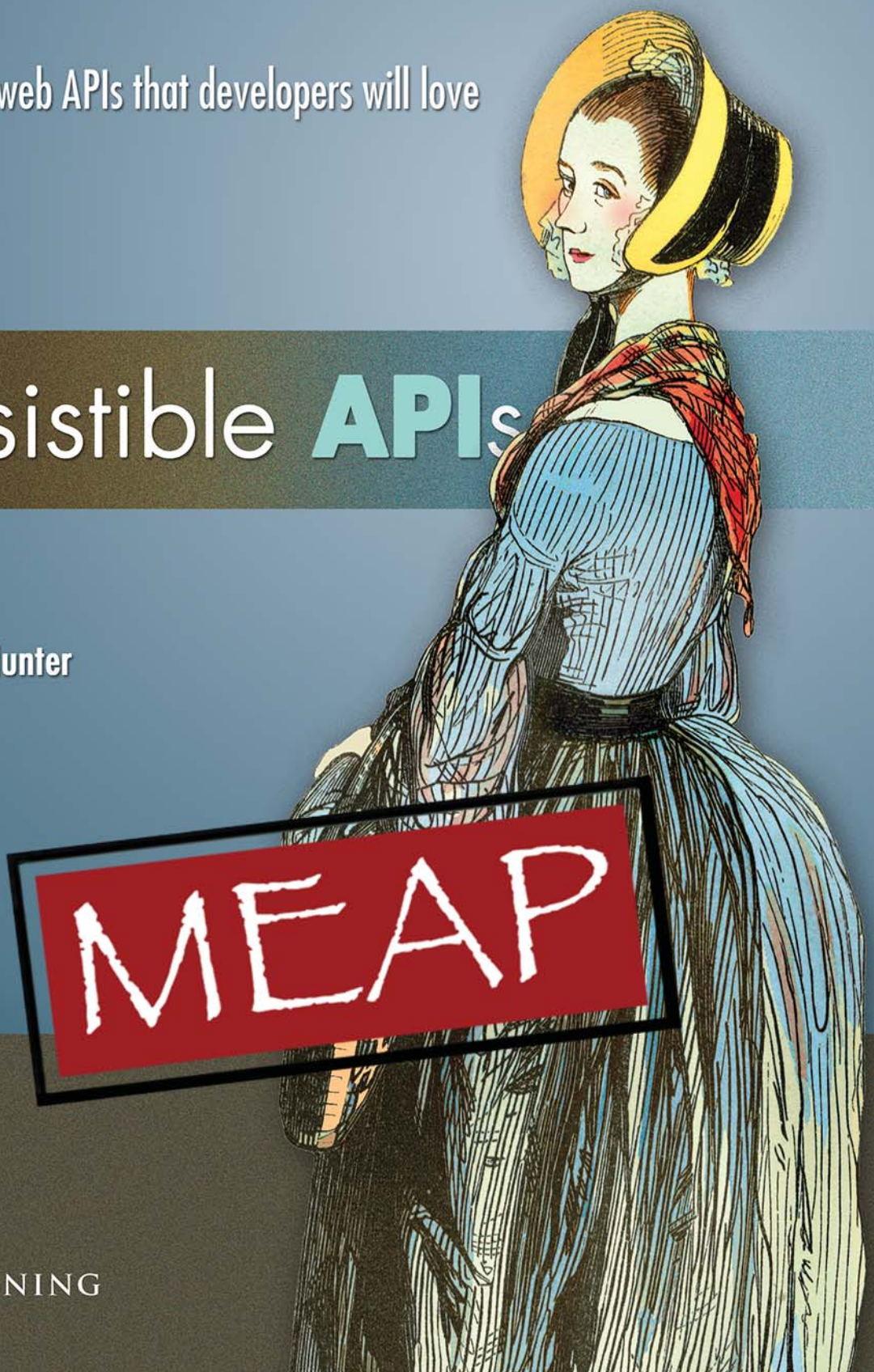
Designing web APIs that developers will love

Irresistible APIs

Kirsten L. Hunter



MANNING

MEAP Edition
Manning Early Access Program
Irresistible APIs
Designing web APIs that developers will love
Version 9

For more information on this and other Manning titles go to
www.manning.com

welcome

Thank you for purchasing the MEAP for *Irresistible APIs*! This book was written to be accessible to anyone with interest in Web APIs - specifically, developers or product managers looking to help their organization create an engaging, fantastic platform. There are specific sections for developers which go more deeply into technical topics, but the more general sections don't require any specific technical expertise. As long as you're generally comfortable in an engineering environment you should find what you need in my book.

I've been working with Web APIs for ten years at various companies. Most of my time and energy have gone into helping developers use these platforms without getting frustrated. Several of the earliest APIs made similar mistakes in creating APIs, generating frustration among developers who were trying to integrate them. My goal with this book is to help you learn from this history so that the platform you create is irresistible and provides an amazing experience for your developer partners.

Technologically speaking, it's relatively easy to create a solid Web API using REST, but the real challenge lies elsewhere—how do you create a system that's designed for success? This book outlines a process you can use to ensure that your product is usable and that your developers know exactly how to use the system to achieve their goals. Business value, use cases, metrics and deliberate design are needed to get your API right the first time, every time.

In order to really understand how APIs work, you do need some technical information, so the book covers the basics of REST web services, schema modeling systems, and gives some insight into the ideals you may want to strive for in the design of your platform. The final chapter will focus on a topic frequently skipped during API design—developer support and experience.

Your API should be a first class product. In order to ensure success, you need to treat it like any of your other front line products. This book will teach you how to do that and create an API that lasts and contributes significantly to the bottom line of your company. Engage your developers and partners by giving them a fantastic system that they can't wait to use.

Thanks again for purchasing the book! With your help and your feedback in the book's forum, I hope to create a guide to this process which can be used and leveraged by all its readers.

Regards,
—Kirsten Hunter

brief contents

1

What Makes an API Irresistible?

This chapter covers:

- What is a Web API?
- What can a Web API do?
- Developer Experience
- Common Pitfalls

An API is an interface into a computer system – an Application Programming Interface. Historically APIs started out as highly coupled interfaces between computer systems. Web APIs, which are much freer and less tied together, have been evolving for quite some time, but recently there has been a huge explosion in the web APIs available to developers. However, many of these APIs were developed without the end user (in this case a developer using the API) in mind, resulting in a frustrating developer experience and a less successful Web API.

On the other hand, an irresistible API is straightforward to use, well documented and supported, and the supported use cases are communicated and demonstrated well. Using your API should be a joyful and engaging experience, not a slog through a frustrating and never-ending problem.

This book will help you understand how to create web APIs which re loved by developers, are engaging and purposeful, and will experience success. I'll also discuss what factors you should consider to determine if you should have a platform at all. The guidelines included in this book are meaningful for any kind of Web API, no matter the technology or audience.

When you've finished reading the book, you will have a strong understanding of the process needed to create excellent Web APIs – APIs which enchant customer engineers and extend the platform's reach naturally, as those developers share their experiences with their

colleagues. While there are many different types of APIs in use in the industry, each with their own advantages, this book will focus on the development of RESTful Web APIs. Web APIs decouple the functionality of the server from the client's logic and features, encouraging client developers to use the data in whatever way works best for their application. On the other hand, non-Web APIs which tie the server and the client together tightly work to implement very specific integrations between the client and the server. For instance, SQL, an interface language tied into many databases, represents an API, but the interaction is focused on very specific actions. Exposing the data in a more free-form way wouldn't work for many of the uses people have for databases or other closely coupled systems.

In addition, you will learn a good basic understanding of the technologies involved in creating a RESTful API. REST stands for Representational State Transfer, and refers to APIs which are resource based – where the clients interact with the servers by requesting things, rather than actions. While the creation of Web APIs is technologically simple – a skilled developer can use Flask, Django, Ruby on Rails or Node.js to put together a basic REST API in a few minutes. Without a clear plan, design, and goal, that API will be unlikely to be excellent, usable, or successful. **How** you use those technologies makes all the difference between a successful, irresistible API and one that lies fallow in the ecosystem with no users. The book is made for you, whether you're a product manager, technical lead, engineering manager, API developer, or even a developer who wants to assess APIs you've created, or ones you're looking to use.

