CHAPTER 2



# API Design Best Practices

The practice of API design is a tricky one. Even when there are so many options out there—tools to use, standards to apply, styles to follow—there is one basic question that needs to be answered and needs to be clear in the developer's mind before any kind design and development can begin…

## What Defines a Good API?

As we all know, the concepts of "good" and "bad" are very subjective (one could probably read a couple of books discussing this on its own), and therefore, opinions vary from one person to another. That being said, years of experience of dealing with different kinds of APIs have left the developer community (and this author) with a pretty good sense of the need-to-have features of any good API. (Disclaimer: Things like clean code, good development practices, and other internal considerations will not be mentioned here, but will be assumed, since they should be part of every software undertaking.)

So let's go over this list.

- *Developer friendly*: The developers working with your API should not suffer when dealing with your system.
- *Extensibility*: Your system should be able to handle the addition of new features without breaking your clients.
- *Up-to-date documentation*: Good documentation is key to your API being picked up by new developers.
- *Proper error handling*: Because things will go wrong and you need to be prepared.
- *Provides multiple SDK/libraries*: The more work you simplify for developers, the more they'll like your system.
- *Security*: A key aspect of any global system.
- *Scalability*: The ability to scale up and down is something any good API should have to properly provide its services.

I'll go over these points one by one and show how they affect the API, and how following the REST style help.

## Developer Friendly

By definition, an API is an *application programming interface*, with the key word being *interface*. When thinking about designing an API that will be used by developers other than yourself, there is a key aspect that needs to be taken into consideration: the Developer eXperience (or DX).


F. Doglio, *Pro REST API Development with Node.js*, DOI 10.1007/978-1-4842-0917-2_2

Even when the API will be used by another system, the integration into that system is first done by one or more developers—human beings that bring the human factor into that integration. This means you'll want the API to be as easy to use as possible, which makes for a great DX, and which should translate into more developers and client applications using the API.

There is a trade-off, though, since simplifying things for humans could lead into an oversimplification of the interface, which in turn could lead to design issues when dealing with complex functionalities.

It is important to consider the DX as one of the major aspects of an API (let's be honest, without developers using it, there is no point to an API), but there are other aspects that have to be taken into consideration and have weight in the design decisions. *Make it simple, but not dummy simple.*

The next sections provide some pointers for a good DX.

## Communication's Protocol

This is one of the most basic aspects of the interface. When choosing a communication protocol, it's always a good idea to go with one that is familiar to the developers using the API. There are several standards that already have libraries and modules available in many programming languages (e.g., HTTP, FTP, SSH, etc.).

A custom-made protocol isn't always a good idea because you'll lose that instant portability in so many existing technologies. That said, if you're ready to create support libraries for the most used languages, and your custom protocol is more efficient for your use case, it could be the right choice.

In the end, it's up to the API designer to evaluate the best solution based on the context he's working in.

In this book, you're working under the assumption that the protocol chosen for REST is HTTP.[1] It's a very well-known protocol; any modern programming language supports it and it's the basis for the entire Internet. You can rest assured that most developers have a basic understanding of how to use it. And if not, there is plenty of information out there to get to know it better.

In summary, there is no silver bullet protocol out there perfect for every scenario. Think about your API needs, make sure that whatever you choose is compatible with REST, and you'll be fine.

## Easy-to-Remember Access Points

The points of contact between all client apps and the API are called *endpoints*. The API needs to provide them to allow clients to access its functionalities. This can be done through whatever communications protocol is chosen. These access points should have mnemotechnic names to help the developer understand their purpose just by reading them.

Of course, the name by itself should never be a replacement for a detailed documentation, but it is normally considered a good idea to reference the resource being used, and to have some kind of indicator of the action being taken when calling that access point.

The following is a good example of a badly named access point (meant to list the books in a bookstore):

```
GET /books/action1
```

This example uses the HTTP protocol to specify the access point, and even though the entity used (books) is being referenced, the action name is not clear; `action1` could mean anything, or even worst, the meaning could change in the future, but the name would still be suitable, so any existing client would undoubtedly break.

A better example—one that follows REST and the standards discussed in Chapter 1—would be this:

```
GET /books
```

[1] See http://www.w3.org/Protocols/rfc2616/rfc2616.html.

This should present the developer with more than enough information to understand that a GET request into the root of a resource (`/books`) will always yield a list of items of this type; then the developer can replicate this pattern into other resources, as long as the interface is kept uniform across all other endpoints.

## Uniform Interface

Easy-to-remember access points are important, but so is being consistent when defining them. Again, you have to go back to the human factor when consuming an API: you have it. So making their lives easier is a must if you want anyone to use it, you can't forget about the DX. That means you need to be consistent when defining endpoints' names, request formats, and response formats. There can be more than one for the latter two (more specifically, the response format is directly tied to the various representations a resource can have), but as long as the default is always the same, there will be no problems.

A good example of an inconsistent interface, even though not on an API, can be seen in the programming language PHP. It has underscore notation on most functions' names, but the underscore is not used on some, so the developer is always forced to go back to the documentation to check how to write these functions (or worst, rely on his/her memory).

For example, `str_replace` is a function that uses an underscore to separate both words (`str` and `replace`), whereas `htmlentities` has no separation of words at all.

Another example of bad design practice in an API is to name the endpoints based on the actions taken instead of the resources handled; for example:

```
/getAllBooks
/submitNewBook
/updateAuthor
/getBooksAuthors
/getNumberOfBooksOnStock
```

These examples clearly show the pattern that this API is following. And at a first glance, they might not seem that bad, but consider how poor the interface is going to become as new features and resources are added to the system (not to mention if the actions are modified). Each new addition to the system causes extra endpoints to the API's interface. The developers of client apps will have no clue as to how these new endpoints are named. For instance, if the API is extended to support the cover images of books, with the current naming scheme, these are all possible new endpoints:

```
/addNewImageToBook
/getBooksImages
/addCoverImage
/listBooksCovers
```

And the list can go on. So for any real-world application, you can safely assume that following this type of pattern will yield a really big list of endpoints, increasing the complexity of both server-side code and client-side code. It will also hurt the system's ability to capture new developers, due to the inherited complexity that it will have over the years.

To solve this problem, and generate an easy-to-use and uniform interface across the entire API, you can apply the REST style to the endpoints. If you remember the constraints proposed by REST from Chapter 1, you end up with a resource-centric interface. And thanks to HTTP, you also have verbs to indicate actions.

Table 2-1 shows how the previous interface changes using REST.

