

[Technologies ▾](#)[References & Guides ▾](#)[Feedback ▾](#)[Sign in](#) 

Introduction to web APIs

[↑ Overview: Client-side web APIs](#)[Next →](#)

First up, we'll start by looking at APIs from a high level — what are they, how do they work, how do you use them in your code, and how are they structured? We'll also take a look at what the different main classes of APIs are, and what kind of uses they have.

Prerequisites: Basic computer literacy, a basic understanding of HTML and CSS, JavaScript basics (see [first steps](#), [building blocks](#), [JavaScript objects](#)).

Objective: To gain familiarity with APIs, what they can do, and how you can use them in your code.

What are APIs?

Application Programming Interfaces (APIs) are constructs made available in programming languages to allow developers to create complex functionality more easily. They abstract more complex code away from you, providing some easier syntax to use in its place.

As a real-world example, think about the electricity supply in your house, apartment, or other dwellings. If you want to use an appliance in your house, you simply plug it into a plug socket and it works. You don't try to wire it directly into the power supply — to do so would be really inefficient and, if you are not an electrician, difficult and dangerous to attempt.

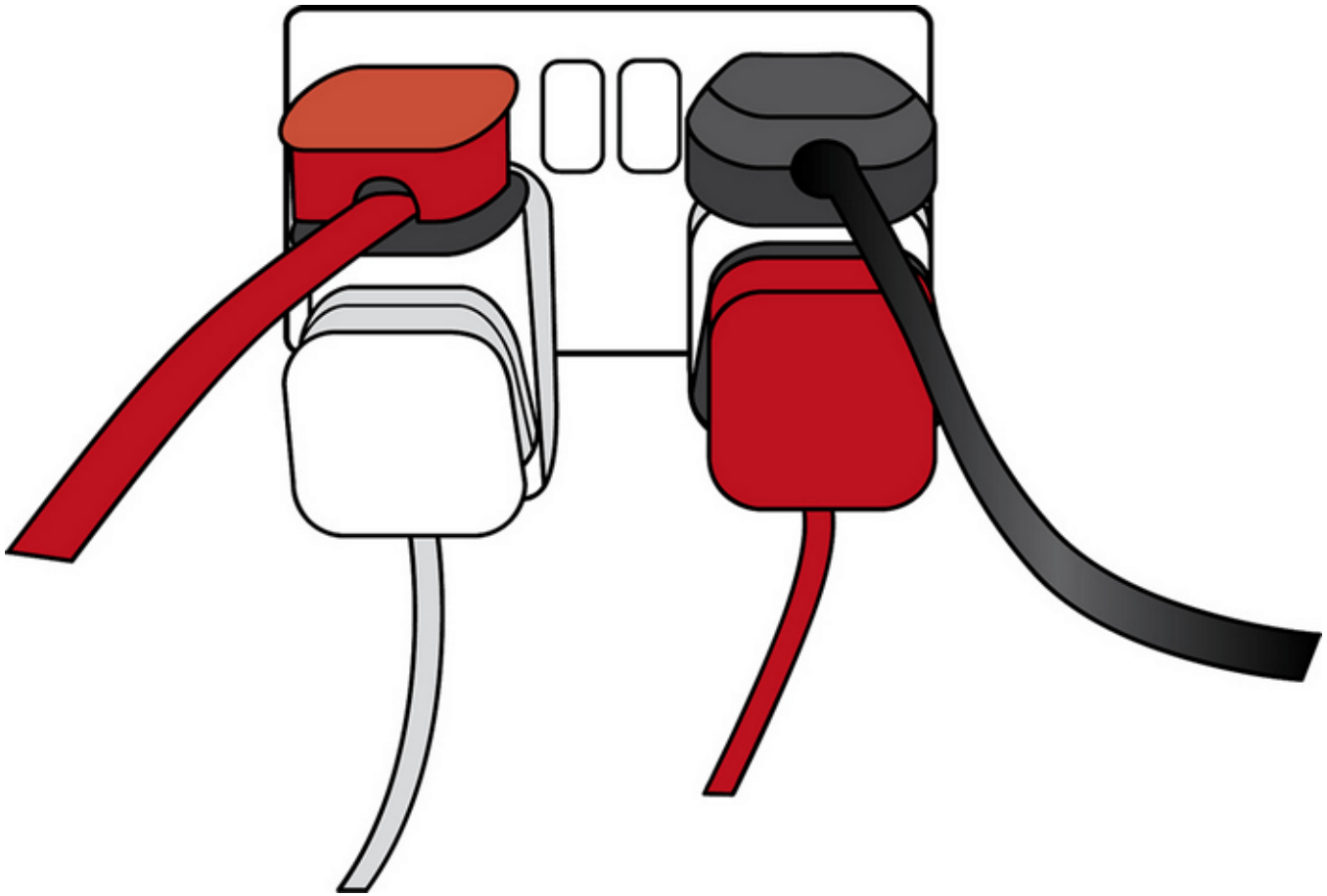


Image source: [Overloaded plug socket](#) by [The Clear Communication People](#), on Flickr.

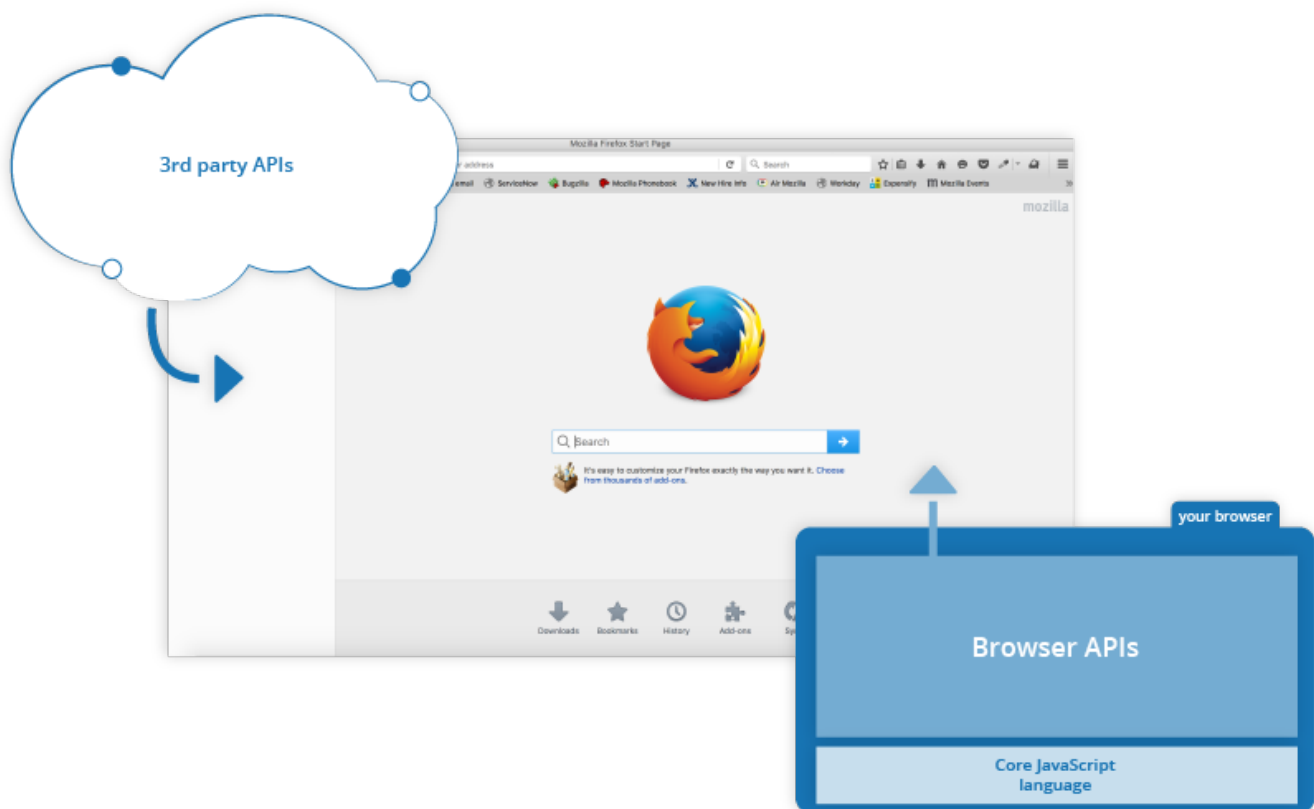
In the same way, if you want to say, program some 3D graphics, it is a lot easier to do it using an API written in a higher level language such as JavaScript or Python, rather than try to directly write low level code (say C or C++) that directly controls the computer's GPU or other graphics functions.

Note: See also the [API glossary entry](#) for further description.