This chapter is focused on helping you understand the overall ecosystem of Web APIS – what the terms mean, and what things you want to accomplish (and avoid!) in creating your own API, and how to decide whether you actually need an API.

1.1 Integrating Social APIs into Web Content

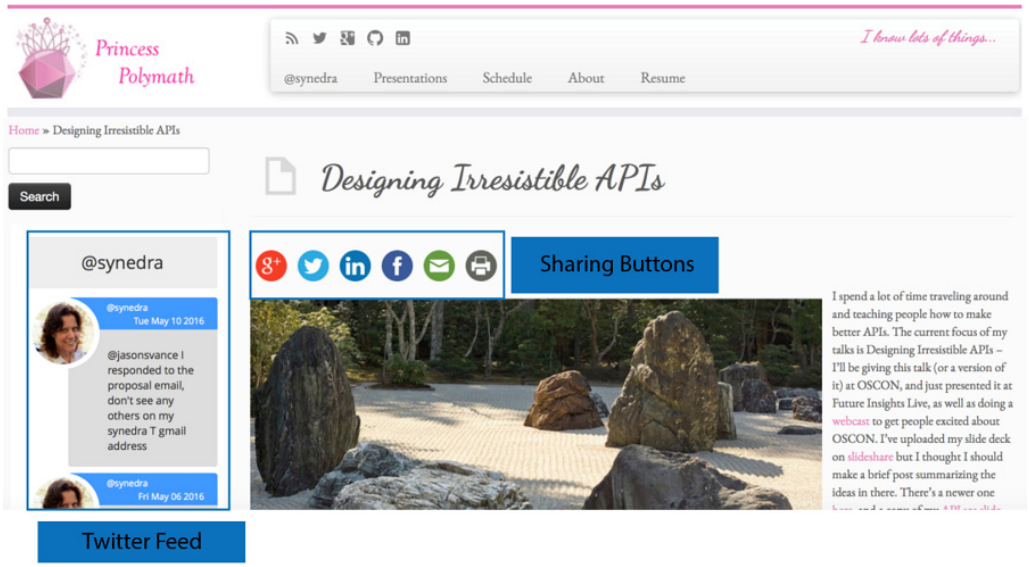


Figure 1.1 – The above shows a blog incorporating a twitter feed as well as API functionality for sharing to multiple social networks. A widget provides buttons for each of the social networks – Google+, Twitter, LinkedIn and Facebook, and this widget incorporates that network's API into the site in a manner that's easy to implement.

You likely use products that are incorporating APIs all the time. The "share" button you see on news sites and blogs uses the APIs for those social sites - like Twitter, Facebook, or LinkedIn. If you can "Login with Twitter" the site you're visiting is using Twitter's API to identify you – this makes for a much better user experience, as you don't need to remember more usernames and passwords, and you can jump right in to enjoying the system.

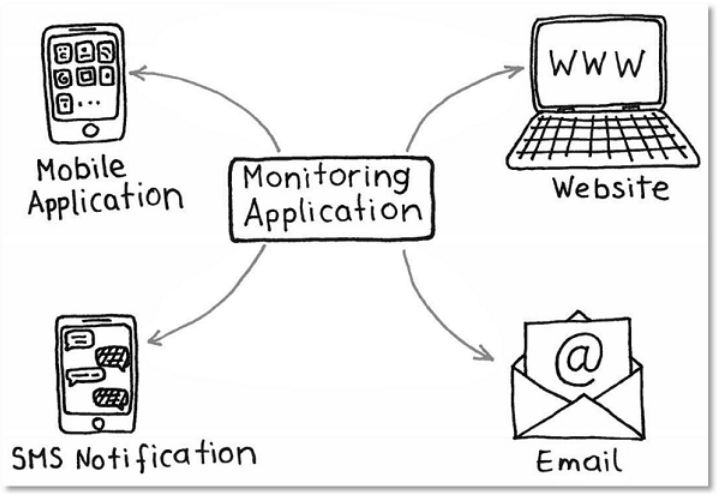


Figure 1.2 – Example API interactions with a monitoring application. When the monitoring application detects a change, it can propagate it to a website, mobile application, SMS notification or email.

A RESTful API is a platform that exposes data as resources on which to operate. For instance, a Contact Records Management application might make it possible for you to interact with users, contacts and locations. Each of these would be exposed as a resource, or object, you can interact with – whether reading or writing changes. When you create a well-designed RESTful API, developer users can create applications commonly referred to as "mashups." A mashup combines multiple APIs together to create a new user experience.

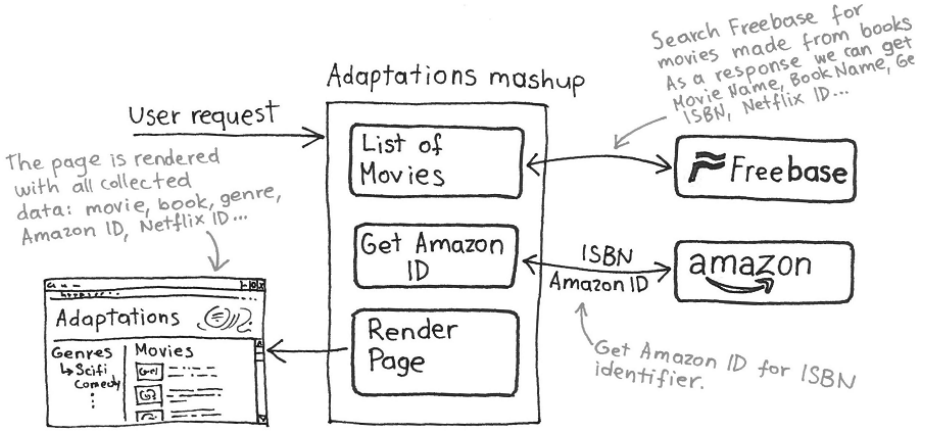


Figure 1.3 – This describes an adaptations mashup using Freebase, Amazon and Netflix. The user is presented with a grid of book-based movies matching a specific genre, and for each of the movies, they are given the opportunity to watch the movie on Netflix or purchase the book on Amazon. This is an example of a mashup, combining multiple APIs to create an integrated experience for the user.

As an example, I created a mashup using Freebase (think of this as a database of the world's information), Amazon and Netflix. The mashup allowed users to find books that had been made into movies, and then buy the book at Amazon or add it to their Netflix queue. I used information from Freebase to add genre information for the movies so users could browse around and find the movies that interested them most. This was a simple example of mashing up three different APIs to create a new way for users to explore a data space. Frequently, websites or applications are also leveraging the APIs of social networks for logins, sharing and showing your user feed.

This is a great place to give you one of the main concepts of this book. The developer experience for your customers is the most important factor in the success of your API. if you're trying to encourage creativity and engagement with a larger community, or inspire developers to help you reimagine your main product via the APIs, REST might be your best bet. On the other hand, if your API needs action-based methods that do a very small number of things in a very specific manner, you may well want to make a non-REST API, using SOAP (Simple object access protocol) or another technology designed for more tightly coupled clients and servers. Either way, the most important things to take into account are what your customers want and how you want them to use your API. Just remember, developers are people too.

1.2 What is a Web API?

The term "API" has been used for most of the history of computing to refer to the interfaces between computer systems, or different programs on an existing system. Frequently these systems were "peers" where neither of the systems was specifically a server or client. For instance, a mail server might have used a database to store information, but the systems were inherently designed together, tightly coupled to work together seamlessly. However, more recently the term has been expanded to include Web APIs, systems where a client – which could be anything from a web browser to a mobile application – contacts a web server, and operates on the data on that server. The main difference here is that the developers who are writing the clients are not the same as the developers writing the interface – the system is truly decoupled.