***Table 2-1.*** *List of Endpoints and How They Change When the REST Style Is Applied*

| Old Style | REST Style |
|---|---|
| /getAllBooks | GET /books |
| /submitNewBook | POST /books |
| /updateAuthor | PUT /authors/:id |
| /getBooksAuthors | GET /books/:id/authors |
| /getNumberOfBooksOnStock | GET /books (This number can easily be returned as part of this endpoint.) |
| /addNewImageToBook | PUT /books/:id |
| /getBooksImages | GET /books/:id/images |
| /addCoverImage | POST /books/:id/cover_image |
| /listBooksCovers | GET /books (This information can be returned in this endpoint using subresources.) |

You went from having to remember nine different endpoints to just two, with the added bonus of having all HTTP verbs being the same in all cases once you defined the standard; now there is no need to remember specific roles in each case (they'll always mean the same thing).

## Transport Language

Another aspect of the interface to consider is the *transport language* used. For many years, the de facto standard was XML; it provided a technology-agnostic way of expressing data that could easily be sent between clients and servers. Nowadays, there is a new standard gaining popularity over XML—JSON.

### Why JSON?

JSON has been gaining traction over the past few years (see Figure 2-1) as the standard Data Transfer Format. This is mainly due to the advantages that it provides. The following lists just a few:

- It's lightweight. There is very little data in a JSON file that is not directly related to the information being transferred. This is a major winning point over more verbose formats like XML.[2]
- It's human readable. The format itself is so simple that it can easily be read and written by a human. This is particularly important considering that a focus point of the interface of any good API is the human factor (otherwise known as the DX).
- It supports different data types. Because not everything being transferred is a string, this feature allows the developer to provide extra meaning to the information transferred.

[2]XML is not strictly a Data Transfer Format, but it's being used as one.

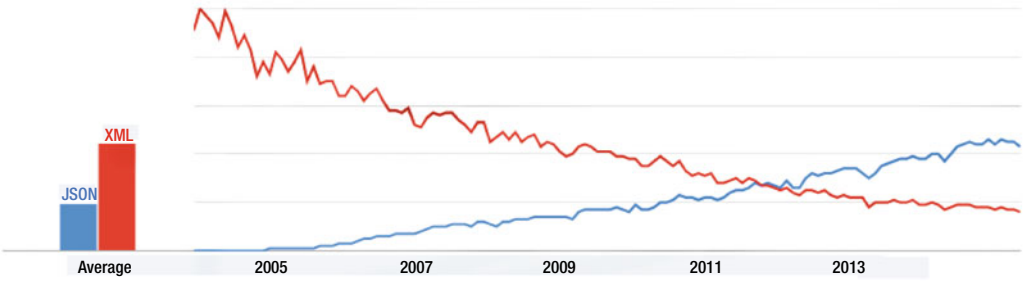


***Figure 2-1.*** *Trend of Google searches for "JSON" vs. "XML" over the last few years*

The list could go on, but these are the three main aspects that are helping JSON win so many followers in the developer community.

Even though JSON is a great format and is gaining traction, it's not the silver bullet that will always solve all of your problems; so it's also important to provide clients with options. And here is where REST comes to help.

Since the protocol you're basing REST on is HTTP, developers can use a mechanism called *content negotiation* to allow clients to specify which of the supported formats they want to receive (as discussed in Chapter 1). This allows for more flexibility on the API, and still keeps the interface uniform.

Going back to the list of endpoints, the last one talks about using a subresource as the solution. That can be interpreted in several ways, because not only is the language used to transfer the data important, but so is the structure that you give the data being transferred. My final advice for a uniform interface is to standardize the format used, or even better, follow an existing one, like HAL.

This was covered in the previous chapter, so refer back to it for more information.

# Extensibility

A good API is never fully finished. This might be a bold claim, but it's one that comes from the experience of the community. Let's look at some of the big ones.[3]

- Google APIs – 5 billion calls a day[4]; launched in 2005
- Facebook APIs – 5 billion calls a day[5]; launched in 2007
- Twitter APIs – 13 billion calls a day[6]; launched in 2006

These examples show that even when a great team is behind the API, the APIs will keep growing and changing because the client apps developers find new ways to use it, the business model of the API owner changes over time, or simply because features are added and removed.

When any of this happens, the API may need to be extended or changed, and new access points added or old ones changed. If the original design is right, then going from v1 to v2 should be no problem, but if it's not, then that migration could spell disaster for everyone.

---

[3] Source: http://www.slideshare.net/3scale/apis-for-biz-dev-20-which-business-model-15473323.
[4] April 2010.
[5] October 2009.
[6] May 2011.

## How Is Extensibility Managed?

When extending the API, you're basically releasing a new version of your software, so the first thing you need to do is let your users (the developers) know what will happen once the new version is out. Will their apps still work? Are the changes backward-compatible? Will you maintain several versions of your API online, or just the latest one?

A good API should take the following points into consideration:

- How easily can new endpoints be added?
- Is the new version backward-compatible?
- Can clients continue to work with older versions of the API while their code is being updated?
- What will happen to existing clients targeting the new API?
- How easy will it be for clients to target the new version of the API?

Once all these points are settled, then you can safely grow and extend the API .

Normally, going from version A to version B of an API by instantly deprecating version A and taking it offline in favor of version B is considered a bad move, unless you have very few client applications using that version, of course.

A better approach for this type of situation is to allow developers to choose which version of the API they want to use, keeping the old version long enough to let everyone migrate into the newer one. And in order to do this, an API would include its version number in the resource identifier (i.e., the URL of each resource). This approach makes the version number a mandatory part of the URL to clearly show the version in use.

Another approach, which may not be as clear, is to provide a versionless URL that points to the latest version of the API, and an optional URL parameter to overwrite the version. Both approaches have pros and cons that have to be weighted by the developer creating the API.

***Table 2-2.*** *Pros and Cons of Having the Version of the API As Part of the URL*

| Pros | Cons |
|---|---|
| The version number is clearly visible, helping avoid confusion about the version being used. | URLs are more verbose. |
| Easy to migrate from one version to another, from a client perspective (all URLs change the same portion—the version number ) | A wrong implementation on the API code could cause a huge amount of work when migrating from one version to the other (i.e., if the version is hardcoded on the endpoint's URL template, individually for every endpoint). |
| Allows cleaner architecture when more than one version of the API needs to be kept working. | |
| Clear and simple migration from one version to the next from the API perspective, since both versions could be kept working in parallel for a period of time, allowing slower clients to migrate without breaking. | |
| The right versioning scheme can make fixes and backward-compatible new features instantly available without the need to update on the client's part. | |

***Table 2-3.*** *Pros and Cons of Having the API Version Hidden from the User*

| Pros | Cons |
|---|---|
| Simpler URLs. | A Hidden version number might lead to confusion about the version being used. |
| Instant migration to latest working code of the API. | Non-backward-compatible changes will break the clients that are not referencing a specific version of the API. |
| Simple migration from one version to the next from the client's perspective (only change the value of the attribute). | Complex architecture required to make version selection available. |
| Easy test of client code against the latest version (just don't send version-specific parameters). | |

Keeping this in mind, there are several versioning schemes to use when it comes to setting the version of a software product:

Ubuntu's[7] version numbers represent the year and month of the release; so version 14.04 means it was released in April 2014.

