Technically, API stands for **Application Programming Interface**. At some point or another, most large ***companies have built APIs*** for ***their customers***, or for ***internal use***.

But how do you explain API in plain English? And is there a broader meaning than the one used in development and business? First, let’s pull back and look at how the web itself works.

**WWW** and **remote servers** [**Remote Server** is ***just part*** of ***remotely located computer*** that is***optimised*** to ***process requests***].

**Web** means we can imagine a large network of **connected *servers****.*

**Every page** on the internet is **stored somewhere** on a **remote server**. A **remote server** is not so mystical after all — it’s just a **part of a remotely located computer** that is optimized to ***process requests***.

To put things in perspective, you can spin up a **server** **on your laptop capable** of **serving an entire website to the Web** (**in fact, a *local* server is what engineers use to develop websites before releasing them to the public**).

When you type [www.facebook.com](https://www.facebook.com/) into your browser, **a request goes out** to Facebook’s **remote server**. Once your **browser receives** the **response**, it **interprets** the **code** and **displays** the page.

To the browser, also known as the ***client***, Facebook’s ***server*** is an ***API***. This means that **every time** you **visit** a **page** on the **Web**, you **interact** with some **remote server’s API**.

An ***API*** ***isn’t*** the ***same as*** the **remote server** — rather it is the ***part*** of the **server** that **receives requests** and **sends responses**.

[**\*\*** Notes: **APIs** as a **way** to **serve** your **customers,** it is **not same** as a **remote server,** butit is the **part** of the **server** that **receives request** and **sends responses]**

You’ve probably heard of **companies packaging APIs** as **products**. ( For example, Weather Underground sells access to its [weather data API](https://www.wunderground.com/weather/api)).

Example scenario: Your small business’s website has a form used to sign clients up for appointments. You want to give your clients the ability to automatically create a Google calendar event with the details for that appointment.

**API use:** The idea is to have your website’s server talk directly to Google’s server with a request to create an event with the given details. Your server would then receive Google’s response, process it, and send back relevant information to the browser, such as a confirmation message to the user.

Alternatively, your browser can often send an API request directly to Google’s server bypassing your server.

How is this Google Calendar’s API different from the API of any other remote server out there?

**In technical terms**, the ***difference*** is the ***format*** of the **request** and the **response**.

To render the whole web page, your browser expects a response in *HTML,*which contains presentational code, while Google Calendar’s API call would just return the data — likely in a format like *JSON*.

If your website’s server is making the API request, then your website’s server is the client (similar to your browser being the client when you use it to navigate to a website).

**From your users perspective,** APIs allow them to complete the action without leaving your website.

Most modern websites consume at least some third-party APIs.

Many problems already have a third-party solution, be it in the form of a library or service. It’s often just easier and more reliable to use an existing solution.

It’s not uncommon for **development teams** to **break up** their **application** into **multiple servers** that **talk to each** other via **APIs**. The **servers** that perform ***helper functions*** for the **main application server** are commonly referred to as [***microservices***](http://microservices.io/patterns/microservices.html)*.*

To summarize, when a ***company offers*** an **API** to ***their customers***, it just **means** that they’ve ***built a set of dedicated URLs*** that ***return pure data responses*** — meaning the **responses won’t** contain the kind of **presentational overhead** that you would expect in a **graphical user interface** like a website.

Can you make these requests with your browser? Often, yes. Since the actual HTTP transmission happens in text, your browser will always do the best it can to display the response.

For example, you can access GitHub’s API directly with your browser without even needing an access token. Here’s the JSON response you get when you visit a GitHub user’s API route in your browser (<https://api.github.com/users/petrgazarov>):

