APIS FOR THE REST OF US

August 9th, 2021

API HTTP REQUEST POSTMAN

Understanding what APIs are, let alone actually using and getting value out of them, is usually a topic reserved for engineers or otherwise technical people. But that doesn't need to be true! This guide will walk through everything you need to get started, from a technical foundation down to debugging common errors.

Like a database, each API has its own specifics and quirks; no guide is going to cover them all. But use this as a jumping off point, ask engineers when you have questions, and you'll be golden.

WHAT'S AN API EXACTLY?

Applications are just a bunch of functions that get things done: APIs **wrap those functions in easy to use interfaces** so you can work with them without being an expert. Let's start with an example. If you're an e-commerce company, there are a bunch of things you need to get done internally that power your site:

- Show available items and sizes
- Create orders
- Update an email address

All of these tasks are completed through a bunch of code behind the scenes – adding rows to a database, generating and shipping orders, and updating data, in our case. Companies build APIs *on top of* these complex workflows so that *actually using them* is as simple as a few lines of code. Sometimes, an API is just a really simple way to select rows from a database; other times, it can encapsulate thousands of lines of hairy code.

> 🚨 **CONFUSION ALERT** 🚨
>
> The technical definition of an API is *very different* than how people use it in conversation. If this is your first time really trying to grasp the concept, I wrote a full explainer here that might help.

Generally, there are two types of APIs you'll come across: **internal APIs and public APIs**. Your company will have APIs for internal operations (like our e-commerce example) – those aren't available to the public, obviously. They drive business operations. But sometimes, companies with interesting datasets will release public APIs so that developers can build cool stuff on top of their data. A good example is the Twitter API, which lets people like you interact with tweet data programmatically. Even the government has public APIs.

A third type of API that's a bit harder to classify (and getting more popular!) is the **vendor API**. Companies like Stripe sell a product that's basically an API: you use their interface to implement payments into your app. Products like this usually come with some sort of admin panel or frontend to manage the data you've put into them too. These APIs aren't quite available in the open, but they aren't built by you either, so they're neither public nor internal.

COMMON API IMPLEMENTATIONS: SOAP, REST, GRPC

Regardless of what type of API you're dealing with (internal, public, vendor) there are three major protocols / technologies that developers use to build APIs: **SOAP, REST, and gRPC** (yes, obviously they all need to be acronyms). Interacting with each of them brings its own challenges, formats, and tools.

1. **SOAP**

SOAP (Simple Object Access Protocol) is the oldest of the API protocols: it was originally released in 1998 (!) by a few Microsoft engineers. It's based on XML, which sort of looks like HTML. Here's an example of what kind of code you'd need to write to work with it:

```
<?xml version="1.0"?>
<soap:Envelope xmlns:soap="http://www.w3.org/2003/05/soap-envelope" xmln
  <soap:Header>
  </soap:Header>
  <soap:Body>
    <m:GetStockPrice>
      <m:StockName>T</m:StockName>
    </m:GetStockPrice>
  </soap:Body>
</soap:Envelope>
```

If you're thinking this looks…awful, you're not alone. SOAP has all but fallen out of favor, and is generally considered a legacy or enterprise (scary word) kind of technology.

2. REST

REST stands for **Re**presentational **S**tate **T**ransfer (just an awful acronym), and it, too, is a protocol for building APIs. Most APIs you use these days will likely be RESTful in some form or another, but it's not that much younger than SOAP - REST was originally released as a dissertation just two years after SOAP, in 2000. It dictates how you build the systems that power your APIs and what formats they need to adhere to.

The internals of REST are complicated and beyond the scope of this post (i.e. I don't understand them), but there's one important part of it that's worth mentioning: REST is all about **resources**. Every endpoint – the single "unit" of an API – is a URL, which you're probably already used to if you…use the internet. And if you've ever wondered, URL stands for **Uniform Resource Locator**, and this is the "resource" that we're locating.

3. gRPC

REST and SOAP are actually really *old* compared to how quickly everything else in software development changes. The cool kids are talking about something new these days, and it's getting adopted quickly at larger companies with more complex apps. It's called **gRPC** – which loosely stands for **Remote Procedure Call** – and it was originally released by Google in 2015.

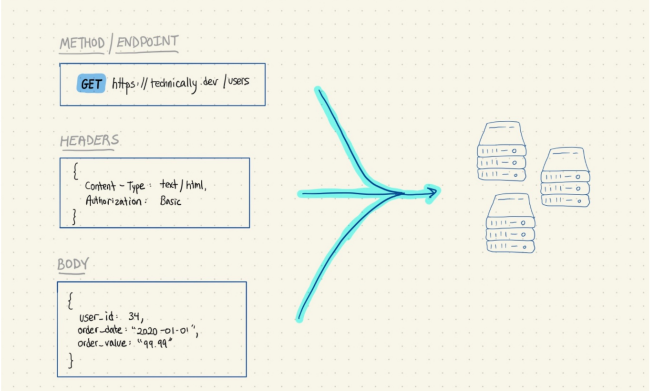
> 🔍 **DEEPER LOOK** 🔍
>
> If you're actually reading this, you may have noticed that gRPC and Remote Procedure Call don't quite match – what's the "g" for? The acronym is actually a joke: it stands for gRPC Remote Procedure Calls, which is a play on recursive humor. Gotta love open source.

gRPC is made mostly for distributed systems that optimize for scale and low latency. If you don't work at a company with those constraints, it's not really something you're going to come across often. You're definitely not going to find public APIs like Twitter using gRPC.