APIs in client-side JavaScript

Client-side JavaScript, in particular, has many APIs available to it — these are not part of the JavaScript language itself, rather they are built on top of the core JavaScript language, providing you with extra superpowers to use in your JavaScript code. They generally fall into two categories:

- **Browser APIs** are built into your web browser and are able to expose data from the browser and surrounding computer environment and do useful complex things with it. For example, the [Geolocation API](#) provides some simple JavaScript constructs for retrieving location data so you can say, plot your location on a Google Map. In the background, the browser is actually using some complex lower-level code (e.g. C++) to communicate with the device's GPS hardware (or whatever is available to determine position data), retrieve position data, and return it to the browser environment to use in your code. But again, this complexity is abstracted away from you by the API.
- **Third party APIs** are not built into the browser by default, and you generally have to grab their code and information from somewhere on the Web. For example, the [Twitter API](#) allows you to do things like displaying your latest tweets on your website. It provides a special set of constructs you can use to query the Twitter service and return specific information.



Relationship between JavaScript, APIs, and other JavaScript tools

So above, we talked about what client-side JavaScript APIs are, and how they relate to the JavaScript language. Let's recap this to make it clearer, and also mention where other JavaScript tools fit in:

- **JavaScript** — A high-level scripting language built into browsers that allows you to implement functionality on web pages/apps. Note that JavaScript is also available in other programming environments, such as [Node](#). But don't worry about that for now.
 - **Browser APIs** — constructs built into the browser that sit on top of the JavaScript language and allow you to implement functionality more easily.
 - **Third party APIs** — constructs built into third-party platforms (e.g. Twitter, Facebook) that allow you to use some of those platform's functionality in your own web pages (for example, display your latest Tweets on your web page).
 - **JavaScript libraries** — Usually one or more JavaScript files containing [custom functions](#) that you can attach to your web page to speed up or enable writing common functionality. Examples include jQuery, Mootools and React.
 - **JavaScript frameworks** — The next step up from libraries, JavaScript frameworks (e.g. Angular and Ember) tend to be packages of HTML, CSS, JavaScript, and other technologies that you install and then use to write an entire web application from scratch. The key difference between a library and a framework is “Inversion of Control”. When calling a method from a library, the developer is in control. With a framework, the control is inverted: the framework calls the developer's code.
-

What can APIs do?

There are a huge number of APIs available in modern browsers that allow you to do a wide variety of things in your code. You can see this by taking a look at the [MDN APIs index page](#).

Common browser APIs

In particular, the most common categories of browser APIs you'll use (and which we'll cover in this module in greater detail) are:

- **APIs for manipulating documents** loaded into the browser. The most obvious example is the [DOM \(Document Object Model\) API](#), which allows you to manipulate HTML and CSS — creating, removing and changing HTML, dynamically applying new styles to your page, etc. Every time you see a popup window appear on a page, or


some new content displayed, for example, that's the DOM in action. Find out more about these types of API in [Manipulating documents](#).

- **APIs that fetch data from the server** to update small sections of a webpage on their own are very commonly used. This seemingly small detail has had a huge impact on the performance and behaviour of sites — if you just need to update a stock listing or list of available new stories, doing it instantly without having to reload the whole entire page from the server can make the site or app feel much more responsive and "snappy". APIs that make this possible include [XMLHttpRequest](#) and the [Fetch API](#). You may also come across the term [Ajax](#), which describes this technique. Find out more about such APIs in [Fetching data from the server](#).
- **APIs for drawing and manipulating graphics** are now widely supported in browsers — the most popular ones are [Canvas](#) and [WebGL](#), which allow you to programmatically update the pixel data contained in an HTML `<canvas>` element to create 2D and 3D scenes. For example, you might draw shapes such as rectangles or circles, import an image onto the canvas, and apply a filter to it such as sepia or grayscale using the Canvas API, or create a complex 3D scene with lighting and textures using WebGL. Such APIs are often combined with APIs for creating animation loops (such as `window.requestAnimationFrame()`) and others to make constantly updating scenes like cartoons and games.
- **Audio and Video APIs** like [HTMLMediaElement](#), the [Web Audio API](#), and [WebRTC](#) allow you to do really interesting things with multimedia such as creating custom UI controls for playing audio and video, displaying text tracks like captions and subtitles along with your videos, grabbing video from your web camera to be manipulated via a canvas (see above) or displayed on someone else's computer in a web conference, or adding effects to audio tracks (such as gain, distortion, panning, etc).
- **Device APIs** are basically APIs for manipulating and retrieving data from modern device hardware in a way that is useful for web apps. We've already talked about the [Geolocation API](#) accessing the device's location data so you can plot your position on a map. Other examples include telling the user that a useful update is available on a web app via system notifications (see the [Notifications API](#)) or vibration hardware (see the [Vibration API](#)).
- **Client-side storage APIs** are becoming a lot more widespread in web browsers — the ability to store data on the client-side is very useful if you want to create an app that will save its state between page loads, and perhaps even work when the device is offline. There are a number of options available, e.g. simple name/value storage with the [Web Storage API](#), and more complex tabular data storage with the [IndexedDB API](#).

