An Introduction to APIs

By Brian Cooksey

ZAPIER.COM/LEARN

zapier

An Introduction to APIs

by Brian Cooksey

Edited by Bryan Landers and Danny Schreiber.
Cover art by Stephanie Briones.
Clipart images from *FCIT*.

Originally published April 23, 2014

Zapier, Inc.

This book is available for free at *zapier.com/learn/apis*, with interactive exercises and additional resources.

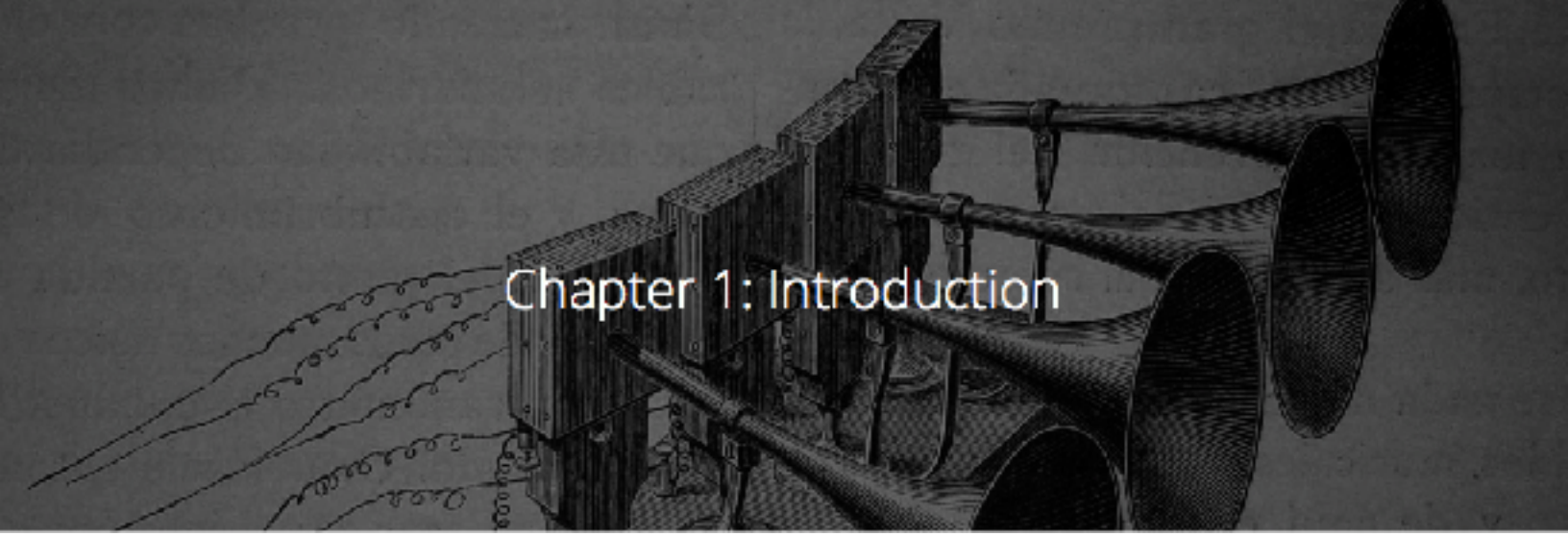
Have you ever wondered how Facebook is able to automatically display your Instagram photos? How about how Evernote syncs notes between your computer and smartphone? If so, then it’s time to get excited!

In this course, we walk you through what it takes for companies to link their systems together. We start off easy, defining some of the tech lingo you may have heard before, but didn’t fully understand. From there, each lesson introduces something new, slowly building up to the point where you are confident about what an API is and, for the brave, could actually take a stab at using one.

Who Is This Book For?

If you are a non-technical person, you should feel right at home with the lesson structure. For software developers, the first lesson or two may feel like a mandatory new employee orientation, but stick with it – you'll get your fill of useful information, too.

Table of Contents



Chapter 1: Introduction

APIs (application programming interfaces) are a big part of the web. In 2013 there were over 10,000 APIs published by companies for open consumption [1]. That is quadruple the number available in 2010 [2].

With so many companies investing in this new area of business, possessing a working understanding of APIs becomes increasingly relevant to careers in the software industry. Through this course, we hope to give you that knowledge by building up from the very basics. In this chapter, we start by looking at some fundamental concepts around APIs. We define what an API is, where it lives, and give a high level picture of how one is used.

A Frame of Reference

When talking about APIs, a lot of the conversation focuses on abstract concepts. To anchor ourselves, let's start with something that is physical: the server. A server is nothing more than a big computer. It has all the same parts as the laptop or desktop you use for work, it’s just faster and more powerful. Typically, servers don't have a monitor, keyboard, or mouse, which makes them look unapproachable. The reality is that IT folks connect to them remotely — think remote desktop-style — to work on them.

Servers are used for all sorts of things. Some store data; others send email. The kind people interact with the most are web servers. These are the servers that give you a web page when you visit a website. To better understand how that works, here's a simple analogy:

In the same way that a program like Solitaire waits for you to click on a card to do something, a web server runs a program that waits for a person to ask it for a web page.

There's really nothing magical or spectacular about it. A software developer writes a program, copies it to a server, and the server runs the program continuously.

What An API Is and Why It's Valuable

Websites are designed to cater to people's strengths. Humans have an incredible ability to take visual information, combine it with our experiences to derive meaning, and then act on that meaning. It's why you can look at a form on a website and know that the little box with the phrase "First Name" above it means you are supposed to type in the word you use to informally identify yourself.

Yet, what happens when you face a very time-intensive task, like copying the contact info for a thousand customers from one site to another? You would love to delegate this work to a computer so it can be done quickly and accurately. Unfortunately, the characteristics that make websites optimal for humans make them difficult for computers to use.

The solution is an API. An API is the tool that makes a website's data digestible for a computer. Through it, a computer can view and edit data, just like a person can by loading pages and submitting forms.

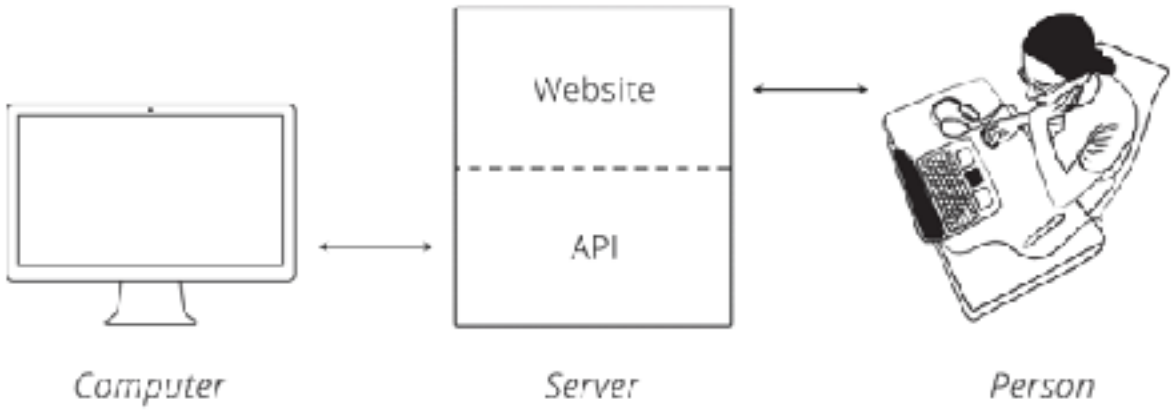


Figure 1. *Communicating with a server.*

Making data easier to work with is good because it means people can write software to automate tedious and labor-intensive tasks. What might take a human hours to accomplish can take a computer seconds through an API.

How An API Is Used

When two systems (websites, desktops, smartphones) link up through an API, we say they are "integrated." In an integration, you have two sides, each with a special name. One side we have already talked about: the server. This is the side that actually provides the API. It helps to remember that the API is simply another program running on the server [3]. It may be part of the same program that handles web traffic, or it can be a completely separate one. In either case, it is sitting, waiting for others to ask it for data.

The other side is the "client." This is a separate program that knows what data is available through the API and can manipulate it, typically at the request of a user. A great example is a smartphone app that syncs with a website. When you push the refresh button your app, it talks to a server via an API and fetches the newest info.

The same principle applies to websites that are integrated. When one site pulls in data from the other, the site providing the data is acting as the server, and the site fetching the data is the client.

Chapter 1 Recap

This chapter focused on providing some foundational terminology and a mental model of what an API is and how it is used.

The key terms we learned were:

- **Server:** A powerful computer that runs an API
- **API:** The "hidden" portion of a website that is meant for computer consumption
- **Client:** A program that exchanges data with a server through an API

Homework

Normally, each chapter has a mini homework assignment where you apply what you learned. Today, however, you get a pass. Go enjoy your favorite TV show!

Next

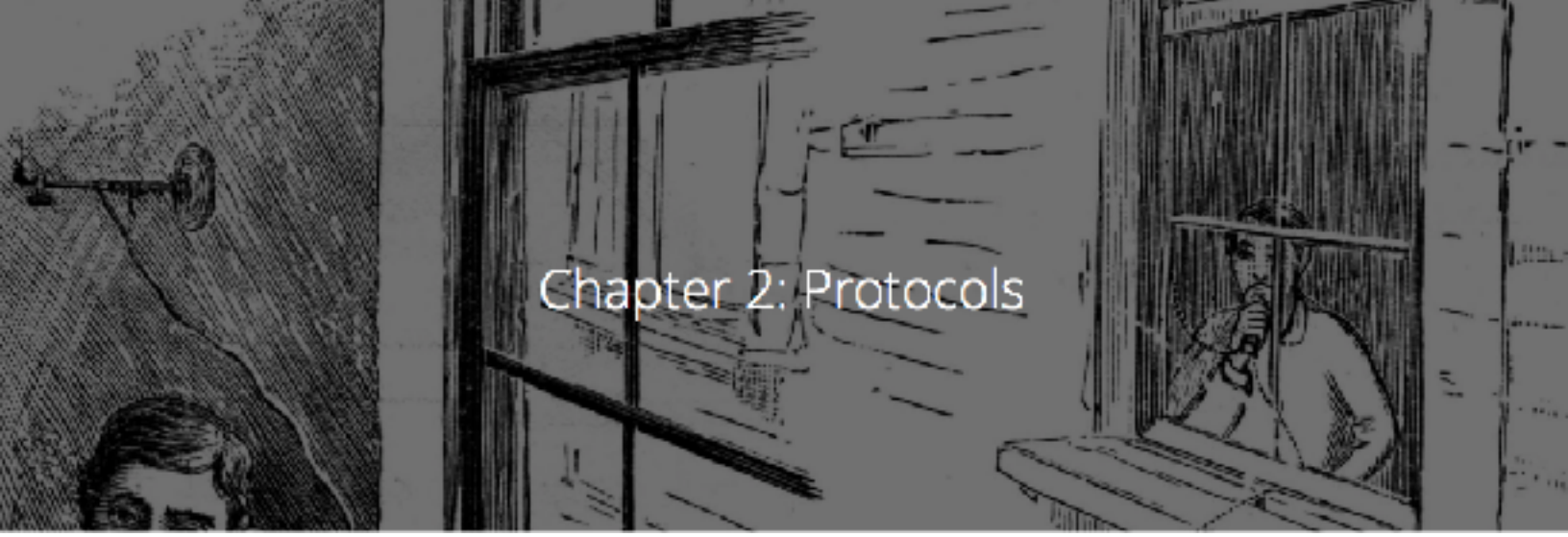
In the next chapter, we dive into the mechanics of how a client talks with an API.

NOTES

1. David Berlind, ProgrammableWeb's Directory Hits 10,000 APIs. And Counting. ProgrammableWeb. September 23, 2013.

2. Adam DuVander, API Growth Doubles in 2010, Social and Mobile are Trends. ProgrammableWeb. January 3, 2011.

3. Technically, an API is just a set of rules (interface) that the two sides agree to follow. The company publishing the API then implements their side by writing a program and putting it on a server. In practice, lumping the interface in with the implementation is an easier way to think about it.



Chapter 2: Protocols

In Chapter 1, we got our bearings by forming a picture of the two sides involved in an API, the server and the client. With a solid grasp on the who, we are ready to look deeper into how these two communicate. For context, we first look at the human model of communication and compare it to computers. After that, we move on to the specifics of a common protocol used in APIs.

Knowing the Rules

People create social etiquette to guide their interactions. One example is how we talk to each other on the phone. Imagine yourself chatting with a friend. While they are speaking, you know to be silent. You know to allow them brief pauses. If they ask a question and then remain quiet, you know they are expecting a response and it is now your turn to talk.

Computers have a similar etiquette, though it goes by the term "protocol." A computer protocol is an accepted set of rules that govern how two computers can speak to each other. Compared to our standards, however, a computer protocol is extremely rigid. Think for a moment of the two sentences "My favorite color is blue" and "Blue is my favorite color." People are able to break down each sentence and

see that they mean the same thing, despite the words being in different orders. Unfortunately, computers are not that smart.

For two computers to communicate effectively, the server has to know exactly how the client will arrange its messages. You can think of it like a person asking for a mailing address. When you ask for the location of a place, you assume the first thing you are told is the street address, followed by the city, the state, and lastly, the zip code. You also have certain expectations about each piece of the address, like the fact that the zip code should only consist of numbers. A similar level of specificity is required for a computer protocol to work.

The Protocol of the Web

There is a protocol for just about everything; each one tailored to do different jobs. You may have already heard of some: Bluetooth for connecting devices, and POP or IMAP for fetching emails.