Because most of the APIs you'll use will be RESTful in one way or another, we'll assume as much for the rest of this post.

BASICS OF AN API REQUEST

REST APIs work via HTTP, which means you need to use a request-response model: you make a request with a bunch of details, and the server sends back the data you asked for (or a message telling you that something went wrong). Here's how it breaks down:



1. The endpoint

An endpoint is just a URL, behind which all of the cool, important logic happens that actually gets you the response you need. Here's what our e-commerce endpoints might look like:

- `https://technically.dev/orders` – list orders
- `https://technically.dev/users` – list users

For some requests, literally all you need is the endpoint: there's only one thing you can do with it, and making a request to it will get you the data or operation you need. For an example, check out Github's Jobs API: just paste `https://jobs.github.com/positions.json` into your browser, and you'll see the response (a bunch of JSON).

2. The method

Every REST request needs a method that tells the server exactly what you want. There are 4 popular ones that you'll need to get comfortable with:

- `GET`: read data (e.g. show all orders)
- `POST`: initiate an action on the server (e.g. create an order)
- `PUT`: put some data on the server (e.g. add a new user)
- `DELETE`: you can guess this one

There are a lot of other methods (defined in the HTTP spec), but these are the most common ones. Each API request will need you to choose one of them; some endpoints are method specific, so it's already baked in (that's what's going on with the Github API I just mentioned).

Sometimes, you'll see a `?` after the endpoint with a bunch of text following. That's a **URL parameter**; it helps more specifically identify which "resource" the API is supposed to return. A common use case is to use it for search, where something like `https://technically.dev/users?query=justin` will look for users named Justin.

3. The body

If you're making a POST or PUT request (and for some rare GET requests), there's probably some data that you'll need to associate with it. If you want to create a new order, your API might require a user name, a date, and an order value. You put that information in the **request body** so that the API can parse and use it. Here's an example of what that might look like:

```
{
    user_id: 34,
    order_date: "2020-01-01",
    order_value: "99.99"
}
```

Request bodies are generally formatted as JSON (this format of `name`:`value`).

4. The headers

In addition to the request body, there's a bunch of other higher level information (also called metadata) that you'll sometimes need to include. The most popular header content is authentication information: your API needs you to prove that you're allowed to use it. This is how companies make sure random people don't access their internal APIs! Here's an example of what a header might look like:

```
{
    Content-Type: text/html; charset=UTF-8,
    Authorization: Basic YWxhZGRpbjpvcGVuc2VzYW1l
}
```

This example also sends our server a value called `Content-Type`: it helps clarify to the server what kind of data we're sending over. We'll talk more about authentication down below.

> 🚨 **CONFUSION ALERT** 🚨
>
> If you're confused about what JSON is or how to read this code format we covered in bodies and headers, think of it like you're filling in a form. What's your content type? What authorization are you using? Your values answer the questions.

If you've got these down, you understand everything you need to successfully make an API request. You bundle these things all together – your method, your endpoint, your request body, and your headers – and send it to the server. Because REST endpoints need to resolve into URLs, you can pretty much make these requests anywhere: the next section covers how developers work with and debug their APIs in

practice.

WHERE AND HOW TO CALL APIS

Because an API is just…well…a place, you can make requests from almost anywhere.

1. Browsers

Not all are, but a bunch of APIs are accessible simply through your browser address bar. Sometimes you can include query parameters right through a URL. If we built our API to allow it, you might be able to list the orders made yesterday by hitting `https://technically.dev/orders?order_date=yesterday`. Most developers don't use the browser for making requests consistently.

2. Command Line

`cURL` is a popular command line tool for making API requests: you just choose your method and endpoint, and then attach any extra data like headers or a request body. You'd issue a GET request to our endpoint by typing `curl https://technically.dev/posts` into your terminal, or a POST request with `curl -X POST https://technically.dev/posts`. You can also make the requests directly through the command line without `cURL`, but it's much more annoying.

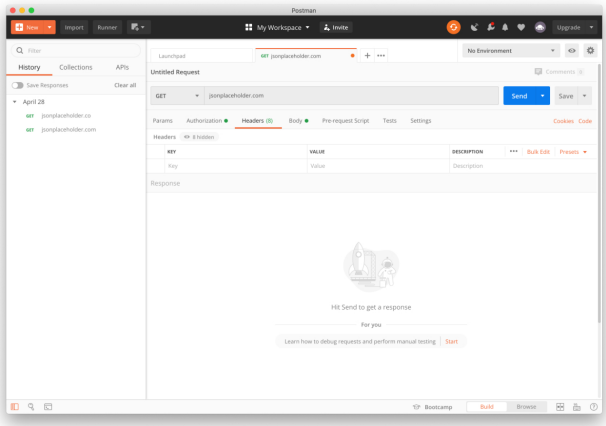
You can try this yourself really easily. Pop open your Terminal and type `curl https://jsonplaceholder.typicode.com/users`. You should see a sample list of users returned.