Common third-party APIs

Third party APIs come in a large variety; some of the more popular ones that you are likely to make use of sooner or later are:

- The [Twitter API](#), which allows you to do things like displaying your latest tweets on your website.
- The [Google Maps API](#) allows you to do all sorts of things with maps on your web pages (funnily enough, it also powers Google Maps). This is now an entire suite of APIs, which handle a wide variety of tasks, as evidenced by the [Google Maps API Picker](#).
- The [Facebook suite of APIs](#) enables you to use various parts of the Facebook ecosystem to benefit your app, for example by providing app login using Facebook login, accepting in-app payments, rolling out targetted ad campaigns, etc.
- The [YouTube API](#), which allows you to embed YouTube videos on your site, search YouTube, build playlists, and more.
- The [Twilio API](#), which provides a framework for building voice and video call functionality into your app, sending SMS/MMS from your apps, and more.


 **Note:** You can find information on a lot more 3rd party APIs at the [Programmable Web API directory](#).

How do APIs work?

Different JavaScript APIs work in slightly different ways, but generally, they have common features and similar themes to how they work.

They are based on objects

APIs are interacted with in your code using one or more [JavaScript objects](#), which serve as containers for the data the API uses (contained in object properties), and the functionality the API makes available (contained in object methods).

 **Note:** If you are not already familiar with how objects work, you should go back and work through our [JavaScript objects](#) module before continuing.

Let's return to the example of the Geolocation API — this is a very simple API that consists of a few simple objects:

- [Geolocation](#), which contains three methods for controlling the retrieval of geodata.

- **Position**, which represents the position of a device at a given time — this contains a **Coordinates** object that contains the actual position information, plus a timestamp representing the given time.
- **Coordinates**, which contains a whole lot of useful data on the device position, including latitude and longitude, altitude, velocity and direction of movement, and more.

So how do these objects interact? If you look at our [maps-example.html](#) example ([see it live also](#)), you'll see the following code:

```
1  navigator.geolocation.getCurrentPosition(function(position) {
2    var latlng = new google.maps.LatLng(position.coords.latitude,position.co
3    var myOptions = {
4      zoom: 8,
5      center: latlng,
6      mapTypeId: google.maps.MapTypeId.TERRAIN,
7      disableDefaultUI: true
8    }
9    var map = new google.maps.Map(document.querySelector("#map_canvas"), myO
10  });
```

Note: When you first load up the above example, you should be given a dialog box asking if you are happy to share your location with this application (see the [They have additional security mechanisms where appropriate](#) section later in the article). You need to agree to this to be able to plot your location on the map. If you still can't see the map, you may need to set your permissions manually; you can do this in various ways depending on what browser you are using; for example in Firefox go to *> Tools > Page Info > Permissions*, then change the setting for *Share Location*; in Chrome go to *Settings > Privacy > Show advanced settings > Content settings* then change the settings for *Location*.

We first want to use the `Geolocation.getCurrentPosition()` method to return the current location of our device. The browser's `Geolocation` object is accessed by calling the `Navigator.geolocation` property, so we start off by using


```
1  navigator.geolocation.getCurrentPosition(function(position) { ... });
```

This is equivalent to doing something like

```
1 | var myGeo = navigator.geolocation;  
2 | myGeo.getCurrentPosition(function(position) { ... });
```

But we can use the dot syntax to chain our property/method access together, reducing the number of lines we have to write.

The `Geolocation.getCurrentPosition()` method only has a single mandatory parameter, which is an anonymous function that will run when the device's current position has been successfully retrieved. This function itself has a parameter, which contains a `Position` object representing the current position data.

