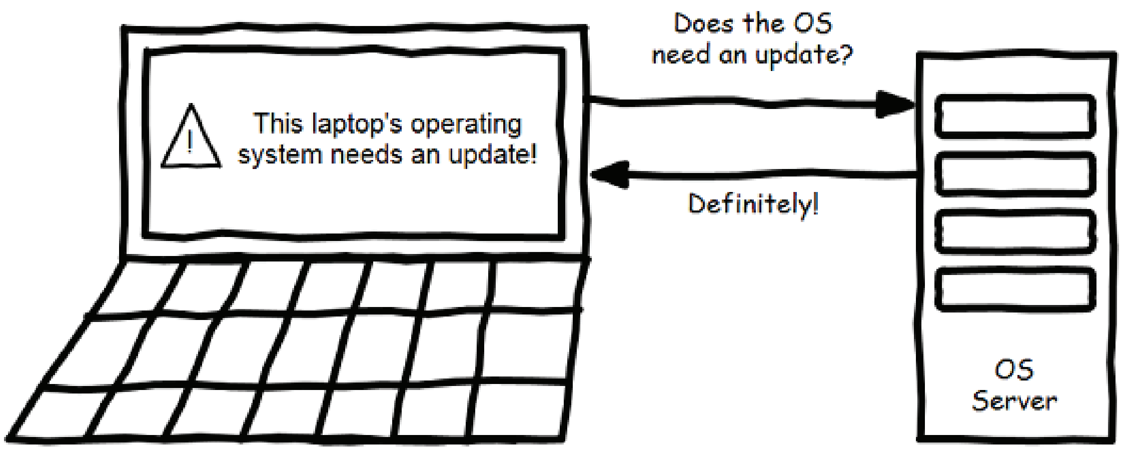

This laptop's operating system needs an update!
Does the OS need an update?
Definitely!
OS Server

Figure 1.8 A laptop checking if its operating system needs an update

Device such as computers or smartphones contain a main software called an operating system or firmware which communicates with the server of the company which created it to know if it needs to update itself.

What do all these use cases have in common? All these use cases relate to software interacting with, talking to or communicating with other software. Weither it's a mobile application interacting with its backend server application, another mobile application talking to internet of thing device's software or server applications communicating with each other.

If we look into a dictionary, the *place at which to independent and often unrelated systems meet and interact or communicate* is called an *interface*. When these *systems* are software, this *interface* is called an *API*. That's why almost everyone is a modern Molière's Mr Jourdain. Almost everyone uses software that interact with other software through their *interfaces*. Almost everyone (indirectly) uses *APIs* without knowledge of it.

NOTE **Keep this in mind!**

An API is what allows a software to interact with another software. We use them indirectly everyday with our connected objects such as smartphone or computers.

But what is an interface for software? How does it look like?

1.1.2 An API is an interface for software

So an API is an interface wich is used by a software to interact with another software. To get a better vision of what this means, we can make an analogy with how we interact with mobile applications on our smartphones. As users of a mobile application we interact with it by using touching the smartphone's screen which displays the mobile application's *user interface* or *UI*.

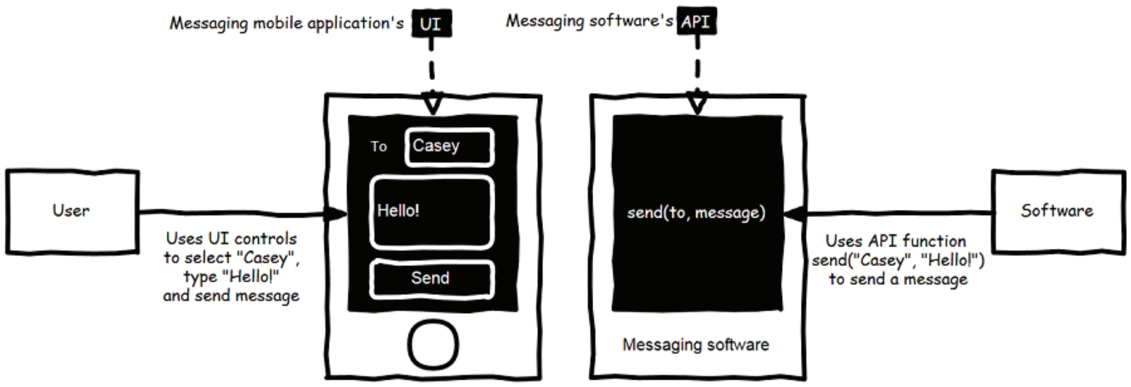

Messaging mobile application's UI
Messaging software's API
To
Casey
Hello!
Send
send(to, message)
Messaging software
User
Uses UI controls to select "Casey", type "Hello!" and send message
Software
Uses API function send("Casey", "Hello!") to send a message

Figure 1.9 Comparing user interface and software interface

A mobile application's user interface may provide elements such as buttons, text fields or labels on the smartphone's screen for its users. These elements allow a user to interact with the application to see or provide information or to trigger actions. A software's interface (API) will provide functions that may need input data or may return output data. These functions allow another software to interact with the software providing the API to retrieve or send information or to trigger actions.

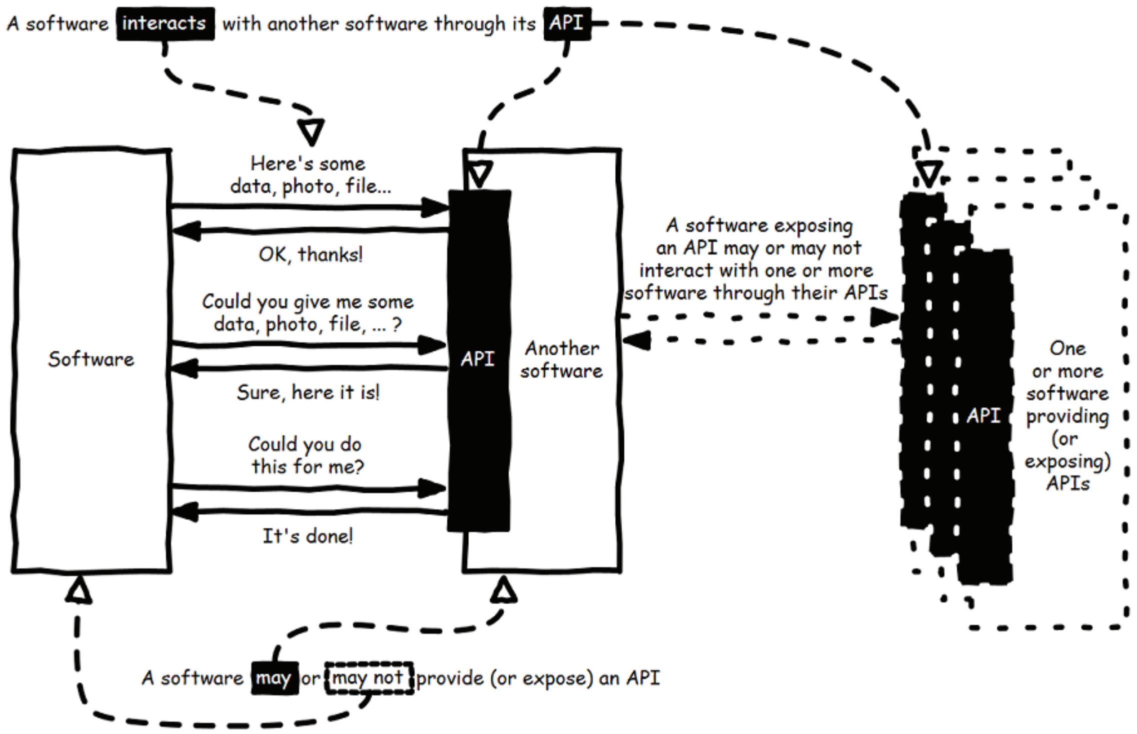

A software interacts with another software through its API
Here's some data, photo, file...
OK, thanks!
Could you give me some data, photo, file, ... ?
Sure, here it is!
Could you do this for me?
It's done!
Software
API
Another software
A software exposing an API may or may not interact with one or more software through their APIs
API
One or more software providing (or exposing) APIs
A software may or may not provide (or expose) an API

Figure 1.10 Software interactions are done through APIs

Just like the UI which is a part of a mobile application, the API is a part of a software. It is the part a software may expose or provide to other software to give a way to interact with itself. By using another software's API, a software may retrieve data or information (to get *a room's price in a hotel* or *a parcel's status*), send data (to share a *photo*) or trigger actions (to *turn a smart bulb on*). Note that a software exposing an API may even rely on other software to do its job.

