Client-side web APIs

When writing client-side JavaScript for websites or applications, you will quickly encounter Application Programming Interfaces (APIs). APIs are programming features for manipulating different aspects of the browser and operating system the site is running on, or manipulating data from other websites or services. In this module, we will explore what APIs are, and how to use some of the most common APIs you'll come across often in your development work.

Introduction to web APIs

First up, we'll start by looking at APIs from a high level — what are they, how do they work, how to 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.

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. If you want to use an appliance in your house, you 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.

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.

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 Web Audio API provides JavaScript constructs for manipulating audio in the browser — taking an audio track, altering its volume, applying effects to it, etc. In the background, the browser is actually using some complex lower-level code (e.g. C++ or Rust) to do the actual audio processing. 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 retrieve 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 allow you to implement functionality on web pages/apps. Note that JavaScript is also available in other programming environments, such as Node.
- Browser APIs — constructs built into the browser that sits on top of the JavaScript language and allows 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.

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.
- **APIs that fetch data from the server** to update parts of a webpage without reloading the entire page. The main API used for this is the Fetch API, although older code might still use the XMLHttpRequest API. You may also come across the term **Ajax**, which describes this technique.
- **APIs for drawing and manipulating graphics** are 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.
- **Audio and Video APIs** allow manipulation and control of multimedia elements like audio and video, including adding effects and custom UI controls.
- **Device APIs** enable you to interact with device hardware: for example, accessing the device GPS to find the user's position using the Geolocation API.
- **Client-side storage APIs** allow storing data on the client-side, enabling offline functionality and state persistence between page loads.

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.
- Map APIs, like Mapquest and the Google Maps API, which allow you to do all sorts of things with maps on your web pages.
- The Facebook suite of APIs, which enables you to use various parts of the Facebook ecosystem to benefit your app, such as by providing app login using Facebook login, accepting in-app payments, rolling out targeted 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.

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

Your code interacts with APIs 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).

Let's return to the example of the Web Audio API — this is a fairly complex API, which consists of a number of objects. The most obvious ones are:

- `AudioContext`, which represents an audio graph that can be used to manipulate audio playing inside the browser, and has a number of methods and properties available to manipulate that audio.
- `MediaElementAudioSourceNode`, which represents an `<audio>` element containing sound you want to play and manipulate inside the audio context.
- `AudioDestinationNode`, which represents the destination of the audio, i.e. the device on your computer that will actually output it — usually your speakers or headphones.

So how do these objects interact? In this web audio example, we have an HTML page with an audio player. Here's the HTML code:

```
<audio src="outfoxing.mp3"></audio>

<button class="paused">Play</button>
<br />
<input type="range" min="0" max="1" step="0.01" value="1" class="volume" />
```

The JavaScript code does the following:

1. It creates an `AudioContext` to manipulate the audio track:

```
const AudioContext = window.AudioContext || window.webkitAudioContext;
const audioCtx = new AudioContext();
```

2. It selects the audio, play button, and volume slider elements, and creates a `MediaElementAudioSourceNode` representing the audio source:

```
const audioElement = document.querySelector("audio");
const playBtn = document.querySelector("button");
const volumeSlider = document.querySelector(".volume");

const audioSource = audioCtx.createMediaElementSource(audioElement);
```

3. It sets up event handlers to toggle play/pause and reset the display when the audio finishes:

```
playBtn.addEventListener("click", () => {
  // Check if the audio context is in the suspended state (autoplay policy)
  if (audioCtx.state === "suspended") {
    audioCtx.resume();
  }

  // Toggle play/pause
  if (playBtn.getAttribute("class") === "paused") {
    audioElement.play();
    playBtn.setAttribute("class", "playing");
    playBtn.textContent = "Pause";
  } else if (playBtn.getAttribute("class") === "playing") {
    audioElement.pause();
    playBtn.setAttribute("class", "paused");
    playBtn.textContent = "Play";
  }
});

audioElement.addEventListener("ended", () => {
  // Reset display when audio finishes
  playBtn.setAttribute("class", "paused");
  playBtn.textContent = "Play";
});
```

