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

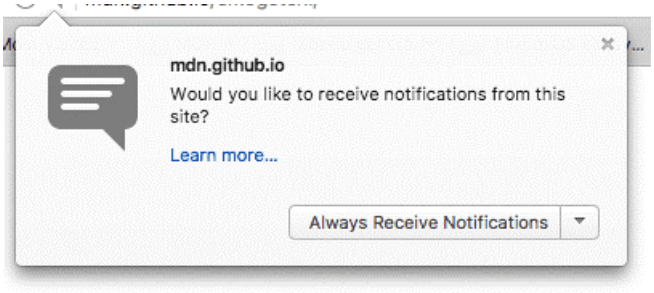
Some web APIs contain no events, but most contain at least a few. We already saw a number of event handlers in use in our Web Audio API example above:

```
// 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.

Manipulating documents

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.

The important parts of a web browser

Web browsers are very complicated pieces of software with a lot of moving parts, many of which can't be controlled or manipulated by a web developer using JavaScript.

Despite the limitations, Web APIs still give us access to a lot of functionality that enables us to do a great many things with web pages. There are a few really obvious bits you'll reference regularly in your code — consider the following diagram, which represents the main parts of a browser directly involved in viewing web pages:

- The window is the browser tab that a web page is loaded into; this is represented in JavaScript by the `Window` object. Using methods available on this object you can do things like return the window's size (see `Window.innerWidth` and `Window.innerHeight`), manipulate the document loaded into that window, store data specific to that document on the client-side, and more.
- The navigator represents the state and identity of the browser (i.e. the user-agent) as it exists on the web. In JavaScript, this is represented by the `Navigator` object. You can use this object to retrieve things like the user's preferred language, a media stream from the user's webcam, etc.
- The document (represented by the DOM in browsers) is the actual page loaded into the window, and is represented in JavaScript by the `Document` object. You can use this object to return and manipulate information on the HTML and CSS that comprises the document, for example get a reference to an element in the DOM, change its text content, apply new styles to it, create new elements and add them to the current element as children, or even delete it altogether.

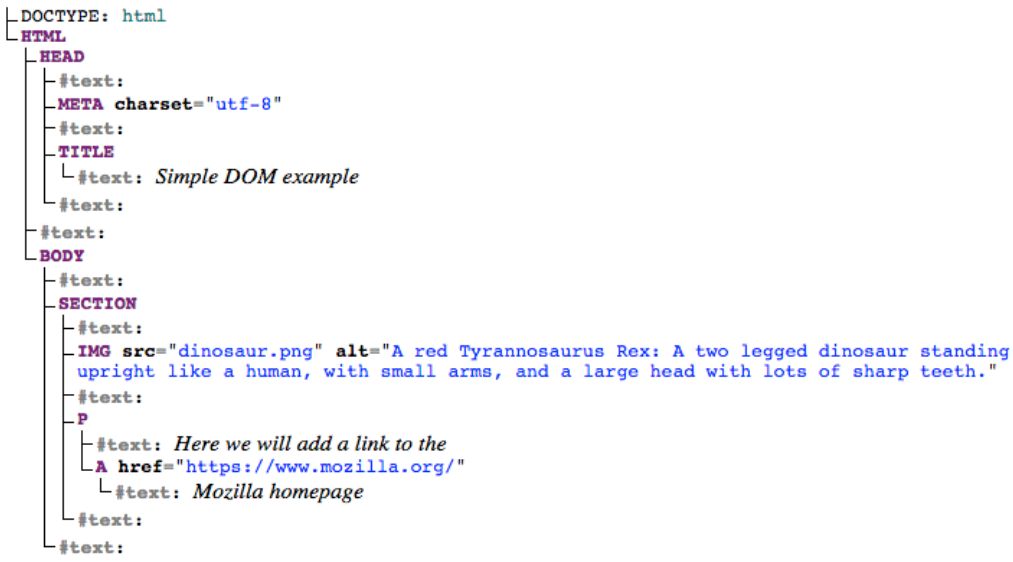
The document object model

The document currently loaded in each one of your browser tabs is represented by a document object model. This is a "tree structure" representation created by the browser that enables the HTML structure to be easily accessed by programming languages.

We have created a simple example page, the HTML source code looks like this:

```
<!DOCTYPE html>
<html lang="en-US">
  <head>
    <meta charset="utf-8" />
    <title>Simple DOM example</title>
  </head>
  <body>
    <section>
      <img src="dinosaur.png" alt="Two-legged dinosaur standing upright."/>
      <p>
        Here we will add a link to the
        <a href="https://www.mozilla.org/">Mozilla homepage</a>
      </p>
    </section>
  </body>
</html>
```

The DOM on the other hand looks like this:



Each entry in the tree is called a **node**. You can see in the diagram above that some nodes represent elements (identified as `HTML`, `HEAD`, `META` and so on) and others represent text (identified as #text). There are other types of nodes as well, but these are the main ones you'll encounter.

Nodes are also referred to by their position in the tree relative to other nodes:

- **Root node**: The top node in the tree, which in the case of `HTML` is always the `HTML` node (other markup vocabularies like `SVG` and custom `XML` will have different root elements).

- **Child node**: A node *directly* inside another node. For example, `IMG` is a child of `SECTION` in the above example.
- **Descendant node**: A node *anywhere* inside another node. For example, `IMG` is a child of `SECTION` in the above example, and it is also a descendant. `IMG` is not a child of `BODY`, as it is two levels below it in the tree, but it is a descendant of `BODY`.
- **Parent node**: A node which has another node inside it. For example, `BODY` is the parent node of `SECTION` in the above example.
- **Sibling nodes**: Nodes that sit on the same level in the `DOM` tree. For example, `IMG` and `P` are siblings in the above example.

Basic DOM manipulation

To manipulate an element inside the DOM, you first need to select it and store a reference to it inside a variable. Inside your script element, add the following line:

```
const link = document.querySelector("a");
```

Now we have the element reference stored in a variable, we can start to manipulate it using properties and methods available to it. These properties and methods are defined on interfaces of the DOM specification, which defines the structure and behavior of HTML documents. The DOM API provides a way to interact with and manipulate HTML elements using JavaScript.

First of all, let's change the text inside the link by updating the value of the Node.textContent property. Add the following line below the previous one:

```
link.textContent = "Mozilla Developer Network";
```

We should also change the URL the link is pointing to, so that it doesn't go to the wrong place when it is clicked on. Add the following line, again at the bottom:

```
link.href = "https://developer.mozilla.org";
```

Note that, as with many things in JavaScript, there are many ways to select an element and store a reference to it in a variable. `Document.querySelector()` is the recommended modern approach. It is convenient because it allows you to select elements using CSS selectors. The above `querySelector()` call will match the first `<a>` element that appears in the document. If you

wanted to match and do things to multiple elements, you could use `Document.querySelectorAll()`, which matches every element in the document that matches the selector, and stores references to them in an array-like object called a `NodeList`.

There are older methods available for grabbing element references, such as:

- `Document.getElementById()`, which selects an element with a given `id` attribute value, e.g. `<p id="myId">My paragraph</p>`. The ID is passed to the function as a parameter, i.e. `const elementRef = document.getElementById('myId')`.

- `Document.getElementsByTagName()`, which returns an array-like object containing all the elements on the page of a given type, for example `<p>`s, `<a>`s, etc. The element type is passed to the function as a parameter, i.e. `const elementRefArray = document.getElementsByTagName('p')`.

These two work better in older browsers than the modern methods like `querySelector()`, but are not as convenient. Have a look and see what others you can find!

Creating and placing new nodes

Let's go further and look at how we can create new elements.

1. Going back to the current example, let's start by grabbing a reference to our `<section>` element — add the following code at the bottom of your existing script:

```
const sect = document.querySelector("section");
```

2. Now let's create a new paragraph using `Document.createElement()` and give it some text content in the same way as before:

```
const para = document.createElement("p");
para.textContent = "We hope you enjoyed the ride.";
```

3. You can now append the new paragraph at the end of the section using `Node.appendChild()`:

```
sect.appendChild(para);
```

4. Finally for this part, let's add a text node to the paragraph the link sits inside, to round off the sentence nicely. First we will create the text node using `Document.createTextNode()`:

```
const text = document.createTextNode(
" — the premier source for web development knowledge.",
);
```

5. Now we'll grab a reference to the paragraph the link is inside, and append the text node to it:

```
const linkPara = document.querySelector("p");
linkPara.appendChild(text);
```

That's most of what you need for adding nodes to the DOM — you'll make a lot of use of these methods when building dynamic interfaces (we'll look at some examples later).

Moving and removing elements

There may be times when you want to move nodes or delete them from the DOM altogether. This is perfectly possible.

If we wanted to move the paragraph with the link inside it to the bottom of the section, we could do this:

```
sect.appendChild(linkPara);
```

If you wanted to make a copy and add that as well, you'd need to use `Node.cloneNode()` instead.

Removing a node is pretty simple as well, at least when you have a reference to the node to be removed and its parent. In our current case, we just use `Node.removeChild()`, like this:

```
sect.removeChild(linkPara);
```

When you want to remove a node based only on a reference to itself, which is fairly common, you can use `Element.remove()`:

```
linkPara.remove();
```

This method is not supported in older browsers. They have no method to tell a node to remove itself, so you'd have to do the following:

```
linkPara.parentNode.removeChild(linkPara);
```

Manipulating styles

It is possible to manipulate CSS styles via JavaScript in a variety of ways.

To start with, you can get a list of all the stylesheets attached to a document using `Document.stylesheets`, which returns an array-like object with `CSSStyleSheet` objects. You can then add/remove styles as wished. However, we're not going to expand on those features because they are a somewhat archaic and difficult way to manipulate style. There are much easier ways.

The first way is to add inline styles directly onto elements you want to **dynamically style**. This is done with the `HTMLElement.style` property, which contains inline styling information for each element in the document. You can set properties of this object to directly update element styles.