But if a user interacts with a mobile application running on its smartphone which is at hand, a software may interact with another software even if they are not on the same computer and separated by thousands of miles.

1.1.3 An API is a web interface for software

APIs are not new, they have been there for a very long time and have taken different forms. An API can be used by a software to interact locally with computer hardware, operating system or library.

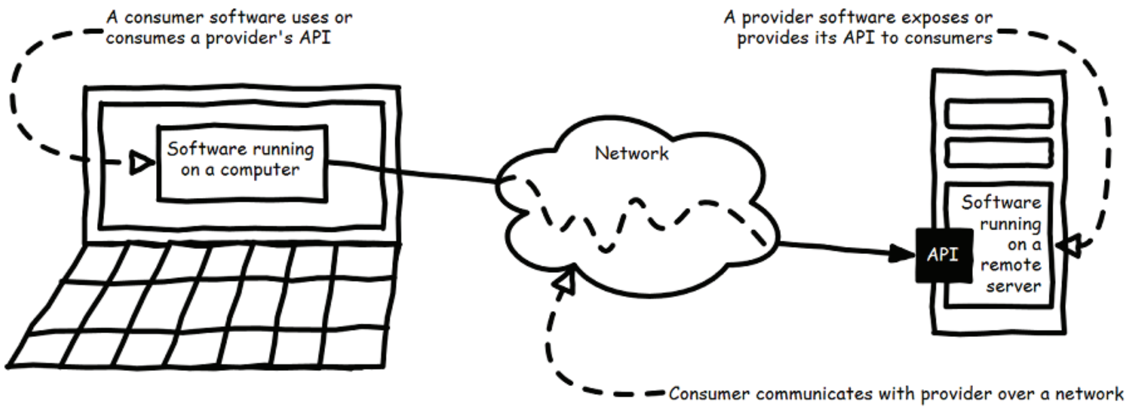

A consumer software uses or consumes a provider's API
A provider software exposes or provides its API to consumers
Software running on a computer
Network
API
Software running on a remote server
Consumer communicates with provider over a network

Figure 1.11 Remote interaction between consumer and provider

But it can also be used to interact with a remote system over a network. In that case, the software using or consuming the API is called a *consumer*. The software exposing or providing the API is called a *provider*.

Such remote API is usually what people are talking about when they use the acronym API nowadays. This is what this book is about, designing APIs that can be used by software over a network. So how does such an API works? Let's analyze how a bank's customer can check his bank account's balance to explain it.

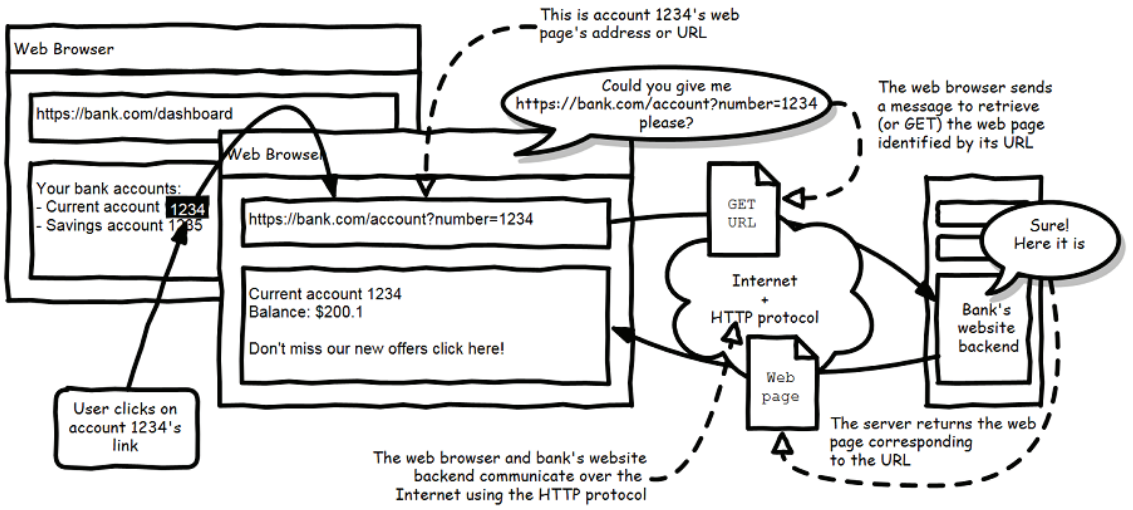

This is account 1234's web page's address or URL
Web Browser
https://bank.com/dashboard
Your bank accounts:
- Current account 1234
- Savings account 1235
Could you give me https://bank.com/account?number=1234 please?
The web browser sends a message to retrieve (or GET) the web page identified by its URL
Web Browser
https://bank.com/account?number=1234
Current account 1234
Balance: $200.1
Don't miss our new offers click here!
GET URL
Internet + HTTP protocol
Sure! Here it is
Bank's website backend
Web page
User clicks on account 1234's link
The web browser and bank's website backend communicate over the Internet using the HTTP protocol
The server returns the web page corresponding to the URL

Figure 1.12 Using a bank's website

When a bank's customer want to know how much money is left on his or her bank account, he or she can use a desktop computer to browse the bank's web site. To see 1234 account's information, the customer clicks on its link, the browser's address bar is updated with the link's URL and after a few hundreds of milliseconds the browser shows the corresponding web page. To show this page, the browser had communicated over the Internet with the bank's website's backend (or bank's web server). To understand each other these two software use a communication protocol called the HTTP protocol. All we need to know for now about this protocol is that the browser sends a message to request the web page corresponding to the account 1234's URL and the server returns the request web page. All websites around the world wide web operates the same way.

But nowadays, a website is not the only way to get such information. A bank may also provide a mobile application to its customers. And as we have seen earlier, this mobile application communicates with its backend using an API. So what's happening when a customer uses the bank's mobile application instead of the web site? It's happening the same thing! The mobile application communicate with its backend application through its API over the Internet using the HTTP protocol.

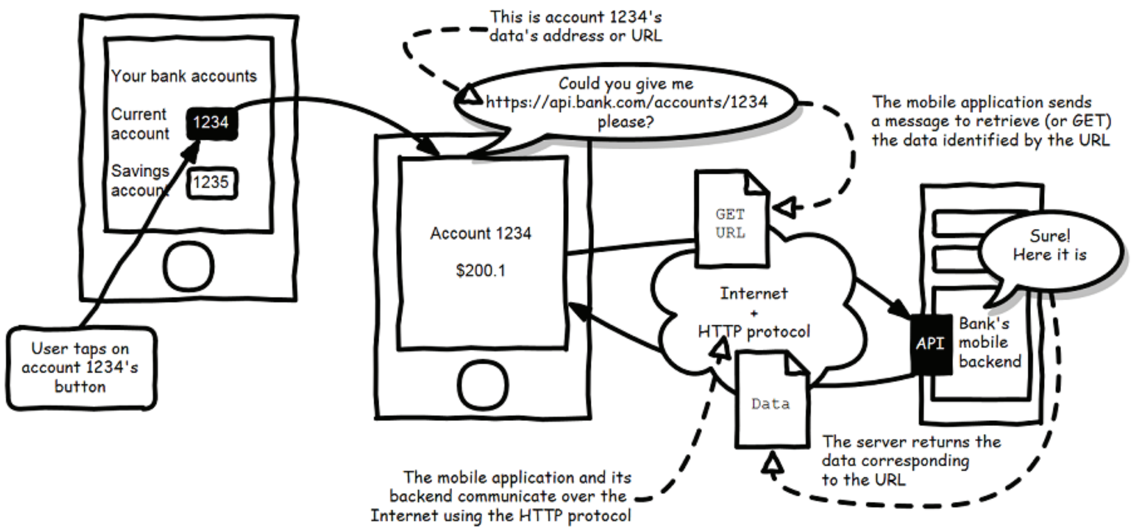

This is account 1234's data's address or URL
Your bank accounts
Current account
1234
Savings account
1235
Could you give me https://api.bank.com/accounts/1234 please?
The mobile application sends a message to retrieve (or GET) the data identified by the URL
Account 1234
$200.1
GET URL
Internet + HTTP protocol
Sure! Here it is
API
Bank's mobile backend
Data
User taps on account 1234's button
The server returns the data corresponding to the URL
The mobile application and its backend communicate over the Internet using the HTTP protocol