In the Chromium project, version numbers have four parts[8]: MAJOR.MINOR.BUILD.PATCH. The following is from the Chromium project's page on versioning: MAJOR and MINOR *may* get updated with any significant Google Chrome release (Beta or Stable update). MAJOR *must* get updated for any backward-incompatible user data change (since this data survives updates). BUILD *must* get updated whenever a release candidate is built from the current trunk (at least weekly for Dev channel release candidates). The BUILD number is an ever-increasing number representing a point in time of the Chromium trunk. PATCH *must* get updated whenever a release candidate is built from the BUILD branch.

Another intermediate approach, known as Semantic Versioning or SemVer,[9] is well accepted by the community. It provides the right amount of information. It has three numbers for each version: MAJOR.MINOR.PATCH.

- MAJOR represents changes that are not backward-compatible.
- MINOR represents new features that leave the API backward-compatible.
- PATCH represents small changes like bug fixes and code optimization.

With that scheme, the first number is the only one that is really relevant to clients, since that'll be the one indicating compatibility with their current version.

By having the latest version of MINOR and PATCH deployed on the API at all times, you're providing clients with the latest compatible features and bug fixes, without making clients update their code.

So with that simple versioning scheme, the endpoints look like this:

```
GET /1/books?limit=10&size=10
POST /v2/photos
GET /books?v=1
```

[7] See https://help.ubuntu.com/community/CommonQuestions#Ubuntu_Releases_and_Version_Numbers.
[8] See http://www.chromium.org/developers/version-numbers.
[9] See semver.org.

When choosing a versioning scheme, please take the following into consideration:

- Using the wrong versioning scheme might cause confusion or problems when implementing a client app, by consuming the wrong version of the API. For instance, using Ubuntu's versioning scheme for your API might not be the best way to communicate what is going on in each new version.
- The wrong versioning scheme might force clients to update a lot, like when a minor fix is deployed or a new backward-compatible feature is added. Those changes shouldn't require a client update. So don't force the client to specify those parts of the version unless your scheme requires it.

# Up-to-Date Documentation

No matter how mnemotechnic your endpoints are, you still need to have documentation explaining everything that your API does. Whether optional parameters or the mechanics of an access point, the documentation is fundamental to having a good DX, which translates into more users.

A good API requires more than just a few lines explaining how to use an access point (there is nothing worse than discovering that you need an access point but it has no documentation at all), but needs a full list of parameters and explanatory examples.

Some providers give developers a simple web interface to try their API without having to write any code. This is particularly useful for newcomers.

There are some online services that allow API developers to upload their documentation, as well as those that provide the web UI to test the API; for example, Mashape provides this service for free (see Figure 2-2).

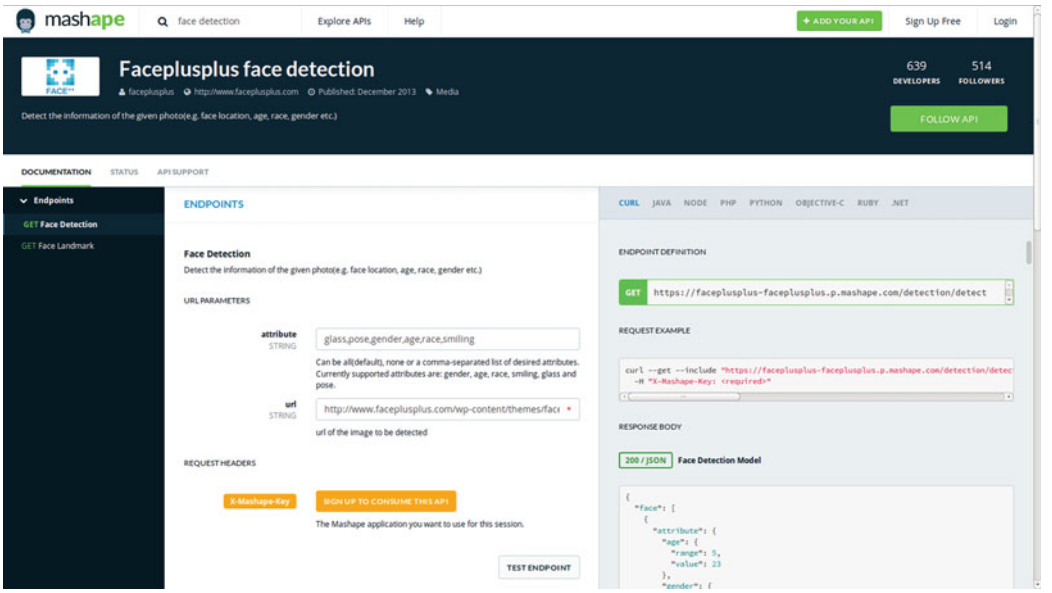


***Figure 2-2.*** *The service provided by Mashape*

Another good example of detailed documentation is at Facebook's developer site.[10] It provides implementation and usage examples for all the platforms that Facebook supports (see Figure 2-3).

[10] See https://developers.facebook.com/docs/graph-api/using-graph-api/v2.1.