```
para.style.color = "white";
para.style.backgroundColor = "black";
para.style.padding = "10px";
para.style.width = "250px";
para.style.textAlign = "center";
```

After page reload if you look at that paragraph in your browser's Page Inspector/DOM inspector, you'll see that these lines are indeed adding inline styles to the document:

```
<p
  style="color: white; background-color: black; padding: 10px; width: 250px;
         text-align: center;">
  We hope you enjoyed the ride.
```

```
  </p>
```

There is another common way to dynamically manipulate styles on your document, which we'll look at now.

Delete the previous five lines you added to the JavaScript, and add the following inside your HTML `<head>`:

```
<style>
  .highlight {
    color: white;
    background-color: black;
    padding: 10px;
    width: 250px;
    text-align: center;
  }
</style>
```

Now we'll turn to a very useful method for general HTML manipulation – `Element.setAttribute()` — this takes two arguments, the attribute you want to set on the element, and the value you want to set it to. In this case, we will set a class name to highlight on our paragraph:

```
para.setAttribute("class", "highlight");
```

Refresh your page, and you'll see no change — the CSS is still applied to the paragraph, but this time by giving it a class that is selected by our CSS rule, not as inline CSS styles.

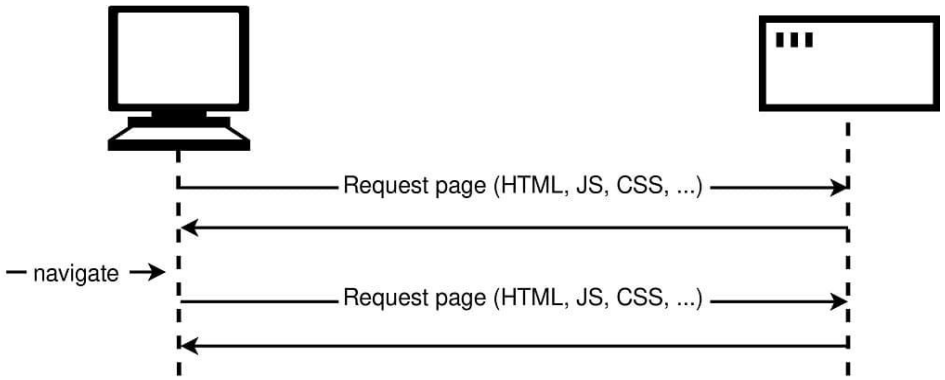
Which method you choose is up to you; both have their advantages and disadvantages. The first method takes less setup and is good for simple uses, whereas the second method is more purist (no mixing CSS and JavaScript, no inline styles, which are seen as a bad practice). As you start building larger and more involved apps, you will probably start using the second method more, but it is really up to you.

Fetching data from the server

Another very common task in modern websites and applications is retrieving individual data items from the server to update sections of a webpage without having to load an entire new page. This seemingly small detail has had a huge impact on the performance and behavior of sites, so in this article, we'll explain the concept and look at technologies that make it possible: in particular, the **Fetch API**.

What is the problem here?

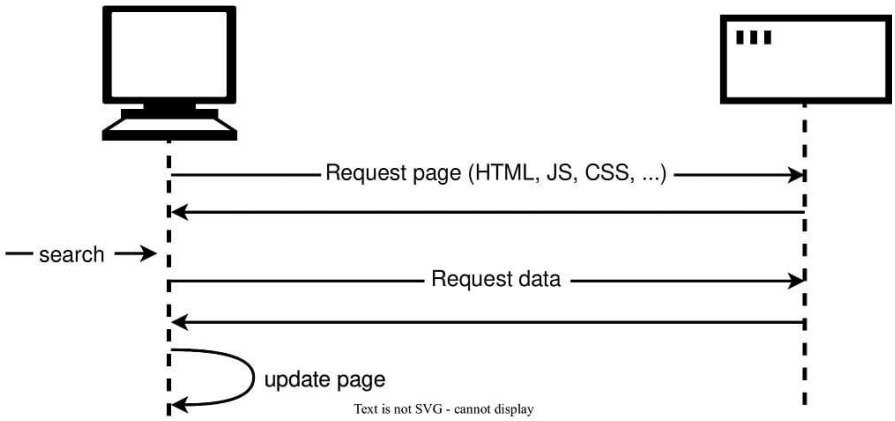
A web page consists of an HTML page and (usually) various other files, such as stylesheets, scripts, and images. The basic model of page loading on the Web is that your browser makes one or more HTTP requests to the server for the files needed to display the page, and the server responds with the requested files. If you visit another page, the browser requests the new files, and the server responds with them.



This model works perfectly well for many sites. But consider a website that's very data-driven.

The trouble with the traditional model here is that we'd **have to fetch and load the entire page**, even when we only need to update one part of it. This is inefficient and can result in a poor user experience.

So instead of the traditional model, many websites use JavaScript APIs to request data from the server and **update the page content without a page load**. So, for example when the user searches for a new product, the browser only requests the data which is needed to update the page — the set of new books to display, for instance.



The main API here is the Fetch API. This enables JavaScript running on a page to make an HTTP request to a server to retrieve specific resources. When the server provides them, the JavaScript can use the data to update the page, typically by using DOM manipulation APIs. The data requested is often JSON, which is a good format for transferring structured data, but can also be HTML or just text.

This is a common pattern for data-driven sites such as Amazon, YouTube, eBay, and so on. With this model:

- Page updates are a lot quicker and you don't have to wait for the page to refresh, meaning that the site feels faster and more responsive.
- Less data is downloaded on each update, meaning less wasted bandwidth. This may not be such a big issue on a desktop on a broadband connection, but it's a major issue on mobile devices and in countries that don't have ubiquitous fast internet service.

Note: In the early days, this general technique was known as Asynchronous JavaScript and XML (Ajax), because it tended to request XML data. This is normally not the case these days (you'd be more likely to request JSON), but the result is still the same, and the term "Ajax" is still often used to describe the technique.

To speed things up even further, some sites also store assets and data on the user's computer when they are first requested, meaning that on subsequent visits they use the local versions instead of downloading fresh copies every time the page is first loaded. The content is only reloaded from the server when it has been updated. This technique is known as Caching or Client-Side Caching.

The Fetch API

The Fetch API is a modern web standard that provides a way to make network requests (e.g., HTTP requests) from a web browser or a JavaScript application. It offers a more powerful and flexible alternative to the older XMLHttpRequest (XHR) object for handling asynchronous data retrieval. The API revolves around concepts such as Request and Response objects, CORS (Cross-Origin Resource Sharing), and the HTTP Origin header semantics.

The Fetch API is incorporated into all modern web browsers, making it available for use in web development across different platforms. The implementation of the Fetch API within browsers is typically done in lower-level languages like C++

With the Fetch API, you can initiate HTTP requests to fetch resources such as JSON data, HTML pages, images, or other types of files from a server. It supports various HTTP methods, including GET, POST, PUT, DELETE, and more.

The Fetch API uses Request and Response objects (and other things involved with network requests), as well as related concepts such as CORS and the HTTP Origin header semantics.

For making a request and fetching a resource, use the `fetch()` method. It is a global method in both `Window` and `Worker` contexts. This makes it available in pretty much any context you might want to fetch resources in.

The `fetch()` method takes one mandatory argument, the path to the resource you want to fetch. It returns a Promise that resolves to the Response to that request — as soon as the server responds with headers — even if the server response is an HTTP error status. You can also optionally pass in an `init` options object as the second argument.

Once a Response is retrieved, there are a number of methods available to define what the body content is and how it should be handled.

Here's a basic example of how to use the Fetch API to make a GET request and handle the response:

```
fetch('https://api.example.com/data') // Initiating a GET request to the specified URL
   .then(response => { // Promise resolves to the Response object
      if (!response.ok) { // Check if the response was successful (status code 200-299)
         throw new Error('Network response fail'); // Throw error if response failed
      }
      return response.json(); // Parsing the response body as JSON and returning
                              another Promise
```