Figure 1.13 Using a bank's mobile application

The account 1234 data is identified by a URL just like the web page. When the user taps on the account 1234, the mobile application sends a requests asking the API server to provide the data corresponding to this account's URL. As a response to this request, the API server returns the data corresponding to this URL just like the web server has returned the web page. Then, the mobile application can show the account's information on the screen.

Most APIs work that way, using the same protocol as the one used around the world wide web by all web sites. This is why these APIs are called *Web APIs* (and before APIs were cool, this is why they were called *Web Services*).

NOTE **Keep this in mind!**

In this book, an API is a web API. It allows a software to interact remotely over a network with another software using the same protocol as web sites, the HTTP protocol.

So an API, or at least the type of API we are studying in this book, is a web interface exposed by a software that can be used remotely by other software. This is not really what you would have have probably imagine by trying to figure what an *Application Programming Interface* is. But what if I tell you that this definition alone, even if correct, does not describe what an API really is?

1.2 What really is an API

If we reduce an API to a web software interface, a technical interface allowing two software to communicate we are definitely missing a very important part of what an API really is. And therefore we will certainly will not be succeed in the design of APIs. Who really uses such interfaces? For what purposes? These are the questions that will reveal the true nature of APIs.

1.2.1 An API is also an interface for developers

An API is an application *programming* interface, it means that it is an interface that can be *programmatically* used. Earlier we have compared UI and API, how a user interact with an mobile application and how a software interact with another software. But if a user has his own will, a software do not decide by himself to use another software's API.

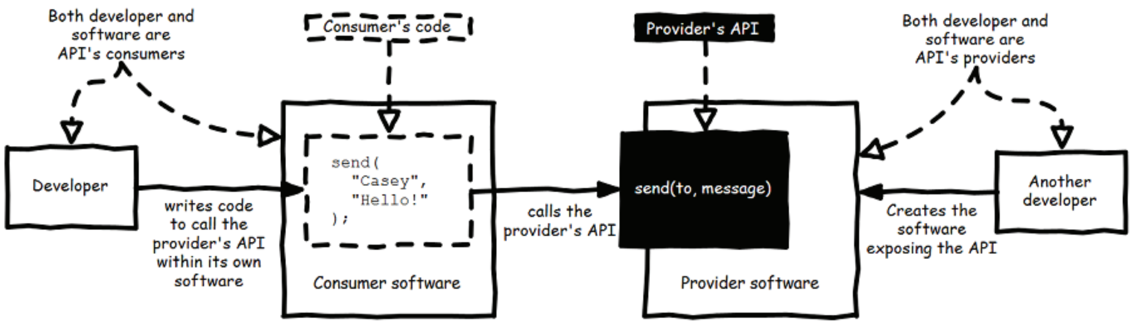

Both developer and software are API's consumers
Consumer's code
Provider's API
Both developer and software are API's providers
Developer
writes code to call the provider's API within its own software
send("Casey", "Hello!");
Consumer software
calls the provider's API
send(to, message)
Provider software
Creates the software exposing the API
Another developer

Figure 1.14 Developers write code to use APIs

If a software uses an application programming interface, it's because it has been programmed to do so. A developer, a human being, has created this software and wrote the code that calls or *consumes* the API. Both developer and its software consuming an API are called *consumers*. You will discover later in this book that contenting these two users is critical when designing an API.

On the opposite side, both the software exposing or *providing* the API and the developer which created it are called *providers*. We'll see later in this book how this developer can be concerned by the design of the API.

NOTE **Keep this in mind!**

An API is an interface exposed by a provider which is both the software exposing the API and the developer (or organization) creating this software. The API's users are called consumers. They are both the software using or consuming the API and the developer (or organization) creating this software.

But why would a developer uses another software into his or her own one?

1.2.2 An API is a screen over complexity

The most important thing an API do is hiding complexity. What does that mean? Let me use a real life analogy to explain that: let's go to a restaurant. What about a french one?

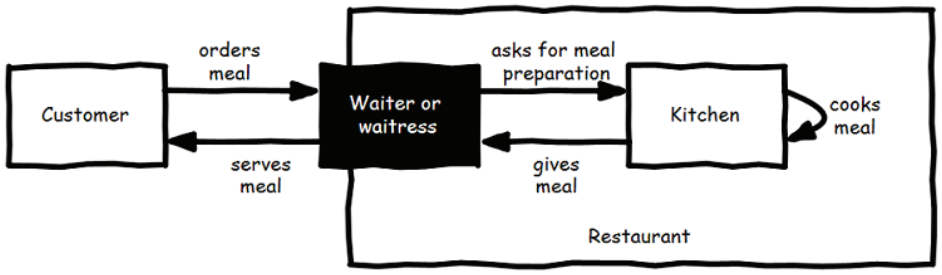


Figure 1.15 Let's go to a restaurant

When you go to a restaurant, you become a *customer*. As a restaurant's customer, you read a menu to know what kind of food you can order. Of course, if you are a regular customer you do not need to read the menu. You should try the Bordeaux style lamprey, it is a famous french fish recipe from the Gascony region. To order the meal you have chosen, you will talk to a usually very kind and friendly person called a *waiter* (or a *waitress*). A moment later, this waiter will come back to give you the meal you have ordered, the Bordeaux style lamprey, which has been prepared in the *kitchen*. While you eat this delicious meal, may I ask you two questions?

First one, do you know how to cook Bordeaux style lamprey?

Probably not, and that can be a 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 is complex and needs 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 question, do you know what happened between the moment the waiter took your order and bring it back to you?

You can guess that the waiter has been to the kitchen to give your order to a cook who works alone. Then, this cook has cooked your meal and then notify the waiter by using a small bell and yelling "order for table number 2 is ready". But it could be slightly different, the waiter may use a smartphone to take your order which is instantly shown on a touch screen in the kitchen. In the kitchen there's not a lonely cook, but a whole kitchen brigade. Once the kitchen brigade has prepared your meal, one of them set your order as "done" on the kitchen's touch screen and the waiter is notified on his smartphone. Whatever the number of cooks, you are also unaware of the recipe and ingredients used to cook your meal. Regardless of the scenario, the meal you have 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 waitress (or waiter) and you don't need to know what happens in the kitchen. What does all this have to do with APIs? Everything.