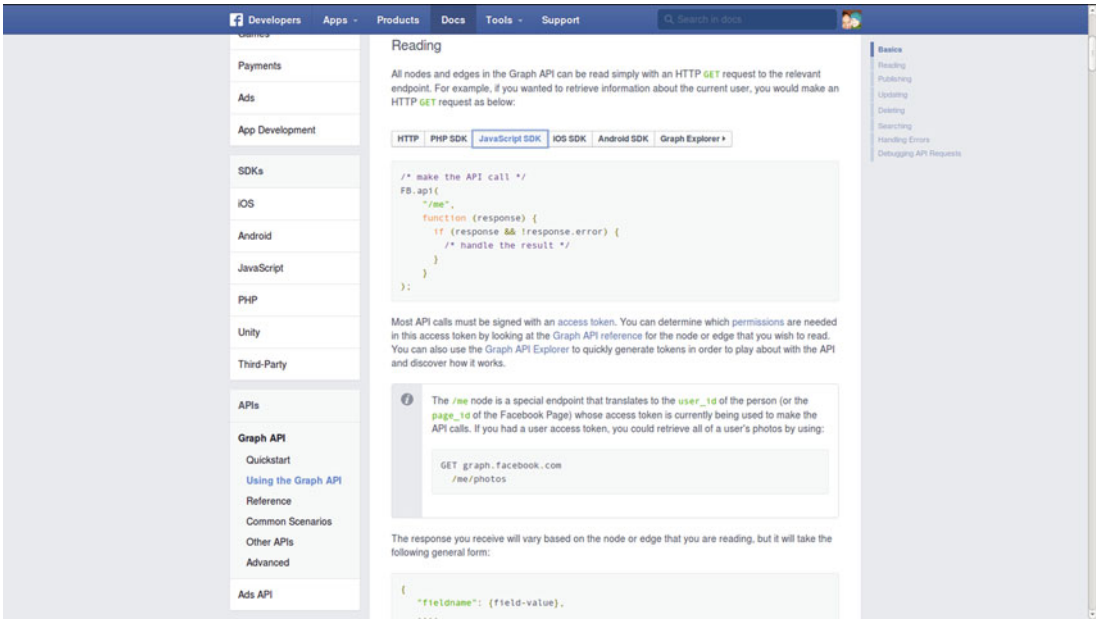


***Figure 2-3.*** *Facebook's API documentation site*

An example of a poorly written documentation is seen in Figure 2-4. It is 4chan's API documentation.[11]

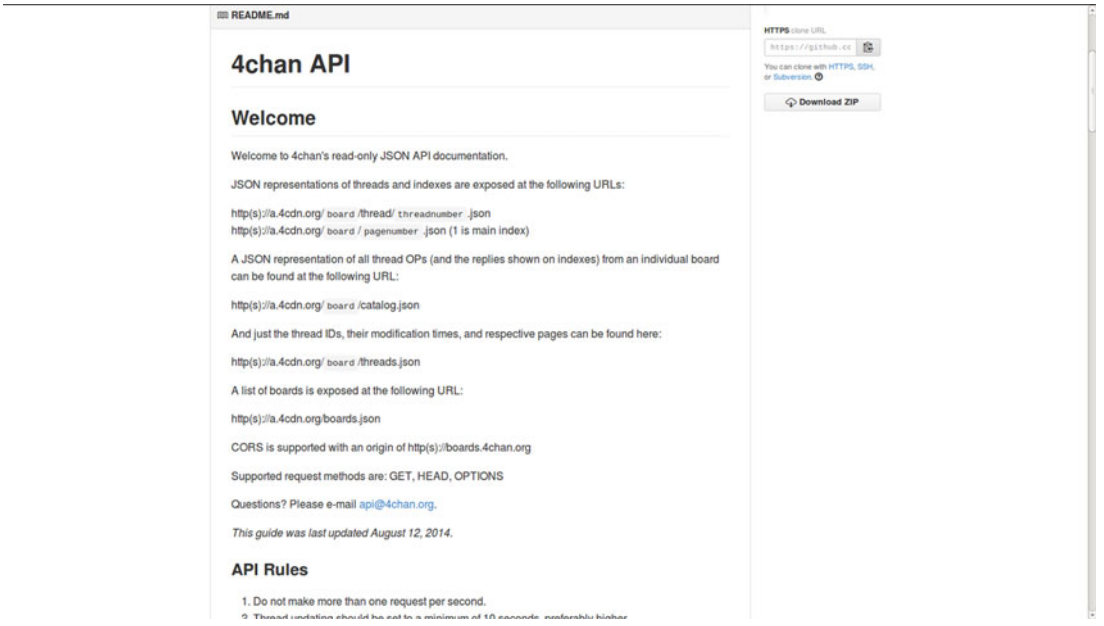


***Figure 2-4.*** *Introduction to 4chan's API documentation*

[11] See https://github.com/4chan/4chan-API.

Yes, the API appears to not be complicated enough to merit writing a whole book about it, but then again, there are no examples provided, only a generic explanation of how to find the endpoints and what parameters to use.

Newcomers might find it hard to understand how to implement a simple client that uses this API.

---

■ **Note** It's unfair to compare 4chan's documentation to that of Facebook's, since the size of the teams and companies are completely different. But you should note the lack of quality in 4chan's documentation.

---

Although it might not seem like the most productive idea while developing an API, the team needs to consider working on extensive documentation. It is one of the main things that will assure the success or failure of the API for two main reasons:

- It should help newcomers and advance developers to consume your API without any problems.
- It should serve as a blueprint for the development team, if it is kept up-to-date. Jumping into a project mid-development is easier if there is a well-written and well-explained blueprint of how the API is meant to work.

---

■ **Note** This also applies to updating the documentation when changes are made to the API. You need to keep it updated; otherwise, the effect is the same as not having documentation at all.

---

# Proper Error Handling

Error handling on an API is incredibly important, because if it is done right, it can help the client app understand how to handle errors; and on the human side of it (the DX), it can help developers understand what it is they're doing wrong and how to fix it.

There are two very distinct moments during the life cycle of an API client that you need to consider error handling:

- *Phase 1*: The development of the client.
- *Phase 2*: The client is implemented and being used by end users.

## Phase 1: Development of the Client

During the first phase, developers implement the required code to consume the API. It is very likely that a developer will have errors on the requests (things like missing parameters, wrong endpoint names, etc.) during this stage.

Those errors need to be handled properly, which means returning enough information to let developers know what they did wrong and how they can fix it.

A common problem with some systems is that their creators ignore this stage, and when there is a problem with the request, the API crashes, and the returned information is just an error message with the stack trace and the status code 500.

The response in Figure 2-5 shows what happens when you forget to add error handling in the client development stage. The stack trace returned might give the developer some sort of clue (at best) as to what exactly went wrong, but it also shows a lot of unnecessary information, so it ends up being confusing. This certainly hurts development time, and no doubt would be a major point against the DX of the API.


***Figure 2-5.*** *A classic example of a crash on the API returning the stack trace*

On the other hand, let's take a look at a proper error response for the same error in Figure 2-6.


***Figure 2-6.*** *A proper error response would look like this*

Figure 2-6 clearly shows that there has been an error, what the error is, and an error code. The response only has three attributes, but they're all helpful:

- The error indicator gives the developer a clear way to check whether or not the response is an error message (you could also check against the status code of the response).
- The error message is clearly intended for the developer, and not only states what's missing, but also explains how to fix it.
- A custom error code, if explained in the documentation, could help a developer automate actions when this type of response happens again.