{  
 "login": "petrgazarov",  
 "id": 5581195,  
 "avatar\_url": "https://avatars.githubusercontent.com/u/5581195?v=3",  
 "gravatar\_id": "",  
 "url": "https://api.github.com/users/petrgazarov",  
 "html\_url": "https://github.com/petrgazarov",  
 "followers\_url": "https://api.github.com/users/petrgazarov/followers",  
 "following\_url": "https://api.github.com/users/petrgazarov/following{/other\_user}",  
 "gists\_url": "https://api.github.com/users/petrgazarov/gists{/gist\_id}",  
 "starred\_url": "https://api.github.com/users/petrgazarov/starred{/owner}{/repo}",  
 "subscriptions\_url": "https://api.github.com/users/petrgazarov/subscriptions",  
 "organizations\_url": "https://api.github.com/users/petrgazarov/orgs",  
 "repos\_url": "https://api.github.com/users/petrgazarov/repos",  
 "events\_url": "https://api.github.com/users/petrgazarov/events{/privacy}",  
 "received\_events\_url": "https://api.github.com/users/petrgazarov/received\_events",  
 "type": "User",  
 "site\_admin": false,  
 "name": "Petr Gazarov",  
 "company": "PolicyGenius",  
 "blog": "http://petrgazarov.com/",  
 "location": "NYC",  
 "email": "petrgazarov@gmail.com",  
 "hireable": null,  
 "bio": null,  
 "public\_repos": 23,  
 "public\_gists": 0,  
 "followers": 7,  
 "following": 14,  
 "created\_at": "2013-10-01T00:33:23Z",  
 "updated\_at": "2016-08-02T05:44:01Z"  
}

The browser seems to have done just fine displaying a JSON response. ***A JSON response is ready for use in your code***. It‘s easy to extract data from this text. Then you can do whatever you want with the data.

**A** is for **“Application”.** some examples of APIs.

“Application” can refer to many things. Here are some of them in the context of API:

* A piece of software with a distinct function.
* The whole server, the whole app, or just a small part of an app.
* Basically, any piece of software that can be distinctively separated from its environment, can be an “A” in API, and will probably also have some sort of API.

Let’s say you’re using a third-party library in your code. Once incorporated into your code, a library becomes part of your overall app. Being a distinct piece of software, the library would likely have an API which allows it to interact with the rest of your code.

Here’s another example: In **Object Oriented Design**, code is organized into objects. Your application may have hundreds of objects defined that can interact with one another.

Each object has an API — a set of *public* methods and properties that it uses to interact with other objects in your application.

An object may also have inner logic that is *private,*meaning that it’shidden from the outside scope (and not an API).

@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@

**APIs:**

* An API is an application programming interface. It is a set of rules that allow programs to talk to each other.
* The developer creates the API on the server and allows the client to talk to it.

**REST** determines how the API looks like. It stands for “Representational State Transfer”. It is a set of rules that developers follow when they create their API. One of these rules states that you should be able to get a piece of data (called a resource) when you link to a specific URL.

Each **URL** is called a **request** while the **data sent back** to you is called a **response**.

There’s a high chance you came across the term “REST API” if you’ve thought about getting data from another source on the internet, such as Twitter or Github. But what is a REST API? What can it do for you? How do you use it?

What Is A REST API?

Let’s say you’re trying to find videos about Batman on Youtube. You open up Youtube, type “Batman” into a search field, hit enter, and you see a list of videos about Batman. A REST API works in a similar way. You search for something, and you get a list of results back from the service you’re requesting from.

**The Anatomy of a Request:**

It’s important to know that a request is made up of four things:

* The endpoint (URI- REST APIs use URI to address resources).
* The method (Resource methods).
* The headers
* The data (or body)

1. **The endpoint** (or route) is the url you request for. It follows this structure:

root-endpoint/?

The root-endpoint is the starting point of the API you’re requesting from. The root-endpoint of Github’s API is https://api.github.com while the root-endpoint Twitter’s API is https://api.twitter.com.

The path determines the resource you’re requesting for. Think of it like an automatic answering machine that asks you to press 1 for a service, press 2 for another service, 3 for yet another service and so on.

/users/:username/repos

Any colons (:) on a path denotes a variable.

Technically, query parameters are not part of the REST architecture, but you’ll see lots of APIs use them. Query parameters give you the option to modify your request with key-value pairs. They always begin with a question mark (?). Each parameter pair is then separated with an ampersand (&), like this:

?query1=value1&query2=value2

**JSON:**

JSON (JavaScript Object Notation) a common format for sending and requesting data through a REST API.

A JSON object looks like a JavaScript Object. In JSON, each property and value must be wrapped with double quotation marks.

1. **The** **Resource Methods:**

The method is the type of request you send to the server. You can choose from these five types below:

GET

POST

PUT

PATCH

DELETE

These methods provide meaning for the request you’re making. They are used to perform four possible actions: Create, Read, Update and Delete (CRUD).