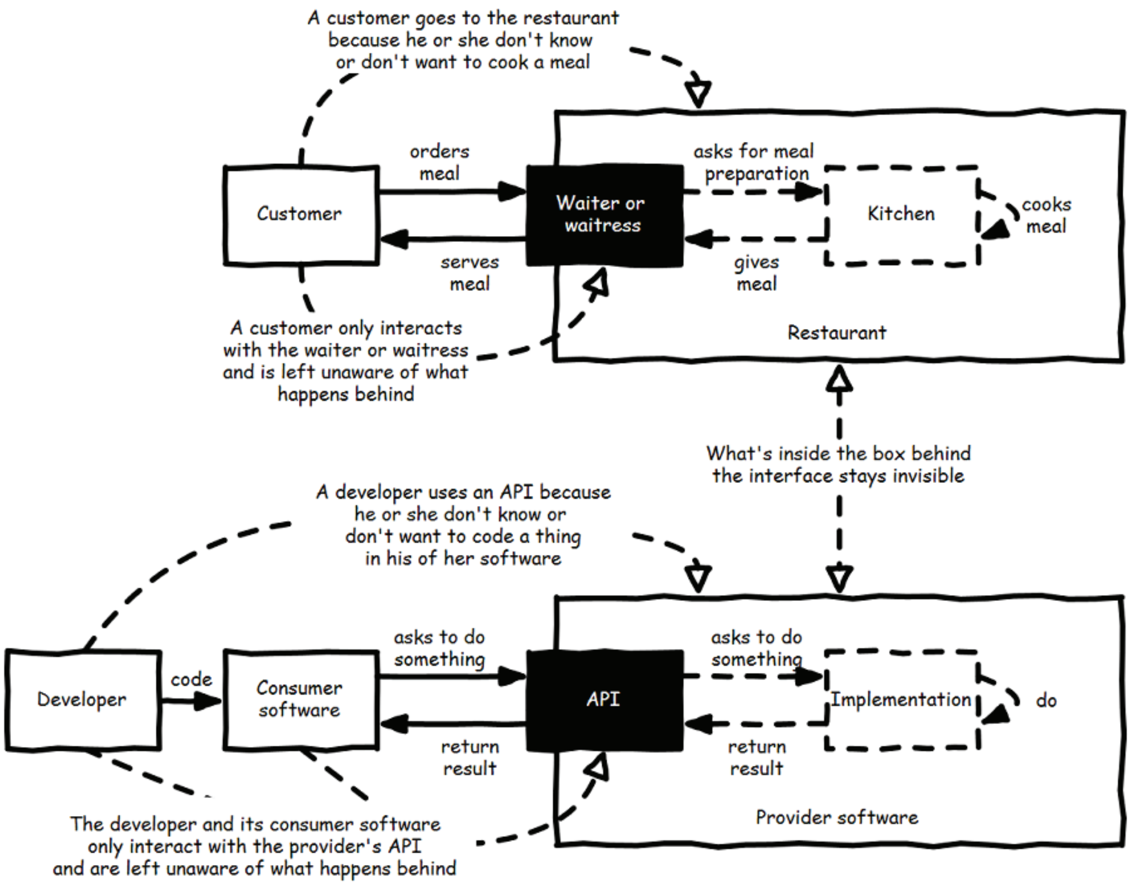

A customer goes to the restaurant because he or she don't know or don't want to cook a meal
Customer
orders meal
Waiter or waitress
serves meal
asks for meal preparation
Kitchen
cooks meal
gives meal
Restaurant
A customer only interacts with the waiter or waitress and is left unaware of what happens behind
What's inside the box behind the interface stays invisible
A developer uses an API because he or she don't know or don't want to code a thing in his of her software
Developer
code
Consumer software
asks to do something
API
return result
asks to do something
Implementation
do
return result
Provider software
The developer and its consumer software only interact with the provider's API and are left unaware of what happens behind

Figure 1.16 Let's go to a restaurant

When a developer creates a *consumer software* (the *customer*) that use a *provider software* (the *restaurant*) through its API (the *waiter* or *waitress*) to *do something* (like *orders meal*), the developer and its consumer software are only aware of the API and don't need to know how to *do something* nor how the provider software will actually do it.

NOTE **Keep this in mind!**

An API hides complexity because it helps you and your software do things you don't know how to do or don't want to do by delegating this thing to another software. And also because it leaves you totally unaware of how the other software will handle the tasks you have delegated to it.

APIs let you create a software than can do almost anything by just delegating a task you cannot do or do not want to do to another software without bothering how it will be done. That’s a first reason why a developer would use an API, to simplify its work. Having this capability to delegate actions to another software offers another possibility to developers that is absolutely awesome.

1.2.3 An API is a LEGO® brick connector

Have you ever played with LEGO® bricks? This toy is an awesome invention. All these simple bricks that you can connect easily to each other to create new things. Aristotle was probably playing with some when he said that *the whole is bigger than the sum of its parts*.

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 to start a new thing from scratch. I could also transform it by replacing some parts. I could even assemble existing structures easily to create a massive spaceship for example. At each birthday I was hoping to have a new set to add new bricks to my collection and have new possibilities. This toy is really wonderful to unleash creativity. You don’t care about how you connect the blocks physically as they are so well designed that you can connect and disconnect them easily. If new types of LEGO® pieces are created they share some common features with others, so its really easy to understand how to use them. When you play with such well designed toy, you only think about what awesome things you can do with them and do not struggle to understand how to use them.

TIP **Pro tip!**

Components that share common design features are easy to understand and use once you have worked with a first one.

Components that you can connect together easily to create something new that is better than the sum of its part. This is definitely what you can do with software thanks to APIs.

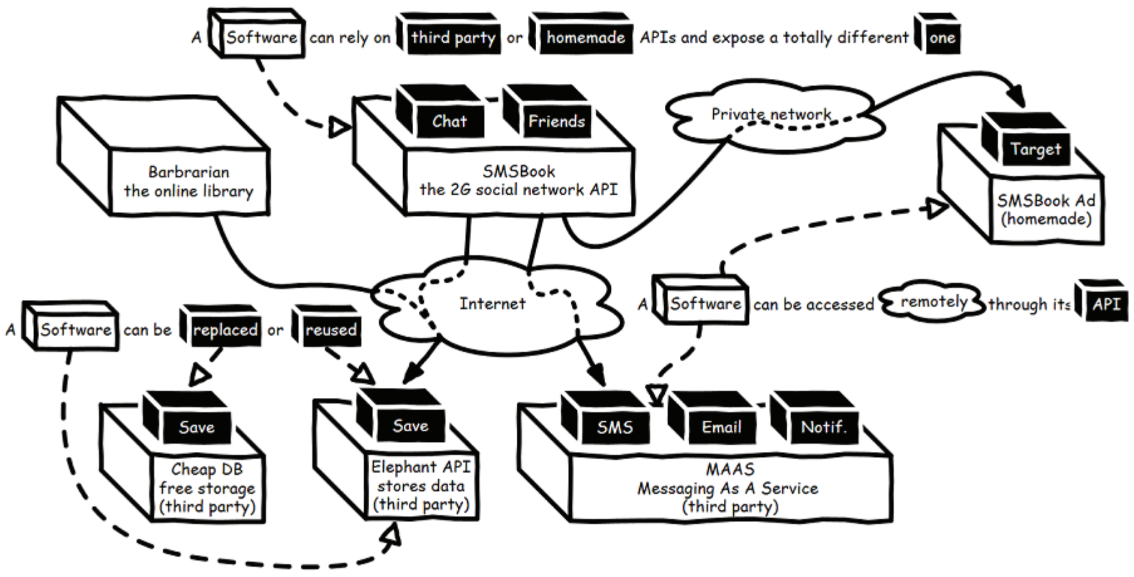

A Software can rely on third party or homemade APIs and expose a totally different one
Chat
Friends
SMSBook the 2G social network API
Barbrarian the online library
Private network
Target
SMSBook Ad (homemade)
Internet
A Software can be accessed remotely through its API
A Software can be replaced or reused
Save
Cheap DB free storage (third party)
Save
Elephant API stores data (third party)
SMS
Email
Notif.
MAAS Messaging As A Service (third party)

Figure 1.17 Software LEGO® bricks

You can create your own software bricks exposing APIs (*SMSBook Ad*), pick some other software bricks also exposing APIs (*Elephant API* and *MAAS*) and assemble them to create a new software… that may expose another API (*SMSBook*). The possibilities are endless, the only limit is your imagination.