On the web, the main protocol is the Hyper-Text Transfer Protocol, better known by its acronym, HTTP. When you type an address like `http://example.com` into a web browser, the "http" tells the browser to use the rules of HTTP when talking with the server.

With the ubiquity of HTTP on the web, many companies choose to adopt it as the protocol underlying their APIs. One benefit of using a familiar protocol is that it lowers the learning curve for developers, which encourages usage of the API. Another benefit is that HTTP has several features useful in building a good API, as we'll see later. Right now, let's brave the water and take a look at how HTTP works!

HTTP Requests

Communication in HTTP centers around a concept called the Request-Response Cycle. The client sends the server a request to do something. The server, in turn, sends the client a response saying whether or not the server could do what the client asked.

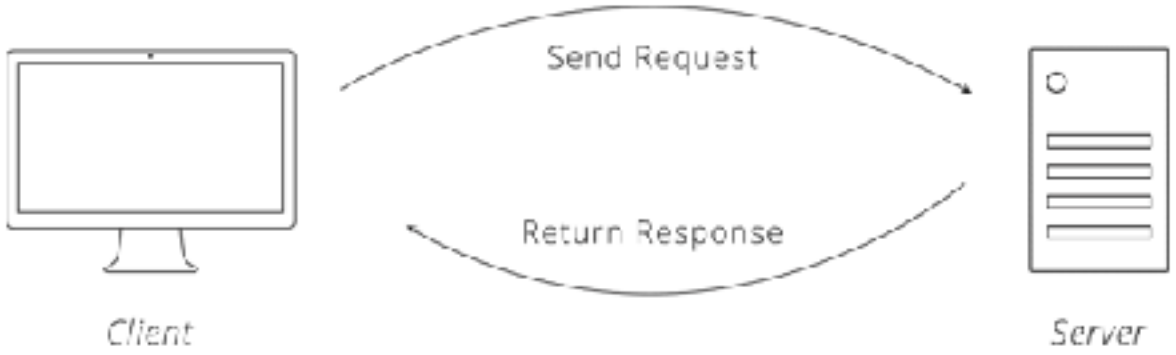


Figure 1. *The Request-Response Cycle.*

To make a valid request, the client needs to include four things:

1. **URL** (Uniform Resource Locator) [1]
2. **Method**
3. List of **Headers**
4. **Body**

That may sound like a lot of details just to pass along a message, but remember, computers have to be very specific to communicate with one another.

URL

URLs are familiar to us through our daily use of the web, but have you ever taken a moment to consider their structure? In HTTP, a URL is a unique address for a thing (a noun). Which things get addresses is entirely up to the business running the server. They can make URLs for web pages, images, or even videos of cute animals.

APIs extend this idea a bit further to include nouns like customers, products, and tweets. In doing so, URLs become an easy way for the client to tell the server which thing it wants to interact with. Of course, APIs also do not call them "things", but give them the technical name "resources."

Method

The request method tells the server what kind of action the client wants the server to take. In fact, the method is commonly referred to as the request "verb."

The four methods most commonly seen in APIs are:

- **GET** - Asks the server to retrieve a resource
- **POST** - Asks the server to create a new resource
- **PUT** - Asks the server to edit/update an existing resource
- **DELETE** - Asks the server to delete a resource

Here's an example to help illustrate these methods. Let's say there is a pizza parlor with an API you can use to place orders. You place an order by making a POST request to the restaurant's server with your order details, asking them to create your pizza. As soon as you send the request, however, you realize you picked the wrong style crust, so you make a PUT request to change it.

While waiting on your order, you make a bunch of GET requests to check the status. After an hour of waiting, you decide you've had enough and make a DELETE request to cancel your order.

Headers

Headers provide meta-information about a request. They are a simple list of items like the time the client sent the request and the size of the request body.

Have you ever visited a website on your smartphone that was specially formatted for mobile devices? That is made possible by an HTTP header called "User-Agent." The client uses this header to tell the server what type of device you are using, and websites smart enough to detect it can send you the best format for your device.

There are quite a few HTTP headers that clients and servers deal with, so we will wait to talk about other ones until they are relevant in later chapters.

Body

The request body contains the data the client wants to send the server. Continuing our pizza ordering example above, the body is where the order details go.

A unique trait about the body is that the client has complete control over this part of the request. Unlike the method, URL, or headers, where the HTTP protocol requires a rigid structure, the body allows the client to send anything it needs.

These four pieces — URL, method, headers, and body — make up a complete HTTP request.

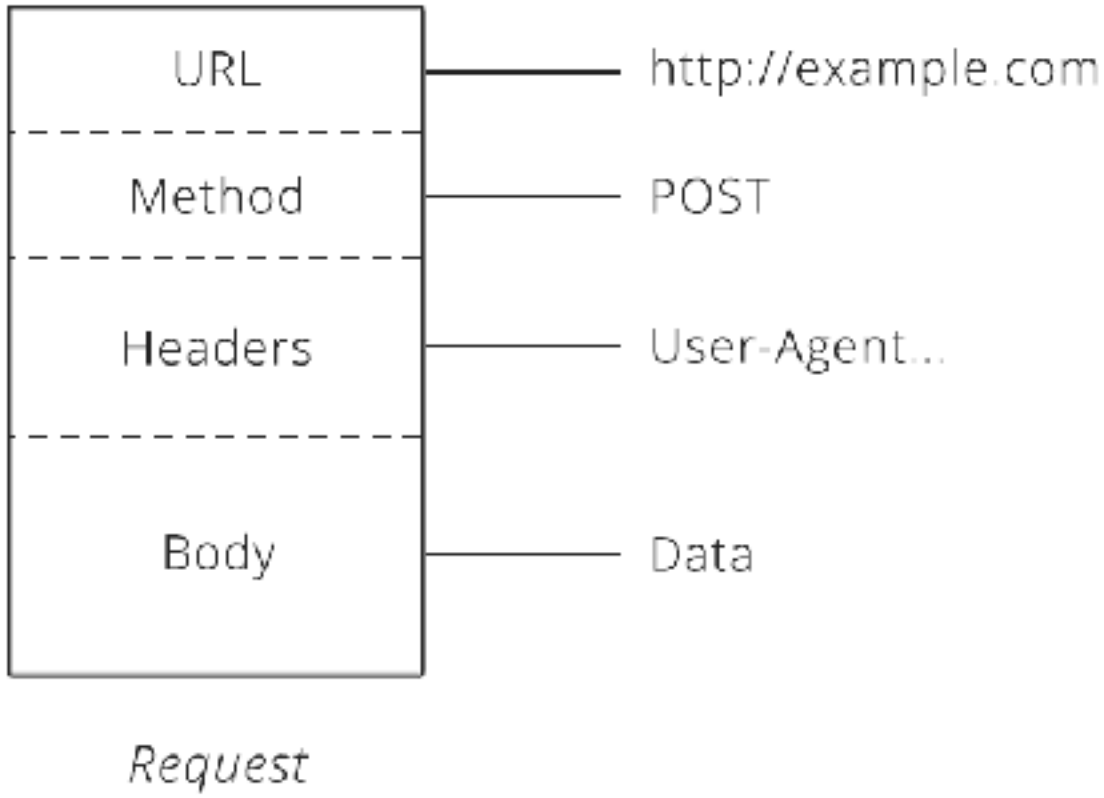


Figure 2. *The structure of an HTTP request.*

HTTP Responses

After the server receives a request from the client, it attempts to fulfill the request and send the client back a response. HTTP responses have a very similar structure to requests. The main difference is that instead of a method and a URL, the response includes a status code. Beyond that, the response headers and body follow the same format as requests.

Status Codes

Status codes are three-digit numbers that each have a unique meaning. When used correctly in an API, this little number can communicate a lot of info to the client. For example, you may have seen this page during your internet wanderings:

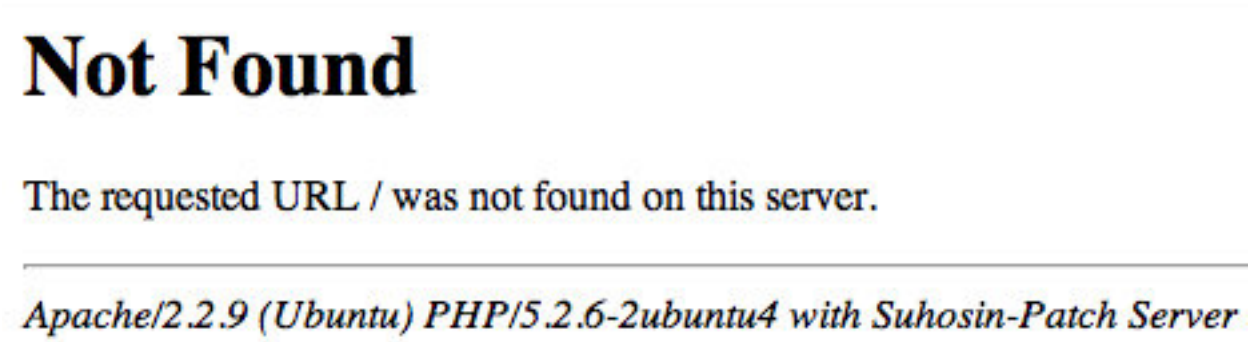


Figure 3. *A default 404 web page.*

The status code behind this response is 404, which means "Not Found." Whenever the client makes a request for a resource that does not exist, the server responds with a 404 status code to let the client know: "that resource doesn't exist, so please don't ask for it again!"

There is a slew of other statuses in the HTTP protocol, including 200 ("success! that request was good") to 503 ("our website/API is currently down.") We'll learn a few more of them as they come up in later chapters.

After a response is delivered to the client, the Request-Response Cycle is completed and that round of communication over. It is now up to the client to initiate any further interactions. The server will not send the client any more data until it receives a new request.

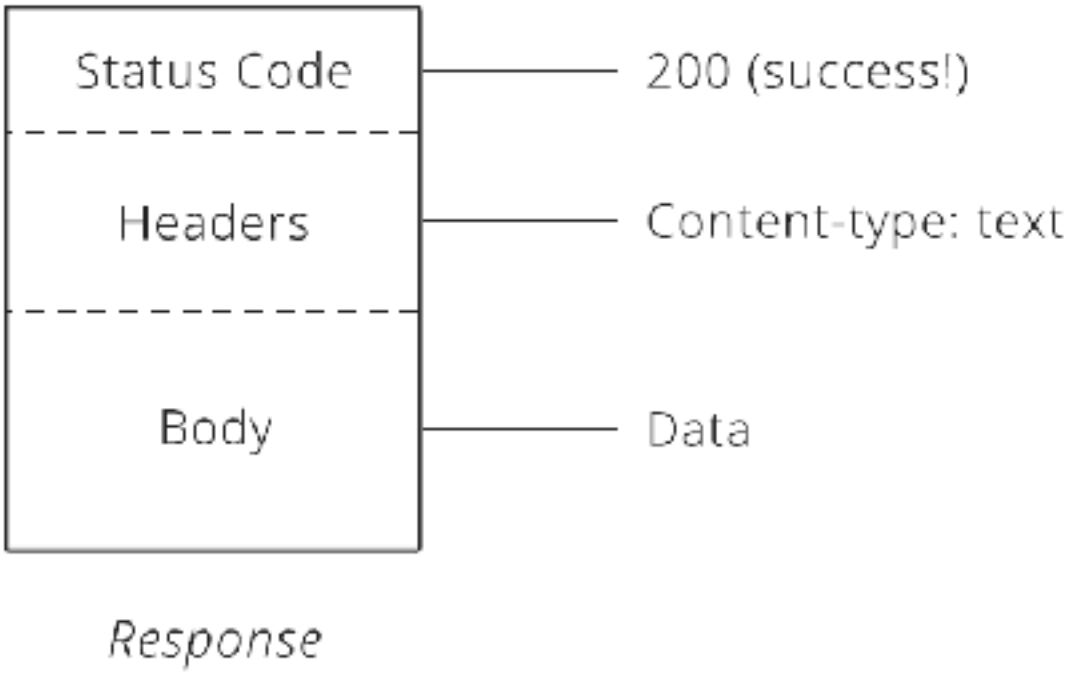


Figure 4. *The structure of an HTTP response.*

How APIs Build on HTTP

By now, you can see that HTTP supports a wide range of permutations to help the client and server talk. So, how does this help us with APIs? The flexibility of HTTP means that APIs built on it can provide clients with a lot of business potential. We saw that potential in the pizza ordering example above. A simple tweak to the request method was the difference between telling the server to create a new order or cancel an existing one. It was easy to turn the desired business outcome into an instruction the server could understand. Very powerful!

This versatility in the HTTP protocol extends to other parts of a request, too. Some APIs require a particular header, while others require specific information inside the request body. Being able to use APIs hinges on knowing how to make the correct HTTP request to get the result you want.

Chapter 2 Recap

The goal of this chapter was to give you a basic understanding of HTTP. The key concept was the Request-Response Cycle, which we broke down into the following parts:

- **Request** - consists of a URL (http://...), a method (GET, POST, PUT, DELETE), a list of headers (User-Agent...), and a body (data).
- **Response** - consists of a status code (200, 404...), a list of headers, and a body.

Throughout the rest of the course, we will revisit these fundamentals as we discover how APIs rely on them to deliver power and flexibility.

Homework

Use the form on the site for Chapter 2 to make the following list of requests and see what responses you are given.