To understand the idea of a Web API, it's first useful to understand the protocol – the way that the systems talk to each other. For this, think about the switchboard phone system from days long past. Your phone only knew how to do two things: connect to the switchboard and make noise. If you wanted to call your Aunt Mae, you would pick up the handset and make noise with the ringer. After the operator answered, you would give her Aunt Mae's phone number and she would cause Aunt Mae's phone to make noise itself, and then connect up your two phone lines. In this case, you contacted the switchboard (acting in this case as the "server") and gave a very specific identifier to the operator, and that person connected your phones. The protocol for this was well known, and the users of the telephone were able to interact with each other long before auto-switching was technically possible.

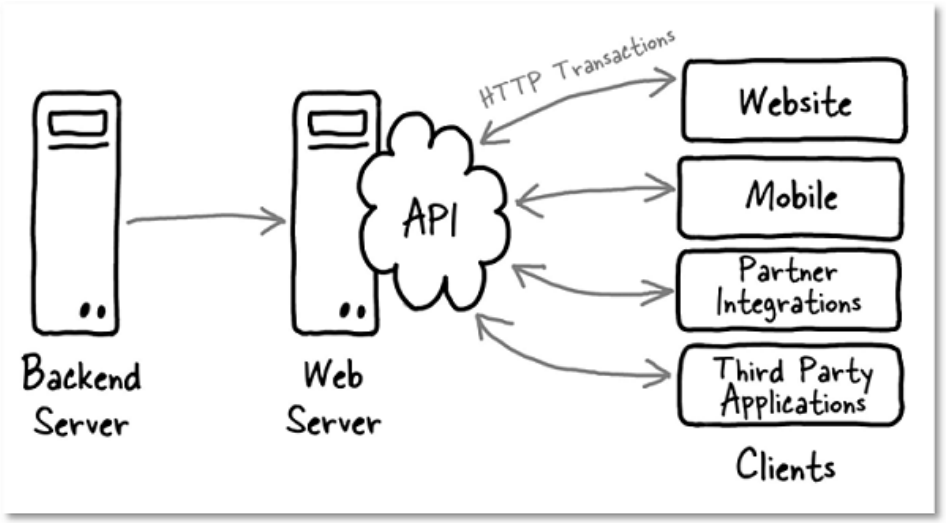


Figure 1.4 – The basic interactions with an API are direct connections with the backend server or servers, and a well defined interaction with clients on the front end. This allows for countless front-end applications, whether mobile, website desktop or system integrations, without changes to the backend servers.

Similarly, HTTP – the HyperText Transfer Protocol - is a well-known protocol, used to drive the web traffic browsers generate. A Web API is a system where clients use a defined interface to interact with a web server via the HTTP protocol – this can be an internal or external system. To understand how this works in the context of a browser, when you type an address into the browser's address bar, you're asking that browser to retrieve a unique resource, just like a phone number. The browser asks the server for the information associated with that identifier, and it's returned and formatted for you to view in the window. Web API clients make similar calls to read and write to the system, but the responses are formatted for programs to process instead of for browsers to display. One of the best known APIs is Twitter, whose APIs are open to third party developers – allowing those developers to create applications that integrate directly with Twitter. I'll be discussing HTTP in detail in chapter 4 of this book – Web Services Explained.

Once you are using a protocol, it's important to have a well described format for the messages that are sent through that protocol – what does a request look like, what response can be expected? To understand what needs to be communicated via the transactions between the client and server, the requests and responses being sent over HTTP, I will bring these questions back into the real world. In order to support the needs of a computer system, a RESTful web API needs to support creating, reading, updating and deleting items on the platform.

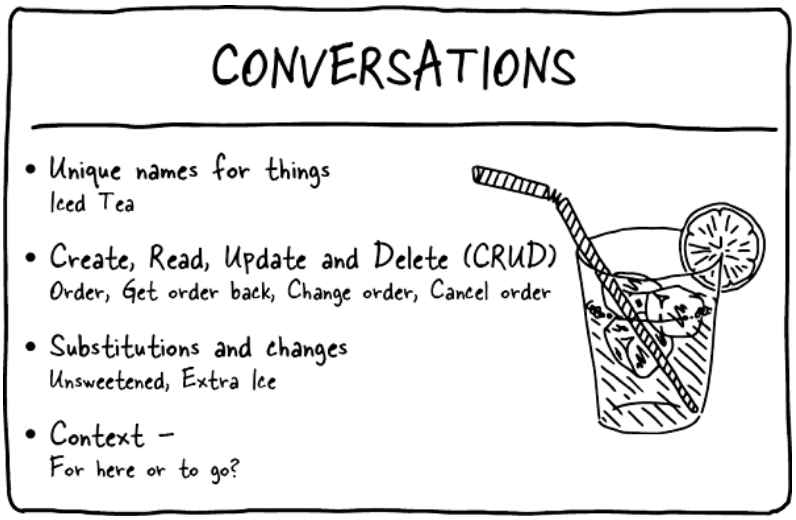


Figure 1.5 – This example shows how common terms map to API concepts. Resources are unique names, the methods are easy to understand, and options and context allow the client to express specific concepts via the API transaction.

To understand how this works, think about what it looks like when you order a drink at a coffee shop.

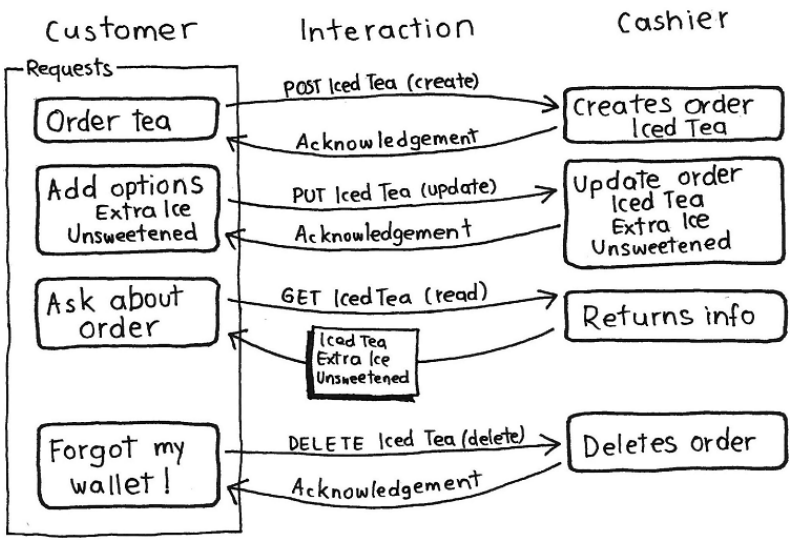


Figure 1.6 – This demonstrates a customer interaction at a coffee shop. While this diagram seems complicated, realize that the interaction represents exchanges you have had multiple times with service providers. An acknowledgement can be as simple as a nod from the cashier, and each of the requests made by the customer is a simple request.

When you request an iced tea, you have created a new item, an order. Adding options to this order, such as extra ice, updates that item. When you ask the cashier to tell you what you've ordered, you've essentially performed a read of that item. And if you've accidentally left your wallet at home and have to cancel the order, that item is deleted. There are more complex factors to the system, but this is the essence of a web API transaction.

There are several advantages to using a RESTful web API, which exposes the data in the system as objects for interactions between the client application and the platform. When two systems are tightly tied together at a deep level, it's hard to make a change on either side without breaking the other. This reduces productivity and creates a vulnerability to unexpected behavior, especially when the applications become out of sync. Writing code where the systems are separated by a documented interface protects both from unexpected changes. It's easier to test an API and easier to document the interface while protecting your internal methods from unexpected use.

So, what kinds of things can an API enable for you? I will dive deeper into business goals later in the book. Understanding the various ways that you can leverage an API also depends on an understanding of what can be done with an API.

