Now that you know what an API is, let's talk about what makes up a **REST API.** We'll be using REST in this course because it is incredibly popular in the software development industry. It is one of the most logical, efficient, and widespread API creation standards.  And according to [Cloud Elements’ 2017 State of API Integration report](https://jaxenter.com/state-of-api-integration-report-136342.html), 83 percent of APIs now use REST.

REST stands for **Re**presentational **S**tate **T**ransfer and is simply a set of **architectural** **standards** or guidelines that structure how you communicate data between your application and the rest of the world, or between different components of your application.

We use RESTful as the adjective to describe REST APIs. All REST APIs are a type of API - but not all APIs are RESTful!

RESTful APIs rely on **HTTP** to transfer information - the same protocol that web communication is based on! So when you see the **http** in the beginning of a URL, like [**http**://twitter.com](https://twitter.com/) - your browser is using HTTP to request that website from a server. REST works in the same way!

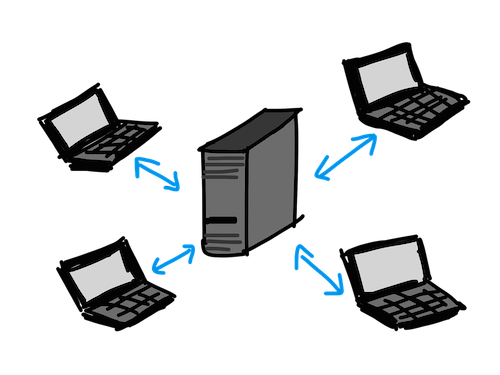
There are a **six** key architectural guidelines to using REST APIs. Let's check them out!

**#1: Client-Server Architecture**

One of the standards of REST is a rule called **separation between client and server.**Now, we spoke a little bit about clients and servers last chapter, but let's flesh out some of the details.

A **client** is a **person** or **software** that uses the API. An example of a client would be you, the developer.  As a client, you might use the Twitter API to view all of Beyonce's tweets.  A client could also be a browser, whether it's Chrome, Safari, or Firefox. When a browser goes to twitter.com, it calls the Twitter API and uses the data from the API so you can view the latest tweets.

Meanwhile, a **server** is a remote computer able to grab data from the database and hand it over to the API, like this big computer in the middle:

A typical client-server relationship

Typically, there's a separation between these two app components. This enables the client to only deal with getting and displaying the information, while the server can focus on storing and manipulating the data.

REST APIs provide a **standardized way of communicating between client and server.**In other words, it doesn't matter how the server is put together or how the client is coded up, as long as they both structure their *communications* according to REST architecture guidelines, using HTTP.  Thanks to REST APIs, any client can communicate with any server, making storing and receiving data super simple! 👌

This is especially useful when you have large teams of developers working on an app. You can have one team work independently on the server-side, and another on the client-side because you have the REST API communicating between them. This enables developers to scale apps more easily and for teams to work more efficiently. 🤝

**#2: Stateless**

Another unique aspect of REST APIs is that they are **stateless** - which means that the server does *not* save any of the previous requests or responses.

But how would that even work?

To return to our metaphor for an API as a waiter, let's say you ask your waiter for french fries. 🍟Your waiter goes to the kitchen, gets your french fries, and comes back with your order. Success!

But wait!  You just remembered you also want salt on those french fries. So you ask your waiter, "Hey, can I get salt with that?" A **stateless** waiter would have no idea what you're talking about...because he wouldn't remember that you've just ordered fries!

So what does that mean for REST APIs?

Because each message is isolated, you would need to make sure to send all the **necessary data** with the request that you make. This could go something like - "Hey, can I get salt on the **french fries** **I** ordered at**my**table?" And then your waiter would be able to identify which fries to add salt to!

**Statelessness**makes each request and response very **purposeful** and **understandable.**So, if you're a developer and you see someone else's API request in existing code, you will be able to understand what the request is for without any other context.

**#3: Cacheable**

In REST, the server response should contain information about whether or not the client can **cache,** or save, the data. If it is **cacheable,** the response should come with a version number. That way, if your user makes the same request twice (i.e., wants to see a page again), your server doesn't have to do double the workload to get all the data. Instead, the client can just cache the data the first time and then reload the same data the second time. 💪

An effective cache can reduce the number of times a client and server need to interact, which can help speed up load time for the user! 👏

You may have heard the word **cache** when referring to something like "Refresh your browser cache." A cache is simply a way to **save**data so future requests can be returned faster. When you visit a bunch of websites on your browser, it can save those requests so it can autocomplete the site you want to go to or load the page faster the next time you visit it.

**#4: Uniform Interface**

When building a REST API, developers agree to use the same standards, so every API has a **uniform interface.** This interface is like a contract between the client and the service that all REST APIs share. It's useful because when developers use APIs, they need global concepts to make sure they can understand each other.

A REST API from one application can communicate in the *same way* to an entirely different application. This makes communication much easier and more efficient.

**#5: Layered System**

Every component that uses REST does not have access to components beyond the specific one it is interacting with. That means a client that connects to an intermediate component has no idea what that component is interacting with afterward. This encourages developers to create independent components, making each one easier to replace or update.

**#6: Code on Demand (Optional)**

Code on demand means that the server can extend its functionality by sending the code to the client to download. This is optional because not all clients will be able to download and run the same code - so this is not usually used.

**REST API Alternatives**

REST is only **one** type of API. There are other alternatives that are also good for you to know, notably **SOAP** APIs.

SOAP stands for**Simple Object Access Protocol.**Unlike REST, it is considered a protocol, not an architectural style. SOAP was the most common API before REST came along. While REST uses HTTP to communicate, SOAP can use multiple means of communication.  This adds a lot of complexity because developers have to coordinate to ensure they're communicating the same way. Additionally, SOAP can require more bandwidth, which leads to slower page load times. REST was created to resolve some of these issues by being lighter and more flexible.

SOAP is currently used more often in enterprise applications because you can add additional layers of security, data privacy, and integrity. REST can be just as secure, but it needs to be implemented instead of being built in like it is with SOAP.

**Let's Recap!**

* Not all APIs are RESTful and REST APIs have specific architecture guidelines.
* Some key advantages to REST APIs are:
  + Separation between client and server, which helps apps be more scalable.
  + Statelessness, which makes API requests very specific and detail-driven.
  + Cacheable, which lets clients save data so they don't have to constantly query servers.
* Another popular API type is SOAP.