Instructions

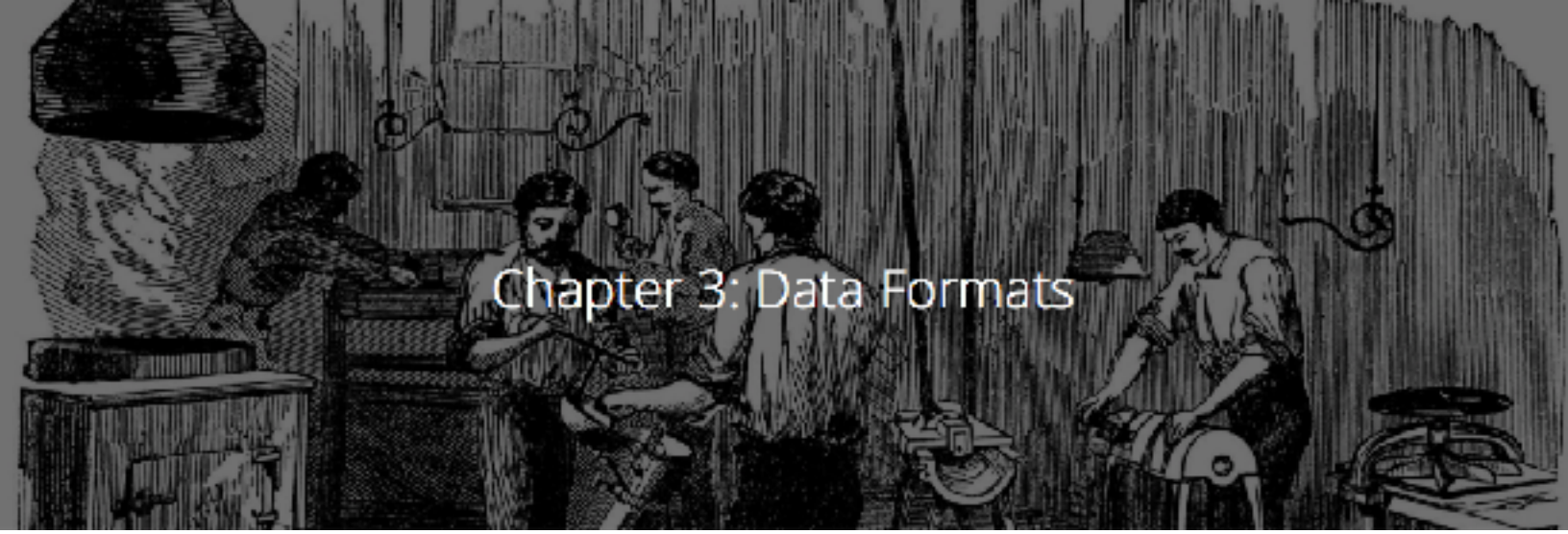
1. Send a GET request without any body data.
2. Send a POST request and type your favorite kind of pizza in the body field.
3. Send a PUT request and type a new ingredient to add to your pizza in the body field.
4. Send a DELETE request without any body data.

Next

In the next chapter, we explore what kind of data APIs pass between the client and the server.

NOTES

1. The HTTP specification actually requires a request to have a URI (Universal Resource Identifier), of which URLs are a subset, along with URNs (Uniform Resource Names). We chose URL because it is the acronym readers already know. The subtle differences between these three are beyond the scope of the course.



Chapter 3: Data Formats

So far, we've learned that HTTP (Hyper-Text Transfer Protocol) is the underpinning of APIs on the web and that to use them, we need to know how HTTP works. In this chapter, we explore the data APIs provide, how it's formatted, and how HTTP makes it possible.

Representing Data

When sharing data with people, the possibilities for how to display the information is limited only by human imagination. Recall the pizza parlor from last chapter — how might they format their menu? It could be a text-only, bulleted list; it could be a series of photos with captions; or it could even be only photos, which foreign patrons could point at to place their order.

A well-designed format is dictated by what makes the information the easiest for the intended audience to understand.

The same principle applies when sharing data between computers. One computer has to put the data in a format that the other will understand. Generally, this means some kind of text format. The most

common formats found in modern APIs are **JSON (JavaScript Object Notation)** and **XML (Extensible Markup Language)**.

JSON

Many new APIs have adopted JSON as a format because it's built on the popular Javascript programming language, which is ubiquitous on the web and usable on both the front- and back-end of a web app or service. JSON is a very simple format that has two pieces: *keys* and *values*. Keys represent an attribute about the object being described. A pizza order can be an object. It has attributes (keys), such as crust type, toppings, and order status. These attributes have corresponding values (thick crust, pepperoni, and out-for-delivery).

Let's see how this pizza order could look in JSON:

```
{
    "crust": "original",
    "toppings": ["cheese", "pepperoni", "garlic"],
    "status": "cooking"
}
```

In the JSON example above, the keys are the words on the left: toppings, crust, and status. They tell us what attributes the pizza order contains. The values are the parts to the right. These are the actual details of the order.

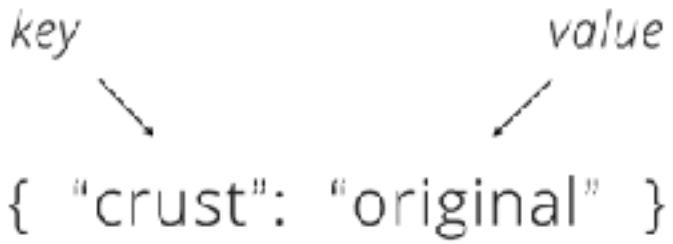


Figure 1. *JSON key and value.*

If you read a line from left to right, you get a fairly natural English sentence. Taking the first line as an example, we could read it as, "the crust for this pizza is original style." The second line can also be read — in JSON, a value that starts and ends with square brackets ([]) is a list of values. So, we read the second line of the order as, "the toppings for this order are: cheese, pepperoni, and garlic."

Sometimes, you want to use an object as the value for a key. Let's extend our pizza order with customer details so you can see what this might look like:

```
{
  "crust": "original",
  "toppings": ["cheese", "pepperoni", "garlic"],
  "status": "cooking",
  "customer": {
    "name": "Brian",
    "phone": "573-111-1111"
```

```
  }
}
```

In this updated version, we see that a new key, "customer", is added. The value for this key is another set of keys and values that provide details about the customer that placed the order. Cool trick, huh?! This is called an *Associative Array*. Don't let the technical term intimidate you though - an associative array is just a nested object.

XML

XML has been around since 1996 [1]. With age, it has become a very mature and powerful data format. Like JSON, XML provides a few simple building blocks that API makers use to structure their data. The main block is called a *node*.

Let's see what our pizza order might look like in XML:

```
<order>
    <crust>original</crust>
    <toppings>
        <topping>cheese</topping>
        <topping>pepperoni</topping>
        <topping>garlic</topping>
    </toppings>
    <status>cooking</status>
```

```
</order>
```

XML always starts with a root node, which in our pizza example is "order." Inside the order are more "child" nodes. The name of each node tells us the attribute of the order (like the key in JSON) and the data inside is the actual detail (like the value in JSON).

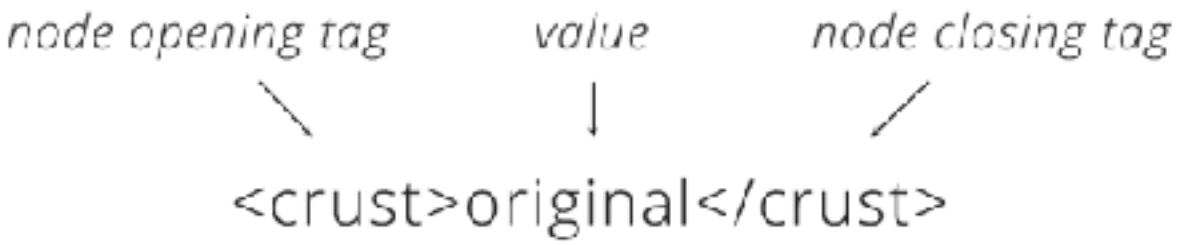


Figure 2. *XML node and value.*

You can also infer English sentences by reading XML. Looking at the line with "crust", we could read, "the crust for the pizza is original style." Notice how in XML, every item in the list of toppings is wrapped by a node. You can see how the XML format requires a lot more text to communicate than JSON does.

How Data Formats Are Used In HTTP

Now that we've explored some available data formats, we need to know how to use them in HTTP. To do so, we will say hello again to one of the fundamentals of HTTP: headers. In Chapter 2, we learned that headers are a list of information about a request or response. There is a header for saying what format the data is in: *Content-Type*.

When the client sends the Content-Type header in a request, it is telling the server that the data in the body of the request is formatted a

particular way. If the client wants to send the server JSON data, it will set the Content-Type to "application/json." Upon receiving the request and seeing that Content-Type, the server will first check if it understands that format, and, if so, it will know how to read the data. Likewise, when the server sends the client a response, it will also set the Content-Type to tell the client how to read the body of the response.

Sometimes, the client can only speak one data format. If the server sends back anything other than that format, the client will fail and throw an error. Fortunately, a second HTTP header comes to the rescue. The client can set the *Accept* header to tell the server what data formats it is able to accept. If the client can only speak JSON, it can set the Accept header to "application/json." The server will then send back its response in JSON. If the server doesn't support the format the client requests, it can send back an error to the client to let it know the request is not going to work.

With these two headers, Content-Type and Accept, the client and server can work with the data formats they understand and need to work properly.

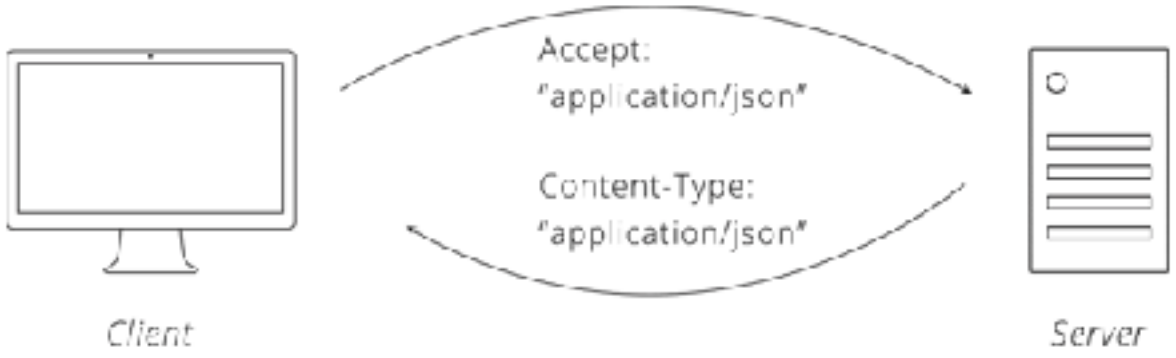


Figure 3. *Data format headers.*

Chapter 3 Recap

In this chapter, we learned that for two computers to communicate, they need to be able to understand the data format passed to them. We were introduced to 2 common data formats used by APIs, JSON and XML. We also learned that the Content-Type HTTP header is a useful way to specify what data format is being sent in a request and the Accept header specifies the requested format for a response.

The key terms we learned were:

- **JSON:** JavaScript Object Notation
- **Object:** a thing or noun (person, pizza order...)
- **Key:** an attribute about an object (color, toppings...)
- **Value:** the value of an attribute (blue, pepperoni...)
- **Associative array:** a nested object
- **XML:** Extensible Markup Language

Homework

Use the form in our site for Chapter 3 to make the following list of requests and see what responses you are given.

Instructions

1. Send a request with: Content-Type header = "application/json", Accept header = "application/json", and data format = "XML".

2. Send a request with: Content-Type header = "application/json", Accept header = "application/json", and data format = "JSON".
3. Ok, now just try changing things around and seeing what happens! :)

Next

In the next chapter, we find out how two computers can establish trust using *Authentication* in order to pass along sensitive data, like customer details or private content.

Notes:

1. http://en.wikipedia.org/wiki/XML

Chapter 4: Authentication, Part 1

Things are starting to pick up in our understanding of APIs. We know who the client and server are, we know they use HTTP to talk to each other, and we know they speak in specific data formats to understand each other. Knowing how to talk, though, leaves an important question: how does the server know the client is who it claims to be? In this chapter, we explore two ways that the client can prove its identity to the server.

Identities in a Virtual World

You've probably registered for an account on a website before. The process involves the site asking you for some personal information, most notably a username and a password. These two pieces of information become your identifying marks. We call these your **credentials**. When you visit the website again, you can login by providing these credentials.

Logging-in with a username and password is one example of a technical process known as **authentication**. When you authenticate with a server, you prove your identity to the server by telling it information that only you know (at least we hope only you know it). Once the server knows who you are, it can trust you and divulge the private data in your account.

There are several techniques APIs use to authenticate a client. These are called **authentication schemes**. Let's take a look at two of these schemes now.

Basic Authentication

The logging-in example above is the most basic form of authentication. In fact, the official name for it is **Basic Authentication** ("Basic Auth" to its friends). Though the name has not garnered any creativity awards, the scheme is a perfectly acceptable way for the server to authenticate the client in an API.

Basic Auth only requires a username and password. The client takes these two credentials, smooshes them together to form a single value [1], and passes that along in the request in an HTTP header called **Authorization**.