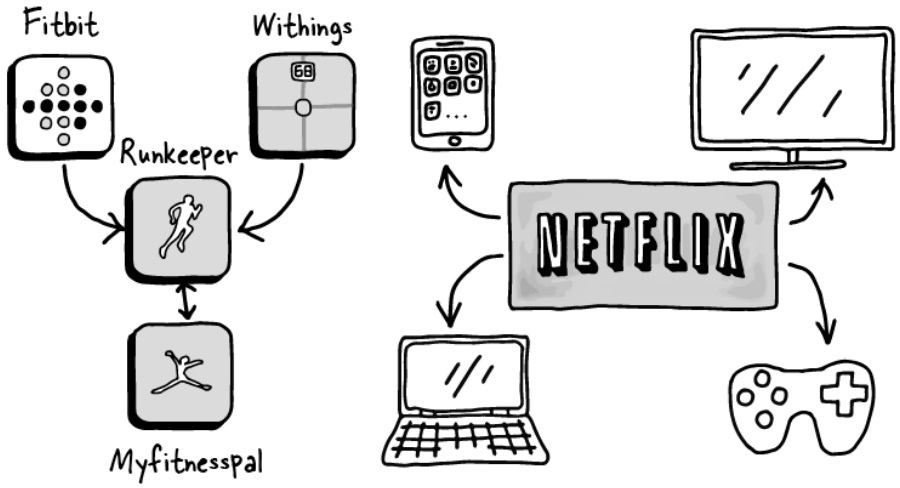


Figure 1.7 – This diagram shows multiple systems integrated together. On the left, you can see Fitbit, MyFitnessPal and Withings feeding into the Runkeeper system, which can aggregate all of your fitness data in one place. The picture on the right shows Netflix and all of the different types of devices they have integrated with, cementing their leadership in the video industry.

Figure 1.7 demonstrates just a small number of the types of integrations that are possible with Web APIs. Mobile devices are really the main driving force between APIs – in almost all cases, the need for an API is driven by the requirement to have a responsive mobile application. Mobile is an almost universal use case for companies developing APIs – for this reason, it is frequently the main use case. Allowing your customer developers to integrate their system

with your platform saves you the development resources needed to create custom implementations for each separate partner.

1.2.1 Do You Need an API?

One of the first questions you'll want to answer is whether to create a web API at all. Creating an API without any purpose is destined to fail, as is any other product. This question should be clear once you go through the process of determining the business value for your API. Whether you are trying to improve engagement, support a mobile strategy, or integrate with other systems, an API takes time and resources to create correctly.

APIs can improve velocity, engage partners, and enable mobile devices, but if your business does not (and will not) need these things a web API may not be right for you. However, it is still worth going through the exercise to determine at what point an API may be needed.

1.2.2 Choosing REST APIs

When deciding whether to use REST for your API you'll need to take into account your customers' needs and the systems you need to interface with. Over the last several years, REST style APIs have become the most popular type of Web API. The main difference between previous API structures and REST is that REST APIs are designed around the idea of "nouns" in the system, instead of "verbs."

REST APIs are designed to encourage creativity and innovation by allowing developer consumers to decide how to use the data available. As with open source, this openness can cause concern for large enterprise companies who are focused on protecting their proprietary information, but security and privacy can be retained in both open and internal RESTful APIs when mindfully designed. Another term for an API of this sort is "platform," describing the underlying system as a foundation for developers to use to build their integration or application.

1.2.3 JSON

The most popular response format for a REST API is currently JSON, the Javascript Object Notation, which is an efficient way to represent data passed between the server and the client.

JSON Sample

```
{ ❶
   "glossary": {
      "title": "example glossary", ❷
        "GlossDiv": {
        "title": "S",
   "GlossList": { ❸
       "GlossEntry": {
          "ID": "SGML",
          "SortAs": "SGML",
          "GlossTerm": "Standard Generalized Markup Language",
```

```
                            "Acronym": "SGML",
                            "Abbrev": "ISO 8879:1986",
                            "GlossDef": {
                        "para": "A meta-markup language, used to create
                                 markup languages such as DocBook.",
                                 "GlossSeeAlso": ["GML", "XML"]
                },
                                 "GlossSee": "markup"
            }
        }
    }
}
```

1. The top level item of a JSON item is usually an object (designated with curly braces)
2. Objects within objects are name : value
3. These objects can be objects contained within the main object. Nesting of objects is supported at every level

The JSON format is compact and thus easy to transmit on a slow network. For this reason, it is manageable on devices with questionable network connectivity (such as mobile devices). Interpreted languages such as JavaScript, PHP, Perl and Python work with JSON quite easily – in fact, the objects look very familiar to those developers, which is one reason it has become so popular. At this point, if you are creating a new API and must choose one format, you likely want to use JSON. Adding a second output type adds more overhead and duplication, so unless your customers explicitly need some other format (such as XML) for some reason it's best to stick with just the one. Choosing to support two different formats will incur a reasonably large amount of technical debt – creating the need to do extra work as you develop your product - as you need to double your use case testing and increase development time.

1.3 Developer Experience

The experience of the developers using your API is **the** most critical factor in the success of your APIs. An API which is open to external developers, that has a poor developer experience may well drive third party developers to your competitors. Even an internal API needs to think about developer experience. Creating internal APIs without excellent documentation and tutorials will create a huge support burden as your consumers struggle to write their implementation and flounder without understanding how best to use your APIs.

A great developer experience begins with understanding your goals for your API and communicating those to the developers you want to engage. The more information you can share with them, the better able they will be to decide if they want to use your APIs. Remember, when a developer comes to use your API, whether it be an internal, partner or open API, that developer is giving you the only thing they can never get more of – time. Demonstrating that you recognize this is key to developing a trusting and mutually beneficial relationship with those users.

Once you have determined the business value, metrics and use cases for your API, communicate this information to your developers. A trusting relationship can most easily be accomplished by practicing transparency wherever you can – as we work through each of the sections in the book, consider carefully if there is a strong reason not to share information about your business values, metrics and use cases with your developers. If not, share it! You may not think that a developer will want to understand your business value or use cases, but giving them this information communicates to them clearly that you are serious about your API. Allowing your customers to view your design documents before the API is complete gives you the opportunity to get valuable feedback during the development process. Telling developers exactly how you're going to measure the success of your API – whether it's user engagement, number of end users, or some other factor - gives them the opportunity for them to work on your team, improving those metrics you've identified as critical to the success of the API.

Providing tools, example code and tutorials are also critical to the developer experience. Twilio, a company that provides APIs for voice calls and SMS messages, has a goal that any developer entering their site should be able to make a successful call from their system within 5 minutes. Having this kind of exercise available grabs your potential developer and engages them in your system. While they may have only planned to browse your site for 5-10 minutes, making that call will encourage them to invest more time in understanding what they can do.

1.3.1 Versioning

Versioning is one of the hardest facets of API management. It may seem like a newly created API can be "pretty decent," assuming that you can fix it up later. Unfortunately, the choice of which version to use is in the hands of developers – developers you likely don't control. Once a developer has made an integration or application with your system, they're not likely to be motivated to move to a new, incompatible version unless there is new functionality that's critical to them. You need to really take your first version seriously, knowing that you may be supporting that version for a very long time.

Compatible versions that don't change or delete existing functionality are generally well received, but as soon as you need to make a change that breaks existing implementations, you are going to be facing resistance from your developer community, especially if you deprecate a version that they've stuck with – even if you give them a long warning period to migrate, they may not want to re-do the work that they've done. On the flip side, maintaining older versions creates duplicate code and requires that you make fixes to the earlier code as well as the existing code. Troubleshooting and support becomes much more difficult when you have developers on multiple versions. This means that your first version needs to be a solid contender, something you intend to use for a very long time. The website model of releasing a new version every week or two simply does not work for an API. While the API does provide a decoupled world, developers will still rely on the version they started with, so you need to be aware of the price of new versions.

1.3.2 Marketing to Developers

Finally, make sure that your marketing is targeted appropriately to developers. A developer will have one main question when coming to your portal: Why do I care? This usually can be broken down into "What can I do?" and "How do I do X?" Unfortunately, documentation is frequently stuck in the realm of "What does each piece do?" which doesn't meet the needs of the questions above. Developers don't tend to be inspired by pretty pictures or taglines – they want to play with toys. Give them example code and applications, the building blocks they need to get started making magic with your platform.