After a while, you may want to replace one of those software bricks (*Elephant API*) by another one more efficient or less expensive (*Cheap DB*). A single API (*Elephant API*) may also be used by multiple consumers (*SMSBook* and *Barbrarian*)

NOTE **Keep this in mind!**

Existing APIs can be used to build new software easily and quickly. They can also help to decompose huge software into smaller and more specialized module communicating with each other that can be easily reused or replaced.

APIs allow to create modular software that can be assembled easily in many different ways. But unlike LEGO® bricks which are tighly assembled, software bricks exposing APIs are loosely coupled. As APIs are remote interfaces, all software which are part of a system may run on different computers connected by some networks weither it is a private one or the Internet. It allows to rely on third party systems which are completely externalized without having to handle them inside your infrastructure. Having all these softwares running on different computers contribute to build more efficient systems, because each software hardware may be tuned according to its own specific needs.

NOTE **Keep this in mind!**

APIs contribute to build efficient and flexible systems based on loosely coupled software bricks running on different computers.

This is awesome, APIs turn software into software remote LEGO® bricks that help you create anything easily and efficiently. We have cover many concepts, let's try to sum up what we have seen so far to grasp what really is an API.

1.2.4 API definitions

So yes, an API is an Application Programming Interface. But these three words do not show the true nature of what an API is. An API is:

- an interface a software may expose so that other software may interact with it
- an interface that can be used remotely over the web
- an interface that turns software into LEGO® brick
- an interface which hides what really happens behind it
- an interface that lets you do things you don't want to or don't know how to do
- an interface for developers which create software that use it

Once aware of that, you may feel that the most important letter in API is I. It is definitely true, the most important word in Application Programming Interface is the last one. An API is an interface and its design is a critical issue.

1.3 Why the design of APIs is so important

APIs seems to allow to do awesome things but why would their design be a critical issue? Well, there are two main reasons why API design is really, absolutely, definitely important. The first one is a simple matter of volume. As time pass, more and more APIs are created. And when I say more and more I'm talking about millions of them. So the world needs many, many, many API designers. The second one is a matter of survival. Yes, survival. Good API design is what separates success from failure. A poor designed API, whatever its purpose, may be cursed by all its consumers before slowly sliding from indifference to oblivion. It can even be lethal for its provider.

1.3.1 More and more public and private APIs

Yes, there are millions of APIs to design out there. They can be split into two categories: *public* and *private*.

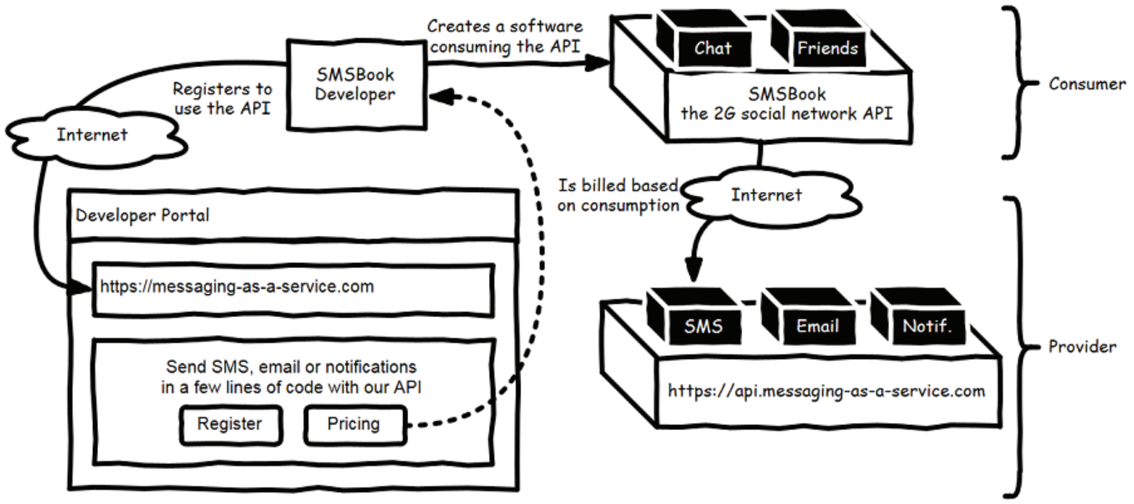

Creates a software consuming the API
Chat
Friends
SMSBook the 2G social network API
Consumer
Registers to use the API
SMSBook Developer
Internet
Is billed based on consumption
Internet
Developer Portal
https://messaging-as-a-service.com
Send SMS, email or notifications in a few lines of code with our API
Register
Pricing
SMS
Email
Notif.
https://api.messaging-as-a-service.com
Provider

Figure 1.18 A public API

A public API is created by a company or an organization to be used by anyone. This kind of API comes with a website called a *developer portal* where any willing user can register to use it. This portal proposes all what developers need: documentation, code samples, SDKs, sandbox environment to test their development, support… Some public APIs are free to use for anyone (like government agencies APIs for example), they are also called open APIs. Some others are sold *as a service* or *as a product* (like an API allowing you to send SMS for example), you have to pay to use them. This kind of API can be billed based on the number of API calls made or on the exchanged data volume. Some public APIs are called *partner APIs* because their providers only allow selected users to use them. Many companies build entirely their business on public APIs, many companies are extending their business with public APIs.

NOTE **Keep this in mind!**

A public API is exposed on the Internet and can be used by anybody. A partner API is like a public API but limited to selected users.

But these public APIs are only the tip of the iceberg, the visible part of the API world. If public APIs are count with tens of thousand, there are millions of private ones.

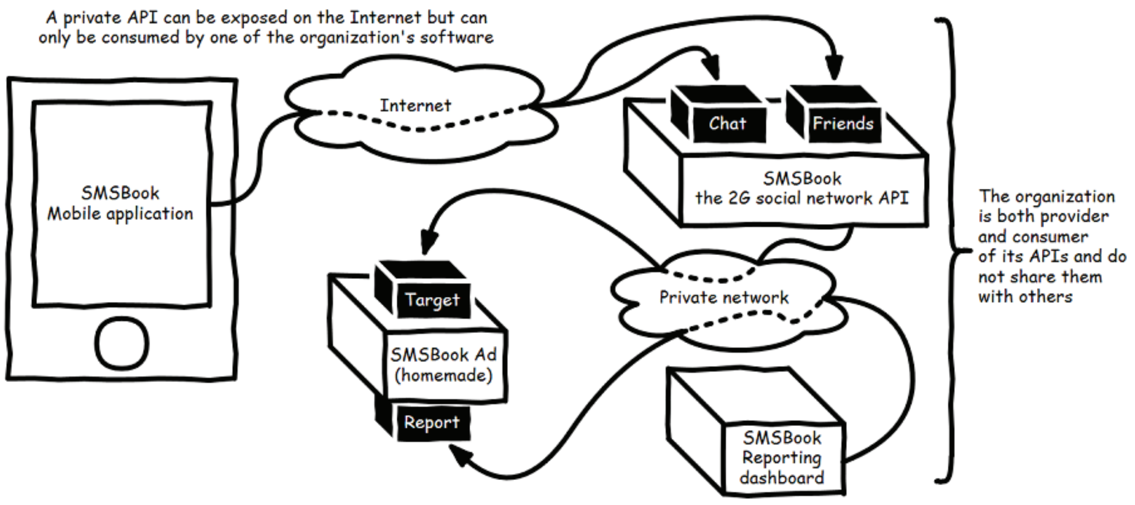

A private API can be exposed on the Internet but can only be consumed by one of the organization's software
Internet
SMSBook Mobile application
Chat
Friends
SMSBook the 2G social network API
Target
SMSBook Ad (homemade)
Report
Private network
SMSBook Reporting dashboard
The organization is both provider and consumer of its APIs and do not share them with others

Figure 1.19 Private APIs

A private or internal API is an API which will be only used by the company or organization which has created it. A mobile application or website backend are the most obvious examples of private API. Depending on the organization's size but also its maturity regarding APIs, these private APIs may come with developer portal, documentation and sandbox like public ones. Unfortunately, providing such ecosystem for private APIs is often neglected, which is a terrible mistake as it dramatically harden API discovery and reusability within an organization.