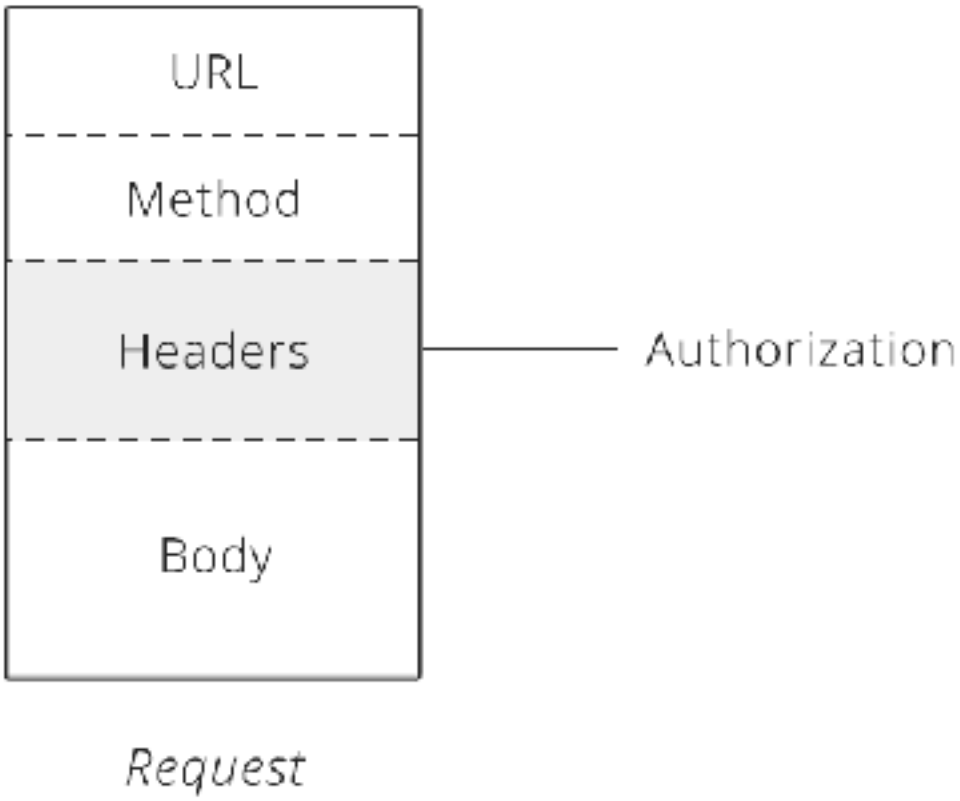


Figure 1. *The Authorization HTTP header.*

When the server receives the request, it looks at the Authorization header and compares it to the credentials it has stored. If the username and password match one of the users in the server's list, the server fulfills the client's request as that user. If there is no match, the server returns a special status code (401) to let the client know that authentication failed and the request is denied.

Though Basic Auth is a valid authentication scheme, the fact that it uses same username and password to access the API and manage the account is not ideal. That is like a hotel handing a guest the keys to the whole building rather than to a room.

Similarly with APIs, there may be times when the client should have different permissions than the account owner. Take for example a business owner who hires a contractor to write a program that uses an API on their behalf. Trusting the contractor with the account credentials puts the owner at risk because an unscrupulous contractor could change the password, locking the business owner out of their own account. Clearly, it would be nice to have an alternative.

API Key Authentication

API Key authentication is a technique that overcomes the weakness of using shared credentials by requiring the API to be accessed with a unique key. In this scheme, the key is usually a long series of letters and numbers that is distinct from the account owner's login password. The owner gives the key to the client, very much like a hotel gives a guest a key to a single room.

When the client authenticates with the API key, the server knows to allow the client access to data, but now has the option to limit administrative functions, like changing passwords or deleting accounts. Sometimes, keys are used simply so the user does not have to give out

their password. The flexibility is there with API Key authentication to limit control as well as protect user passwords.

So, where does the API key go? There is a header for it, too, right? Unfortunately, there is not. Unlike Basic Auth, which is an established standard with strict rules, API keys were conceived at multiple companies in the early days of the web. As a result, API key authentication is a bit like the wild west; everybody has their own way of doing it.

Over time, however, a few common approaches have emerged. One is to have the client put the key in the Authorization header, in lieu of a username and password. Another is to add the key onto the URL (`http://example.com?api_key=my_secret_key`). Less common is to bury the key somewhere in the request body next to the data. Wherever the key goes, the effect is the same - it lets the server authenticate the client.

Chapter 4 Recap

In this chapter, we learned how the client can prove its identity to the server, a process known as authentication. We looked at two techniques, or schemes, APIs use to authenticate.

The key terms we learned were:

- **Authentication:** process of the client proving its identity to the server
- **Credentials:** secret pieces of info used to prove the client's identity (username, password...)

- **Basic Auth:** scheme that uses an encoded username and password for credentials
- **API Key Auth:** scheme that uses a unique key for credentials
- **Authorization Header:** the HTTP header used to hold credentials

Homework

Use the form in our site for Chapter 4 below to explore locations using the Google Maps API.

Next

In the next chapter, we continue the discussion of authentication by looking at a third technique; one that is quickly becoming the standard of the web.

1. The actual process involves combining the username with a colon, followed by the password, and then running the whole string through the base64 encoding algorithm. Thus "user" and "password" becomes "user:password" and, after encoding, you have "dXNlcjpwYXNzd29yZAo=".

Chapter 5: Authentication, Part 2

In Chapter 4, we mentioned most websites use a username and password for authentication credentials. We also discussed how reusing these credentials for API access is insecure, so APIs often require a different set of credentials from the ones used to login to the website. A common example is API keys. In this chapter, we look at another solution, **Open Authorization (OAuth)**, which is becoming the most widely used authentication scheme on the web.

Making Life Easy for People

Have you ever had to complete a registration form like the one below?

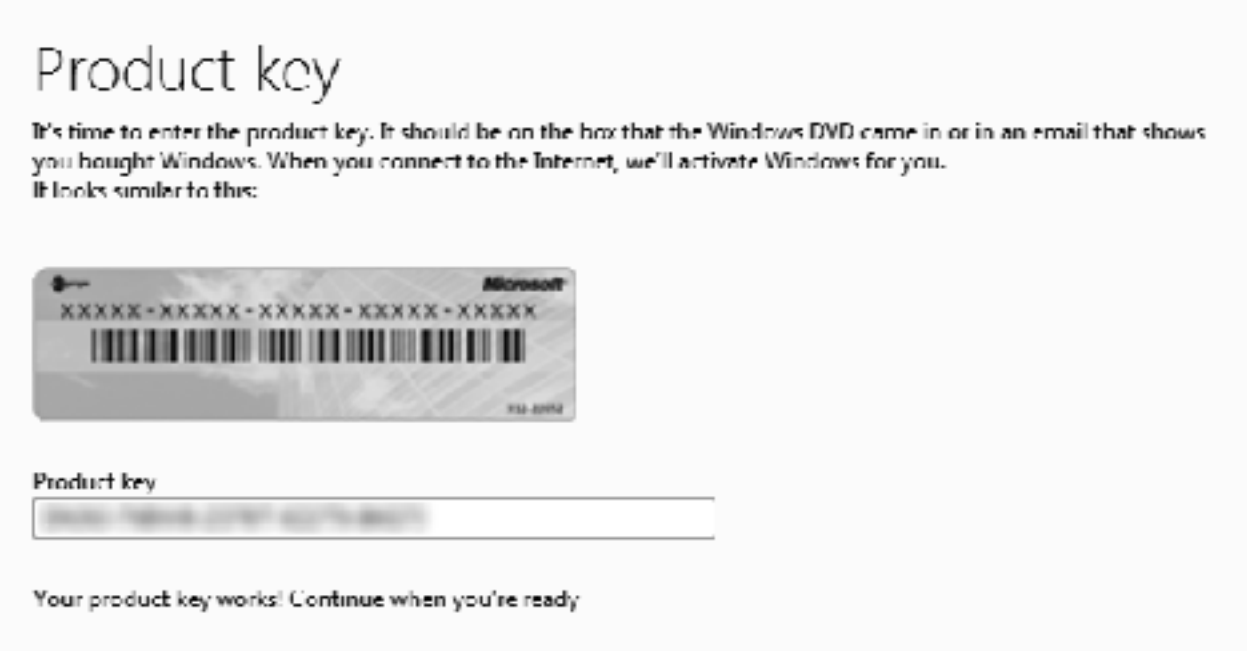


Figure 1. *A product key as seen on Microsoft's Windows 8 registration form.*

Typing a long key into a form field like the one above makes for a poor user-experience. First, you have to find the required the key. Sure, it was right in your inbox when you bought the software, but a year later, you're scrambling to find it (What email was it sent from? Which email did I use to register?!) Once located, you have to enter the darned thing perfectly - making a typo or missing a single character will result in failure, or might even get you locked out of your unregistered software!

Forcing users to work with API keys is a similarly poor experience. Typos are a common problem and it requires the user to do part of the setup between the client and server manually. The user must obtain the key from the server, then give it to the client. For tools meant to automate work, surely there's a better solution.

Enter OAuth. Automating the key exchange is one of the main problems OAuth solves. It provides a standard way for the client to get a key from the server by walking the user through a simple set of steps. From the

user's perspective, all OAuth requires is entering credentials. Behind the scenes, the client and server are chattering back and forth to get the client a valid key.

There are currently two versions of OAuth, aptly named **OAuth 1** and **OAuth 2**. Understanding the steps in each is necessary to be able to interact with APIs that use them for authentication. Since they share a common workflow, we will walk through the steps of OAuth 2, then point out the ways in which OAuth 1 differs.

OAuth 2

To get started, we first need to know the cast of characters involved in an OAuth exchange:

- The User - A person who wants to connect two websites they use
- The Client - The website that will be granted access to the user's data
- The Server - The website that has the user's data

Next, we need to give a quick disclaimer. One goal of OAuth 2 is to allow businesses to adapt the authentication process to their needs. Due to this extendable nature, APIs can have slightly different steps. The workflow shown below is a common one found among web-based apps. Mobile and desktop applications might use slight variations on this process.

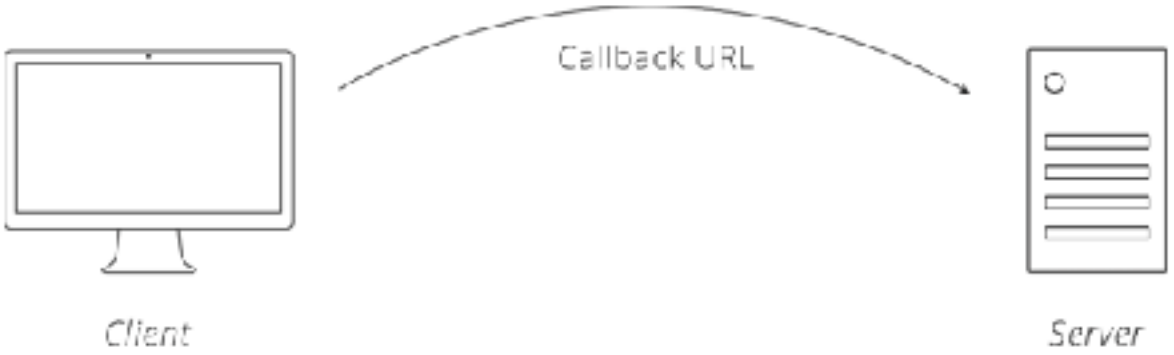
With that, here are the steps of OAuth 2.

Step 1 - User Tells Client to Connect to Server

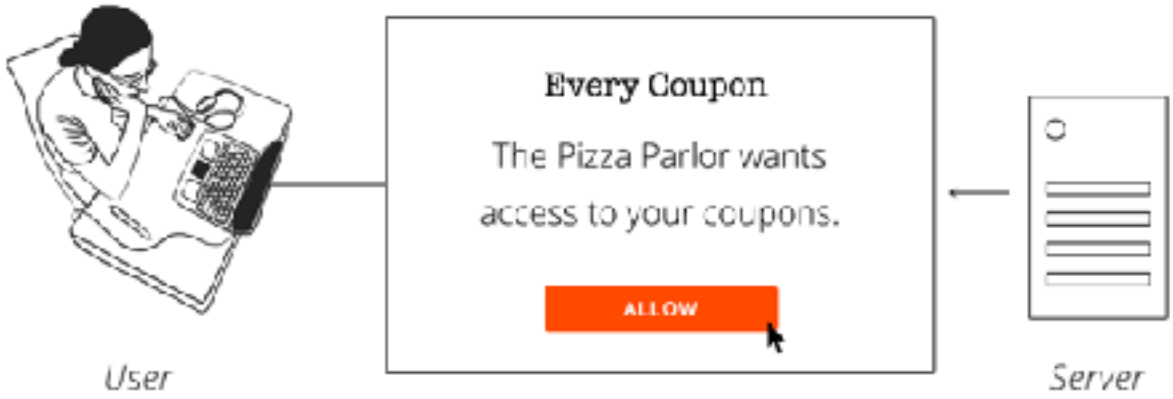
The user kicks off the process by letting the client know they want it to connect to the server. Usually, this is by clicking a button.

Step 2 - Client Directs User to Server

The client sends the user over to the server's website, along with a URL that the server will send the user back to once the user authenticates, called the **callback URL**.