All of these ideas will be revisited throughout the book – developer experience should drive most of the decisions you make. The specific topic of developer support will be covered in more detail in chapter 9: Developer Support. Following these guidelines will help you to avoid some of the commonly encountered problems with the first generation of APIs.

1.4 Common Pitfalls of Organic APIs

While it may seem like APIs are a new thing, REST APIs have been out in the world for over 10 years. At this point the industry is moving from innovators who are blazing the trail into more stable and mindful API creators who can leverage the learnings of previous platforms. This means that you have the advantage of learning from the mistakes made by the first APIs, which were created without a view toward how the APIs would be used and measured. Twitter, Netflix and Flickr… these very heavily used APIs are well known, but you may not realize that at this point tens of thousands of APIs have been created – and as APIs become more common those numbers will rise into the hundreds of thousands. Many of these APIs have been left by the wayside, and even more are struggling to achieve success in a relatively crowded marketplace. Distinguishing your platform from your competitors by having a usable API with a fantastic developer experience will help you rise to the top of the stack.

I'm going to discuss a few specific examples that illustrate common mistakes made by platform companies. Remember that the APIs I'm using as examples have actually attained success in the long run, but the early stumbles cost them money, resources and credibility, and in some cases have prevented them from making design changes to improve their API.

1.4.1 Lack of Vision

When the Netflix API was first released, it was open to the developer community at large. The goal of this product was to "let a thousand flowers bloom," to enable third party developers to create amazing applications which would bring in new subscribers and make money for the applications while increasing Netflix's subscriber base. Guidance for developers on how to use the API was minimal, focusing on what the API did rather than providing tutorials and use cases. Developers were attracted to the platform and created many applications, but the revenue benefit never occurred. Since the company's goal for the API was unfocused, developers didn't know what the company wanted them to accomplish with the API. As a

result, many clients recreated the functionality of Netflix's website without adding new functionality. Netflix employed a large team to support the open API and encouraged developers to jump in with both feet and create applications and integrations into the system. However, a somewhat restrictive Terms of Use, limiting the way that third party developers could integrate with the system muted the innovation they were hoping for. The main issues in this case were that Netflix required attribution, disallowed the ability to combine information with other vendors, and most critically, required that advertisements were not associated with Netflix content.

Through partner use of their APIs – using the APIs to integrate Netflix into various video devices - Netflix discovered that the API was an excellent way to establish market dominance by creating efficient integrations into devices (such as the Xbox, Blu-ray players and smart TVs) and partner products (such as Windows). This business decision was a great one for Netflix, but their API was now focused on a very specific market segment. The existence of third party developers, however, meant they needed to continue to use resources to enable and support alternative use cases. This turned out to be a costly use of resources, for very little business value. As time went on, Netflix focused the API more and more on the device market and less on the open version of the API. Support for open developers declined, new features were offered only to partners and device manufacturers, and the open API was eventually decommissioned entirely.

This is an excellent example of an innovator creating a product which had a strong negative developer experience, but that turned out to have an excellent value for the business. That business value was not defined at the beginning – communication with developers encouraged them to innovate and create, and those developers trusted that the resource would continue to be available to them. This situation soured many developers on platforms in general, as they had spent a great deal of time and money implementing applications that eventually failed entirely.

1.4.2 Prioritizing the Developer Experience

Twitter started with a single web page where you typed your message, and you could follow people or send messages – that was the extent of the features. There was no image management, no lists, and none of the other features social systems feature at this point. Developers loved the API – while the developer experience was a little awkward at first, it was easy to imagine what you could do with such a system. Many Twitter features were initially created by external developers as part of their product, and Twitter adopted the new features because users liked them. This was a decent setup, although developers were somewhat unhappy with the feeling that their ideas were being "stolen" by Twitter.

Eventually, that situation changed for the worse. Twitter rewrote their terms of use so that developers could not create applications that competed directly with their product, and existing applications of that type needed to be killed. This meant that when Twitter adopted a new feature, any existing applications that relied on that feature to distinguish themselves from Twitter could no longer exist. Twitter wanted users to integrate sharing and social media

into their own applications rather than creating applications based directly on Twitter. Unfortunately, because this message wasn't shared until after several missteps the developer community was, by and large, quite unhappy with Twitter, and their credibility suffered as a result. Fortunately, with their enormous user base, they weren't seriously affected by the fallout.

Additionally, Twitter eventually realized that there were aspects of their API which were costing them time and resources without being used by the majority of their developer customers. For instance, all of their calls could be retrieved using JSON or XML – a more expansive structure preferred by some customers. The XML version, however, was used by less than 5% of the developers, so Twitter created a new incompatible version, which refined the API into something matching more closely with what most of their developers were using. As I mentioned, the creation of backwards incompatible version changes has a high cost. Twitter took a long time to "sunset" the older API, and still there were many unhappy developers as a result of this change.

As an innovator, Twitter was bound to experience some growing pains as they developed their API. At this point, they have one of the best developer portals around, with excellent tutorials, support in the form of forums, and a well documented API that is straightforward and consistent. If they had known at the beginning what they have learned about how users would use their platform – or had another API on which they could model their platform - they would have been able to create the right API at the start. On the other hand, they learned valuable lessons from the developers, watching them use the API and seeing what they did. Sometimes you need to break the rules to figure out the right answer – but knowing what the right plan is can help you avoid unnecessary mistakes and help you achieve the success you want.

1.4.3 Bad API Design

Flickr – one of the first photo sharihng serveces. was one of the first APIs, as well. While they were attempting to create a RESTful API, instead they created an API which worked with actions instead of objects (verbs instead of nouns). They have since made improvements in their API, but there are many developers who had already implemented the old API so it was difficult for them to create a new, better version. The original design choices are helpful examples to understand some of the problems that come with a lack of understanding of the basic technology to create RESTful APIs.

To understand this, I'll describe an example of the non-RESTful choices made by Flickr. To delete a photo from Flickr, a consumer needs to make a request to the 'delete_photo' resource related to the photo in question. Originally, this photo would be deleted by sending a GET request to this `delete_photo` resource. The RESTful way to make a change to the server is to use the appropriate HTTP verb – in this case, DELETE is the right way to delete a resource, where GET should be used only to read the existing value of a resource. The distinction may seem minor, but when you break these rules you create a situation where your API behaves in an unexpected way for the developers consuming that API. Just as in designing for the end

users of a product, you do not want to create a situation where the behavior of the system is a surprise to the developer.

REST

```
DELETE /photos/1234
```

FLICKR

```
GET /photos/1234/delete_photo
```

The specifics of this will be covered in the chapter on Web Services, but I will point out some issues with the Flickr implementation.

First, in REST APIs, a request using the GET method should never change the data on the server. One reason why this is important is that if your API doesn't follow this rule, a web crawler or other automated system could accidentally change or delete all of the items in your system. Since the development of strong authentication, this has become less of an issue, but it's important to realize that this rule is known by most application and website developers, so making this kind of design choice creates a situation where the behavior of the API is not what the developer expects. HTTP provides a set of "verbs" which are well defined. In this case, there is a DELETE method that deletes the resource from the system, so that is the right way to accomplish this task – a DELETE on the resource rather than a GET to the action.

Another problem here is that by exposing actions instead of objects, the developer is constrained in their use of the API. Exposing nouns instead of verbs means that the developer is free to imagine new and creative uses for that data. Additionally, operations on the same object are at the same address, which matches much better with object oriented design and helps enforce consistency in the developer's experience. If a particular API object doesn't support a particular method (like DELETE), the system can send back an error code that is consistent with the HTTP specification, and the developer will know how to handle that error.

It may not seem terribly important to follow the REST specification for your API. It's your system, you can choose how to implement it. However, remember that your developer customers may be familiar with how REST APIs tend to work, and whenever you change the way things work, they may well get confused or make incorrect assumptions. Wherever possible, follow the general principles will be laid out in chapter 4, the Web Services chapter. The more similar your API is to other existing APIs, the better the developer experience will be. Once you have worked through these prerequisites, you are ready to create your API.