### Phase 2: The Client Is Implemented and Being Used by End Users

During this stage in the life cycle of the client, you're not expecting any more developer errors, such as using the wrong endpoint, missing parameters, and the like, but there could still be problems caused by the data generated by the user.

Client applications that request some kind of input from the user are always subject to errors on the user's part, and even though there are always ways to validate that input before it reaches the API layer, it's not safe to assume all clients will do that. So the safest bet for any API designer and developer is to assume there is no validation done by the client, and anything that could go wrong, will go wrong with the data. This is also a safe assumption to make from a security point of view, so it's providing a minor security improvement as a side effect.

With that mindset, the API implemented should be rock-solid and able to handle any type of errors in the input data.

The response should mimic the one from phase 1: there should be an error indicator, an error message stating what's wrong (and if possible, how to fix it), and a custom error code. The custom error code is especially useful in this stage, since it'll provide the client with the ability to customize the error shown to the end user (even showing a different but still relevant error message).

## Multiple SDK/Libraries

If you expect your API to be massively used across different technologies and platforms, it might be a good idea to develop and provide support for libraries and SDKs that can be used with your system.

By doing so, you provide developers with the means to consume your services, so all they have to do is use them to create their client apps. Essentially, you're shaving off potential weeks or months (depending on the size of your system) of development time.

Another benefit is that most developers will inherently trust your libraries over others that do the same, because you're the owner of the service those libraries are consuming.

Finally, consider open sourcing the code of your libraries. These days, the open source community is thriving. Developers will undoubtedly help maintain and improve your libraries if they're of use to them.

Let's look again at some of the biggest APIs out there:

- Facebook API provides SDKs for iOS, Android, JavaScript, PHP, and Unity.[12]
- Google Maps API provides SDKs for several technologies, including iOS, the Web, and Android.[13]

[12] See https://developers.facebook.com (see the bottom of the page for the list of SDKs).
[13] See https://developers.google.com/maps/.

- Twitter API provides SDKs for several of their APIs, including Java, ASP, C++, Clojure, .NET, Go, JavaScript, and a lot of other languages.[14]
- Amazon provides SDKs for their AWS service, including PHP, Ruby, .NET, and iOS. They even have those SDKs on GitHub for anyone to see.[15]

# Security

Securing your API is a very important step in the development process and it should not be ignored, unless what you're building is small enough and has no sensitive data to merit the effort.

There are two big security issues to deal with when designing an API:

- *Authentication*: Who's going to access the API?
- *Authorization*: What will they be able to access once logged in?

Authentication deals with letting valid users access the features provided by the API. Authorization deals with handling what those authenticated users can actually do inside the system.

Before going into details about each specific issue, there are some common aspects that need to be remembered when dealing with security on RESTful systems (at least, those based on HTTP):

- *RESTful systems are meant to be stateless*. Remember that REST defines the server as stateless, which means that storing the user data in session after the initial login is not a good idea (if you want to stay within the guidelines provided by REST that is).
- *Remember to use HTTPS*. On RESTful systems based on HTTP, HTTPS should be used to assure encryption of the channel, making it harder to capture and read data traffic (man-in-the-middle attack).

## Accessing the System

There are some widely used authentication schemes out there meant to provide different levels of security when signing users into a system. Some of the most commonly known ones are Basic Auth with TSL, Digest Auth, OAuth 1.0a, and OAuth 2.0.

I'll go over these and talk about each of their pros and cons. I'll also cover an alternative method that should prove to be the most RESTful, in the sense that it's 100% stateless.

### Almost Stateless Methods

OAuth 1.0a, OAuth 2.0, Digest Auth, and Basic Auth + TSL are the go-to methods of authentication these days. They work wonderfully, they have been implemented in all of the modern programming languages, and they have proven to be the right choice for the job (when used for the right use-case). That being said, as you're about to see, none of them are 100% stateless.

They all depend on having the user have information stored on some kind of cache layer on the server side. This little detail, especially for the purists out there, means a no-go when designing a RESTful system, because it goes against one of the most basic of the constraints imposed by REST: *Communication between client and server must be stateless.*

[14] See https://dev.twitter.com/overview/api/twitter-libraries.
[15] See https://github.com/aws.

This means the state of the user should not be stored anywhere.

You will look the other way in this particular case, however. I'll cover the basics of each method anyway, because in real life, you have to compromise and you have to find a balance between purism and practicality. But don't worry. I'll go over an alternative design that will solve authentication and stay true to REST.

## Basic Auth with TSL

Thanks to the fact that you're basing REST on HTTP for the purpose of this book, the latter provides a basic authentication method that most of the languages have support for.

Keep in mind, though, that this method is aptly named, since it's quite basic and works by sending the username and password unencrypted over HTTP. So the only way to make it secure is to use it with a secured connection over HTTPS (HTTP + TSL).

This authentication method works as follows (see Figure 2-7):

1. First, a client makes a request for a resource without any special header.
2. The server responds with a 401 unauthorized response, and within it, a WWW-Authenticate header, specifying the method to use (Basic or Digest) and the realm name.
3. The client then sends the same request, but adds the Authorization header, with the string USERNAME:PASSWORD encoded in base 64.

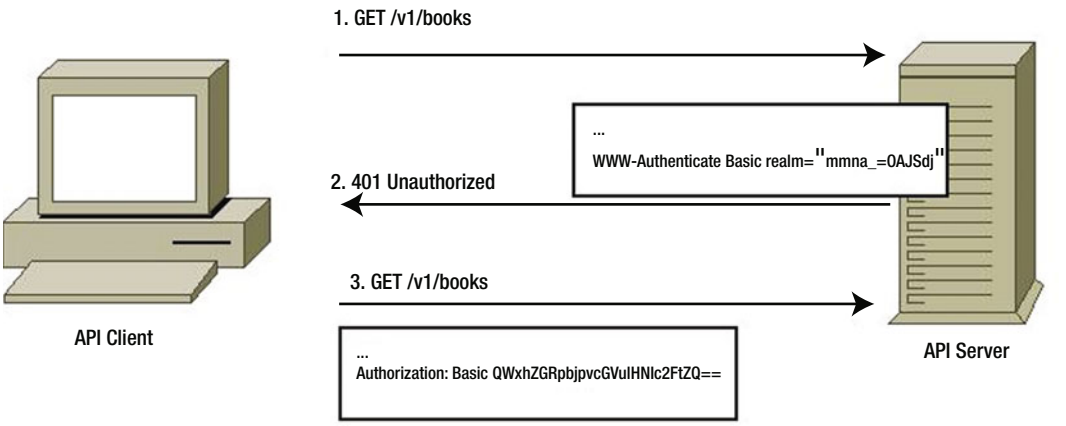


***Figure 2-7.*** *The steps between client and server on Basic Auth*

On the server side, there needs to be some code to decode the authentication string and load the user data from the session storage used (normally a database).

