APIS

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. Makes data easier to work with.

**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 sitting, waiting for others to ask it for data.

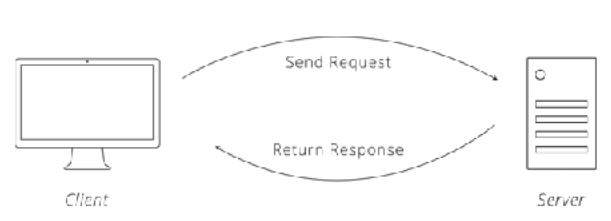
The other side is the "client." This is a separate program that knows what data is available through the API and can manipulate it, typically at the request of a user. A great example is a smartphone app that syncs with a website. When you push the refresh button your app, it talks to a server via an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

**Knowing the Rules**

People create social etiquette to guide their interactions. One example is how we talk to each other on the phone. 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. For two computers to communicate effectively, the server has to know exactly how the client will arrange its messages.

**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.



**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.

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

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

**URLS**

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

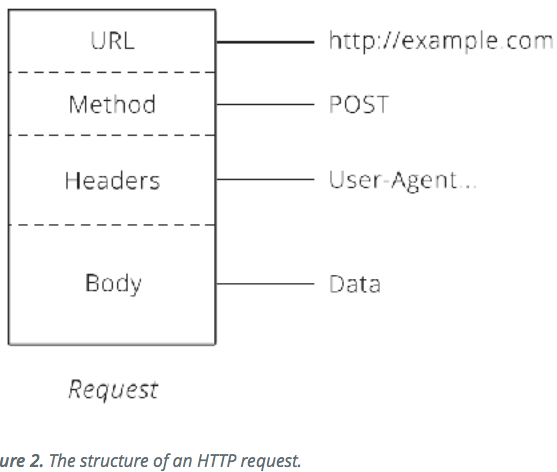
**Headers**

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.

**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.



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:

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 w

*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*.

REPRESENTING DATA

*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).

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](https://zapier.com/learn/apis/chapter-3-data-formats/#footnote-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).

## 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.

Things are starting to pick up in our understanding of APIs. We know who the [client and server](https://zapier.com/learn/apis/chapter-1-introduction/) are, we know they use [HTTP](https://zapier.com/learn/apis/chapter-2-protocols/) to talk to each other, and we know they speak in specific [data formats](https://zapier.com/learn/apis/chapter-3-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.

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).

Basic Auth only requires a username and password. The client takes these two credentials, smooshes them together to form a single value [1](https://zapier.com/learn/apis/chapter-4-authentication-part-1/#footnote-1), and passes that along in the request in an HTTP header called **Authorization**.

Though Basic Auth is a valid authentication scheme, the fact that it uses the 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.

## OAuth 2

Forcing users to work with API keys is a similarly poor experience. Typos are a common problem. 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.

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

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. 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.

ORGANIZING DATA

Companies give similar thought to organization when building their APIs. As we mentioned in [Chapter 1](https://zapier.com/learn/apis/chapter-1-introduction-to-apis/), 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](https://zapier.com/learn/apis/chapter-6-api-design/#footnote-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.

Back in [Chapter 2](https://zapier.com/learn/apis/chapter-2-protocols/), 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.

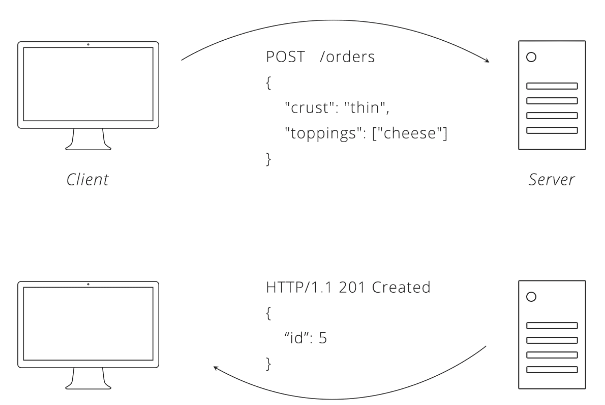
 Example API: In this context, menu, pizza, customer, and order all sound like good candidates for resources.

The next step is assigning URLs to the resource. 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>.

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 |



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

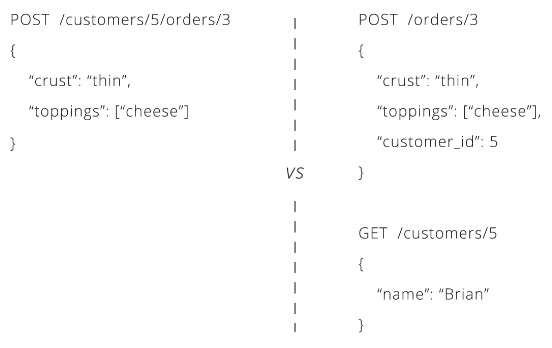
## Linking Resources Together

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.

But 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.

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&crust=thin>.