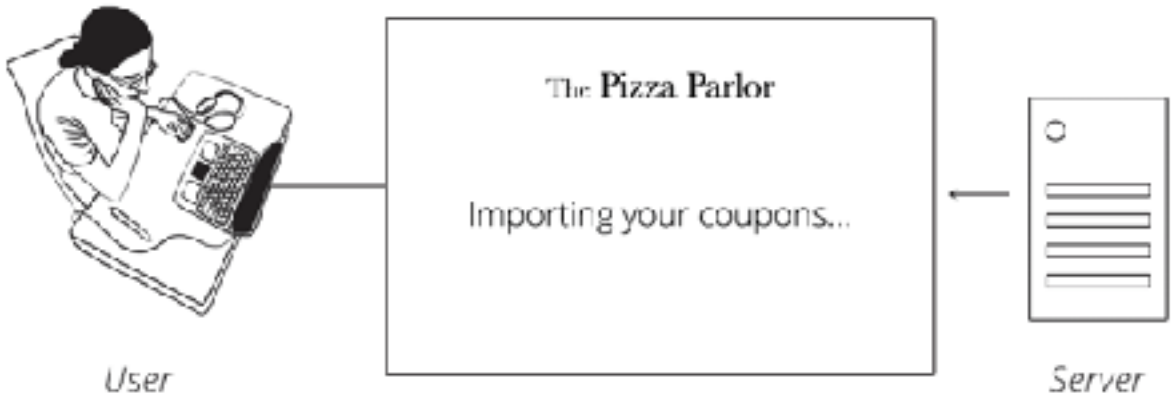


Step 3 - User Logs-in to Server and Grants Client Access

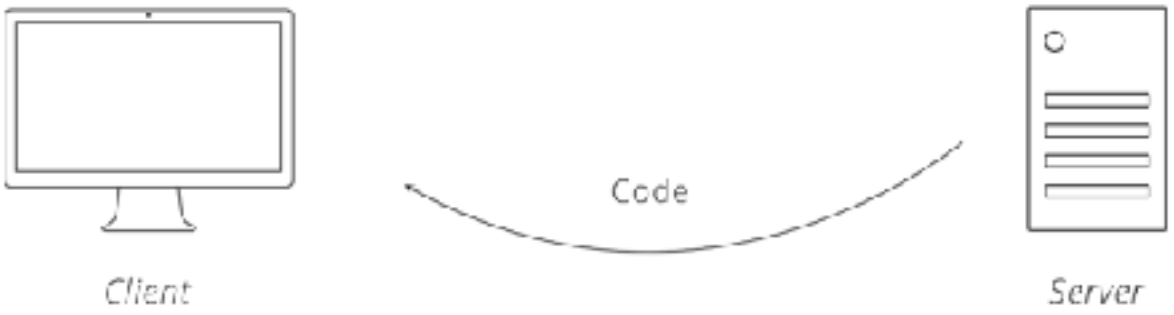


With their normal username and password, the user authenticates with the server. The server is now certain that one of its own users is requesting that the client be given access to the user's account and related data.

Step 4 - Server Sends User Back to Client, Along with Code

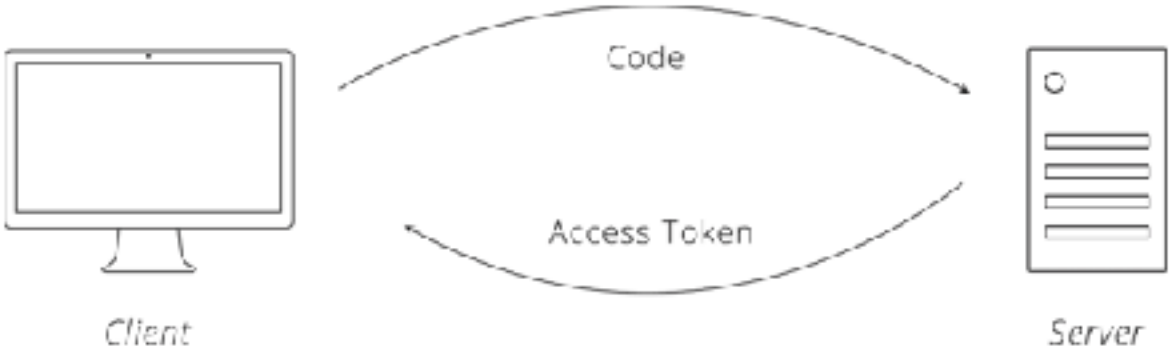


The server sends the user back to the client (to the Callback URL from Step 2). Hidden in the response is a unique **authorization code** for the client.

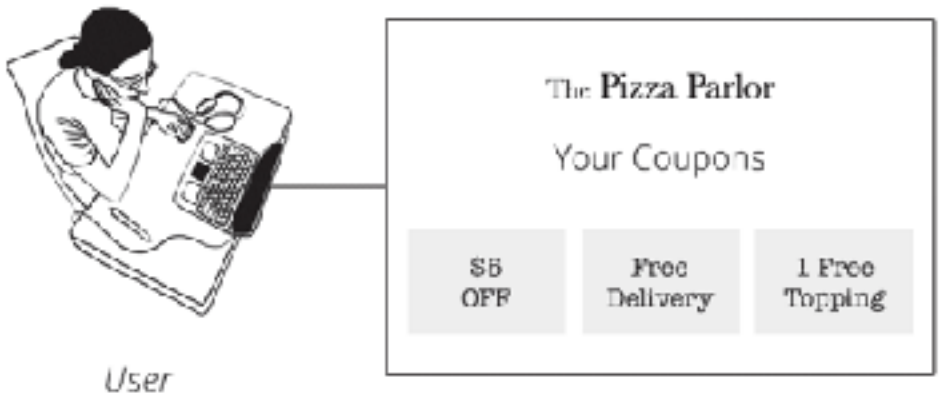


Step 5 - Client Exchanges Code + Secret Key for Access Token

The client takes the authorization code it receives and makes another request to the server. This request includes the client's **secret key**. When the server sees a valid authorization code and a trusted client secret key, it is certain that the client is who it claims to be and that it is acting on behalf of a real user. The server responds back with an **access token**.



Step 6 - Client Fetches Data from Server



At this point, the client is free to access the server on the user's behalf. The access token from Step 6 is essentially another password into the user's account on the server. The client includes the access token with every request so it can authenticate directly with the server.

Client Refreshes Token (Optional)

A feature introduced in OAuth 2 is the option to have access tokens expire. This is helpful in protecting users' accounts by strengthening security - the faster a token expires, the less time a stolen token might be used maliciously, similar to how a credit card number expires after a certain time. The lifespan of a token is set by the server. APIs in the wild use anything from hours to months. Once the lifespan is reached, the client must ask the server for a new token.

How OAuth 1 Is Different

There are several key differences between the versions of OAuth. One we already mentioned; access tokens do not expire.

Another distinction is that OAuth 1 includes an extra step. Between Steps 1 and 2 above, OAuth 1 requires the client to ask the server for a **request token**. This token acts like the authorization code in Oauth 2 and is what gets exchanged for the access token.

A third difference is that OAuth 1 requires requests to be digitally signed. We'll skip the details of how signing works (you can find code libraries to do this for you), but it is worth knowing why it is in one version and not the other. Request signing is a way to protect data from being tampered with while it moves between the client and the server. Signatures allow the server to verify the authenticity of the requests.

Today, however, most API traffic happens over a channel that is already secure (HTTPS). Recognizing this, OAuth 2 eliminates signatures in an effort to make version two easier to use. The trade-off is that OAuth 2 relies on other measures to provide security to the data in transit.

Authorization

An element of OAuth 2 that deserves special attention is the concept limiting access, known formally as **authorization**. Back in Step 2, when the user clicks the button to allow the client access, buried in the fine print are the exact permissions the client is asking for. Those permissions, called **scope**, are another important feature of OAuth 2. They provide a way for the client to request limited access to the user's data, thereby making it easier for the user to trust the client.

What makes scope powerful is that it is client-based restrictions. Unlike an API Key, where limits placed on the key affect every client equally, OAuth scope allows one client to have permission X and another permissions X and Y. That means one website might be able to view your contacts, while another site can view *and* edit them.

Chapter 5 Recap

In this chapter, we learned the flow of the OAuth authentication process. We compared the two versions, pointing out the major difference between them.

The key terms we learned were:

- **OAuth:** an authentication scheme that automates the key exchange between client and server.
- **Access Token:** a secret that the client obtains upon successfully completing the OAuth process.
- **Scope:** permissions that determine what access the client has to user's data.

Homework

This exercise walks you through the OAuth 2 flow for Facebook. In this scenario, you play the role of the user, we are the client, and Facebook is the server.

You will connect your real account and have us pull a few friends from your list to illustrate how the process works.

We will not store any of the data we get, nor do we keep the access token. As soon as you leave this page, we will disconnect with Facebook and all temporary data will be removed.

Visit our site for Chapter 5 to run the exercise.

Next

In the next chapter, we look at some of the basics concepts in API design, including how APIs organize their data so the client can easily access what it wants.

This chapter marks a turning point in our adventure with APIs. We are finished covering fundamentals and are now ready to see how the previous concepts combine to form an API. In this chapter, we discuss the components of an API by designing one.

Organizing Data

National Geographic estimated that in 2011, Americans snapped 80 billion photos [1]. With so many photos, you can imagine the different approaches people have to organizing them on their computers. Some people prefer to dump everything into a single folder. Others meticulously arrange their pictures into a hierarchy of folders by year, month, and event.

Companies give similar thought to organization when building their APIs. As we mentioned in Chapter 1, the purpose of an API is to make it easy for computers to work with the company's data. With ease of use in mind, one company may decide to have a single URL for all the data and make it searchable (sort of like having one folder for all your photos). Another may decide to give each piece of data its own URL, organized in a hierarchy (like having folders and sub-folders for photos). Each company chooses the best way to structure its API for its particular situation, guided by existing industry best practices.

Start with an Architectural Style

When discussing APIs, you might hear talk of "soap" and "rest" and wonder whether the software developers are doing work or planning a vacation. The truth is that these are the names of the two most common architectures for web-based APIs. **SOAP** (formerly an acronym [2]) is an XML-based design that has standardized structures for requests and responses. **REST**, which stands for Representational State Transfer, is a more open approach, providing lots of conventions, but leaving many decisions to the person designing the API.

Throughout this course, you may have noticed we've had an inclination for REST APIs. The preference is largely due to REST's incredible rate of adoption [3]. This is not to say that SOAP is evil; it has its strong points [4]. However, the focus of our discussion will stay on REST as this will likely be the kind of API you encounter. In the remaining sections, we walk through the components that make up a REST API.

Our First Resource

Back in Chapter 2, we talked a little bit about **resources**. Recall that resources are the nouns of APIs (customers and pizzas). These are the things we want the world to be able to interact with through our API.

To get a feel for how a company would design an API, let's try our hand at it with our pizza parlor. We'll start by adding the ability to order a pizza.

For the client to be able to talk pizzas with us, we need to do several things:

1. Decide what resource(s) need to be available.
2. Assign URLs to those resources.
3. Decide what actions the client should be allowed to perform on those resources.
4. Figure out what pieces of data are required for each action and what format they should be in.

Picking resources can be a difficult first task. One way to approach the problem is to step through what a typical interaction involves. For our pizza parlor, we probably have a menu. On that menu are pizzas. When a customer wants us to make one of the pizzas for them, they place an order. In this context, menu, pizza, customer, and order all sound like good candidates for resources. Let's start with order.

The next step is assigning URLs to the resource. There are lots of possibilities, but luckily REST conventions give some guidance. In a typical REST API, a resource will have two URL patterns assigned to it. The first is the plural of the resource name, like `/orders`. The second is the plural of the resource name plus a unique identifier to specify a single resource, like `/orders/<order_id>`, where `<order_id>` is the unique identifier for an order. These two URL patterns make up the first **endpoints** that our API will support. These are called endpoints simply because they go at the end of the URL, as in `http://example.com/<endpoint_goes_here>`.

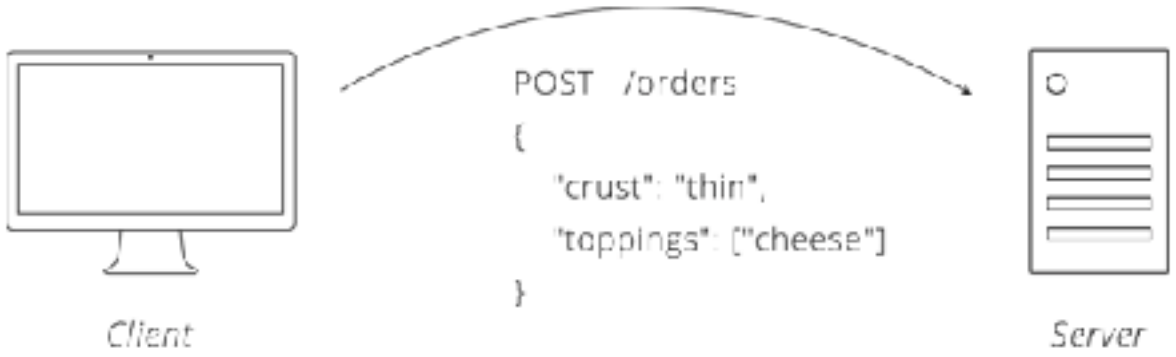
Now that we picked our resource and assigned it URLs, we need to decide what actions the client can perform. Following REST conventions, we say that the plural endpoint (`/orders`) is for listing existing orders and creating new ones. The plural with a unique identifier endpoint (`/orders/<order_id>`), is for retrieving, updating, or cancelling a specific order. The client tells the server which action to perform by passing the appropriate HTTP verb (GET, POST, PUT or DELETE) in the request.

Altogether, our API now looks like this:

HTTP verb	Endpoint	Action
GET	/orders	List existing orders
POST	/orders	Place a new order
GET	/orders/1	Get details for order #1
GET	/orders/2	Get details for order #2
PUT	/orders/1	Update order #1
DELETE	/orders/1	Cancel order #1

With the actions for our order endpoints fleshed out, the final step is to decide what data needs to be exchanged between the client and the server. Borrowing from our pizza parlor example in Chapter 3, we can say that an order needs a crust and toppings. We also need to select a data format that the client and server can use to pass this information back and forth. XML and JSON are both good choices, but for readability sake, we'll go with JSON.