 **Note:** A function that is taken by another function as an argument is called a `callback function`.

This pattern of invoking a function only when an operation has been completed is very common in JavaScript APIs — making sure one operation has completed before trying to use the data the operation returns in another operation. These are called **asynchronous operations**. Because getting the device's current position relies on an external component (the device's GPS or other geolocation hardware), we can't guarantee that it will be done in time to just immediately use the data it returns. Therefore, something like this wouldn't work:

```
1 | var position = navigator.geolocation.getCurrentPosition();  
2 | var myLatitude = position.coords.latitude;
```



If the first line had not yet returned its result, the second line would throw an error, because the position data would not yet be available. For this reason, APIs involving asynchronous operations are designed to use callback functions, or the more modern system of `Promises`, which were made available in ECMAScript 6 and are widely used in newer APIs.

We are combining the Geolocation API with a third party API — the Google Maps API — which we are using to plot the location returned by `getCurrentPosition()` on a Google Map. We make this API available on our page by linking to it — you'll find this line in the HTML:

```
1 | <script type="text/javascript" src="https://maps.google.com/maps/api/js?ke
```


To use the API, we first create a `LatLng` object instance using the `google.maps.LatLng()` constructor, which takes our geolocated `Coordinates.latitude` and `Coordinates.longitude` values as parameters:

```
1 | var latlng = new google.maps.LatLng(position.coords.latitude,position.coords.longitude);
```

This object is itself set as the value of the `center` property of an options object that we've called `myOptions`. We then create an object instance to represent our map by calling the `google.maps.Map()` constructor, passing it two parameters — a reference to the `<div>` element we want to render the map on (with an ID of `map_canvas`), and the options object we defined just above it.

```
1 | var myOptions = {  
2 |   zoom: 8,  
3 |   center: latlng,  
4 |   mapTypeId: google.maps.MapTypeId.TERRAIN,  
5 |   disableDefaultUI: true  
6 | }  
7 |  
8 | var map = new google.maps.Map(document.querySelector("#map_canvas"), myOptions);
```

With this done, our map now renders.

This last block of code highlights two common patterns you'll see across many APIs. First of all, API objects commonly contain constructors, which are invoked to create instances of those objects that you'll use to write your program. Second, API objects often have several options available that can be tweaked to get the exact environment you want for your program. API constructors commonly accept options objects as parameters, which is where you'd set such options.

Note: Don't worry if you don't understand all the details of this example immediately. We'll cover using third party APIs in a lot more detail in a future article.

They have recognizable entry points

When using an API, you should make sure you know where the entry point is for the API. In The Geolocation API, this is pretty simple — it is the `Navigator.geolocation` property,

which returns the browser's `Geolocation` object that all the useful geolocation methods are available inside.

The Document Object Model (DOM) API has an even simpler entry point — its features tend to be found hanging off the `Document` object, or an instance of an HTML element that you want to affect in some way, for example:


```
1 | var em = document.createElement('em'); // create a new em element
2 | var para = document.querySelector('p'); // reference an existing p element
3 | em.textContent = 'Hello there!'; // give em some text content
4 | para.appendChild(em); // embed em inside para
```

Other APIs have slightly more complex entry points, often involving creating a specific context for the API code to be written in. For example, the Canvas API's context object is created by getting a reference to the `<canvas>` element you want to draw on, and then calling its `HTMLCanvasElement.getContext()` method:

```
1 | var canvas = document.querySelector('canvas');
2 | var ctx = canvas.getContext('2d');
```

Anything that we want to do to the canvas is then achieved by calling properties and methods of the content object (which is an instance of `CanvasRenderingContext2D`), for example:

```
1 | Ball.prototype.draw = function() {
2 |   ctx.beginPath();
3 |   ctx.fillStyle = this.color;
4 |   ctx.arc(this.x, this.y, this.size, 0, 2 * Math.PI);
5 |
6 |   ctx.fill();
   | };
```

 **Note:** You can see this code in action in our [bouncing balls demo](#) (see it [running live](#) also).


They use events to handle changes in state

We already discussed events earlier on in the course, in our [Introduction to events](#) article — this article looks in detail at what client-side web events are and how they are used in your code. If you are not already familiar with how client-side web API events work, you should go and read this article first before continuing.

Some web APIs contain no events, but some contain a number of events. The handler properties that allow us to run functions when events fire are generally listed in our reference material in separate "Event handlers" sections. As a simple example, instances of the `XMLHttpRequest` object (each one represents an HTTP request to the server to retrieve a new resource of some kind) have a number of events available on them, for example the `load` event is fired when a response has been successfully returned containing the requested resource, and it is now available.