1.5 API Creation Process

Chapter 7 on API Design and Schema Modeling is where it all comes together, where you learn the steps you need in order to create a successful and engaging Web API. These steps include:

- Determining your Business Value
- Choosing your Metrics

- Defining your Use Cases
- Designing your API and Creating a Schema Model

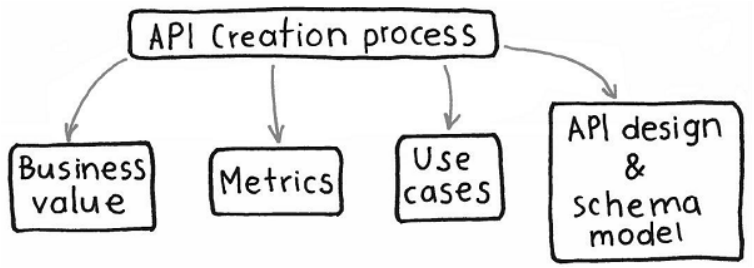


Figure 1.8 The process must contain all of these factors in order to become successful. Threat you platform like a first class product and watch it thrive.

Many times it's tempting to skip the steps I am covering this book, but just like your main product, your Web API should be a first class citizen – with the same effort given to designing, implementing and supporting your platform as you do for your website or application. Treating your API like a real product will help you immensely in getting your platform right the first time.

1.5.1 Determine your Business Value

Many of the first platform products had very vague goals. Third party developers experienced a world where they had to struggle to understand what the company wanted them to do with the API, and even worse, the platforms they were relying on underwent drastic changes or were deprecated all together. This could have been avoided had they clearly determined their business goals from the outset.

When thinking about business value, think of the "elevator pitch" so commonly targeted by new startups. If you get in the elevator with the CEO of your company and he asks why you have an API, you need to be able to succinctly answer the question in a way that convinces him that it's a valuable product. "Developer engagement" or "API use" is not a great goal. You need a tangible goal related to the larger business such as "Increase user engagement," "Establish industry leadership," "Move activity off the main product to the API," or "Engage and retain partners." Monetization is a nice aim, but unless your API is your main product you need to see it as a supporting product, enhancing the value of your product.

Another compelling reason to have a great handle on your business value is resource contention. Your company can only afford to pay a certain number of engineers, and usually those engineers are divided into revenue producing products and support engineers. An API is an awkward sort of product – a great API can exist that doesn't add directly to the bottom line. If you can't consistently and succinctly explain why your API is valuable to the company, and how it is supporting and enhancing the existing revenue producing products, your API may well suffer. When the leaders of the company don't understand the value of an API, it

may be relegated to the back burner to get updated "when the engineers have time" or worse yet, decommissioned entirely.

As you go through the book, keep thinking about what you want to accomplish with your API, and what you want your developers to do, so you can mindfully create your API to achieve success.

1.5.2 Choose your Metrics

Whatever the business value of your platform, your CEO is going to want to know whether you're achieving it, and getting buy-in from other teams is much easier if you can demonstrate how your API is doing. You need this information so you can quickly evaluate the API you're creating, or the effectiveness of changes you make.

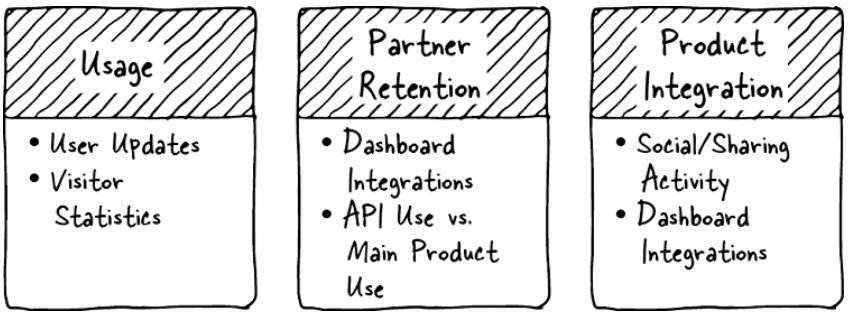


Figure 1.9 – These are examples of different business values with the associated metrics. The most important concept to understand is that you want to have a business value that's meaningful to people outside of the API team, and metrics should support that value in a way that's meaningful to your overall organization or company.

Frequently you'll see API metrics such as "Number of developer keys in use" or "Number of applications developed" but that's a very internal type of metric, unlikely to be meaningful to the business value of the platform itself. Developers have to get keys in order to try out your system, and 90% of those keys may be completely inactive and meaningless. Instead, would it be useful to know what activity the platform enabling on your system? How many users are interacting via the API? How many times/day are users interacting with your system? How many of your partners have created integrations into their systems? Increasing engagement with your system is a great goal, and relatively easy to measure, though you need to make sure that the actions that are happening are good for your business. Just try to think about how your API is enhancing the goals of the company as a whole rather than simply determining how many people have begun to integrate with your system.

1.5.3 Define your Use Cases

Once you've identified your business value and how you're going to measure it, it's time to figure out what use cases you want to support. Your main product is frequently a great use

case to target – whether it's the entire product or a subset. For instance, if there's a sharing component to your API, you may want to highlight and improve engagement for that feature through your platform. Thinking of your main product in terms of API features that might be useful is a great way to start thinking about what use cases you want to support.

If you want to engage mobile developers – and you probably do, as people spend an inordinate amount of time on their smartphones and have become conditioned to expect an excellent user experience on the smaller screen – you need to understand the needs of mobile developers and address them. This use case is important enough that I strongly suggest you consider it even if mobile isn't in your immediate future, as this use case will have a strong effect on choices you make when designing your system. Mobile developers need the API to be performant and robust, and they are very unlikely to use an API that cannot deliver all the information they need from the system for a single screen in one call – ideally, with all the relevant information and no extra data. Your user could walk into a tunnel or elevator and lose connectivity, and the developer will want the application to be quick and robust enough to handle these cases – unless the API is designed with these developers as a use case, it's easy to create an API that's not usable for mobile cases.

Partner integration is also a strong candidate for use cases. When designing an API, partner engagement is frequently a great business goal – these partnerships bring in consistent and reliable income and you want to make it as easy as possible for partners to integrate your API into the systems that they use. If your main product is a service they rely on, you may expose the metrics on those systems so they can integrate with your dashboard. If you have a social or communication element, they may want to display news streams within their employee portals. Whatever the case, these use cases are important to evaluate – partner engagement is generally a key goal for the company and supporting that will make your product central to that end.

1.5.4 Design your API

Developers are generally chomping at the bit to create the API as soon as the topic comes up. Their aversion to taking the time to create designs stems from the fact that Web APIs, especially REST APIs, are very easy to implement, and it's hard to damp down their enthusiasm for the idea of creating a product they can code so quickly.

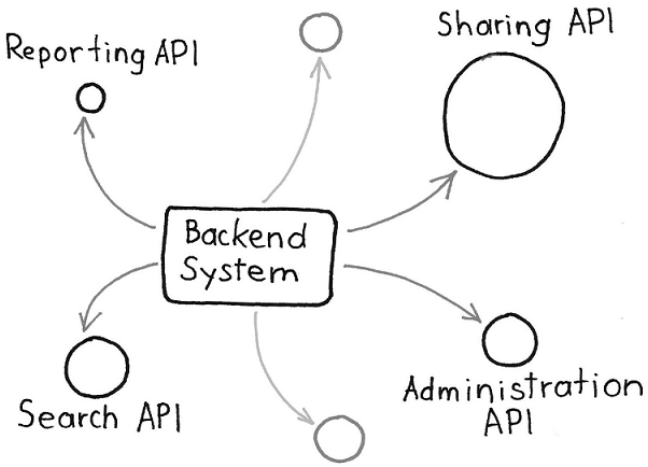


Figure 1.10 – Organically grown APIs are unrelated to each other. This results in duplicate (and sometimes inconsistent) code, different interfaces to the same information, and extra work for the consumers of the various APIs to implement integrations with applications.