NOTE **Keep this in mind!**

A private API is only used by the organization which has created it. It can be exposed on the Internet or a private network.

Imagine how many mobile applications and websites exists, millions of them, even if they were all API based that would only be the tip of the private API iceberg. Almost all modern information systems are build upon various software communicating with each other using APIs (or Web Services).

NOTE **Keep this in mind!**

Around 2002, Jeff Bezos, the Amazon CEO issued a mandate that would be the corner stone of this company's transformation from a book seller to a cloud computing leader. This mandate was enforcing the use of APIs (which were call web services at that time) between teams in the company:

- All teams will henceforth expose their data and functionality through service interfaces (*APIs*)
- Teams must communicate with each other through these interfaces.
- There will be no other form of interprocess communication allowed: no direct linking, no direct reads of another team's data store, no shared-memory model, no back-doors whatsoever. The only communication allowed is via service interface (*API*) calls over the network.
- It doesn't matter what technology they use.
- All service interfaces (*APIs*), without exception, must be designed from the ground up to be externalizable. That is to say, the team must plan and design to be able to expose the interface (*API*) to developers in the outside world. No exceptions.

The legend holds that this mandate finished with:

> Anyone who doesn't do this will be fired.
> Thank you; have a nice day!
>
> -- Jeff Bezos Amazon CEO

The rise of microservice architecture drastically increases the number of private APIs. In this kind or architecture, huge monolithic catch-all softwares which are hard to evolve and scale are replaced by smaller and scalable pieces of software focusing on narrow business capabilities: microservices. As all these microservices interact with each other using APIs, the more microservices are created, the more APIs are created.

So, as days and even hours pass, more and more public and even more private APIs are created. APIs have become a key element in the software industry. That explains why

API is not a niche but it does not explain why all these millions of APIs, wether private or public, must be properly designed.

1.3.2 Poor API design has terrible consequences

Why should we care about the design of an API? It's quite simple. What do you do when you use an everyday thing, wheither it's a microwave oven, a web site, a mobile application, a TV remote or an ATM? You use its interface. 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 experience. And this is where design matters. An API is an *interface* which is supposed to be used by developers within their software. Design matters whatever the type of interface and APIs are no exception.

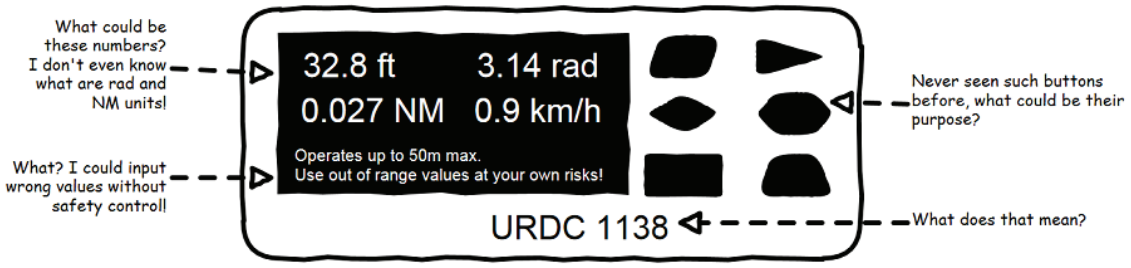
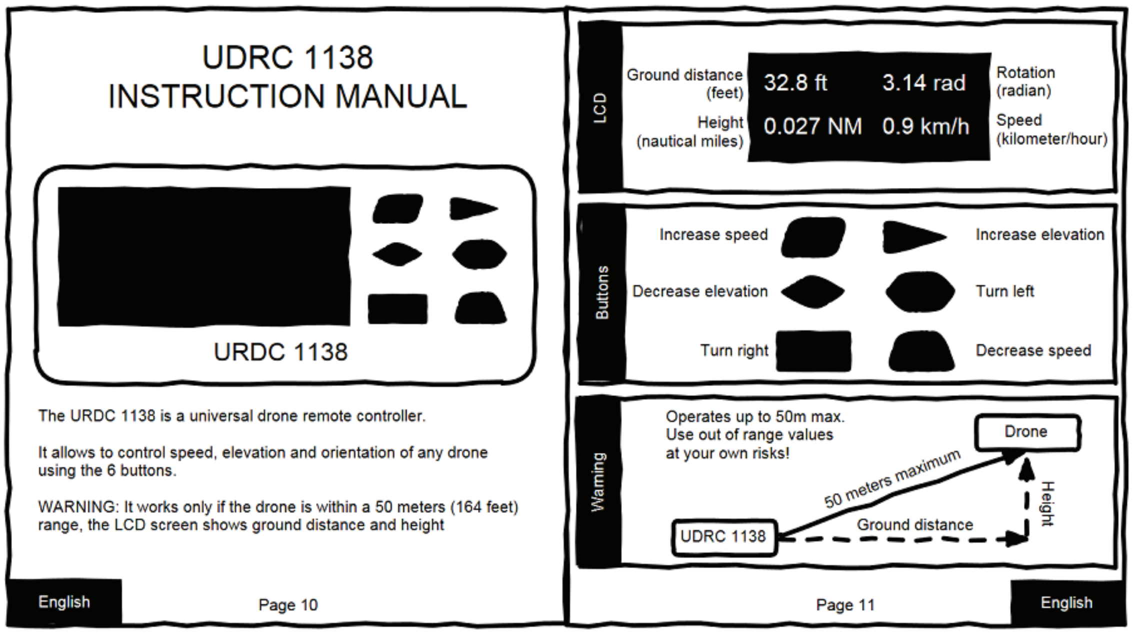

What could be these numbers? I don't even know what are rad and NM units!
What? I could input wrong values without safety control!
32.8 ft
3.14 rad
0.027 NM
0.9 km/h
Operates up to 50m max.
Use out of range values at your own risks!
URDC 1138
Never seen such buttons before, what could be their purpose?
What does that mean?

Figure 1.20 A cryptic device's interface

What could be this *UDRC 1138* device? What could be its purpose? It is not easy to guess by looking at its name, maybe its interface can help us. There are six unlabeled buttons with unfamiliar forms that do not give any hint about their purposes. The LCD display screen shows four, also unlabeled values. I'm not familiar with all these units. And there is even a warning message telling us that we can input wrong values without safety control. This interface is hard to decipher and is not very engaging. Let's take a look at the documentation to see what this device could be.



The URDC 1138 is a universal drone remote controller.

It allows to control speed, elevation and orientation of any drone using the 6 buttons.

WARNING: It works only if the drone is within a 50 meters (164 feet) range, the LCD screen shows ground distance and height

Operates up to 50m max.
Use out of range values
at your own risks!

Figure 1.21 A cryptic device's documentation

The UDRC 1138 is a universal drone remote controller. Well, it sounds interesting.

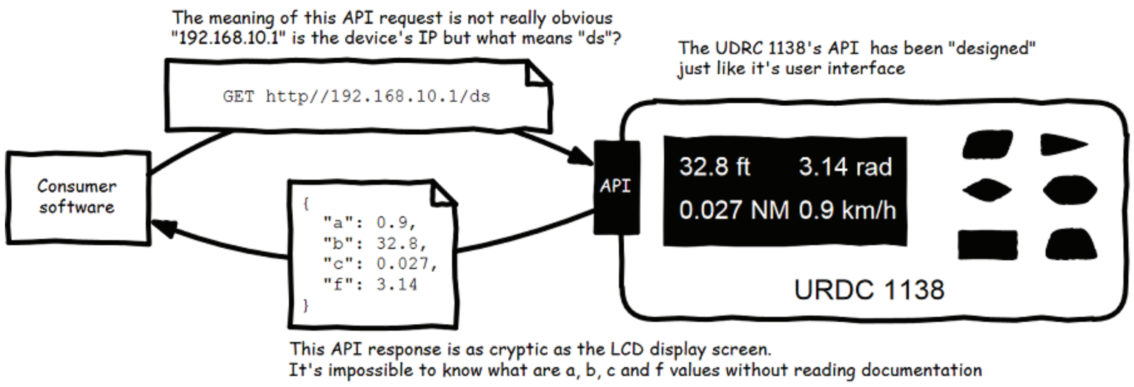
It allows to control speed, elevation and orientation of any drone using the 6 buttons. This is insane how a user is supposed to use that easily? Buttons shapes do not mean anything and they are not even ordered correctly!