```
})
.then(data => { // Promise resolves to the parsed JSON data
   // Process the retrieved data
   console.log(data); // Logging the retrieved data to the console
})
.catch(error => { // Handling any errors that occurred during the request or parsing
   // Handle any errors that occurred during the request
   console.error('Error:', error); // Logging the error message to the console
});
```

In this example, `fetch()` is called with the URL of the resource you want to fetch. The fetch() function returns a Promise that resolves to a Response object representing the server's response. The Promise is resolved asynchronously when the response is received from the server.

You can then chain `.then()` methods to handle the response. In the first `.then()`, we check if the response was successful (`response.ok`). If not, an error is thrown. Otherwise, we call `response.json()` to parse the response body as JSON and return another Promise.

The second `.then()` receives the parsed JSON data as the argument, and you can process it as needed.

If any errors occur during the request, they can be caught and handled in the `.catch()` block.

The Fetch API also allows you to set request headers, pass data in the request body, handle different types of responses (e.g., text, blob, array buffer), and more. It provides a flexible and modern way to handle network requests in JavaScript applications.

The Fetch API includes various interfaces that you can utilize:

- **fetch()**: The method used to fetch a resource.
- **Headers**: Represents response/request headers, allowing you to query them and perform actions based on the results.
- **Request**: Represents a resource request.
- **Response**: Represents the response to a request.

By leveraging these interfaces and methods, you can efficiently handle network requests, retrieve resources from servers, and process the received data in JavaScript applications.

The XMLHttpRequest API

The `XMLHttpRequest` API is a built-in browser API that allows you to make HTTP requests from JavaScript in a browser. It provides a way to communicate with servers and retrieve data without reloading the entire web page.

Here's an example of how to use the `XMLHttpRequest` API to make a GET request and handle the response:

```
// Create a new XMLHttpRequest object
const xhr = new XMLHttpRequest();

// Configure the request: GET method, URL
xhr.open('GET', 'https://api.example.com/data', true);

// Set up a callback function to handle the response
xhr.onload = function() {
  // Check if the request was successful (status code 200)
  if (xhr.status === 200) {
    // Process the response data
    const responseData = JSON.parse(xhr.responseText);
    console.log(responseData);
  } else {
    // Handle the error or non-200 status code
    console.error('Request failed. Status:', xhr.status);
  }
};

// Send the request
xhr.send();
```

In this example:

1. We create a new `XMLHttpRequest` object using the `new XMLHttpRequest()` constructor.
2. We configure the request using the `open()` method. Here, we specify the HTTP method (GET in this case), the URL of the resource we want to retrieve (`https://api.example.com/data`), and `true` to indicate that the request should be asynchronous.
3. We set up an `onload` event handler to handle the response when it's received. Inside the callback function, we check the status code (`xhr.status`) to see if the request was successful (status code 200). If it was, we process the response data by parsing it as JSON and logging it to the console. If the status code is not 200, we handle the error or non-200 status code.

4. Finally, we send the request to the server using the `send()` method.

The `XMLHttpRequest` API provides various methods and properties to handle different types of requests, set headers, handle progress events, and more. However, it has some limitations and complexities, which led to the development of newer APIs like the Fetch API that provide a more modern and flexible approach to making network requests in JavaScript applications.

The `XMLHttpRequest` API is commonly used to implement AJAX (Asynchronous JavaScript and XML) functionality in web applications. It allows you to make asynchronous HTTP requests from JavaScript, enabling dynamic updates to web content without requiring a full page reload. AJAX, using the `XMLHttpRequest` API, enhances interactivity and responsiveness of web applications, enabling features like auto-complete search, real-time updates, and data fetching without page reloads by allowing data exchange between the client and server without interrupting the user's browsing experience.

It's important to note that the term "AJAX" is often used more broadly to refer to the concept of asynchronous data exchange between the client and server, even if alternative techniques or APIs are used instead of the `XMLHttpRequest` API. For example, newer APIs like the Fetch API and the use of libraries like Axios are also commonly used for implementing AJAX functionality in modern web applications.

Third-party APIs

The APIs we've covered so far are built into the browser, but not all APIs are. Many large websites and services such as Google Maps, Twitter, Facebook, PayPal, etc. provide APIs allowing developers to make use of their data (e.g. displaying your twitter stream on your blog) or services (e.g. using Facebook login to log in your users). This article looks at the difference between browser APIs and 3rd party APIs and shows some typical uses of the latter.

What are third party APIs?

Third party APIs are APIs provided by third parties — generally companies such as Facebook, Twitter, or Google — to allow you to access their functionality via JavaScript and use it on your site. One of the most obvious examples is using mapping APIs to display custom maps on your pages.

Let's look at a Simple Mapquest API example, and use it to illustrate how third-party APIs differ from browser APIs.

They are found on third-party servers

Browser APIs are built into the browser — you can access them from JavaScript immediately. For example, the Web Audio API we saw in the Introductory article is accessed using the native `AudioContext` object. For example:

```
const audioCtx = new AudioContext();
// …
const audioElement = document.querySelector("audio");
// …
const audioSource = audioCtx.createMediaElementSource(audioElement);
// etc.
```

Third party APIs, on the other hand, are located on third party servers. To access them from JavaScript you first need to connect to the API functionality and make it available on your page. This typically involves first linking to a JavaScript library available on the server via a `<script>` element, as seen in our Mapquest example:

```
<script
  src="https://api.mqcdn.com/sdk/mapquest-js/v1.3.2/mapquest.js"
  defer></script>
<link
  rel="stylesheet"
  href="https://api.mqcdn.com/sdk/mapquest-js/v1.3.2/mapquest.css" />
```

You can then start using the objects available in that library. For example:

```
const map = L.mapquest.map("map", {
  center: [53.480759, -2.242631],
  layers: L.mapquest.tileLayer("map"),
  zoom: 12,
});
```

Here we are creating a variable to store the map information in, then creating a new map using the `mapquest.map()` method, which takes as its parameters the ID of a `<div>` element you want to display the map in ('map'), and the coordinates of the center of the map, a map layer of type map to show (created using the `mapquest.tileLayer()` method), and the default zoom level.

They usually require API keys

Security for browser APIs tends to be handled by permission prompts. The purpose of these is so that the user knows what is going on in the websites they visit and is less likely to fall victim to someone using an API in a malicious way.

Third-party APIs have a slightly different permissions system — they tend to use developer keys to allow developers access to the API functionality, which is more to protect the API vendor than the user.

You'll find a line similar to the following in the Mapquest API example:

```
L.mapquest.key = "YOUR-API-KEY-HERE";
```

This line specifies an API or developer key to use in your application — the developer of the application must apply to get a key, and then include it in their code to be allowed access to the API's functionality. In our example, we've just provided a placeholder.]]

Requiring a key enables the API provider to hold users of the API accountable for their actions. When the developer has registered for a key, they are then known to the API provider, and action can be taken if they start to do anything malicious with the API (such as tracking people's location or trying to spam the API with loads of requests to stop it working, for example). The easiest action would be to just revoke their API privileges.

RESTful API

Now let's look at another API example — the New York Times API. This API allows you to retrieve New York Times news story information and display it on your site. This type of API is known as a RESTful API — instead of getting data using the features of a JavaScript library like we did with Mapquest, we get data by making HTTP requests to specific URLs, with data like search terms and other properties encoded in the URL (often as URL parameters). This is a common pattern you'll encounter with APIs.

A RESTful API, or Representational State Transfer API, is an architectural style for designing networked applications. It is commonly used in web development to provide a standardized way for different systems to communicate with each other over the internet.

RESTful APIs are based on a set of principles and constraints that define how the API should behave. These principles include:

1. Client-Server Architecture: The API separates the client (user interface) and the server (data storage), allowing them to evolve independently.
2. Stateless: Each request from the client to the server contains all the necessary information for the server to understand and process the request. The server does not maintain any client-specific state between requests.
3. Uniform Interface: The API uses a consistent set of well-defined methods and standard protocols, such as HTTP, to interact with resources. The standard methods include GET (retrieve a resource), POST (create a new resource), PUT (update a resource), and DELETE (remove a resource).
4. Resource-Based: Resources, such as data objects or services, are identified by unique URLs (Uniform Resource Locators). Clients can interact with these resources using the standard HTTP methods.
5. Representations: Resources can have multiple representations, such as JSON, XML, or HTML. Clients can request a specific representation format based on their needs.
6. Hypermedia as the Engine of Application State (HATEOAS): The API provides links or hypermedia in the response, allowing clients to navigate through the API dynamically by following these links.

By following these principles, RESTful APIs promote scalability, simplicity, and interoperability. They have become the de facto standard for building web services and are widely used in various industries for integrating different systems and enabling communication between applications.

REST API vs RESTful API

A REST API is an application programming interface that adheres strictly to the principles and constraints of the REST architectural style. A **REST API strictly implements all the core principles/rules of REST**, providing a consistent and standardized approach to interact with web services.

A RESTful API is an informal term used to describe an API that is designed and implemented following the principles of REST architecture. **RESTful API tries to implement the principles of REST, but it might not be 100% following all guidelines**. It allows for some flexibility or adaptations in its design and implementation while still being inspired by the REST architectural style.

In summary, the main difference lies in the strictness to the REST principles. A REST API strictly adheres to all the principles and constraints, while a RESTful API aims to follow the principles but allows for some flexibility in its implementation. Both terms refer to APIs that share the fundamental characteristics of REST, providing a scalable and standardized way for clients to interact with servers over the internet.

Overview of Different API Types and Their Characteristics

There are various types of APIs, each with its own characteristics and purposes. Here are some common API types:

1. **RESTful API**: A RESTful API follows the principles of the REST architectural style, using HTTP methods like GET, POST, PUT, and DELETE to interact with resources over the internet. It emphasizes a resource-centric approach and is widely used for building web services.
2. **RPC-style API**: RPC (Remote Procedure Call) APIs focus on exposing remote functions or procedures that can be invoked by clients. Clients send requests specifying the function to be executed along with any necessary parameters. The server processes the request and returns the result.
3. **SOAP-based API**: SOAP (Simple Object Access Protocol) APIs use the SOAP protocol for communication. They typically use XML for data exchange and rely on a rigid contract-based approach. SOAP APIs often have complex payloads and provide extensive support for features like security, reliability, and transactions.
4. **GraphQL API**: GraphQL is a query language and runtime for APIs. With GraphQL APIs, clients can specify precisely the data they need using a flexible and powerful query syntax. The server responds with a JSON payload containing only the requested data, reducing over-fetching or under-fetching of data.
5. **WebSocket API**: WebSocket APIs provide full-duplex communication channels between clients and servers. They enable real-time, bidirectional communication, allowing for constant data updates without the need for frequent polling. WebSocket APIs are suitable for applications requiring live data, such as chat applications or collaborative tools.

These are just a few examples of API types, and there are others as well, such as gRPC (a high-performance RPC framework) and event-driven APIs. The choice of API type depends on the specific requirements of your application, the level of real-time interaction needed, and the preferences of your development team.

Drawing graphics

The browser contains some very powerful graphics programming tools, from the Scalable Vector Graphics (SVG) language, to APIs for drawing on HTML <canvas> elements, (see The Canvas API and WebGL). This article provides an introduction to canvas, and further resources to allow you to learn more.

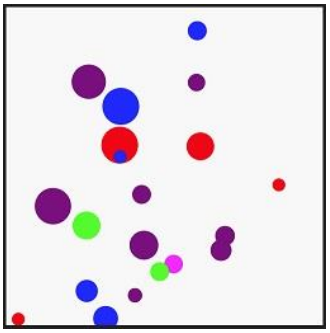
Graphics on the Web

The Web was originally just text, which was very boring, so images were introduced — first via the <img> element and later via CSS properties such as background-image, and SVG.

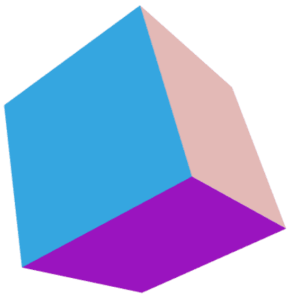
This however was still not enough. While you could use CSS and JavaScript to animate (and otherwise manipulate) SVG vector images — as they are represented by markup — there was still no way to do the same for bitmap images, and the tools available were rather limited. The Web still had no way to effectively create animations, games, 3D scenes, and other requirements commonly handled by lower level languages such as C++ or Java.

The situation started to improve when browsers began to support the <canvas> element and associated **Canvas API** in 2004. As you'll see below, canvas provides some useful tools for creating 2D animations, games, data visualizations, and other types of applications, especially when combined with some of the other APIs the web platform provides, but can be difficult or impossible to make accessible.

The below example shows a simple 2D canvas-based bouncing balls animation that we originally met in our Introducing JavaScript objects module:



Around 2006–2007, Mozilla started work on an experimental 3D canvas implementation. This became **WebGL**, which gained traction among browser vendors, and was standardized around 2009–2010. WebGL allows you to create real 3D graphics inside your web browser; the below example shows a simple rotating WebGL cube:



Active learning: Getting started with a <canvas>

This example demonstrates how to get started with an HTML <canvas> element and perform some basic setup for drawing on it using JavaScript.

```
<!DOCTYPE html>
<html>
<head>
  <title>Canvas Example</title>
  <style>
    canvas {
      border: 1px solid black;
    }
  </style>
</head>
<body>
  <canvas id="myCanvas" width="320" height="240">
    <p>Canvas not supported.</p>
  </canvas>

  <script>
    const canvas = document.getElementById("myCanvas");
    const ctx = canvas.getContext("2d");

    ctx.fillStyle = "rgb(0, 0, 0)";
    ctx.fillRect(0, 0, canvas.width, canvas.height);
  </script>
</body>
</html>
```

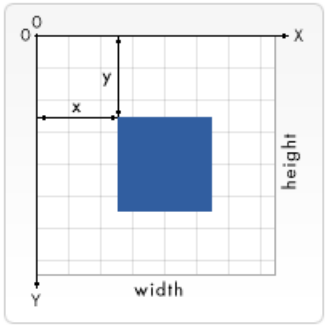
This code sets up a canvas element with a fallback content (`<p>Canvas not supported.</p>`) for browsers that don't support the canvas feature. Inside the script tag, it retrieves the canvas

element using its ID, gets the 2D rendering context, sets the fill style to black, and fills the entire canvas with a black rectangle.

When you open this HTML file in a browser, it will display a black rectangle inside the canvas element with a size of 320 pixels by 240 pixels. The canvas element provides a drawing area where you can create various 2D and 3D graphics using JavaScript and the HTML5 canvas API.

2D canvas basics

As we said above, all drawing operations are done by manipulating a `CanvasRenderingContext2D` object (in our case, ctx). Many operations need to be given coordinates to pinpoint exactly where to draw something — the top left of the canvas is point (0, 0), the horizontal (x) axis runs from left to right, and the vertical (y) axis runs from top to bottom.



Drawing shapes tends to be done using the rectangle shape primitive, or by tracing a line along a certain path and then filling in the shape. Below we'll show how to do both.

Simple rectangles

```
<!DOCTYPE html>
<html>
<head>
  <title>Canvas Example</title>
  <style>
    canvas {
      border: 1px solid black;
    }
  </style>
</head>
<body>
  <canvas id="myCanvas" width="320" height="240">
```

```
      <p>Canvas not supported.</p>
    </canvas>

    <script>
      const canvas = document.getElementById("myCanvas");
      const ctx = canvas.getContext("2d");

      ctx.fillStyle = "rgb(255, 0, 0)";
      ctx.fillRect(50, 50, 100, 150);

      ctx.fillStyle = "rgb(0, 255, 0)";
      ctx.fillRect(75, 75, 100, 100);

      ctx.fillStyle = "rgba(255, 0, 255, 0.75)";
      ctx.fillRect(25, 100, 175, 50);
    </script>
  </body>
  </html>
```

In this example, we first retrieve the canvas element and its 2D rendering context. Then we use the `fillStyle` property to set the fill color, and the `fillRect` method to draw rectangles on the canvas.

The first rectangle is red (`rgb(255, 0, 0)`) and has its top-left corner positioned at (`50, 50`). It is 100 pixels wide and 150 pixels tall.

The second rectangle is green (`rgb(0, 255, 0)`) and has its top-left corner positioned at (`75, 75`). It is 100 pixels wide and 100 pixels tall.

The third rectangle is semi-transparent purple (`rgba(255, 0, 255, 0.75)`) and has its top-left corner positioned at (`25, 100`). It is 175 pixels wide and 50 pixels tall.

Save and refresh the HTML file in a browser, and you will see the canvas displaying the three rectangles in the specified positions and colors. Feel free to experiment and add more rectangles of your own!

Strokes and line widths

So far we've looked at drawing filled rectangles, but you can also draw rectangles that are just outlines (called strokes in graphic design). To set the color you want for your stroke, you use the `strokeStyle` property; drawing a stroke rectangle is done using `strokeRect`.

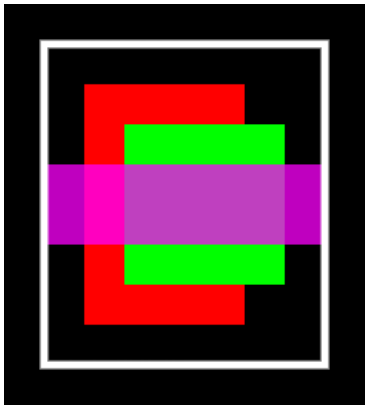
Add the following to the previous example, again below the previous JavaScript lines:

```
ctx.strokeStyle = "rgb(255, 255, 255)";
ctx.strokeRect(25, 25, 175, 200);
```

The default width of strokes is 1 pixel; you can adjust the `lineWidth` property value to change this (it takes a number representing the number of pixels wide the stroke is). Add the following line in between the previous two lines:

```
ctx.lineWidth = 5;
```

Now you should see that your white outline has become much thicker! That's it for now. At this point your example should look like this:



Drawing paths

If you want to draw anything more complex than a rectangle, you need to draw a path. Basically, this involves writing code to specify exactly what path the pen should move along on your canvas to trace the shape you want to draw. Canvas includes functions for drawing straight lines, circles, Bézier curves, and more.

Let's start the section off by making a fresh copy of our canvas template (1_canvas_template), in which to draw the new example.

We'll be using some common methods and properties across all of the below sections:

- `beginPath()` — start drawing a path at the point where the pen currently is on the canvas. On a new canvas, the pen starts out at (0, 0).
- `moveTo()` — move the pen to a different point on the canvas, without recording or tracing the line; the pen "jumps" to the new position.
- `fill()` — draw a filled shape by filling in the path you've traced so far.
- `stroke()` — draw an outline shape by drawing a stroke along the path you've drawn so far.
- You can also use features like `lineWidth` and `fillStyle/strokeStyle` with paths as well as rectangles.

A typical, simple path-drawing operation would look something like so:

```
ctx.fillStyle = "rgb(255, 0, 0)";
ctx.beginPath();
ctx.moveTo(50, 50);
// draw your path
ctx.fill();
```

Drawing lines

Let's draw an equilateral triangle on the canvas.

1. First of all, add the following helper function to the bottom of your code. This converts degree values to radians, which is useful because whenever you need to provide an angle value in JavaScript, it will nearly always be in radians, but humans usually think in degrees.