Aside from the fact that this approach is one of the many that will break the nonstateless constraint, it's easy and fast to implement.

---

■ **Note** When using this method, if the password for a logged in user is reset, then the login data sent on the request becomes old and the current session is terminated.

---

## Digest Auth

This method is an improvement over the previous one, in the sense that it adds an extra layer of security by encrypting the login information. The communication with the server works the same way, by sending the same headers back and forth.

With this methodology, upon receiving a request for a protected resource, the server will respond with a WWW-Authenticate header and some specific parameters. Here are some of the most interesting:

- *Nounce*: A uniquely generated string. This string needs to be unique on every 401 response.
- *Opaque*: A string returned by the server that has to be sent back by the client unaltered.
- *Qop*: Even though optional, this parameter should be sent to specify the quality of protection needed (more than one token can be sent in this value). Sending `auth` back would imply a simple authentication, whereas sending `auth-int` implies authentication with integrity check.
- *Algorithm*: This string specifies the algorithm used to calculate the checksum response form the client. If not present, then MD5 should be assumed.

For the full list of parameters and implementation details, please refer to the RFC.[16] Here is a list of some of the most interesting ones:

- *Username*: The unencrypted username.
- *URI*: The URI you're trying to access.
- *Response*: The encrypted portion of the response. This proves that you are who you say you are.
- *Qop*: If present, it should be one of the supported values sent by the server.

To calculate the response, the following logic needs to be applied:

```
MD5(HA1:STRING:HA2)
```

Those values for HA1 are calculated as follows:

- If no algorithm is specified on the response, then `MD5(username:realm:password)` should be used.
- If the algorithm is MD5-less, then it should be `MD5(MD5(username:realm:password):nonce:cnonce)`

Those values for HA2 are calculated as follows:

- If `qop` is `auth`, then `MD5(method:digestURI)` should be used.
- If `qop` is `auth-int`, then `MD5(method:digestURI:MD5(entityBody))`

Finally, the response will be as follows:

```
MD5(HA1:nonce:nonceCount:clientNonce:HA2) //for the case when "qop" is "auth" or "auth-int"
MD5(HA1:nonce:HA2) //when "qop" is unspecified.
```

[16] See https://www.ietf.org/rfc/rfc2617.txt.

The main issue with this method is that the encryption used is based on MD5, and in 2004 it was proven that this algorithm is not collision resistant, which basically means a man-in-the-middle attack would make it possible for an attacker to get the necessary information and generate a set of valid credentials.

A possible improvement over this method, just like with its "Basic" brother, would be adding TSL; this would definitely help make it more secure.

## OAuth 1.0a

OAuth 1.0a is the most secure of the four nonstateless methodologies described in this section. The process is a bit more tedious than the ones described earlier (see Figure 2-8), but the trade-off here is a significantly increased level of security.

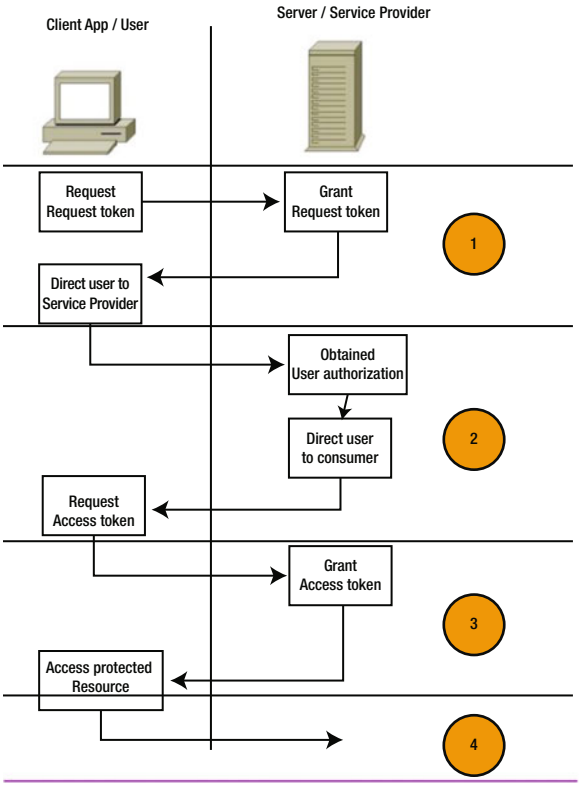


***Figure 2-8.*** *The interaction between client and server*

In this case, the service provider has to allow the developer of the client app to register the app on the provider's web site. By doing so, the developer obtains a consumer key (a unique identifying key for his application) and the consumer secret. Once that process is done, the following steps are required:

- The client app needs a request token. The purpose is to receive the user's approval and then request an access token. To get the request token, a specific URL must be provided by the server; in this step, the consumer key and the consumer secret are used.

- Once the request token is obtained, the client must make a request using the token on a specific server URL (i.e., `http://provider.com/oauth/authorize`) to get authorization from the end user.
- After authorization from the user is given, then the client app makes a request to the provider for an access token and a token secret key.
- Once the access token and secret token are obtained, the client app is able to request protected resources for the provider on behalf of the user by signing each request.

For more details on how this method works, please refer to the complete documentation.[17]

## OAuth 2.0

OAuth 2.0 is meant to be the evolution of OAuth 1.0a; it focuses on client developer simplicity. The main problem with implementations of systems that worked with OAuth 1.0 was the complexity implied in the last step: signing every request.

Due to its complexity, the last step is the key weak point of the algorithm: if either the client or server makes a tiny mistake, then the requests will not validate. Even when the same aspect made it the only methodology that didn't need to work on top of SSL (or TSL), this benefit wasn't enough.

OAuth 2.0 tries to simplify the last step by making some key changes, mainly:

- It relies on SSL (or TSL) to ensure that the information sent back and forth is encrypted.
- Signatures are not required for requests after the token has been generated.

To summarize, this version of OAuth tries to simplify the complexity introduced by OAuth 1.0, while sacrificing security at the same time (by relying on TSL to ensure data encryption). It is the preferred method over OAuth 1.0 if the devices you're dealing with have support for TSL (computers, mobile devices, etc.); otherwise, you might want to consider using other options.

## A Stateless Alternative

As you've seen, the alternatives you have when it comes to implementing a security protocol to allow users to sign into a RESTful API are not stateless, and even though you should be prepared to make that commitment in order to gain the benefits of tried and tested ways of securing your application, there is a fully REST compatible way of doing it as well.