Unfortunately, APIs that are created without a deliberate design frequently end up looking a little bit like a potato you left in the garage for a couple of months, with sprouts coming out at weird angles, completely unrelated to each other, unattractive, with no clear goal or consistency. I frequently refer to the created APIs as the Wild Wild REST, because it's so easy to create a system that is functionally complete but essentially unusable. Backend architects tend to think of development in terms of reliability, scalability and efficiency. However, a Web API is a form of user interface, so it is critical that you take the time to define how the API will be used, what it should look like, and defer prototyping or development until this task is complete.

The process of designing an API can be challenging unless you have some kind of system available to describe what the API will look like when it's done. Unlike earlier API structures, REST doesn't require any documentation about what an API does or how to use it. Historically this has meant that REST APIs are created without an explicit overall design, but this creates a variety of issues. First, the product manager for the API is generally the person most knowledgeable about the use cases for the API itself, but if there is no tangible description that can be discussed, those product managers may be unable to weigh in on the appropriateness of the proposed API.

The answer to this problem lies in schema modeling systems. These products allow you to describe the interface for the API before any development starts. Deliberate design before starting development encourages open communication and prevents the frustration of duplicating development work and misunderstood requirements. When a model is created and used to drive the development process, use cases can be closely mapped to specific sets of endpoints (each named API resource is referred to as an endpoint), and work can be

prioritized in a meaningful manner. The API can be reviewed to make sure that the endpoints are consistent and that there aren't any obvious holes or feature mismatches. A mock server can be deployed for customers or tests to determine if the use cases will be easy, and documentation, tests and clients can use this structured documentation to stay current with new API functions and features.

Modeling schemas for REST APIs isn't necessarily an easy task – making sure that different products are presenting the same information in the same way is something that takes time and work. For instance, an endpoint associated with accounting and reporting might present a "user" and their accounting information differently than the document management system, which would show ownership and editorial information. This process helps you to avoid those clashes during development and meet your deadlines with a minimum of difficulty. Backend systems usually keep all of this information segregated, but when designing an API it's important to think of your data as items within a cohesive system rather than distinctive items within separate systems. This is one of the main guidelines frequently missed when API design is left to backend architects, who are generally focused on scalability, performance and precision as opposed to usability. Your API design should include a human readable definition for each endpoint, the methods, fields and formatting for the response, and any other metadata that is important within your system.

In chapter 7, where I discuss modeling schemas, I'll discuss the two of the three main schema modeling languages – RAML and OpenAPI (previously Swagger). Each of them uses markdown or JSON to define the behavior of API endpoints, and they all enable communication, documentation and testing. Each one is an open standard, but each are championed by the company that originally created them, and if you are using one of the associated systems for API management, design or documentation you'll be best served by using the matching modeling language. What follows here are examples of the schema modeling languages, to give you an idea of what information is maintained in each one.

```
FORMAT: 1A

# Polls

Polls is a simple API allowing consumers to view polls and vote in them.

## Questions Collection [/questions]

### List All Questions [GET]

+ Response 200 (application/json)

        {
            "question": "Favourite programming language?",
            "choices": [
                {
                    "choice": "Swift",
                    "votes": 2048
                }, {
                    "choice": "Python",
                    "votes": 1024
                }
            ]
        }
```

Figure 1.11 – This is a blueprint definition for a simple "notes" API – Blueprint uses markdown for the formatting, and as you can see this is a very human readable document. As long as someone understands the basics of HTTP interactions, they will be able to parse this document – which means that your product manager, customers and the other development teams you interact with will be able to understand your API long before you even start coding.

Apiary is a company that provides API design tooling and support. Blueprint, the schema modeling language championed by Apiary, uses Markdown. This system focuses on the entire development cycle as described in this book. In addition to providing a structured language to describe the API, Apiary makes it easy to run a mock server and allow customers to comment on upcoming API changes. Apiary provides intuitive design tooling to make it easy for anyone to describe the API, and has a handy starter template to play around with when you're getting started.

```
{
  "title": "Swagger Sample App",
  "description": "This is a sample server Petstore server.",
  "termsOfService": "http://swagger.io/terms/",
  "contact": {
    "name": "API Support",
    "url": "http://www.swagger.io/support",
    "email": "support@swagger.io",
  },
  "license": {
    "name": "Apache 2.0",
    "url": "http://www.apache.org/licenses/LICENSE-2.0.html"
  },
  "version": "1.0.1"
}
```

Figure 1.12 – OpenAPI sample markup – unlike Blueprint, OpenAPI (previously Swagger) supports JSON and YAML as their markup languages, and includes the ability to include abstractions for objects in the system (such as a "user" or a "contact"), which encourages readers to consider the resources of the API in an object-oriented way.

One of the most popular and vibrant schema modeling languages is the Open API Format, known previously as Swagger. This is the only schema modeling language using JSON rather than Markdown, and it additionally supports YAML, a third markup language (YAML actually stands for Yet Another Markup Language). While the functionality of this system matches that of the other two, there aren't any easy tools to use to create Swagger documents, and there aren't readily apparent templates to help you get started. This leads to a much steeper learning curve for new users and can discourage you from using the system.

```
/books:
  /{bookTitle}:
    get:
      description: Retrieve a specific book title
      responses:
        200:
          body:
            application/json:
             example: |
                {
                  "data": {
                    "id": "SbBGk",
                    "title": "Stiff: The Curious Lives of Human Cadavers",
                    "description": null,
                    "datetime": 1341533193,
                    "genre": "science",
                    "author": "Mary Roach",
                    "link": "http://e-bookmobile.com/books/Stiff",
                  },
                  "success": true,
                  "status": 200
                }
```

Figure 1.13 - RAML from Mulesoft – RAML supports markdown, just as Blueprint does, but the schema that's created is more expressive than Blueprint. Additionally, like the OpenAPI framework, they support abstract objects more natively than Blueprint, making it easier to implement and maintain consistent APIs across a complex system.

Mulesoft is an API management system, and RAML is their schema modeling language. This language, is designed to encourage re-use of best practices among API providers. Additionally, the language and tooling are designed to make API discovery and exploration easier. As with Apiary, templates and tooling are provided by Mulesoft to soften the learning curve.

None of these options is necessarily better than the others, and they each have different features and focus. I will walk through two of the main schema modeling languages – RAML and OpenAPI – in chapter 7. Whatever system you use, having schema model will create an artifact representing the design choices you have made for your API.

1.5.5 Industry Standards

As you design your API, one thing to keep in mind is that an open API, unlike a website or application, will be public once released. You won't be able to hide the schema from your competitors. This is actually not a problem – many companies believe that the model for their

API is their competitive advantage, but those companies are wrong. The data you're presenting and the algorithms you use to make it more engaging are the things that separate your API from the other companies in your industry. The strong notion that your API should be secret until released has resulted in a huge disparity in schemas between companies in the same industry – this has created a situation where developers have to work much harder than they should to integrate similar APIs together to create a better client, because each API requires that they start anew with the development, and the task of combining data that's structured differently falls to the developer.

If you think about your API as a public interface, it becomes obvious that you should strive to learn from other companies in your industry and work towards best practices with them. This will help the entire API ecosystem to mature much more rapidly, and will encourage more developers to try out your API. If you consider an industry such as fitness, for example, it's easy to see that weight or number of steps per day are items which are quite simple – creating different ways to access, manage and interact with those items between Fitbit and Runkeeper simply increases the difficulty that a developer will encounter in integrating these systems.