At this point, you should pat yourself on the back; we have designed a functional API! Here is what an interaction between the client and server might look like using this API:



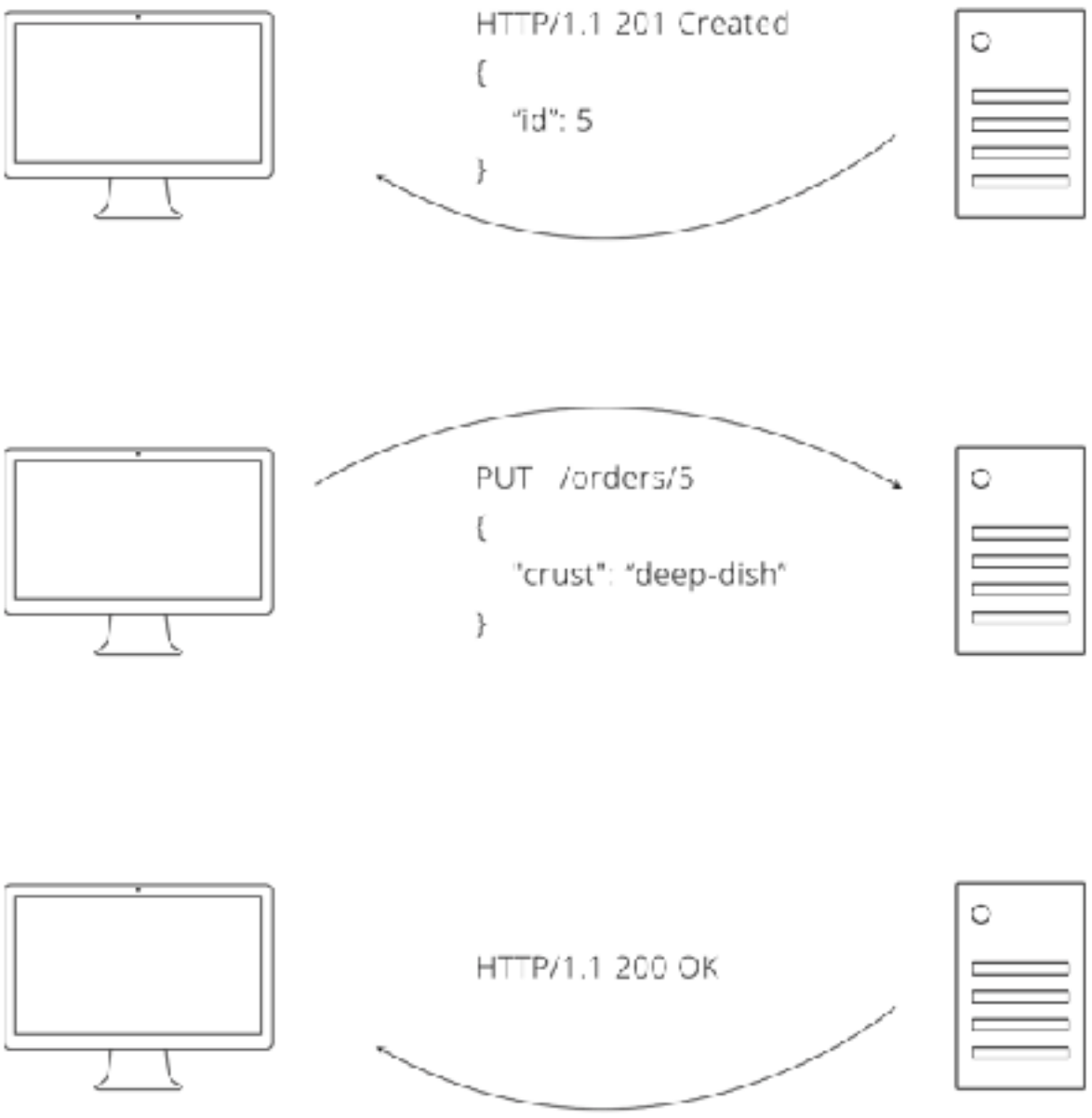


Figure 1. *Example interaction between the client and server using our API.*

Linking Resources Together

Our pizza parlor API is looking sharp. Orders are coming in like never before. Business is so good in fact, we decide we want to start tracking orders by customer to gauge loyalty. An easy way to do this is to add a new customer resource.

Just like with orders, our customer resource needs some endpoints. Following convention, `/customers` and `/customers/<id>` fit nicely. We'll skip the details, but let's say we decide which actions make sense for each endpoint and what data represents a customer. Assuming we do all of that, we come to an interesting question: how do we associate orders with customers?

REST practitioners are split on how to solve the problem of associating resources. Some say that the hierarchy should continue to grow, giving endpoints like `/customers/5/orders` for all of customer #5's orders and `/customers/5/orders/3` for customer #5's third order. Others argue to keep things flat by including associated details in the data for a resource. Under this paradigm, creating an order requires a `customer_id` field to be sent with the order details. Both solutions are used by REST APIs in the wild, so it is worth knowing about each.

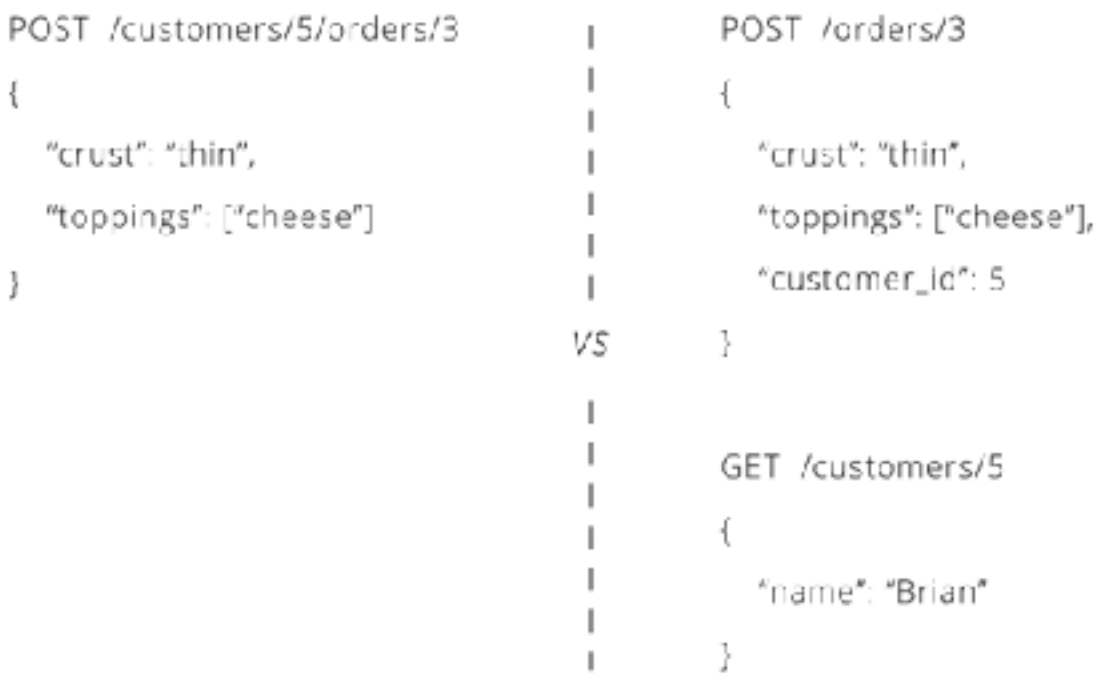


Figure 2. *Two ways to handle associated data in API design.*

Searching Data

As data in a system grows, endpoints that list all records become impractical. Imagine if our pizza parlor had three million completed orders and you wanted to find out how many had pepperoni as a topping. Sending a GET request to `/orders` and receiving all three million orders would not be very helpful. Thankfully, REST has a nifty way for searching through data.

URLs have another component that we have not mentioned yet, the **query string**. Query means search and string means text. The query string is a bit of text that goes onto the end of a URL to pass things along to the API. For example, everything after the question mark is the query string in `http://example.com/orders?key=value`.

REST APIs use the query string to define details of a search. These details are called query **parameters**. The API dictates what parameters it will accept, and the exact names of those parameters need to be used for them to effect the search. Our pizza parlor API could allow the client to search for orders by topping by using this URL: `http://example.com/orders?topping=pepperoni`. The client can include multiple query parameters by listing one after another, separating them by an ampersand ("&"). For example: `http://example.com/orders?topping=pepperoni&crust=thin`.

Another use of the query string is to limit the amount of data returned in each request. Often, APIs will split results into sets (say of 100 or 500 records) and return one set at a time. This process of splitting up the data is known as **pagination** (an analogy to breaking up words into pages for books). To allow the client to page through all the data, the API will support query parameters that allow the client to specify which page of data it wants. In our pizza parlor API, we can support paging by allowing the client to specify two parameters: page and size. If the client makes a request like `GET /orders?page=2&size=200`, we know they want the second page of results, with 200 results per page, so orders 201-400.

Chapter 6 Recap

In this chapter, we learned how to design a REST API. We showed the basic functions an API supports and how to organize the data so that it can be easily consumed by a computer.

The key terms we learned were:

- **SOAP:** API architecture known for standardized message formats

- **REST:** API architecture that centers around manipulating resources
- **Resource:** API term for a business noun like customer or order
- **Endpoint:** A URL that makes up part of an API. In REST, each resource gets its own endpoints
- **Query String:** A portion of the URL that is used to pass data to the server
- **Query Parameters:** A key-value pair found in the query string (topping=cheese)
- **Pagination:** Process of splitting up results into manageable chunks

Homework

Your homework for this chapter is an exploration of API design. We'll look at a few examples using 3 notable APIs to see what's available and how things are structured.

Example 1: The Instagram API

Answer the following questions about Instagram's API design.

To find the answer to the following 3 questions, open the Instagram API docs.

1. What resources does Instagram make available (hint: look at the endpoints)?
2. What is unique identifier for users?

3. For the endpoint `users/self/media/liked`, what is the name of the parameter that limits the number of media results returned?

Example 2: The Facebook API

Answer the following questions about Facebooks's API design.

1. What 3 terms does Facebook use to describe what the Graph API is composed of?

   Open "Quickstart" to find the answer.

2. What does 'me' in the `/me` endpoint translate to as a convenience?

   Open "Using the Graph API" to find the answer.

Example 3: The Twitter API

Answer the following questions about Twitter's API design.

1. What 4 resources, referred to as "objects", does Twitter make available?

   Open the docs index to find the answer.

2. What parameter is required to create a new favorite?

   Open the "POST favorites/create" to find the answer.

Next

In the next chapter, we explore ways to make the client react to changes on the server in real-time.

Notes:

1. Unknown, Image Obsessed. National Geographic. April, 2012.

2. SOAP stood for Simple Object Access Protocol. It was originally used for a very specific type of API access. As developers found ways to apply it to more situations, the name no longer fit, so in SOAP version 1.2 the acronym was dropped.

3. Abel Avram, Is REST Successful in the Enterprise?. InfoQ. June 1, 2011.

4. SOAP provides a very structured architecture. The structure provides system reliability, standard extensions for adding functionality to the protocol, and makes it possible for tools to generate code, saving on development time.

Chapter 7: Real-Time Communication

In Chapter 6, we learned about designing APIs by building our own and exploring a few real-world examples. At this point, we have a lot of hard-earned knowledge and it's time for it to start paying off. We are ready to see how we can put APIs to work for us. In this chapter, we learn four ways to achieve real-time communication through APIs.

Integrations

To set the stage for our discussion, let's remind ourselves why APIs are useful. Back in Chapter 1 we said that APIs make it easy to share data between two systems (websites, desktops, smartphones). Straightforward sharing allows us to link systems together to form an **integration**. People like integrations because they make life easier. With an integration, you can do something in one system and the other will automatically update.

For our purposes, we will split integrations into two broad categories[1]. The first we call "client-driven," where a person interacts with the client and wants the server's data to update. The other we call "server-driven", where a person does something on the server and needs the client to be aware of the change.

The reason for dividing integrations in this manner comes down to one simple fact: the client is the only one who can initiate communication. Remember, the client makes requests and the server just responds. A consequence of this limitation is that changes are easy to send from the client to the server, but hard to do in the reverse direction.

Client-Driven Integration

To demonstrate why client-driven integrations are easy, let's turn to our trusty pizza parlor and its API for ordering pizzas. Say we release a smartphone app that uses the API. In this scenario, the pizza parlor API is the server and the smartphone app is the client. A customer uses the app to choose a pizza and then hits a button to place the order. As soon as the button is pressed, the app knows it needs to make a request to the pizza parlor API.

Figure 1. *Example of a client-driven interaction.*

More generally speaking, when a person interacts with the client, the client knows exactly when data changes, so it can call the API immediately to let the server know. There's no delay (hence it's real-time) and the process is efficient because only one request is made for each action taken by a person.

Server-Driven Integration

Once the pizza order is placed, the customer might want to know when the pizza is ready. How do we use the API to provide them with updates? Well, that is a bit harder. The customer has nothing to do with making the pizza. They are waiting on the pizza parlor to prepare the pizza and update the order status. In other words, the data is changing on the server and the client needs to know about it. Yet, if server can't make requests, we appear to be stuck!

Solving this type of problem is where we utilize the second category of integrations. There are a number of solutions software developers use to get around the client-only requests limitation. Let's take a look at each.

Polling

When the client is the only one who can make requests, the simplest solution to keep it up-to-date with the server is for the client to simply ask the server for updates. This can be accomplished by repeatedly requesting the same resource, a technique known as **polling**.