Method Name Request Meaning

**GET**

This request is used to **get a resource** from **a server**. If you perform a `GET` request, the **server looks for the data you requested** and **sends it back to you**. In other words, a `GET` request performs a `READ` operation. This is the default request method.

**POST**

This request is used to **create a new resource** on a **server**. If you perform a `POST` request, the **server creates a new entry** in the **database** and **tells you whether the creation is successful**. In other words, a `POST` request performs an `CREATE` operation.

**PUT and PATCH**

These two requests are used to **update a resource** on a **server**. If you perform a `PUT` or `PATCH` request, the **server updates an entry** in the **database** and **tells you whether the update is successful**. In other words, a `PUT` or `PATCH` request performs an `UPDATE` operation.

**DELETE**

This request is used to **delete a resource** from a **server**. If you perform a `DELETE` request, the **server deletes an entry** in the **database** and **tells you whether the deletion is successful**. In other words, a `DELETE` request performs a `DELETE` operation.

1. **The Headers:**

Headers are used to **provide information** **to both the client and server**. It can be used for many purposes, such as **authentication** and **providing information** about the **body content**.

HTTP Headers are property-value pairs that are separated by a colon.

1. **The Data (Or “Body”):**

The data (sometimes called “body” or “message”) contains information you want to be sent to the server. This option is only used with POST, PUT, PATCH or DELETE requests.

**Authentication:**

You wouldn’t allow anyone to access your bank account without your permission, would you? On the same line of thought, developers put measures in place to ensure you perform actions only when you’re authorized to do. This prevents others from impersonating you.

Since POST, PUT, PATCH and DELETE requests alter the database, developers almost always put them behind an authentication wall. In some cases, a GET request also requires authentication (like when you access your bank account to check your current balance, for example).

**HTTP Status Codes and Error Messages**

Some of the messages you’ve received earlier, like “Requires authentication” and “Problems parsing JSON” are error messages. They only appear when something is wrong with your request. HTTP status codes let you tell the status of the response quickly. The range from 100+ to 500+.

In general, the numbers follow the following rules:

200+ means the request has succeeded.

300+ means the request is redirected to another URL

400+ means an error that originates from the client has occurred

500+ means an error that originates from the server has occurred

**API Versions:**

Developers update their APIs from time to time. Sometimes, the API can change so much that the developer decides to upgrade their API to another version. If this happens, and your application breaks, it’s usually because you’ve written code for an older API, but your request points to the newer API.

You can request for a specific API version in two ways. Which way you choose depends on how the API is written.

These two ways are:

* Directly in the endpoint
* In a request header

Twitter, for example, uses the first method. At the time of writing, Twitter’s API is at version 1.1, which is evident through its endpoint:

On the web, there are two main ways to authenticate yourself:

* With a username and password (also called basic authentication)
* With a secret token

The secret token method includes oAuth, which lets you to authenticate yourself with social media networks like Github, Google, Twitter, Facebook, etc

**Web API Request/Response Data Formats**:( html, xml, plain text, json, jpeg, pdf etc).

Media Type (aka MIME type) specifies the format of the data as **type/subtype**

e.g. text/html, text/xml, application/json, image/jpeg stc. text/plain

In **HTTP request**, **MIMI type** is specified in the **request header** using **Accept** and **Content-Type** attribute.

The **Accept header attribute** specifies the **format of response data which the client expects,** and the **Content-Type header attribute** specifies the **format of the data in the request body** so that request can parse it into appropriate format.

For example:

if a **client** wants **response data** in **JSON format** the it will **send** following **GET HTTP request** with **Accept header** to the Web API.

**HTTP Get Request:**

GET http://localhost:604464/api/student HTTP/1.1

User-Agent: Fiddler

Host: localhost:1234

Accept: application/json

The same way, if the **client includes JSON data** in the **request body to send** it to the receiver then it will send following **POST HTTP request** with **content-Type header** with JSON data in the body.

**HTTP POST Request:**

POST http://localhost:604464/api/student?age=15 HTTP/1.1

User-Agent: Fiddler

Host: localhost:60464

Content-Type: application/json

Content-Length: 13

{

id:1,

name:'chakra'

}