It works only if the drone is within a 50 meters (164 feet) range, the LCD screen shows ground distance and height. Wait, what? The user has to calculate the distance between the remote controller and the drone based on the information provided on the LCD display screen?! This is a total nonsense. I don't want to use the Pythagorean theorem, the device should do that for me.

The *LCD* description puzzle me too. I'm not familiar with aeronautic units of measurement but I sense there's something wrong with the chosen ones. They seem inconsistent. Mixing feet and meters? And in all movies I have seen about airplanes they use feet to measure height and nautical miles to measure distance not the contrary. This is definitely not engaging. Would you buy or even try to use such a device? Probably not.

Hopefully, such an overly poor designed device's interface couldn't exists. Designers couldn't create such thing… Well, maybe, but and even if it was possible, quality department would never let it go into production… Let's face the truth, poor designed devices, everyday things with poor designed interfaces go into production. Think about how many times you have been puzzled or you have grumble when using a device, a web site or an application because its design was flawed? How many times did you choose not to buy or not to use something because of its design flaws? How many times did you restrain the way you use something that you must use because its interface was incomprehensible? A poor designed product may be misused, underused or not used at all. It can reveal to be dangerous for its user and even the organization which has created it. And on top of that, once such physical device goes into production, it's too late to fix it.

Such terrible design flaws are not reserved to everyday things interface.


The meaning of this API request is not really obvious
"192.168.10.1" is the device's IP but what means "ds"?
GET http//192.168.10.1/ds
The UDRC 1138's API has been "designed" just like it's user interface
Consumer software
API
32.8 ft 3.14 rad
0.027 NM 0.9 km/h
URDC 1138
This API response is as cryptic as the LCD display screen.
It's impossible to know what are a, b, c and f values without reading documentation

```
{
  "a": 0.9,
  "b": 32.8,
  "c": 0.027,
  "f": 3.14
}
```

Figure 1.22 Getting UDRC 1138 status via its terrible API

Unfortunately, APIs can also suffer from this flawed design disease. Poorly designed APIs can look exactly like this URDC 1138's interface. A poorly designed API can be a real pain to understand and use, it can have terrible consequences.

TIP **Pro tip!**

API design has more to do with user interface design than programming.

How do people choose a public API, an API as a product? They go to the developer portal to read the documentation and analyze the API to understand what it allows to do and how to use it to evaluate if it fits their needs *simply*. Even the best documentation will not be able to hide API design flaws which will make the API hard or even dangerous to use. Therefore, these potential users, these potential customers will not choose this API. No customers, no revenue. It may lead to a company's bankruptcy.

What if some people decide to use this flawed designed API or have no other choice and must use it? What will happen? It will increase the time, effort and money needed to build a software using this API. The API may be misused or underused. The API's users may need extensive support from the API provider, therefore raising costs. If the API is a public one, its users may complain publicly or simply stop to use it. Resulting in less customers and less revenue for the API provider. If it's a private one, loosing money on these projects may hinder the company strategy. After too many failures, it may even result in a "APIs are not for us" attitude which could doomed the company in the long run.

Flawed API design may lead to the creation of of security vulnerability in the API. Exposing sensitive data inadvertently, neglecting access rights and group privileges, trusting too much consumers, … What if people decide to exploit such a vulnerability? This is no science fiction, it has really happened.

Hopefully in the API world, it is possible to fix poor API design after the API goes into production. But it comes at a cost. It will take time and money for the provider to fix this mess and may also seriously bother API's consumers.

NOTE **Keep this in mind!**

A poor designed API can be underused, misused and even not used. It can be annoying and even dangerous for both consumer and provider.

This is terrible! Poor designed APIs look definitely terrible and there are potentially millions of them. What can be done to avoid such a doomed fate? It's dead simple: learn how to design APIs properly.

1.4 What really is learning API design

OK, APIs are programming web interfaces for software and developers. There are millions on them. They must be designed like everydays objects interfaces. Flawed design can be a disaster. Woah, that's scary. Designing APIs must be awfully complicated, no?

No. It's not.

Designing APIs is not complicated once you know *how to really* design APIs. The global challenge of API design is being aware of all of its aspects. And the first step of this challenge is to understand what really is *learning* API design.

1.4.1 Learning API design solfeggio

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 doing APIs to enable software communication over a network. You may have heard about RPC, SOAP, REST, gRPC or GraphQL. These are different ways of doing APIs, some are architectural styles, others are protocols or query languages, but to make it more simple let's call them *API styles*. Although these styles may rather be different, they all require the same basic knowledge to design APIs using them. Some people may reduce API design to designing a technical software interface using a specific API style without understanding the real purpose and needs of such a design activity. It's just like playing an instrument without knowing solfeggio.

I am a lazy guitar player, I have some basic solfeggio knowledge but almost didn't use it because I learn songs using their tablatures (or tabs) instead of sheet music.

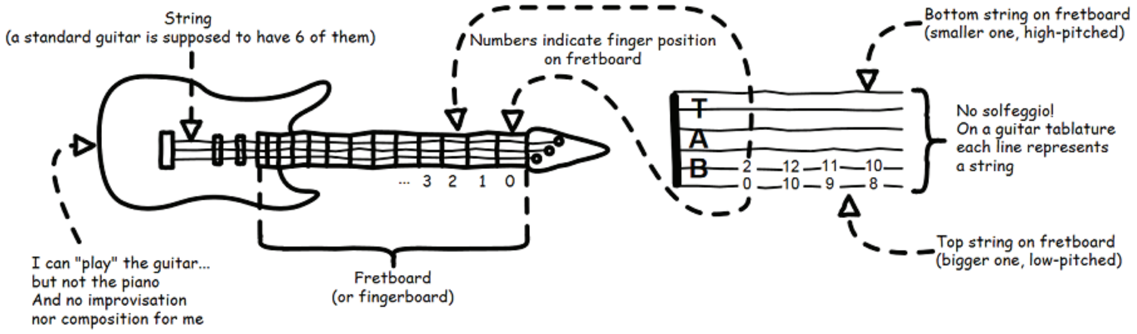

String
(a standard guitar is supposed to have 6 of them)
Numbers indicate finger position on fretboard
Bottom string on fretboard
(smaller one, high-pitched)
No solfeggio!
On a guitar tablature each line represents a string
... 3 2 1 0
T
A
B
2 — 12 — 11 — 10 —
0 — 10 — 9 — 8 —
Top string on fretboard
(bigger one, low-pitched)
I can "play" the guitar...
but not the piano
And no improvisation
nor composition for me
Fretboard
(or fingerboard)

Figure 1.23 Playing the guitar without solgeggio knowledge

On a tablature, there is no clef, nor notes or chords. There 6 lines, each one representing a guitar's string. Numbers on these lines indicate where to put your fingers on the guitar's fretboard. I mostly didn't know what chords and note I am playing but I can play almost any song … if I have its tablature. But I can't do anything else like composition or improvisation. And I also can't play another instrument like the piano for example. A friend of mine is a music teacher, as long as he understands how to do notes and chords, he can play any musical instrument to play any song, improvise and even compose music. He can do that because he masters music basic knowledge: solfeggio. He knows notes, chords, harmony, scales and how to combine them to create music that will make sense and not make you ears bleed with any musical instrument.

Jumping headfirst into programming interface design using a specific API style. Reducing API design to the creation of some "functions" that can be called remotely without any other consideration. All this is like playing the guitar using tablatures. You may possibly be able to achieve the design of some decent APIs in the long run… after a few failures. A few failures that could be easily avoided if you master API design solfeggio.