With our pizza parlor, polling for the status of an order might look like the following.

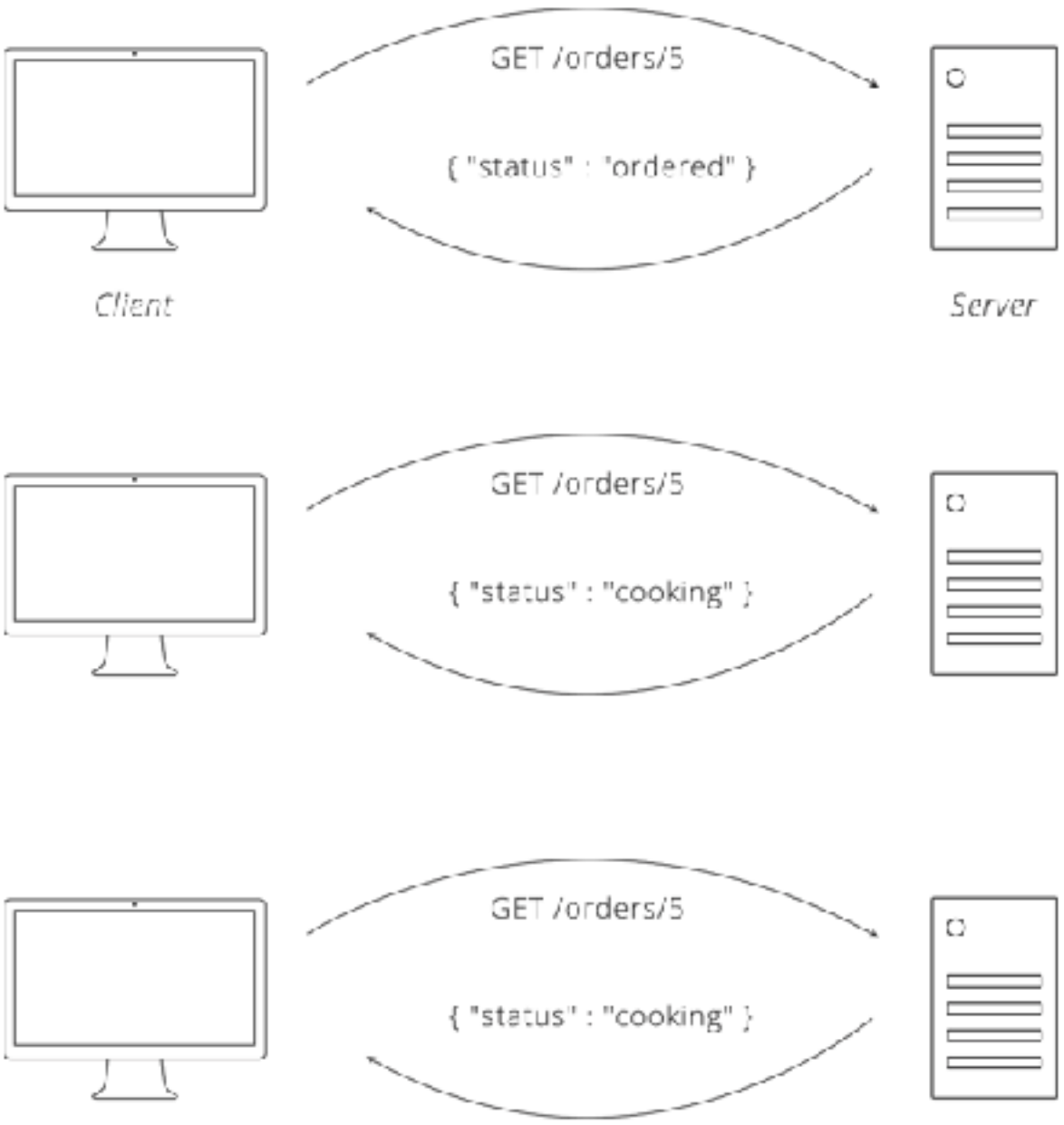


Figure 2. *Example of polling for the status of an order in our Pizza Parlor.*

In this approach, the more frequently the client makes requests (polls), the closer the client gets to real-time communication. If the client polls every hour, at worst, there could be a one-hour delay between a change happening on the server and the client becoming aware of it.

Poll every minute instead and the client and server effectively stay in sync.

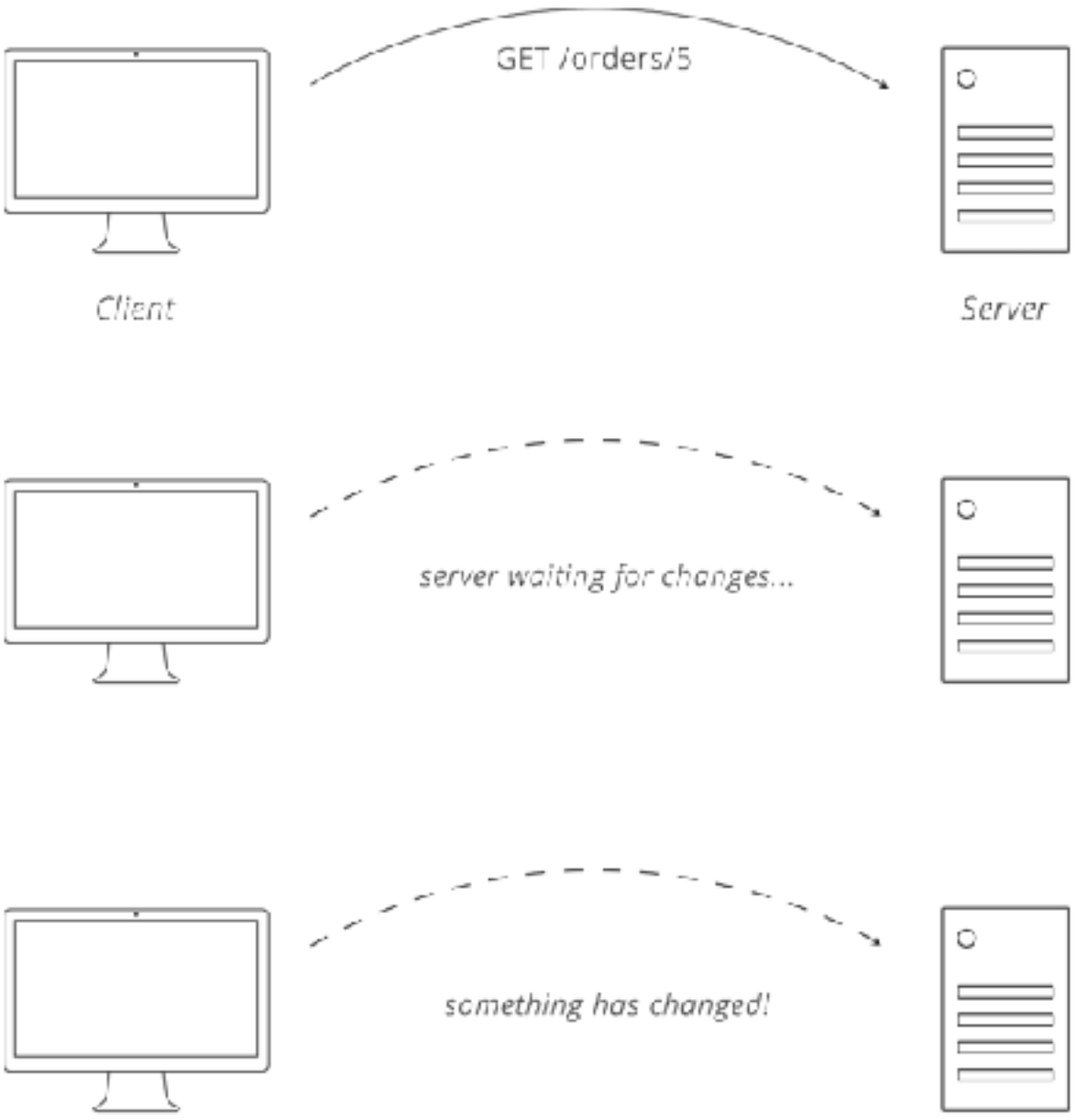
Of course, there is one big flaw with this solution. It is terribly inefficient. Most of the requests the client makes are wasted because nothing has changed. Worse, to get updates sooner, the polling interval has to be shortened, causing the client to make more requests and become even more inefficient. This solution does not scale well.

Long Polling

If requests were free, then nobody would care about efficiency and everyone could just use polling. Unfortunately, handling requests comes at a cost. For an API to handle more requests, it needs to utilize more servers, which costs more money. Scale this cumbersome situation up to Google- or Facebook-sized proportions, and you're paying a lot for inefficiency. Hence, lots of effort has been put into optimizing the way the client can receive updates from the server.

One optimization, which builds off of polling, is called **long polling**. Long polling uses the same idea of the client repeatedly asking the server for updates, but with a twist: the server does not respond immediately. Instead, the server waits until something changes, then responds with the update.

Let's revisit the polling example from above, but this time with a server that uses the long polling trick.


GET /orders/5
Client
Server
server waiting for changes...
something has changed!

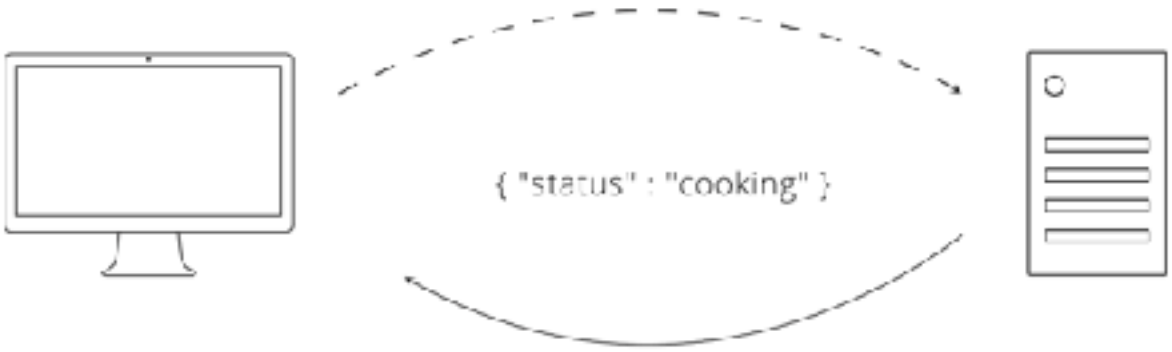


Figure 3. *Long polling example.*

This technique is pretty clever. It obeys the rule of the client making the initial request while leveraging the fact that there is no rule against the server being slow to respond. As long as both the client and the server agree that the server will hold on to the client's request, and the client is able to keep its connection to the server open, it will work.

As resourceful as long polling is, it too has some drawbacks. We'll skip the technical details, but there are concerns like how many requests can the server hold on to at a time, or how to recover if the client or server loses its connection. For now, we'll say that for some scenarios, neither form of polling is sufficient.

Webhooks

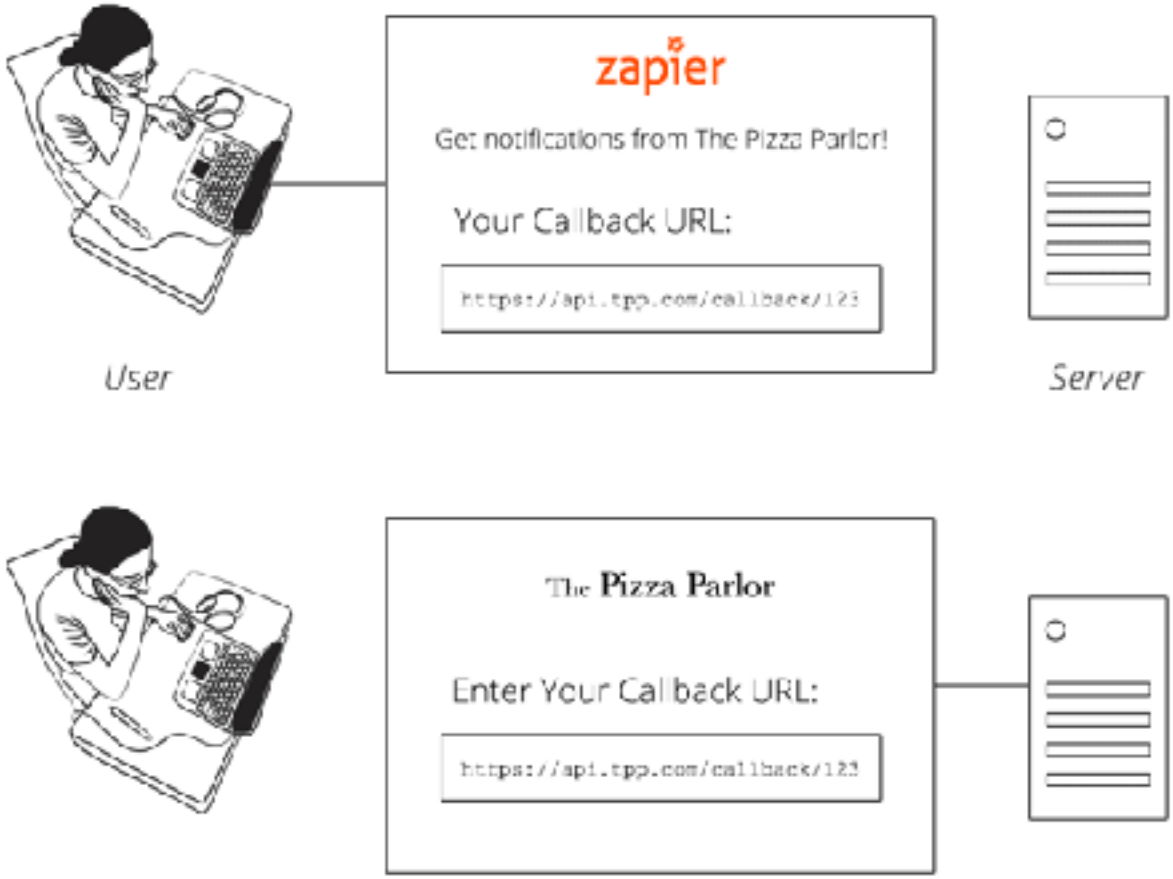
With polling ruled out, some innovative software developers thought, "if all our trouble is because the client is the only one making requests, why not remove that rule?" So they did. The result was **webhooks**, a technique where the client both makes requests and listens for them, allowing the server to easily push updates to it.