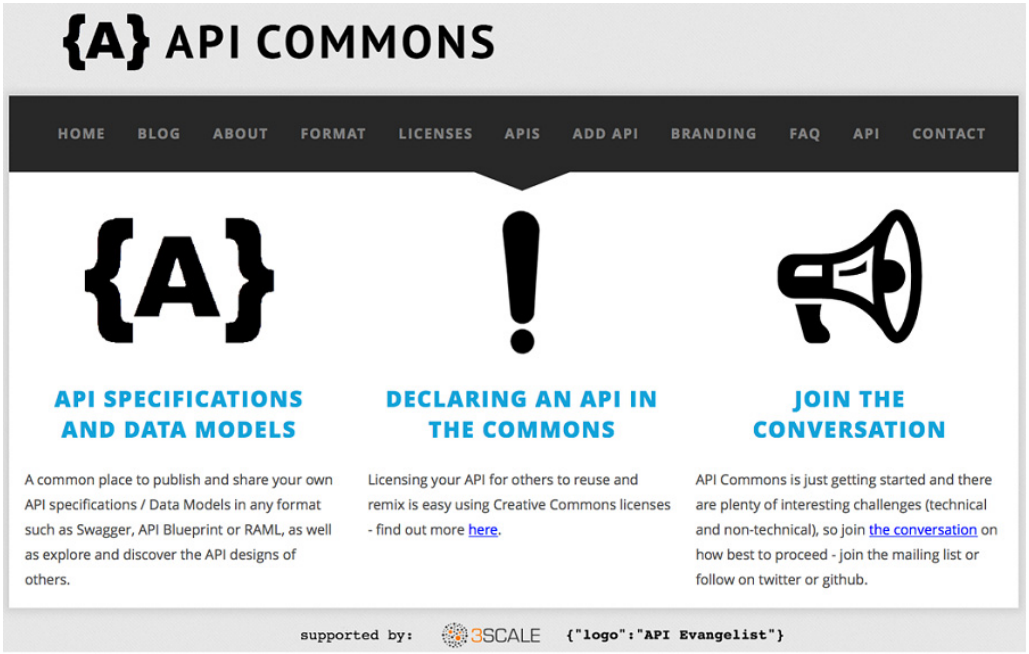


Figure 1.14 – The API Commons was designed for companies to define and store their schema models, so that other companies with similar APIs can leverage these existing schemas when building their own APIs. This allows the developers consuming these APIs to interact with similar systems, rather than needing to translate between numerous different representations of the same object.

In keeping with this idea, API Commons was created as a place to store and share design schemas. The idea of this project is that companies with design schemas can check them into a shared repository, and use these to bootstrap their own schemas.

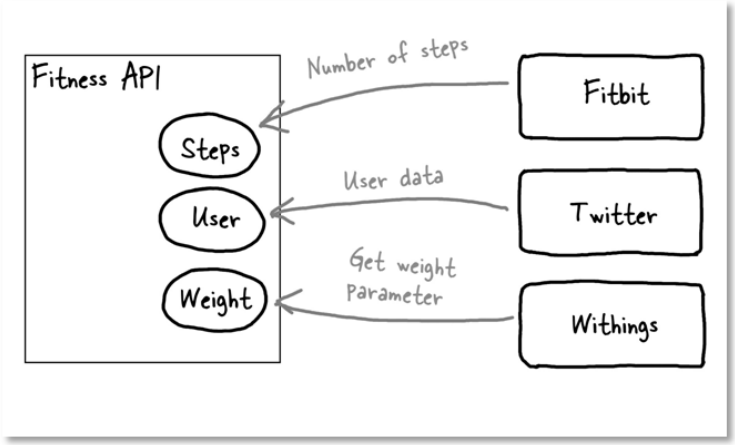


Figure 1.15 – Building an API from other company's schemas can allow you to use the schemas generated by very different API providers to create your own API, while still making it easy for developers who have already implemented these other APIs to integrate with your platform.

A company could choose to use the "user" definition as described by Twitter, along with the "weight" model used by Withings and the "steps" description used by Fitbit to create an application monitoring weight loss – in fact there are several integrations already using these types of information, and normalizing the API interaction would make it significantly easier for developers to create these applications.

This approach has several advantages to consider. Companies considering releasing a new platform can look and see what others in their industry have done. Web APIs with similar purposes can migrate to a compatible structure, easing the learning curve for developers looking to create integrations. Finally, as more and more companies place their schemas into the system, best practices will evolve and newer APIs can present a more consistent interface and improve the developer experience. Without this kind of system, the evolution of APIs into a mature ecosystem will be a long and painful process.

1.5.6 Design Driven Development

When developing a full application, the entire application needs to be deployed at the same time, and new versions need to be as complete as possible, because updating applications puts a burden on the user. On the other end of the spectrum are websites. A website can push out a new version on a daily basis, and broken functionality can generally be fixed very quickly. New features can be added or removed as needed, and users don't need to do

anything in order to take advantage of the newer version (on the other hand, they are rarely given the option to go back to an earlier version if they're not happy with the new one).

APIs live somewhere in the middle of this spectrum. As I mentioned earlier, breaking changes – changes that cause existing implementations to break – are very expensive in terms of time, resources and credibility. On the other hand, new functionality can actually be added within an existing version. While an application would need to "upgrade" in order to grab those new features, a well-designed API can add new functionality or new information within the existing data structure without breaking developers' existing applications.

For this reason, it's possible to create a very targeted API and then build on it. Figure out the most important use case – for instance, mobile – and figure out what the Minimum Viable Product (MVP) – a product which would perform the base functions for the product without any extra bells an whistles - would be for that use case. Perhaps you only want people to be able to look at your product line, or see the activity feed of their friends. Make sure you know what the API will need to do in order to support this use case, and develop your API for that. APIs lend themselves quite well to an agile development process, where short iterations allow for frequent review of changes and additions in order to stay on track.

Frequently, when design isn't determined ahead of time, the resulting product does not meet the use cases of the customers. This can be caught during testing or review of the API, but even when that happens you have to send the product back to development to try again. Nobody – not developers, marketers or management – likes missed deadlines, so making sure that the requirements are well understood helps make sure that the development work is targeted exactly where it should be.

Since you will have created a schema model for your API, it will be easy to make sure that there is documentation for each endpoint, and tests that ensure that the coded product meets the expected behavior. Additionally, the existence of use cases makes it easy to test the integrated product to make sure that those use cases are, in fact, easy. A design-first methodology brings much of the contention to the front of the process and allows for much more streamlined development.

1.5.7 Support your Developers

In a book focused on optimizing the developer experience for your API consumers, I would be remiss not to discuss the support piece of your API. Sometimes APIs are released without a solid support system in place, which can cause a great deal of frustration within your developer community.

Rather than thinking of the developers in your community as interlopers who you have to deal with, consider them valued partners and support them as such. Developer support includes a developer portal that includes documentation, example code and a well-communicated process for finding help. Your documentation should include your use cases, presented as tutorials, your business value and the metrics you're planning to use. Providing this information will help your developer users help you to succeed. The first step to a great

developer experience is the trust they feel when you demonstrate that you're committed to success for the API and for the developers

Engaging developers means performing a lot of the support work up front. While consumers of mainstream products respond well to illustrations and catch phrases, developers want to get started right away. They like building blocks to play with, in the form of example code. When the developer can make an API call in the shortest possible amount of time, it creates engagement and interest, and is invaluable for the success of your platform.

The best portal to study for developer experience is Twitter. They have put a great deal of effort into making sure that they've got clear documentation, excellent tutorials – and in the community there's a huge amount of sample code that developers can leverage in order to write their own applications.

Another great example is Twilio. They have a goal of making it possible for any new developer to be able to make a call to their API in less than 5 minutes. While this is not possible for every API, it is a great goal to strive for, when creating your getting started documentation.

1.6 Summary

This chapter has covered the topic of creating APIs at an extremely high level. At this point you should understand what an API is and what you can do with one. I shared some cautionary tales to explain why a deliberate design process is so critical. The topics covered in this first chapter were:

- What is a Web API? A Web API is distinct from other API systems in that it's designed to decouple the systems from each other, allowing for new and different integrations.
- What can a Web API do? A Web API can add an interface to your system, an integration point that your internal customers, partner developers and external third-party engineers can use to integrate your system with theirs, using well known and mature technology.
- Developer Experience is the most important facet of this process. Focusing on the usability of the platform you're creating will lead to a more successful product.
- Common Pitfalls – Most common pitfalls occur because a part of the first class API process outlined in this chapter is skipped – if there are no use cases, there's no way to check and make sure that the resulting API meets the originally stated goals.

In the following chapter, I'm going to walk you through a "live" API, which you can interact with as a client and see how it works. Advanced developers will have the opportunity to install the system themselves, but everyone will have the chance to see what the interface looks like for a very basic REST Web API.