```
function degToRad(degrees) {
  return (degrees * Math.PI) / 180;
}
```

2. Next, start off your path by adding the following below your previous addition; here we set a color for our triangle, start drawing a path, and then move the pen to (50, 50) without drawing anything. That's where we'll start drawing our triangle.

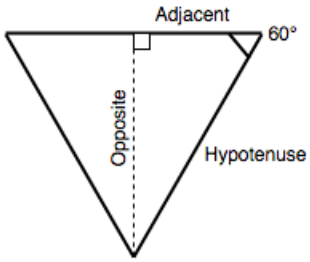
```
ctx.fillStyle = "rgb(255, 0, 0)";
ctx.beginPath();
ctx.moveTo(50, 50);
```

3. Now add the following lines at the bottom of your script:

```
ctx.lineTo(150, 50);
const triHeight = 50 * Math.tan(degToRad(60));
ctx.lineTo(100, 50 + triHeight);
ctx.lineTo(50, 50);
ctx.fill();
```

Let's run through this in order: First we draw a line across to (150, 50) — our path now goes 100 pixels to the right along the x axis. Second, we work out the height of our equilateral triangle, using a bit of simple trigonometry. Basically, we are drawing the triangle pointing downwards. The angles in an equilateral triangle are always 60 degrees; to work out the height we can split it down the middle into two right-angled triangles, which will each have angles of 90 degrees, 60 degrees, and 30 degrees. In terms of the sides:

- The longest side is called the **hypotenuse**
- The side next to the 60 degree angle is called the **adjacent** — which we know is 50 pixels, as it is half of the line we just drew.
- The side opposite the 60 degree angle is called the **opposite**, which is the height of the triangle we want to calculate.



One of the basic trigonometric formulae states that the length of the adjacent multiplied by the tangent of the angle is equal to the opposite, hence we come up with `50 * Math.tan(degToRad(60))`. We use our `degToRad()` function to convert 60 degrees to radians, as `Math.tan()` expects an input value in radians.

4. With the height calculated, we draw another line to `(100, 50 + triHeight)`. The X coordinate is simple; it must be halfway between the previous two X values we set. The Y value on the other hand must be 50 plus the triangle height, as we know the top of the triangle is 50 pixels from the top of the canvas.

5. The next line draws a line back to the starting point of the triangle.

6. Last of all, we run `ctx.fill()` to end the path and fill in the shape.

Drawing circles

Now let's look at how to draw a circle in canvas. This is accomplished using the arc() method, which draws all or part of a circle at a specified point.

1. Let's add an arc to our canvas — add the following to the bottom of your code:

```
ctx.fillStyle = "rgb(0, 0, 255)";
ctx.beginPath();
ctx.arc(150, 106, 50, degToRad(0), degToRad(360), false);
ctx.fill();
```

`arc()` takes six parameters. The first two specify the position of the arc's center (X and Y, respectively). The third is the circle's radius, the fourth and fifth are the start and end angles at which to draw the circle (so specifying 0 and 360 degrees gives us a full circle), and the sixth parameter defines whether the circle should be drawn counterclockwise (anticlockwise) or clockwise (false is clockwise).

2. Let's try adding another arc:

```
ctx.fillStyle = "yellow";
ctx.beginPath();
ctx.arc(200, 106, 50, degToRad(-45), degToRad(45), true);
ctx.lineTo(200, 106);
ctx.fill();
```

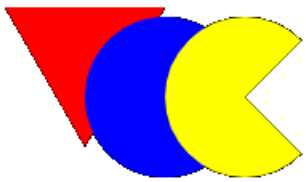
The pattern here is very similar, but with two differences:

- We have set the last parameter of `arc()` to true, meaning that the arc is drawn counterclockwise, which means that even though the arc is specified as starting at -45 degrees and ending at 45 degrees, we draw the arc around the 270 degrees not inside this portion. If you were to change true to false and then re-run the code, only the 90 degree slice of the circle would be drawn.

- Before calling `fill()`, we draw a line to the center of the circle. This means that we get the rather nice Pac-Man-style cutout rendered. If you removed this line (try it!) then re-ran the code, you'd get just an edge of the circle chopped off

between the start and end point of the arc. This illustrates another important point of the canvas — if you try to fill an incomplete path (i.e. one that is not closed), the browser fills in a straight line between the start and end point and then fills it in.

That's it for now; your final example should look like this:



Text

Canvas also has features for drawing text. Text is drawn using two methods:

- fillText() — draws filled text.
- `strokeText()` — draws outline (stroke) text.

Both of these take three properties in their basic usage: the text string to draw and the X and Y coordinates of the point to start drawing the text at. This works out as the **bottom left** corner of the **text box** (literally, the box surrounding the text you draw), which might confuse you as other drawing operations tend to start from the top left corner — bear this in mind.

There are also a number of properties to help control text rendering such as font, which lets you specify font family, size, etc. It takes as its value the same syntax as the CSS font property.

Canvas content is not accessible to screen readers. Text painted to the canvas is not available to the DOM, but must be made available to be accessible. In this example, we include the text as the value for aria-label.

Try adding the following block to the bottom of your JavaScript:

```
ctx.strokeStyle = "white";
ctx.lineWidth = 1;
ctx.font = "36px arial";
ctx.strokeText("Canvas text", 50, 50);

ctx.fillStyle = "red";
ctx.font = "48px georgia";
ctx.fillText("Canvas text", 50, 150);
```

```
canvas.setAttribute("aria-label", "Canvas text");
```

Here we draw two lines of text, one outline and the other stroke. The final example should look like so:

Drawing images onto canvas

The HTML5 canvas element provides a powerful feature for rendering graphics, including images. In this lesson, we will learn how to draw images on a canvas using JavaScript.

Step 1: Setting up the Canvas.

- Create an HTML file and add a canvas element to it. Give it an id for easy reference.

```
<canvas id="myCanvas"></canvas>
```

Step 2: Drawing an Image on the Canvas.

- Get a reference to the element using its id, and 2D rendering context of the canvas.

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

- Create a new Image object and set its source to the path or URL of the image you want to draw.

```
const image = new Image();
image.src = "path/to/image.jpg";
```

- Wait for the image to load using the `load` event. Add an event listener to the image and use the `drawImage()` method inside the event handler.

```
image.addEventListener("load", () => { ctx.drawImage(image, 0, 0); });
```

- Save the changes and open the HTML file in a browser. You should see the image displayed on the canvas.

Step 3: Manipulating the Image.

To display only a part of the image or resize it, you can use the more complex version of `drawImage()`. It takes additional parameters to define the area of the image to cut out and the position and size of the drawn image on the canvas.

```
ctx.drawImage(image, sourceX, sourceY, sourceWidth, sourceHeight, destX, destY,
              destWidth, destHeight);
```

- `sourceX, sourceY`: The coordinates of the top-left corner of the area to cut out from the image.
- `sourceWidth, sourceHeight`: The width and height of the area to cut out.
- `destX, destY`: The coordinates of the top-left corner where the image should be drawn on the canvas.
- `destWidth, destHeight`: The width and height at which to draw the image.

Loops and animations

You won't experience the full power of canvas unless you update or animate it in some way. After all, canvas does provide scriptable images! If you aren't going to change anything, then you might as well just use static images and save yourself all the work.

Creating a loop

In this lesson, we will learn how to use loops in canvas to draw rotating triangles. It's a fun way to play with canvas and create interesting designs.
Step 1: Setting up the Canvas:

- Create an HTML file and add a canvas element to it. Give it an id for easy reference.

```
<canvas id="myCanvas"></canvas>
```

Step 2: Drawing Rotating Triangles:

- Create a JavaScript file (e.g., script.js) and open it.
- Get a reference to the element via id, and get the 2D rendering context of the canvas.

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

- Set the origin point of the canvas to the center using the `translate()` method.

```
const width = canvas.width;
const height = canvas.height;
ctx.translate(width / 2, height / 2);
```

- Define utility functions for converting degrees to radians and generating random numbers, and set initial values for the length and `moveOffset` variables.

```
function degToRad(degrees) {
    return (degrees * Math.PI) / 180;
}

function rand(min, max) {
    return Math.floor(Math.random() * (max - min + 1)) + min;
}

let length = 250; let moveOffset = 20;
```

- Use a for loop to draw the rotating triangles.