If you go back to Chapter 1, the stateless constraints basically imply that any and all states of the communication between client and server should be included on every request made by the client. This of course includes the user information, so if you want to have stateless authentication, you need to include that in your requests as well.

If you want to ensure the authenticity of each request, you can borrow the signature step of OAuth 1.0a and apply it on every request by using a pre-established secret key between the client and the server, and a MAC (Message Authentication Code) algorithm to do the signing (see Figure 2-9).

[17] See `http://oauth.net/core/1.0a/`.

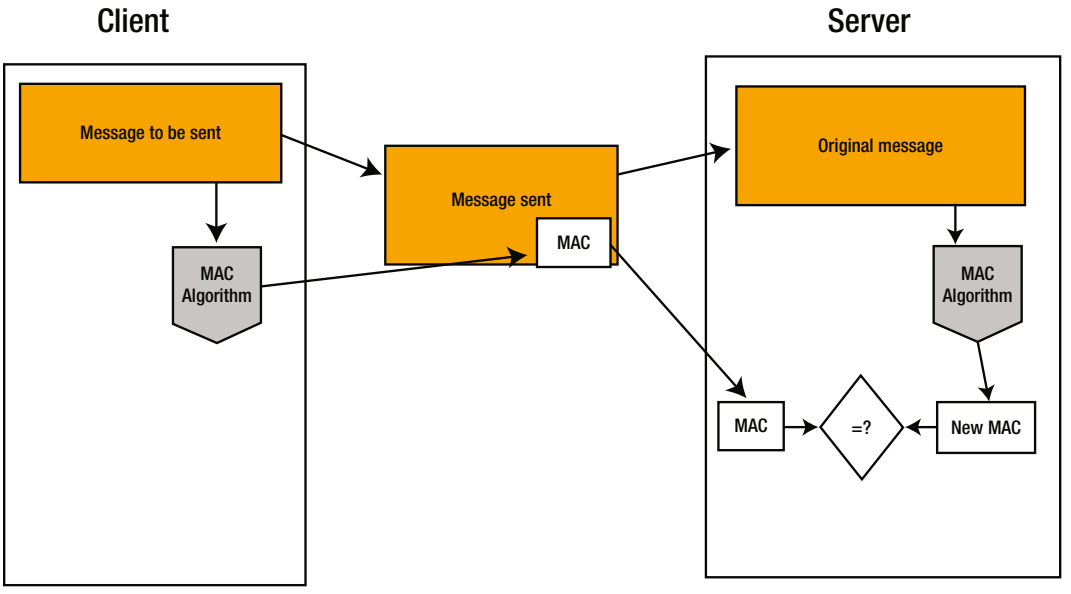


***Figure 2-9.*** *How the MAC signing process works*

As you're keeping it stateless, the information required to generate the MAC needs to also be sent as part of the request, so the server can re-create the result and corroborate its validity.

This approach has some clear advantages in our case, mainly:

- It's simpler than both OAuth 1.0a and OAuth 2.0.
- Zero storage is needed, since any and all required information to validate the encryption needs to be sent on every request.

# Scalability

Last but certainly not least is *scalability*.

Scalability is usually an underestimated aspect of API design, mainly because it's quite difficult to fully understand and predict the reach one API will have before it launches. It might be easier to estimate this if the team has previous experience with similar projects (e.g., Google has probably gotten quite good at calculating their scalability for new APIs before launch day), but if it's their first one, then it might not be as easy.

A good API should be able to scale, that means, it should be able to handle as much traffic as it gets without compromising its performance. But it also means it should not spend resources if they're not needed. This is not only a reflection of the hardware that the API resides on (although that is an important aspect) it's also a reflection of the underlying architecture of that API.

Over the years, the classic monolithic design in software architecture has been migrating into a fully distributed one, so splitting the API into different modules that interact with each other makes sense.

This provides the flexibility needed to not only scale up or down the resources that are affected, but to also provide fault tolerance, help developers maintain cleaner code bases amongst other advantages.

The following image (Figure 2-10) shows a standard monolithic design, having your app inside one server, living like one single entity.

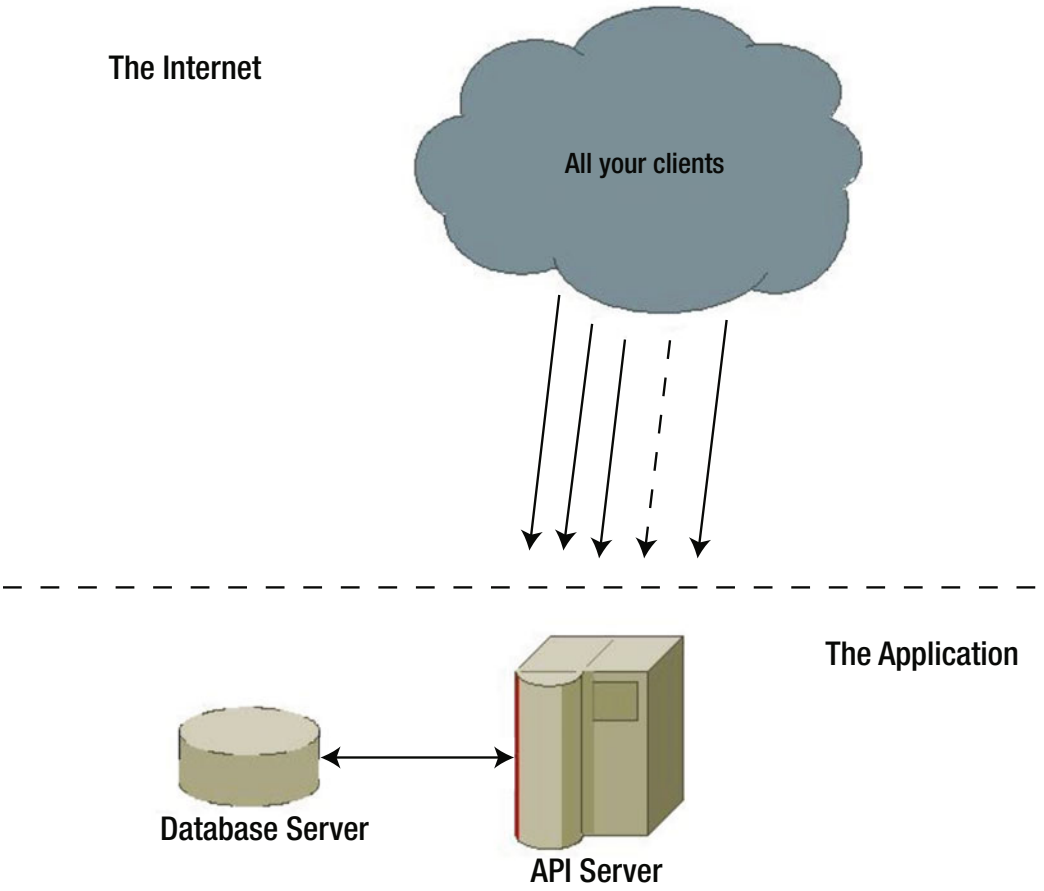


***Figure 2-10.*** *Simple diagram of a monolithic architecture*

In Figure 2-11 you see a distributed design, if compared with the previous one, you can see where the advantages come from (better resource usage, fault tolerance, easier to scale up or down, and so on).