4. It creates a GainNode to adjust the volume and sets up an event listener for the volume slider:

```
const gainNode = audioCtx.createGain();

volumeSlider.addEventListener("input", () => {
  // Adjust the audio graph's gain (volume) based on the slider value
  gainNode.gain.value = volumeSlider.value;
});
```

5. Finally, it connects the audio nodes together:

```
audioSource.connect(gainNode).connect(audioCtx.destination);
```

This code sets up an audio player, handles play/pause functionality, adjusts the volume, and connects the audio elements together to control playback and volume using the Web Audio API.

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 Web Audio API, this is pretty simple — it is the `AudioContext` object, which needs to be used to do any audio manipulation whatsoever.

The Document Object Model (DOM) API also has a simple 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:

```
const em = document.createElement("em"); // create a new em element
const para = document.querySelector("p"); // reference an existing p element
em.textContent = "Hello there!"; // give em some text content
para.appendChild(em); // embed em inside para
```

The Canvas API also relies on getting a context object to use to manipulate things, although in this case, it's a graphical context rather than an audio context. Its context object is created by getting a reference to the `<canvas>` element you want to draw on, and then calling its `HTMLCanvasElement.getContext()` method:

```
const canvas = document.querySelector("canvas");
const ctx = canvas.getContext("2d");
```

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

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

They often use events to handle changes in state

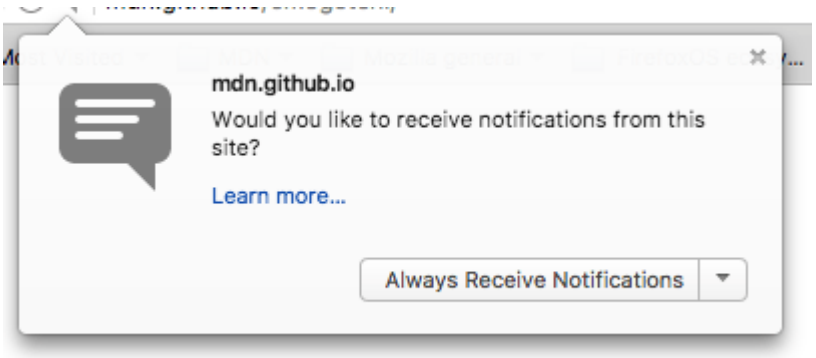
Some web APIs contain no events, but most contain at least a few. We already saw a number of event handlers in use in our Web Audio API example above:

```
// play/pause audio
playBtn.addEventListener("click", () => {
  // check if context is in suspended state (autoplay policy)
  if (audioCtx.state === "suspended") {
    audioCtx.resume();
  }

  // if track is stopped, play it
  if (playBtn.getAttribute("class") === "paused") {
    audioElement.play();
    playBtn.setAttribute("class", "playing");
    playBtn.textContent = "Pause";
    // if track is playing, stop it
  } else if (playBtn.getAttribute("class") === "playing") {
    audioElement.pause();
    playBtn.setAttribute("class", "paused");
    playBtn.textContent = "Play";
  }
});

// if track ends
audioElement.addEventListener("ended", () => {
  playBtn.setAttribute("class", "paused");
  playBtn.textContent = "Play";
});
```

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, the Notifications API asks for permission using a pop-up dialog box:



The Web Audio and `HTMLMediaElement` APIs are subject to a security mechanism called autoplay policy — this basically means that you can't automatically play audio when a page loads — you've got to allow your users to initiate audio play through a control like a button. This is done because autoplaying audio is usually really annoying and we really shouldn't be subjecting our users to it.

Introduction to web APIs

When writing web pages and apps, one of the most common things you'll want to do is manipulate the document structure in some way. This is usually done by using the Document Object Model (DOM), a set of APIs for controlling HTML and styling information that makes heavy use of the Document object. In this article we'll look at how to use the DOM in detail, along with some other interesting APIs that can alter your environment in interesting ways.