```
for (let i = 0; i < length; i++) {
  ctx.fillStyle = `rgba(${255 - length}, 0, ${255 - length}, 0.9)`;
  ctx.beginPath();
  ctx.moveTo(moveOffset, moveOffset);
  ctx.lineTo(moveOffset + length, moveOffset);
  const triHeight = (length / 2) * Math.tan(degToRad(60));
  ctx.lineTo(moveOffset + length / 2, moveOffset + triHeight);
```

```
    ctx.lineTo(moveOffset, moveOffset);
    ctx.fill();

    length--;
    moveOffset += 0.7;
    ctx.rotate(degToRad(5));
  }
```

That's it! The final example should look like so:

Animations

Canvas animations involve continuously updating and redrawing the canvas to create smooth and dynamic visual effects. They create the illusion of movement by continuously updating and redrawing elements on the canvas.

To achieve smooth animations, the `requestAnimationFrame()` method is used to schedule callback functions that update and redraw the canvas before the next repaint. This method ensures synchronization with the browser's rendering loop. If an animation needs to be stopped before completion, the `cancelAnimationFrame()` method can be called, providing the unique identifier returned by `requestAnimationFrame()`. This effectively halts the animation loop and prevents further updates to the canvas.

A simple character animation

```
<canvas id="myCanvas"></canvas>
<script>
  const canvas = document.getElementById("myCanvas");
  const ctx = canvas.getContext("2d");
  const spriteWidth = 100; // Width of each frame in the sprite sheet
  const spriteHeight = 100; // Height of each frame in the sprite sheet
```

```
    const totalFrames = 6; // Total number of frames in the sprite sheet
    const frameWidth = spriteWidth / totalFrames; // Width of each frame on the canvas
    const spriteSheet = new Image();
    spriteSheet.src = "spritesheet.png";
    let currentFrame = 0; // Index of the current frame in the sprite sheet

    function animate() {
      ctx.clearRect(0, 0, canvas.width, canvas.height);
      ctx.drawImage(
        spriteSheet,
        currentFrame * spriteWidth,
        0,
        spriteWidth,
        spriteHeight,
        0,
        0,
        frameWidth,
        spriteHeight
      );

      currentFrame = (currentFrame + 1) % totalFrames;
      requestAnimationFrame(animate);
    }

    spriteSheet.addEventListener("load", () => {
      animate();
    });
  </script>
```

Here how spritesheet looks like:

Explanation:

1. The HTML file includes a canvas element where the animation will be displayed.
2. In JavaScript, we get a reference to the canvas and the 2D rendering context.
3. We define the dimensions of each frame in the sprite sheet (`spriteWidth` and `spriteHeight`) and the total number of frames (totalFrames).

4. We calculate the width of each frame on the canvas (frameWidth) by dividing the sprite width by the total frames.
5. We create a new `Image` object and set its source to the sprite sheet image file (`spritesheet.png`).
6. The `animate` function is called to start the animation loop.
7. Inside the `animate` function, we clear the canvas using `clearRect()`.
8. We use `drawImage()` to draw the current frame of the sprite sheet on the canvas. By specifying the source rectangle from the sprite sheet (`currentFrame * spriteWidth`), we can display the correct portion of the sprite sheet. The destination rectangle on the canvas is specified as `(0, 0, frameWidth, spriteHeight)`.
9. We increment the `currentFrame` index and wrap it around to `0` when it exceeds the total number of frames. This creates the looping effect for the animation.
10. The animation loop is created using `requestAnimationFrame(animate)` to continuously update and redraw the canvas at the browser's optimal frame rate.
11. Finally, we add an event listener to the `spriteSheet` image object to start the animation once the image has finished loading.

This example demonstrates how to animate a character walking using a sprite sheet. By updating the `currentFrame` index and drawing the corresponding frame on the canvas, we create the illusion of movement.

WebGL

It's now time to leave 2D behind, and take a quick look at 3D canvas. 3D canvas content is specified using the **WebGL API**, which is a completely **separate API from the 2D canvas API**, even though they both render onto <canvas> elements.

WebGL is based on OpenGL (Open Graphics Library), and allows you to communicate directly with the computer's GPU. As such, writing raw WebGL is closer to low level languages such as C++ than regular JavaScript; it is quite complex but incredibly powerful.

Using a library

Because of its complexity, most people write 3D graphics code using a third party JavaScript library such as Three.js, PlayCanvas, or Babylon.js. Most of these work in a similar way, providing functionality to create primitive and custom shapes, position viewing cameras and lighting, covering surfaces with textures, and more. They handle the WebGL for you, letting you work on a higher level.

Yes, using one of these means learning another new API (a third-party one, in this case), but they are a lot simpler than coding raw WebGL.

Creating 3D rotating cube in Three.js

```
const scene = new THREE.Scene();
const camera = new THREE.PerspectiveCamera (
   75, window.innerWidth / window.innerHeight, 0.1, 1000);

const renderer = new THREE.WebGLRenderer();
renderer.setSize(window.innerWidth, window.innerHeight);
document.body.appendChild(renderer.domElement);

// Create Cube
const geometry = new THREE.BoxGeometry(1, 1, 1);
const material = new THREE.MeshBasicMaterial({ color: 0x00ff00 });
const cube = new THREE.Mesh(geometry, material);
scene.add(cube);

// Animation
function animate() {
  requestAnimationFrame(animate);
  cube.rotation.x += 0.01;
  cube.rotation.y += 0.01;
  renderer.render(scene, camera);
}

// Start Animation
animate();
```

Explanation:

1. In JavaScript, we set up the basic elements for our scene: `scene`, `camera`, and `renderer`. The `scene` represents the 3D environment, the `camera` determines the perspective and view of the scene, and the `renderer` handles rendering the scene on the canvas.
2. We create a cube using `BoxGeometry` with a width, height, and depth of 1. Then we create a `MeshBasicMaterial` with a green color for the cube.
3. We add the cube to the scene using `scene.add(cube)`.
4. The `animate` function is created to handle the animation loop using `requestAnimationFrame`. It is called recursively to continuously update and render the scene.
5. Inside the `animate` function, we update the rotation of the cube by incrementing its `rotation.x` and rotation.y properties. This creates a smooth rotating effect.

6. Finally, we call `renderer.render(scene, camera)` to render the updated scene with the cube's rotation onto the canvas.

When you open this HTML file in a browser, you should see a rotating 3D cube displayed on the screen. The cube will continue to rotate indefinitely due to the animation loop created by `requestAnimationFrame` in the `animate` function.

Video and Audio APIs

HTML comes with elements for embedding rich media in documents — `<video>` and `<audio>` — which in turn come with their own APIs for controlling playback, seeking, etc. This article shows you how to do common tasks such as creating custom playback controls.

HTML video and audio

The `<video>` and `<audio>` elements allow us to embed video and audio into web pages. As we showed in Video and audio content, a typical implementation looks like this:

```
<video controls>
  <source src="rabbit320.mp4" type="video/mp4" />
  <source src="rabbit320.webm" type="video/webm" />
  <p>
    Your browser doesn't support HTML video. Here is a
    <a href="rabbit320.mp4">link to the video</a> instead.
  </p>
</video>
```

You can review what all the HTML features do in the article linked above; for our purposes here, the most interesting attribute is controls, which enables the default set of playback controls. If you don't specify this, you get no playback controls.

This is not as immediately useful for video playback, but it does have advantages. One big issue with the native browser controls is that they are different in each browser — not very good for cross-browser support! Another big issue is that the native controls in most browsers aren't very keyboard-accessible.

You can solve both these problems by hiding the native controls (by removing the controls attribute), and programming your own with HTML, CSS, and JavaScript. In the next section, we'll look at the basic tools we have available to do this.

The HTMLMediaElement API

The HTMLMediaElement API provides more flexibility and control over the media playback. With the API, you can programmatically control various aspects of the media element, such as **starting and pausing playback, changing the source dynamically, seeking to a specific time, adjusting the volume, and handling custom events**. This is useful when you need to create a custom user interface or implement specific functionality beyond what the default controls offer.

Using the HTMLMediaElement API allows you to have complete control over the media playback experience, but it also requires more coding and implementation effort compared to using the native controls.