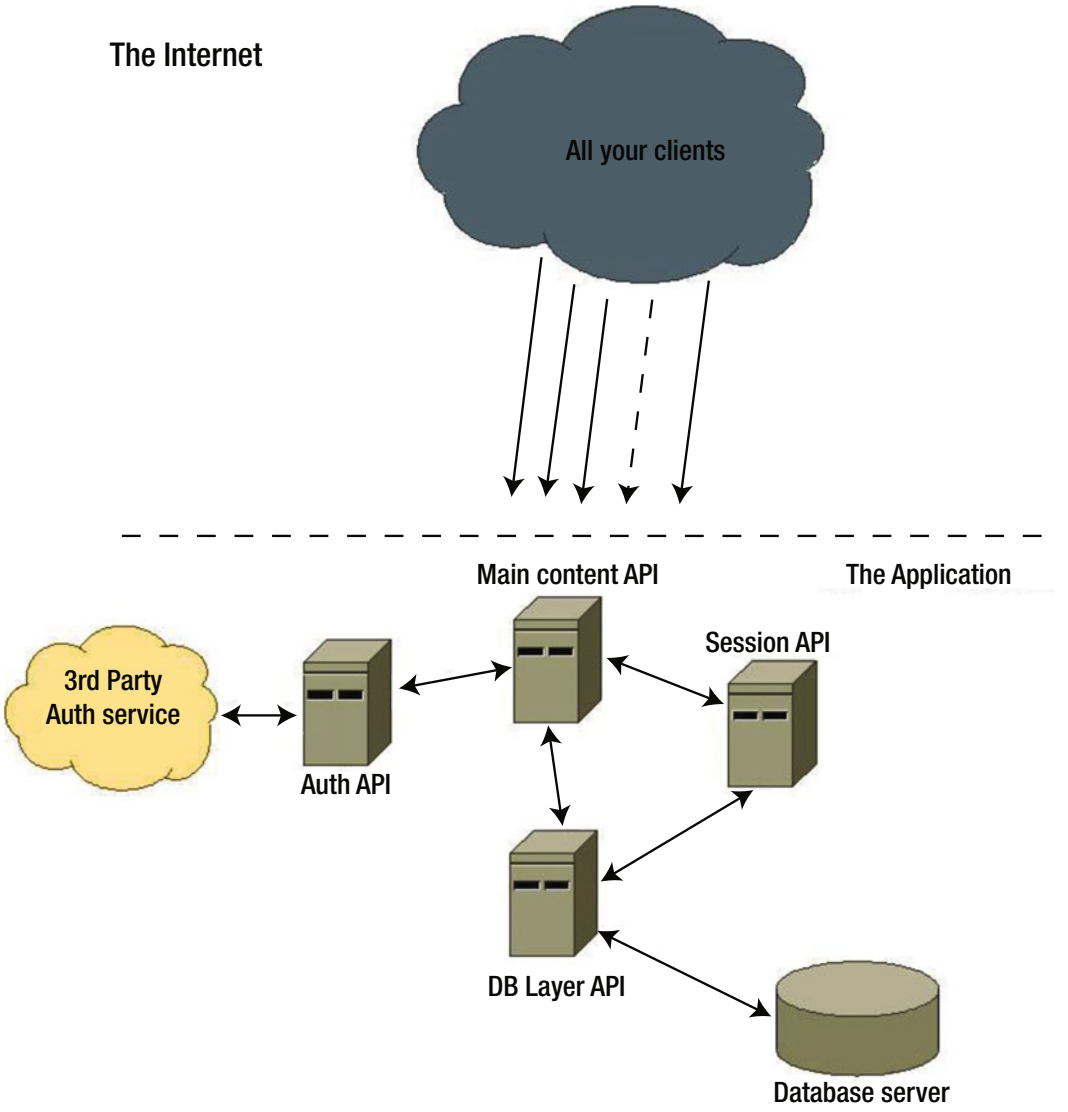


***Figure 2-11.*** *A diagram showing an example of a distributed architecture.*

Achieving a distributed architecture to ensure scalability using REST is quite simple. Fielding's paper proposes a distributed system based on a client-server scheme.

So splitting the entire system into a set of smaller APIs, having them talk to each other when required will ensure the advantages mentioned earlier.

For instance, let's look at an internal system for a bookstore (Table 2-4), the main entities would be:

***Table 2-4.*** *List of Entities and Their Role Inside the System*

| Entity | Description |
|---|---|
| Books | Represents the inventory of the store. It'll control everything from book data, to number of copies, and so forth. |
| Clients | Contact information of clients. |
| Users | Internal bookstore users, they will have access to the system. |
| Purchases | Records information about book sales. |

Now, consider that system on a small bookstore, one that is just starting and has just a few employees. It's very tempting to go with a monolithic design, not a lot of resources will be spent and the design is quite simple.

Now, consider what would happen if the small bookstore suddenly grows so much that it expands into several other bookstores, they go from having one store, to 100, employee numbers grow, books need better tracking, purchases sky rocket.

The simple system from before will not be enough to handle such growth. It would require changes to support networking, centralized data storage, distributed access, better storage capacity, and so forth. In other words scaling it up would be too expensive and probably it would require a complete rewrite.

Finally, consider an alternative beginning, what if you took the time to create the first system using a distributed architecture based on REST? Having each sub-system be a different API and having them talk to each other.

Then you would've been able to scale the whole thing much easier, working independently on each sub-system there would be no need for full rewrites and the system could potentially keep growing to meet new needs.

# Summary

This chapter covered what the developer community considers a "good API," which means the following:

- Remembering the Developer eXperience (DX).
- Being able to grow and improve without breaking existing clients.
- Having up-to-date documentation.
- Providing correct error handling.
- Providing multiple SDK and libraries.
- Thinking about security.
- Being able to scale, both up and down, as needed.

In the next chapter, you'll learn why Node.js is a perfect match for implementing everything you've learned in this chapter.