3. Client Libraries

Internal and vendor APIs often ship with built in functions for specific programming languages. If your company uses mostly Python for the backend, they might create a library to interact with their APIs and build functions like `listOrders()` that you can use directly. Vendor APIs like Stripe or Google AdWords offer multiple of these (Python, Javascript, Ruby, etc.).

4. IDEs

Like with the rest of the programming ecosystem, there are specialized tools for querying REST endpoints. The most popular one out there is Postman: it organizes your request into simple form fields, helps autocomplete your headers and body, and lets you store credentials and settings.



THE ADVANCED STUFF: AUTHENTICATION, ERROR HANDLING, AND SUCH

You can go pretty deep down the API rabbit hole, and developers who have been working with them for decades will probably know things…terrible things. Here are a few areas to dive deeper into yourself:

1. Authentication

Hands down, the most frustrating and time consuming part of building and working with APIs is authentication. APIs need to be secure, especially when they're handling sensitive customer data. But since every HTTP API works through request-response, you need to authenticate yourself through your actual request (at least, at first). There are two popular authentication "schemes" built into HTTP: **basic and bearer**.

Basic auth for HTTP APIs is just username and password. You include a username and password in your request header, and the API checks to make sure you are who you

password in your request header, and the API checks to make sure you are who you say you are. The values are encoded though, so they don't quite look like a username and password; here's an example:

```
Authorization: Basic bG9sOnNlY3VyZQ==
```

Basic auth is popular, but it's a little less than ideal when it comes to security.

Bearer auth follows a similar pattern – the request shows something to the server that "proves" you should have access – but instead of username and password, it's a special string of characters called a **token**. To get that token, you'll often need to log in elsewhere. Here's an example:

```
Authorization: Bearer AbCdEf123456
```

There are other auth schemes that *make use of* bearer authentication (like I said, it's complicated). A commonly used one that you should know is OAuth. **OAuth** is a scheme for authenticating apps or APIs: you set up a special server (or endpoint) for authorizing users, and then that server (or endpoint) forwards along information saying "hey, this user is authorized."

> ⛓ **RELATED CONCEPTS** ⛓
>
> Another common auth scheme – most popular among larger, more secure enterprises – is SAML (Security Assertion Markup Language). If you've heard of Okta or OneLogin, they offer a product that simplifies authentication and can work via SAML.

OAuth was actually included among the original HTTP auth schemes in the spec, but was recently revamped into version 2.0, which has gotten real popular. If you want to dive deeper, this unofficial OAuth site is a fantastic start.

2. Error handling

You will format your requests incorrectly, your authentication won't work, you'll use the wrong endpoint, and your server will have internal problems – and you will get errors. The more you interact with APIs, the more comfortable you'll get with interpreting and fixing these errors. HTTP APIs follow a standard for error messaging with status codes: if your response's status code starts with a 4, it's probably an error. A few common ones:

- `400` **(Bad Request)** – your request is probably formatted incorrectly
- `403` **(Forbidden)** – you're not authorized to access this endpoint
- `404` **(Not Found)** – your endpoint doesn't correspond to a resource on the server
- `405` **(Method Not Allowed)** – your method (e.g. GET) isn't allowed for the endpoint you're using

In practice, your errors are probably going to be uglier and more obscure than clean HTTP problems. Google and Stack Overflow are your friends. You can also scream into your pillow.

3. Caching

One of the core principles of REST is that responses should often be **cache-able** – instead of the server having to interpret your request and whip up a new response every time, it should be able to **store premade responses along the way** (on your computer, in a closer data center, etc.) to speed things up. You can read more about caching HTTP APIs here.

If you're not building the API yourself, you won't need to worry about caching all that often. There is one use case you might run into: if recent changes were made to an API but the responses you're getting don't seem to match those changes, you might need to reset the cache. Ask your local certified developer™.

WHERE TO LEARN MORE

The best way to get better at working with APIs is to – gasp – work with APIs. If you're gainfully employed, ask your engineers to help you practice on internal endpoints. You can also use public APIs (great list here) to get started. Reading is good too, though; here are some resources to help you along your path.

–> Courses / classes

- WebServices/Rest API Testing with SoapUI (Udemy)
- Programming Foundations: APIs and Web Services (Lynda)
- Designing RESTful APIs (Udacity)
- REST APIs with Flask and Python (Udemy)

–> Tutorials

- A beginner's guide to using the Twitter API

- Understanding and using REST APIs (Smashing Magazine)
- REST API Tutorial – REST Client, REST Service, and API Calls Explained With Code Examples (FreeCodeCamp)

→ Tools and IDEs

- Postman (IDE)
- Insomnia (IDE)
- Swagger (for designing APIs and writing documentation)
- Mulesoft API Designer (IDE)

If you liked or hated this, share it on Twitter, Reddit, or HackerNews.