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.

he 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

## Integrations

To set the stage for our discussion, let's remind ourselves why APIs are useful. Back in [Chapter 1](https://zapier.com/learn/apis/chapter-1-introduction-to-apis/) 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.

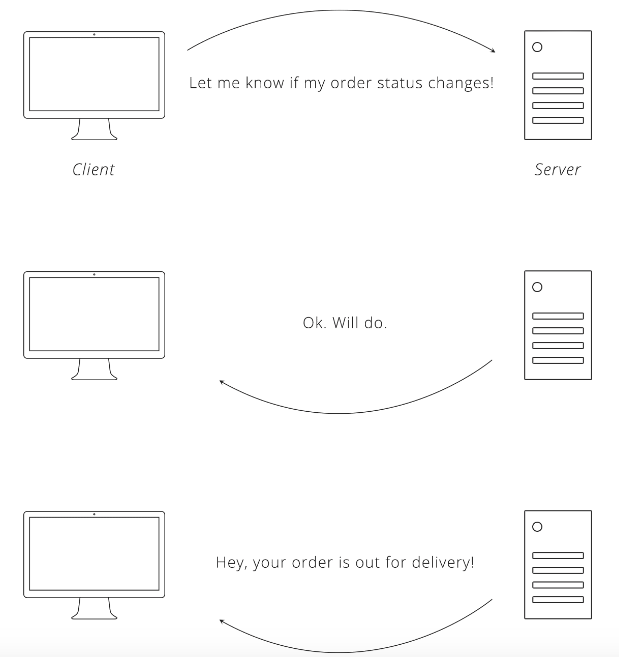
For our purposes, we will split integrations into two broad categories[1](https://zapier.com/learn/apis/chapter-7-real-time-communication/#footnote-1). The first we call "client-driven," where a person interacts with the client and wants the server's data to update. The other we call "server-driven", 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.

### Webhooks

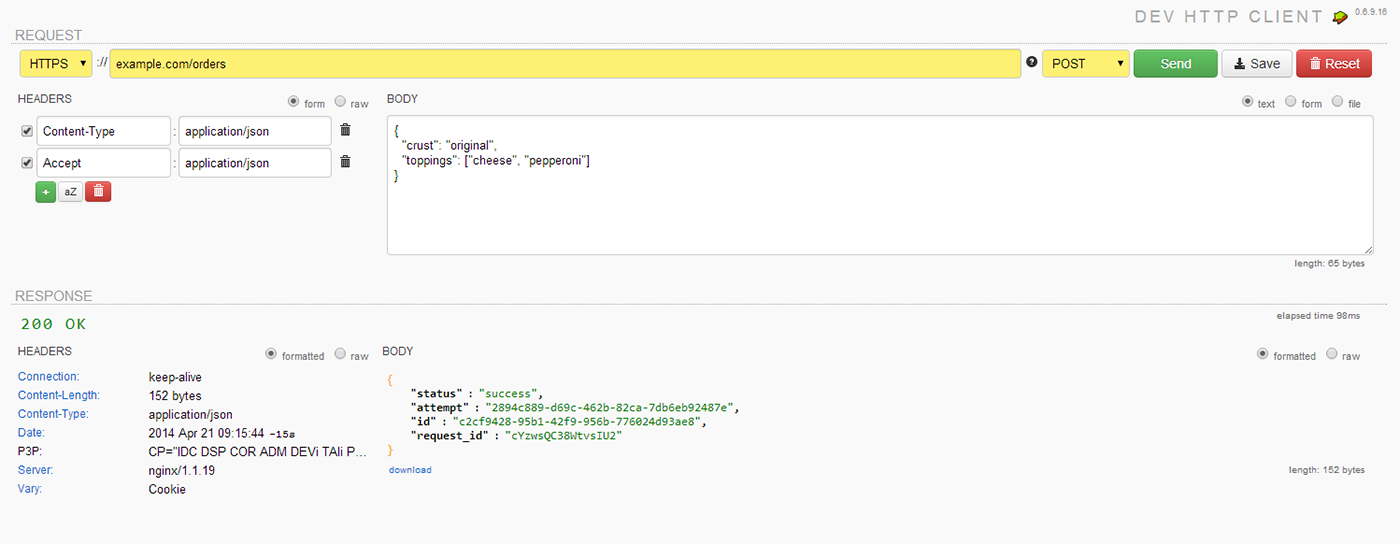
With polling ruled out, some innovative software developers thought, "if all our trouble is because the client is the only one making requests, why not remove that rule?" So they did. The result was **webhooks**, a technique where the client both makes requests and listens for them, allowing the server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



## 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*](https://chrome.google.com/webstore/detail/dev-http-client/aejoelaoggembcahagimdiliamlcdmfm?hl=en&utm_source=zapier.com&utm_medium=referral&utm_campaign=zapier)*, 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.

## Writing Code

To really harness the power of an API, you will eventually need custom software.

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.