```
<!DOCTYPE html>
<html>
<head>
  <title>HTMLMediaElement API Example</title>
</head>
<body>
  <video id="myVideo" src="video.mp4"></video>
  <button id="playButton">Play</button>
  <button id="pauseButton">Pause</button>

  <script>
    // Get references to the video element and control buttons
    const video = document.getElementById('myVideo');
    const playButton = document.getElementById('playButton');
    const pauseButton = document.getElementById('pauseButton');
```

```
    // Add click event listeners to the buttons
    playButton.addEventListener('click', () => {
      video.play(); // play() method is part of the HTMLMediaElement API
    });

    pauseButton.addEventListener('click', () => {
      video.pause(); // pause() method is part of the HTMLMediaElement API
    });
  </script>
</body>
</html>
```

In this example, we have a video element with an id of "myVideo" and two buttons with id attributes of "playButton" and "pauseButton". When the "Play" button is clicked, the play() method is called on the video element, and it starts playing. Similarly, when the "Pause" button is clicked, the pause() method is called to pause the video.

Note that this is a basic example, and the HTMLMediaElement API provides more advanced functionality and additional methods.

Client-side storage

Modern web browsers support a number of ways for websites to store data on the user's computer — with the user's permission — then retrieve it when necessary. This lets you persist data for long-term storage, save sites or documents for offline use, retain user-specific settings for your site, and more. This article explains the very basics of how these work.

Client-side storage?

We talked about the difference between static sites and dynamic sites. Most major modern websites are dynamic — they store data on the server using some kind of database (server-side storage), then run server-side code to retrieve needed data, insert it into static page templates, and serve the resulting HTML to the client to be displayed by the user's browser.

Client-side storage works on similar principles, but has different uses. It consists of JavaScript APIs that allow you to store data on the client (i.e. on the user's machine) and then retrieve it when needed. This has many distinct uses, such as:

- Personalizing site preferences (e.g. showing a user's choice of custom widgets, color scheme, or font size).
- Persisting previous site activity (e.g. storing the contents of a shopping cart from a previous session, remembering if a user was previously logged in).
- Saving data and assets locally so a site will be quicker (and potentially less expensive) to download, or be usable without a network connection.
- Saving web application-generated documents locally for use offline

Often client-side and server-side storage are used together. For example, you could download a batch of music files (perhaps used by a web game or music player application), store them inside a client-side database, and play them as needed. The user would only have to download the music files once — on subsequent visits, they would be retrieved from the database instead.

Old school: Cookies

The concept of client-side storage has been around for a long time. Since the early days of the web, sites have used cookies to store information to personalize user experience on websites. They're the earliest form of client-side storage commonly used on the web.

These days, there are easier mechanisms available for storing client-side data, however, they are still used commonly to store data related to user personalization and state, e.g. session IDs and access tokens.

New school: Web Storage and IndexedDB

The "easier" features we mentioned above are as follows:

- The Web Storage API provides a mechanism for storing and retrieving smaller, data items consisting of a name and a corresponding value. This is useful when you just need to store some simple data, like the user's name, whether they are logged in, what color to use for the background of the screen, etc.
- The IndexedDB API provides the browser with a complete database system for storing complex data. This can be used for things from complete sets of customer records to even complex data types like audio or video files.

The Cache API

The `Cache` API is designed for storing HTTP responses to specific requests, and is very useful for doing things like storing website assets offline so the site can subsequently be used without a network connection. Cache is usually used in combination with the Service Worker API, although it doesn't have to be.

The use of Cache and Service Workers is an advanced topic, and we won't be covering it in great detail in this article, although we will show an example in the Offline asset storage section below.

Storing simple data — web storage

The Web Storage API is very easy to use — you store simple name/value pairs of data (limited to strings, numbers, etc.) and retrieve these values when needed.

Basic syntax

Here's an explanation of the basic syntax with example code:

1. Storing Data: To store data in web storage, you use the `setItem()` method provided by the `localStorage` object. The syntax is as follows:

```
localStorage.setItem("name", "Chris");
```

In this example, the name "`Chris`" is stored with the key "`name`" in the web storage.

2. Retrieving Data: To retrieve data from web storage, you use the `getItem()` method provided by the `localStorage` object. The syntax is as follows:

```
let myName = localStorage.getItem("name");
```

In this example, the value stored with the key "`name`" is retrieved and assigned to the `myName` variable.

3. Removing Data: To remove data from web storage, you use the `removeItem()` method provided by the `localStorage` object. The syntax is as follows:

```
localStorage.removeItem("name");
```

In this example, the item with the key "`name`" is removed from the web storage.

It's important to note that the `localStorage` object is used in this example, which persists data even after the browser is closed and reopened. If you want data to be available only during the browser session, you can use the `sessionStorage` object instead.

After executing these operations, you can check the value of the `myName` variable or use the `getItem()` method again to see if the data item exists or has been removed.

The data persists!

One key feature of web storage is that the **data persists between page loads** (and even when the browser is shut down, in the case of `localStorage`). Let's look at this in action.

1. Storing Data:

```
localStorage.setItem("name", "Chris");
```

This line of code stores the value "`Chris`" with the key "`name`" in the web storage (`localStorage`).

2. Retrieving Data:

```
let myName = localStorage.getItem("name"); myName;
```

These lines retrieve the value associated with the key "`name`" from the web storage and assign it to the variable `myName`. By accessing `myName`, you should see the value "`Chris`" being displayed.

3. Persisting Data Between Browser Sessions: Now, close the browser completely and reopen it.

4. Retrieving Data Again:

```
let myName = localStorage.getItem("name"); myName;
```

After reopening the browser, these lines of code retrieve the value associated with the key "`name`" from the web storage. The variable `myName` should still hold the value "`Chris`".

The key feature of web storage, especially `localStorage`, is that the stored data persists between page loads and even when the browser is shut down and reopened. In this example, the value "`Chris`" remains available in the web storage even after closing and reopening the browser.

Separate storage for each domain

There is a separate data store for each domain (each separate web address loaded in the browser). You will see that if you load two websites (say google.com and amazon.com) and try storing an item on one website, it won't be available on the other website.

This makes sense — you can imagine the security issues that would arise if websites could see each other's data!

Storing complex data — IndexedDB

The IndexedDB API (sometimes abbreviated IDB) is a complete database system available in the browser in which you can store complex related data, the types of which aren't limited to simple values like strings or numbers. You can store videos, images, and pretty much anything else in an IndexedDB instance.

The IndexedDB API allows you to create a database, then create object stores within that database. Object stores are like tables in a relational database, and each object store can contain a number of objects. To learn more about the IndexedDB API, see Using IndexedDB.

However, this does come at a cost: IndexedDB is much more complex to use than the Web Storage API. In this section, we'll really only scratch the surface of what it is capable of, but we will give you enough to get started.

Database Initial Setup

```
let db;
const openRequest = window.indexedDB.open("notes_db", 1);

openRequest.addEventListener(
   "error", () => console.error("Database failed to open"));

openRequest.addEventListener("success", () => {
  db = openRequest.result;
  displayData();
});
```

```
openRequest.addEventListener("upgradeneeded", (e) => {
  db = e.target.result;

  const objectStore = db.createObjectStore(
     "notes_os", { keyPath: "id", autoIncrement: true });
  objectStore.createIndex("title", "title", { unique: false });
  objectStore.createIndex("body", "body", { unique: false });

  console.log("Database setup complete");
});
```

In this example, we initialize the IndexedDB database by creating an instance of it. We specify the database name as "`notes_db`" and the version as 1. The `openRequest` variable holds the request to open the database.

We add event listeners to handle errors, success, and upgrades. If the database opening encounters an error, an error message is logged. If the database opens successfully, the `db` variable is assigned to the result, and the `displayData()` function is called. During an upgrade, the `upgradeneeded` event is triggered, allowing us to define the structure of the object store. Here, we create an object store called "`notes_os`" with an auto-incrementing key and two indexes for the title and body fields.

Adding Data to the Database

```
function addData(note) {
  const transaction = db.transaction(["notes_os"], "readwrite");
  const objectStore = transaction.objectStore("notes_os");
  const addRequest = objectStore.add(note);

  addRequest.addEventListener("success", () => {
    console.log("Note added successfully");
    displayData();
  });

  transaction.addEventListener("error", () =>
    console.log("Transaction not opened due to error")
  );
}
```

The `addData` function takes a `note` object as a parameter and adds it to the "`notes_os`" object store. It starts a read/write transaction, accesses the object store, and uses the `add()` method to add the note.

After the note is successfully added, a success event is triggered, and the `displayData()` function is called to update the display. If an error occurs during the transaction, an error message is logged.

Displaying the Data

```
function displayData() {
  const objectStore = db.transaction("notes_os").objectStore("notes_os");

  while (list.firstChild) {
    list.removeChild(list.firstChild);
  }

  objectStore.openCursor().addEventListener("success", (e) => {
    const cursor = e.target.result;

    if (cursor) {
      const listItem = createListItem(cursor.value);
      list.appendChild(listItem);

      cursor.continue();
    }
  });
}

function createListItem(note) {
  const listItem = document.createElement("li");
  const h3 = document.createElement("h3");
  const para = document.createElement("p");

  h3.textContent = note.title;
  para.textContent = note.body;

  listItem.appendChild(h3);
  listItem.appendChild(para);

  return listItem;
}
```

The `displayData` function retrieves data from the "`notes_os`" object store and displays it on the page. It starts a transaction, accesses the object store, and uses the `openCursor()` method to iterate over the records.

Inside the success event handler for the cursor, the function creates a list item using the `createListItem` helper function. The `createListItem` function takes a note object and creates

HTML elements representing the note's title and body. The list item is then appended to the `<ul>` element.

The cursor continues to the next record until there are no more records.

Deleting a Note

```
function deleteItem(e) {
  const noteId = parseInt(e.target.parentNode.getAttribute("data-note-id"));
  const transaction = db.transaction(["notes_os"], "readwrite");
  const objectStore = transaction.objectStore("notes_os");
  const deleteRequest = objectStore.delete(noteId);

  deleteRequest.addEventListener("success", () => {
    console.log("Note deleted successfully");
    displayData();
  });

  transaction.addEventListener("error", () =>
    console.log("Transaction not opened due to error")
  );
}
```

The `deleteItem` function is triggered when the delete button associated with a note is clicked. It retrieves the note's ID from the data attribute of the parent list item. A read/write transaction is initiated, and the note is deleted from the "`notes_os`" object store using the `delete()` method. After the deletion is successful, the `displayData()` function is called to update the display. If an error occurs during the transaction, an error message is logged.

Storing Complex Data via IndexedDB

As we mentioned above, IndexedDB can be used to store more than just text strings. You can store just about anything you want, including complex objects such as video or image blobs. And it isn't much more difficult to achieve than any other type of data.

In this example, we'll demonstrate how to store video blobs in an IndexedDB database and display them in a web page. The code snippets provided will highlight the most relevant parts of the example.

Storing the Video Data:

First, we define an array of video names that we want to fetch and store in the IndexedDB database:

```
const videos = [
  { name: "crystal" },
  { name: "elf" },
  { name: "frog" },
  { name: "pig" },
  { name: "rabbit" },
];
```

Inside the `init()` function, we iterate through the video names. We check if each video exists in the database using `objectStore.get()`. If the video is present, we display it from IndexedDB using the `displayVideo()` function. Otherwise, we fetch it from the network using `fetchVideoFromNetwork()`:

```
function init() {
  for (const video of videos) {
    const objectStore = db.transaction("videos_os").objectStore("videos_os");
    const request = objectStore.get(video.name);
    request.addEventListener("success", () => {
      if (request.result) {
        displayVideo(
          request.result.mp4,
          request.result.webm,
          request.result.name
        );
      } else {
        fetchVideoFromNetwork(video);
      }
    });
  }
}
```

Inside the `fetchVideoFromNetwork()` function, we use `fetch()` to fetch the MP4 and WebM versions of the video. We extract the response bodies as blobs using the `response.blob()` method. Since both fetch requests are asynchronous, we use `Promise.all()` to wait for both promises to fulfill. Then, we display the video using `displayVideo()` and store it in IndexedDB using `storeVideo()`:

```
const mp4Blob = fetch(`videos/${video.name}.mp4`).then((response) =>
  response.blob()
```

```
  );
  const webmBlob = fetch(`videos/${video.name}.webm`).then((response) =>
    response.blob()
  );

  Promise.all([mp4Blob, webmBlob]).then((values) => {
    displayVideo(values[0], values[1], video.name);
    storeVideo(values[0], values[1], video.name);
  });
```

The `storeVideo()` function opens a `readwrite` transaction, gets a reference to the object store, creates an object representing the video record, and adds it to the database using `IDBObjectStore.add()`:

```
  function storeVideo(mp4, webm, name) {
    const objectStore = db
      .transaction(["videos_os"], "readwrite")
      .objectStore("videos_os");

    const request = objectStore.add({ mp4, webm, name });

    request.addEventListener("success", () =>
      console.log("Record addition attempt finished")
    );
    request.addEventListener("error", () => console.error(request.error));
  }
```

Displaying the Video Data:

The `displayVideo()` function creates the necessary DOM elements to embed the video in the web page. It also converts the video blobs into object URLs using `URL.createObjectURL()` and sets them as the source URLs for the `<source>` elements within the `<video>` element:

```
  function displayVideo(mp4Blob, webmBlob, title) {
    const mp4URL = URL.createObjectURL(mp4Blob);
    const webmURL = URL.createObjectURL(webmBlob);
    const article = document.createElement("article");
    const h2 = document.createElement("h2");
    h2.textContent = title;
    const video = document.createElement("video");
    video.controls = true;
    const source1 = document.createElement("source");
    source1.src = mp4URL;
    source1.type = "video/mp4";
```

```
    const source2 = document.createElement("source");
    source2.src = webmURL;
    source2.type = "video/webm";

    section.appendChild(article);
    article.appendChild(h2);
    article.appendChild(video);
    video.appendChild(source1);
    video.appendChild(source2);
  }
```

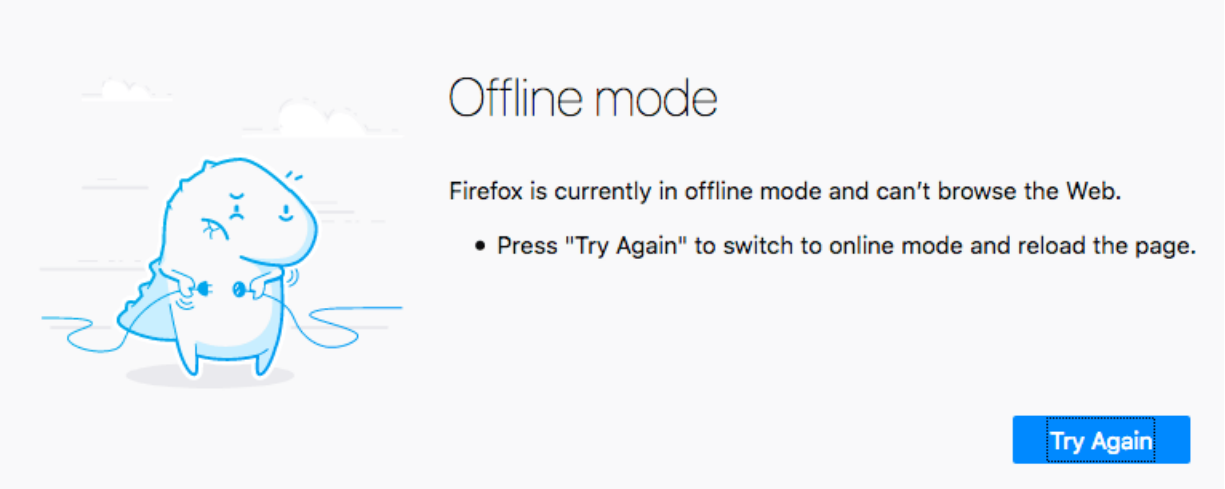
The `displayVideo()` function creates the necessary HTML structure and appends it to the page. The video blobs are displayed within the `<video>` element, allowing users to control and view the videos.

These code snippets demonstrate how to store complex data (video blobs) in IndexedDB and retrieve and display them in a web page.

Please note that this example assumes you have already initialized and set up the IndexedDB database and object store appropriately.

Offline asset storage

The above example already shows how to create an app that will store large assets in an IndexedDB database, avoiding the need to download them more than once. This is already a great improvement to the user experience, but there is still one thing missing — the main HTML, CSS, and JavaScript files still need to be downloaded each time the site is accessed, meaning that it won't work when there is no network connection.



This is where Service workers and the closely-related Cache API come in.

A service worker is a JavaScript file that is registered against a particular origin (website, or part of a website at a certain domain) when it is accessed by a browser. When registered, it can control pages available at that origin. It does this by sitting between a loaded page and the network and intercepting network requests aimed at that origin.

When it intercepts a request, it can do anything you wish to it (see use case ideas), but the classic example is saving the network responses offline and then providing those in response to a request instead of the responses from the network. In effect, it allows you to make a website work completely offline.

The Cache API is another client-side storage mechanism, with a bit of a difference — it is designed to save HTTP responses, and so works very well with service workers.

A service worker example

Here's an explanation and code example that demonstrates offline functionality using the Cache API and IndexedDB.

Registering the Service Worker:

In the main JavaScript file (e.g., index.js), we check if the `serviceWorker` property exists in the `navigator` object to detect if service workers are supported. If supported, we use the `navigator.serviceWorker.register()` method to register the service worker located in the `sw.js` file. This registration allows the service worker to control pages within the same directory or subdirectories:

```
if ("serviceWorker" in navigator) {
    navigator.serviceWorker
        .register("/sw.js")
        .then(() => console.log("Service Worker Registered"));
}
```

Installing the Service Worker:

When a page controlled by the service worker is accessed, the service worker is installed. The `install` event is triggered, and you can write code inside the service worker to respond to the

installation. In the `sw.js` file, we listen for the install event and use the `waitUntil()` method to ensure the installation is completed successfully. Inside the `waitUntil()` promise, we use the `caches.open()` method to open a new cache object, and then we use `cache.addAll()` to fetch and store specific assets in the cache:

```
self.addEventListener("install", (e) => {
  e.waitUntil(
    caches.open("video-store").then((cache) =>
      cache.addAll([
        "/index.html",
        "/style.css",
        "/index.js",
        "/video1.mp4",
        "/video2.mp4",
      ])
    )
  );
});
```

Responding to Requests:

After the service worker is installed, it can respond to further network requests. In the `sw.js` file, we listen for the `fetch` event and log the requested asset's URL. We then use `caches.match()` to check if a matching request is found in the cache. If a match is found, we return the cached response. Otherwise, we fetch the response from the network:

```
self.addEventListener("fetch", (e) => {
   console.log(e.request.url);
   e.respondWith(
      caches.match(e.request).then((response) => response || fetch(e.request))
   );
});
```

Testing Offline Functionality:

To test the offline functionality, you need to load the page multiple times to ensure the service worker is installed. Once installed, you can simulate being offline by disconnecting from the network, enabling the "Work Offline" option in Firefox, or using Chrome DevTools to go offline. When you refresh the page in offline mode, it should still load successfully since the assets are stored in the cache and served by the service worker.

This example demonstrates how a service worker can cache and serve assets, allowing a web app to function offline. The Cache API is used to store static assets, while the service worker intercepts network requests to serve responses from the cache when available.

Please note that you need to adjust the file paths in the example to match your specific file structure.