If this sounds like cheating because now we have the server making requests to the client, don't worry, you've not been told a lie. What

makes webhooks work is that the client becomes a server too! From a technical perspective, it's sometimes very easy to extend the client's functionality to also listen for requests, enabling two-way communication.

Let's look at the basics of webhooks. In its simplest form, webhooks requires the client to provide a **Callback URL** where it can receive events, and the server to have a place for a person to enter that Callback URL. Then, whenever something changes on the server, the server can send a request to the client's Callback URL to let the client know about the update.

For our pizza parlor, the flow might look a little something like the following.



server waiting for changes...

something has changed!

{ 'status' : "out for delivery" }

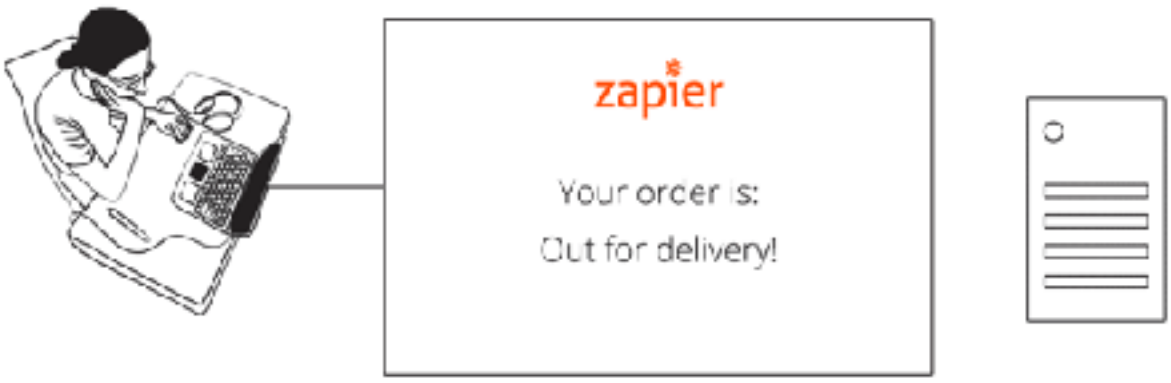


Figure 4. *Using Webhooks to Receive Updates (with Zapier as the client).*

This solution is excellent. Changes happening on the server are sent instantly to the client, so you have true real-time communication. Also, webhooks are efficient since there is only one request per update.

Subscription Webhooks

Building on the idea of webhooks, there have been a variety of solutions that aim to make the setup process dynamic and not require a person to manually enter a Callback URL on the server. You might hear names like HTTP Subscriptions Specification, Restful Webhooks, REST Hooks, and PubSubHubbub. What all of these solutions try to do is define a subscription process, where the client can tell the server what events it is interested in and what Callback URL to send updates to.

Each solution has a slightly different take on the problem, but the general flow looks like the following.


Let me know if my order status changes!
Client
Server
Ok. Will do.
Hey, your order is out for delivery!

Figure 5. *Requests needed for Subscriptions Webhooks.*

Subscription-based webhooks hold a lot of promise. They are efficient, real-time, and easy for people to use. Similar to REST's explosive adoption, a tide is rising behind the movement and it is becoming more common for APIs to support some form of webhooks.

Still, there will likely be a place for polling and long polling for the foreseeable future. Not all clients can also act as servers. Smartphones are a great example where technical constraints rule out webhooks as a possibility. As technology progresses, new ideas will emerge for how to make real-time communication easier between all kinds of devices.

Chapter 7 Recap

In this chapter, we grouped integrations into two broad categories, client-driven and server-driven. We saw how APIs can be used to provide real-time updates between two systems, as well as some of the challenges.

The key terms we learned were:

- **Polling:** Repeatedly requesting a resource at a short interval
- **Long Polling:** Polling, but with a delayed response; improves efficiency
- **Webhooks:** When the client gives the server a Callback URL, so the server can post updates in real time
- **Subscription Webhooks:** Informal name for solutions that make setting up webhooks automatic

Homework

Your homework for this chapter is to play with a little app that introduces the dimension of time to give you a feel for the different methods above.

Check out our site for Chapter 7 to try the app.

Next

In the final chapter of this course, we look at what it takes to turn an API design into working software.

Notes:

1. Client-driven and server-driven are our terms, so don't be surprised if you use one in front of a developer and get only a blank stare in return. Mention polling or webhooks if you want instant credibility.

Chapter 8: Implementation

We made it! We now know everything there is to know about APIs...at an introductory level at least. So, with all this acquired knowledge, how can we put it to good use? In this chapter, we explore how to turn knowledge into working software.

From Plan to Product

As we have seen throughout this course, an API interaction involves two sides. When we are talking at the code-level, though, what we are really saying is that we need two programs that **implement** the API. A program implements an API when it follows the API's rules. In our pizza parlor example, a client that can make requests to the `/orders` endpoint using the correct headers and data format would be a client that implements the pizza parlor's API.

The server program is the responsibility of the company publishing the API. Back in Chapter 6, we looked at the process behind designing the API. After planning, the next step is for the company to implement their side by writing software that follows the design. The last step is to put the resulting program on a server.

Along with the server software, the company publishes **documentation** for the API. The documentation is one or more documents — typically

webpages or PDFs — that explain how to use the API. It includes information like what authentication scheme to use, what endpoints are available, and how the data is formatted. It may also include example responses, code snippets, and an interactive console to play with the available endpoints. Documentation is important because it acts as a guide for building clients. It's where someone interested in using the API goes to learn how the API works.

With documentation in hand, there are a number of ways you can begin to use an API as a client. Let's examine three of those now.

HTTP Clients

An easy way to start using an API is with an HTTP Client, a generic program that lets you quickly build HTTP requests to test with. You specify the URL, headers, and body, and the program sends it to the server properly formatted. These types of programs come in all sorts of flavors, including web apps, desktop apps, web browser extensions, and more.

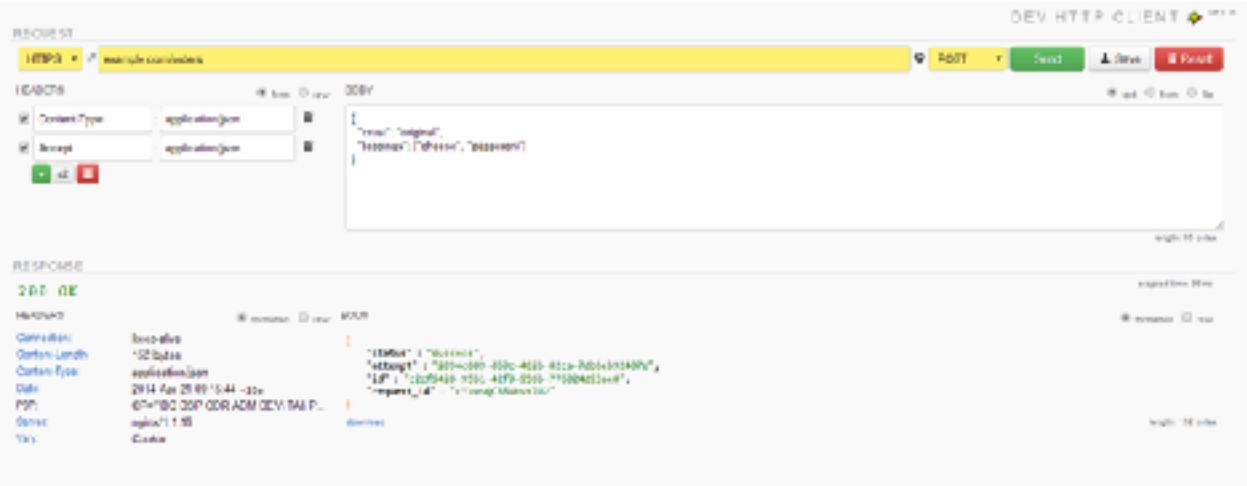


Figure 1. *Screenshot of Dev HTTP Client, a Google Chrome Extension.*

The nice thing about generic HTTP clients is that you do not have to know how to program to use one. With the skills you've attained through this course, you now have the ability to read a company's API documentation and figure out the request you need to make to get the data you want. This small learning curve makes generic clients great for exploration and quick one-off tasks.

There are couple of downsides to this approach, however. First, you usually can't save your work. After you close the program, the requests you made are forgotten and you have to rebuild them the next time you need them. Another disadvantage is that you typically can't do much with the data you get back, other than look at it. At best, you have the ability to save the data into a file, after which it's up to you to do something interesting with it.

Writing Code

To really harness the power of an API, you will eventually need custom software. This is where programming comes in. Being a discipline unto itself, we won't attempt to cover everything about software development, but we can give you some guidance for what writing an API client involves.

The first requirement is to gain some familiarity with a programming language. There are a bunch out there, each with its strengths and weaknesses. For simplicity's sake, it is probably better that you stick to an interpreted language (JavaScript, Python, PHP, Ruby, or similar) instead of a compiled language (C or C++).

If you aren't sure which language to choose, a great way to narrow down the selection can be to find an API you want to implement and see if the company provides a client **library**. A library is code that the API owner publishes that already implements the client side of their

API. Sometimes the library will be individually available for download or it will be bundled in an SDK (Software Development Kit). Using a library saves you time because instead of reading the API documentation and forming raw HTTP requests, you can simply copy and paste a few lines of code and already have a working client.

After you settle on a language, you need to decide where the code will run. If you are automating your own tasks, running the software from your work computer might be acceptable. More frequently, you will want to run the code on a computer better suited for acting as a web server. There are quite a few solutions available, including running your code on shared hosting environments, cloud services (like Amazon Web Services), or even on your own physical servers at a data center.

A third important decision is to figure out what you will do with the data. Saving results into a file is easy enough, but if you want to store the data in a database or send it to another application, things become more complex. Pulling data out of a database to send to an API can also be challenging.

At this point we can pause and remind you to not be too intimidated by all this new information. You should not expect to know everything about implementing APIs on your first attempt. Take solace knowing that there are people who can help (open source communities, developers for-hire, and potential project collaborators) and lots of resources available online to facilitate learning.

Once you master the basics, there are plenty more topics to learn about in the rich realm of software development. For now, if you succeed at learning a programming language and getting a library up and running, you should celebrate. You will be well on your way to making the most of APIs!

Give Zapier A Try

If coding is beyond your current skill set or time constraints, there is a nifty tool we know of that empowers you to easily interact with APIs. OK, you probably saw this coming: it's Zapier! Our Developer Platform offers a way for you to implement an API that you then interact with as an app on Zapier. Through button clicks and filing out forms, you can implement nearly any API you want.

What makes using the Developer Platform easy is that we have done a lot of the programming for you. Zapier has code in place to make requests, all you have to do is fill in the details. Think of using the platform a bit like using a generic HTTP Client; you tell us a bit about the endpoints and we'll do the rest. The additional benefit is that once you have Zapier talking with an API, you have lots of options for what to do with the data you get back. Also, if you get stuck, you are welcome to reach out to the friendly support team, where you have API experts ready to help you out.

Conclusion

We've come a long way in this course. We started with questions like "What is a client?" and end with ideas for how to build them. Even if you decide not to try your hand at implementing an API, we hope you feel comfortable when they come up in conversation. You now have a sense of what an API is, what it does, and how you can leverage one to benefit your business.

Maybe you run a company and see the value in providing an API to your customers. Or perhaps you regularly do pesky, time-consuming tasks that you are excited to have automated. Whatever the case may be, we hope you find the knowledge you've gained valuable. Please share this course with anyone you think could benefit from it and spread the word that APIs are awesome!

Chapter 8 Recap

In this chapter, we discussed how an API design becomes working software. We talked about ways that you can begin using APIs.

The key terms we learned were:

- **Implement:** Writing software that obeys the rules of an API
- **Documentation:** Webpages, PDF's, etc. that explain the rules of an API
- **Library:** Code released by an API publisher that implements the client portion of their API

Homework

Think about ways you might be able to use an API in your working life. To get the juices going, here are a few ideas:

- You need some quick stats from a SaaS (Software as a Service) application you use. Firing up an HTTP Client to make a few requests could be a fast way to get the information you need.
- You have a labor-intensive task that needs to get done and there isn't time to have a developer friend lend a hand. Grabbing a client library and creating a quick program could be a big timesaver.
- You really want to move data between two SaaS apps on a continual basis, but you don't have the resources to build a client for each app from scratch, nor a good place to run that code. Using the Zapier Developer Platform could be a low-cost way to get the applications connected.

The End

This concludes "*An Introduction to APIs*", a free educational course brought to you by Zapier. We hope you've enjoyed reading it. If you think someone else might benefit from this material, please do share.

You can find "*An Introduction to APIs*" for free online and share it at:

https://zapier.com/learn/apis/

This instructional course was crafted for you with love by Zapier.