This book is not intended to be a "how to design *whatever style* API" book, it is intended to teach the API design solfeggio which can be used to design APIs in any existing or yet to come API style. The purpose of this book is to provide the reader the API thinking mindset to be ready to face API design challenges effortlessly like a musician improvising, thinking more about the melody (the API to design) and less about the instrument used to create it (the API style).

NOTE **Keep this in mind!**

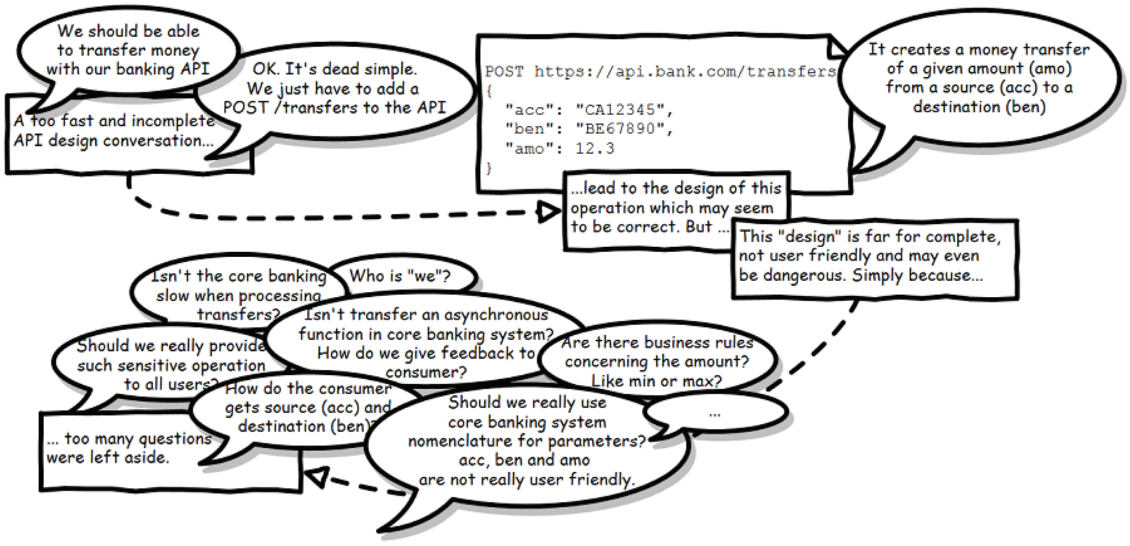
There are different ways of doing APIs but they all require the same basic knowledge.

But, learning solfeggio alone can be incredibly boring and even counter productive without practicing. We need an instrument! We will use the REST API style which will be explained in depth throughout in this book while learning API design.

So there is some basic knowledge to know to master API design. But what are the equivalent of solfeggio, notes, clef, scales and chords in the API design world? What do we need to know to survive to API design and create brilliant APIs?

1.4.2 Mastering all aspects of API design

So what do we need to survive API design? We need to have a keen eye and ask questions. Thousands of them. What happens if we don't do that? What happens it we rush headfirst into the design of a programming interface neglecting others aspects of API design.


We should be able to transfer money with our banking API
OK. It's dead simple. We just have to add a POST /transfers to the API
A too fast and incomplete API design conversation...
It creates a money transfer of a given amount (amo) from a source (acc) to a destination (ben)
...lead to the design of this operation which may seem to be correct. But ...
This "design" is far for complete, not user friendly and may even be dangerous. Simply because...
Isn't the core banking slow when processing transfers?
Who is "we"?
Isn't transfer an asynchronous function in core banking system? How do we give feedback to consumer?
Should we really provide such sensitive operation to all users?
Are there business rules concerning the amount? Like min or max?
How do the consumer gets source (acc) and destination (ben)?
Should we really use core banking system nomenclature for parameters? acc, ben and amo are not really user friendly.
...
... too many questions were left aside.

```
POST https://api.bank.com/transfers
{
  "acc": "CA12345",
  "ben": "BE67890",
  "amo": 12.3
}
```

Figure 1.24 A not so simple API design use case

Such mistake may lead to incomplete, not really user friendly and even dangerous API design. There are many things to think about when designing an API. Designing the programming interface is one of them but there are others. Who are the users? Who owns the data? What flow and information are needed to achieve a specific action? Do we have to evolve existing operations? How does work the underlying system providing the API? Are there security issues to be aware of? Sensitive data? Etc, etc … So many questions, so many aspects. But hopefully they can be grouped into five categories.

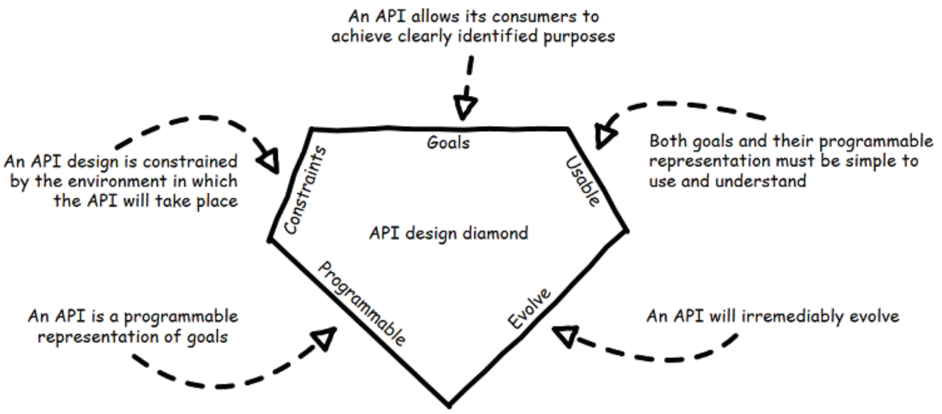

An API allows its consumers to achieve clearly identified purposes
Goals
Both goals and their programmable representation must be simple to use and understand
Usable
An API design is constrained by the environment in which the API will take place
Constraints
API design diamond
An API is a programmable representation of goals
Programmable
Evolve
An API will irremediably evolve

Figure 1.25 The API design diamond

These five categories are *goals*, *usable*, *programmable*, *constraints* and *evolvable*. They form the API design diamond. [1] Designing an API is, first and above all, designing something which as *goals*. An API allows some people do things that make sense for them. These identified goals can then be represented as a *programmable* interface. But, if nobody can understand what can be achieved with this API and how to use it , what is the point of creating it? An interface must be *usable* by both the people and software who will use it . An API has also to observe some *constraints*. An API exists in an environment. It is an interface between two worlds which must be secured by design. It must take into account how it will be consumed (used) and implemented (how the system providing the API is built). And finally, an API will irremediably *evolve*. An API's evolution can be taken care of by wisely creating it and cautiously modifying it after that.

Footnote 1 Some of them have been inspired by a formal definition of *design* that comes from *Design Requirements Engineering: A Ten-Year Perspective* by Kalle Lyytinen, Pericles Loucopoulos, John Mylopoulos and Bill Robinson, 2007, Springer

NOTE

Keep this in mind!

An API is a programmable representation of goals that that may evolve after its creation. Both goals and programmable representation must be easily understandable and usable and must also conforms to constraints imposed by the environment in which the API is built and used.

If you are aware of all API design diamond's facets and know how to deal with them you will be ready to face any API design challenge. You will be able to carve brilliant and shiny APIs.

1.5 What we have learned

In this chapter we have investigated what are these APIs we want to design and we have discovered the surface of what it really means to design them. We have discovered that an API is more than a simple *Application Programming Interface* and that it's design is far more than just defining some functions that can be remotely called by a software:

- APIs we are talking about in this book are web interfaces that allows software to interact with each other over a network to exchange data or trigger actions. They are a pillar of our connected world.
- APIs are interfaces for developers and their design is more close to user interface design than programming
- API design is composed of many aspects and *programming* is only one of them
- These fundamental aspects can be applied to the design of any style of API

In next chapter, we will the deeply explore the *goals* facet of the API design diamond. We will learn to identify an API's real goals and why it must be your very first task when designing an API.