The following code provides a simple example of how this would be used:

```
1  var requestURL = 'https://mdn.github.io/learning-area/javascript/oojs/json
2  var request = new XMLHttpRequest();
3  request.open('GET', requestURL);
4  request.responseType = 'json';
5  request.send();
6
7  request.onload = function() {
8      var superHeroes = request.response;
9      populateHeader(superHeroes);
10     showHeroes(superHeroes);
11 }
```

 **Note:** You can see this code in action in our [ajax.html](#) example ([see it live](#) also).

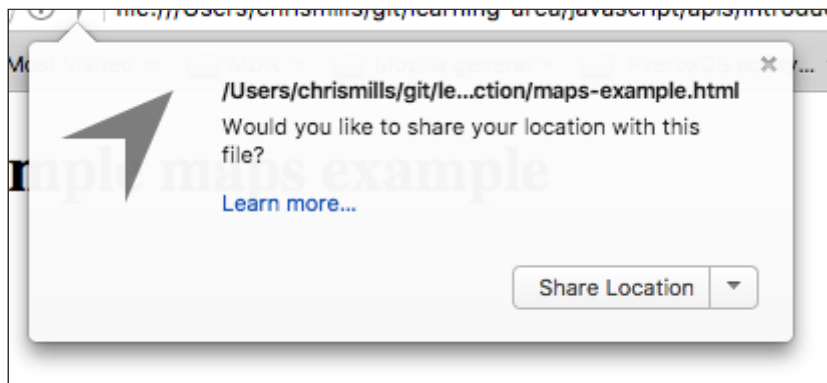
The first five lines specify the location of resource we want to fetch, create a new instance of a request object using the `XMLHttpRequest()` constructor, open an HTTP GET request to retrieve the specified resource, specify that the response should be sent in JSON format, then send the request.

The `onload` handler function then specifies what we do with the response. We know the response will be successfully returned and available after the load event has required (unless an error occurred), so we save the response containing the returned JSON in the `superHeroes` variable, then pass it to two different functions for further processing.

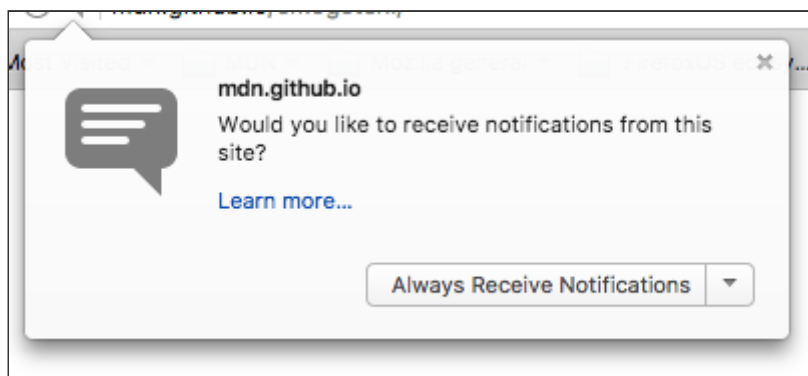
They have additional security mechanisms where appropriate

WebAPI features are subject to the same security considerations as JavaScript and other web technologies (for example [same-origin policy](#)), but they sometimes have additional security mechanisms in place. For example, some of the more modern WebAPIs will only work on pages served over HTTPS due to them transmitting potentially sensitive data (examples include [Service Workers](#) and [Push](#)).

In addition, some WebAPIs request permission to be enabled from the user once calls to them are made in your code. As an example, you may have noticed a dialog like the following when loading up our earlier [Geolocation](#) example:



The Notifications API asks for permission in a similar fashion:



These permission prompts are given to users for security — if they weren't in place, then sites could start secretly tracking your location without you knowing it, or spamming you with a lot of annoying notifications.

Summary

At this point, you should have a good idea of what APIs are, how they work, and what you can do with them in your JavaScript code. You are probably excited to start actually doing some fun things with specific APIs, so let's go! Next up, we'll look at manipulating documents with the Document Object Model (DOM).

[↑ Overview: Client-side web APIs](#)[Next →](#)

In this module

- Introduction to web APIs
 - Manipulating documents
 - Fetching data from the server
 - Third party APIs
 - Drawing graphics
 - Video and audio APIs
 - Client-side storage
-